



Rust at scale: packages, crates, and modules

📅 2022-01-20 ⏪ 2023-06-16 🍷

Dramatis Personae

Modules vs Crates

Advice on code organization

Common pitfalls

- Confusing crates and packages

- Quasi-circular dependencies

Conclusion

Further reading

Good judgment is the result of experience and experience the result of bad judgment.

Attributed to Mark Twain.

The Internet Computer (IC) Rust code base grew from an empty repository in June 2019 to almost 350,000 lines of code in early 2022. This rapid growth taught me that decisions working fine for relatively small projects might start dragging the project down over time. This article evaluates Rust code organization options and suggests ways to use them effectively.

Dramatis Personae

Rust terminology proved to be confusing because the term *crate* is overloaded. For example, the first edition of the venerable The Rust Programming Language book contained the following misleading passage

Rust has two distinct terms that relate to the module system: ‘crate’ and ‘module’. A crate is synonymous with a ‘library’ or ‘package’ in other languages. Hence “Cargo” as the name of Rust’s package management tool: you ship your crates to others with Cargo. Crates can produce an executable or a library, depending on the project.

“The Rust Programming Language”, version 1.25.0

Wait a minute, a library and a package are different concepts, aren’t they? Mixing up these concepts leads to frustration, even if you already have a few months of Rust exposure. Tooling conventions also contribute to the confusion: If a Rust package defines a library crate, cargo automatically derives the library name from the package name¹.

Let’s familiarize ourselves with the concepts we’ll be dealing with.

Module

A module is the unit of code organization. It is a container for functions, types, and nested modules.

Modules also specify the visibility for the names they define or re-export.

Crate

A crate is the unit of compilation and linking. Crates are part of the language (*crate* is a keyword), but you don’t mention them much in the source code. Libraries and executables are the most common crate types.

Package

A package is the unit of software distribution. Packages are not part of the language but artifacts of the Rust package manager, Cargo. Packages can contain one or more crates: at most one library and any number of executables.

Modules vs Crates

When you factor a large codebase into components, there are two extremes: to have a few large packages with lots of modules or to have lots of small packages.

Having few packages with lots of modules has some advantages:

- Adding or removing a module is less work than adding or removing a package.

- Modules are more flexible. For example, modules in the same crate can form a dependency cycle: module `foo` can use definitions from module `bar`, which in turn can use definitions from module `foo`. In contrast, the package dependency graph must be acyclic.

- You don't have to modify your `Cargo.toml` file every time you rearrange your modules.

In the ideal world where Rust compiles instantly, turning the repository into a massive package with many modules would be the most convenient setup. The bitter reality is that Rust takes quite some time to compile, and modules don't help you shorten the compilation time:

- The basic unit of compilation is a *crate*, not a *module*. You must recompile all the modules in a crate even if you change only one. The more code you put in a crate, the longer it

takes to compile.

Cargo parallelizes compilations across crates, not within a crate. You don't use the full potential of your multi-core CPU if you have a few large packages.

It boils down to the tradeoff between convenience and compilation speed. Modules are convenient but don't help the compiler do less work. Packages are less convenient but deliver better compilation speed as the code base grows.

Advice on code organization



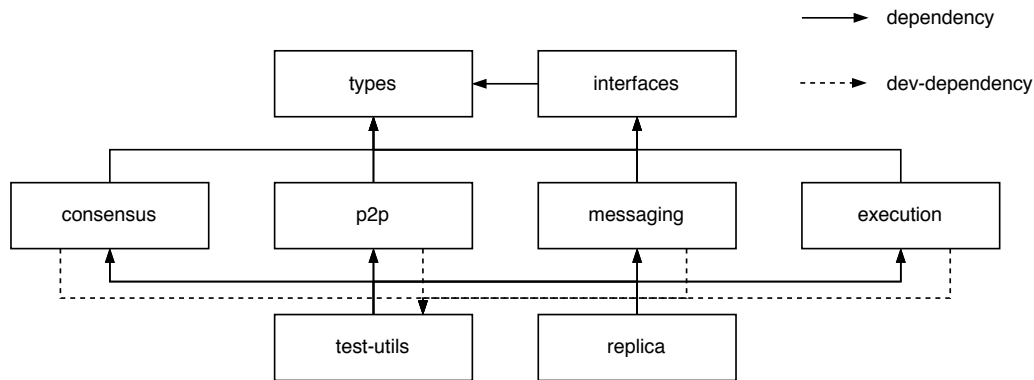
Split dependency hubs.

There are two types of dependency hubs:

Packages with lots of dependencies. Two examples from the IC codebase are the `test-utils` package containing auxiliary code for integration tests (proptest strategies, mock and fake component implementations, helper functions, etc.), and the `replica` package instantiating all the components.

Packages with lots of *reverse dependencies*. Examples from the IC codebase are the `types` package containing common type definitions and the `interfaces` package specifying component interfaces.





Dependency hubs are undesirable because of their devastating effect on incremental compilation. If you modify a package with many reverse dependencies (e.g., `types`), cargo must to recompile all those dependencies to check your change.

Sometimes it is possible to eliminate a dependency hub. For example, the `test-utils` package is a union of independent utilities. We can group these utilities by the component they help to test and factor the code into multiple `<component>-test-utils` packages.

More often, however, dependency hubs will have to stay. Some types from `types` are pervasive. The package containing these types is doomed to be a type-two dependency hub. The `replica` package wiring all the components is doomed to be a type-one dependency hub. The best you can do is to localize the hubs and make them small and stable.

☛ *Consider using generics and associated types to eliminate dependencies.*

This advice needs an example, so bear with me.

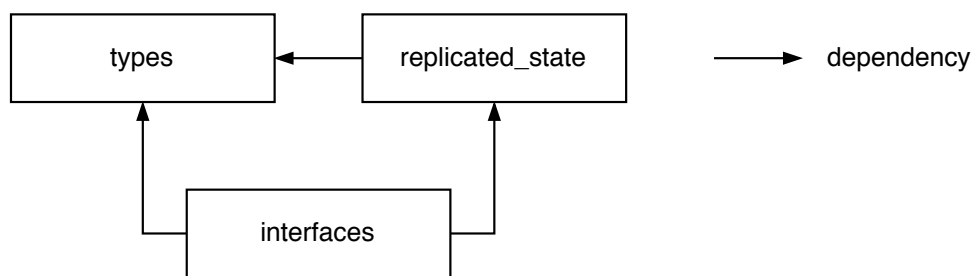
`types`, `interfaces`, and `replicated_state` were among the first packages in the IC code base. The `types` package contains common type definitions, the `interfaces` package defines traits for software

components, and the `replicated_state` package defines IC's replicated state machine data structures, with the `ReplicatedState` type at the root.

But why do we need the `types` package? Types are an integral part of the interface. Why not define them in the `interfaces` package?

The problem is that some interfaces refer to the `ReplicatedState` type. And the `replicated_state` package depends on type definitions from the `types` package. If all the types lived in the `interfaces` package, there would be a circular dependency between `interfaces` and `replicated_state`.

⊕



When we need to break a circular dependency, we can move common definitions into a new package or merge some packages. The `replicated_state` package is heavy; we didn't want to merge its contents with `interfaces`. So we took the first option and moved the types shared between `interfaces` and `replicated_state` into the `types` package.

One property of trait definitions in the `interfaces` package is that the traits depend only on the `ReplicatedState` type *name*. The traits do not need to know `ReplicatedState`'s definition.

⊕

```
trait StateManager {  
  fn get_latest_state(&self) → ReplicatedState;  
  
  fn commit_state(&self, state: ReplicatedState, version: Version
```

}

This property allows us to break the direct dependency between interfaces and `replicated_state`. We only need to replace the exact type with a generic type argument.

⊕

```
trait StateManager {  
    type State; //< We turned a specific type into an associated ty  
  
    fn get_latest_state(&self) → State;  
  
    fn commit_state(&self, state: State, version: Version);  
}
```

Now, we don't need to recompile the interfaces package and its numerous dependencies every time we add a new field to the replicated state.



Prefer runtime polymorphism.

One of the design choices we had was how to connect software components. Should we pass instances of components as `Arc<dyn Interface>` (runtime polymorphism) or as generic type arguments (compile-time polymorphism)?

⊕

```
pub struct Consensus {  
    artifact_pool: Arc<dyn ArtifactPool>,  
    state_manager: Arc<dyn StateManager>,  
}
```

⊕

```
pub struct Consensus<AP: ArtifactPool, SM: StateManager> {  
    artifact_pool: AP,  
    state_manager: SM,  
}
```

Compile-time polymorphism is an essential tool but a heavy-weight one. Runtime polymorphism requires less code and results in less binary bloat. Most team members also found the `dyn` version easier to read.



Prefer explicit dependencies.

One of the most common questions new developers ask on the dev channel is “Why do we explicitly pass around loggers? Global loggers seem to work pretty well.” What a great question. I would ask the same thing in 2019!

Global variables are *bad*, but my previous experience suggested that loggers and metric sinks are special. Oh well, they aren’t, after all.

The usual problems with implicit state dependencies are especially prominent in Rust.

Most Rust libraries do not rely on true global variables. The usual way to pass an implicit state is to use a thread-local variable, which can become problematic when you spawn a new thread. New threads tend to inherit and retain unexpected values of thread locals.

Cargo runs tests within a test binary in parallel by default. The test output might become an intangible mess if you’re not careful with threading loggers through the call stack. The problem usually manifests when a background thread needs to access the log. Explicitly passing loggers eliminates that

problem.

Testing code relying on an implicit state often becomes hard or impossible in a multi-threaded environment. The code recording your metrics is, well, *code*. It also deserves to be tested.

If you use a library relying on implicit state, you can introduce subtle bugs if you depend on incompatible library versions in different packages.

The latter point desperately needs an example. So here is a little detective story.

We use the prometheus package for metrics recording. This package can keep the metrics registry in a global variable.

One day, we discovered a bug: we could not see metrics from some of our components. Our code seemed correct, yet the metrics were missing.

One of the packages depended on prometheus version 0.9, while all other packages used 0.10. According to semver, these versions are incompatible, so cargo linked both versions into the binary, introducing *two* implicit registries. We exposed only the 0.10 version registry over the HTTP interface. As you correctly guessed, the missing components recorded metrics to the 0.9 registry.

Passing loggers, metrics registries, and async runtimes explicitly turns a runtime bug into a compile-time error. Switching to explicit passing the metrics registry helped me find and fix the bug.

The official documentation of the venerable slog package also recommends passing loggers explicitly:

The reason is: manually passing `Logger` gives maximum flexibility. Using `slog_scope` ties the logging data structure to the stacktrace, which is not the same a logical structure of your

software. Especially libraries should expose full flexibility to their users, and not use implicit logging behaviour.

Usually `Logger` instances fit pretty neatly into data structures in your code representing resources, so it's not that hard to pass them in constructors, and use `info!(self.log, ...)` everywhere.

slog FAQ

By passing state implicitly, you gain temporary convenience but make your code less clear, less testable, and more error-prone. Every type of resource we passed implicitly² caused hard-to-diagnose issues and wasted a lot of engineering time.

People in other programming communities also realized that global loggers are evil. You might enjoy reading *Logging Without a Static Logger*.



Deduplicate dependencies.

Cargo makes it easy to add dependencies, but this convenience comes with a cost. You might accidentally introduce incompatible version of the same package.

Multiple versions of the same package might result in correctness issues, especially with packages with zero major version component (`0.y.z`). If you depend on versions `0.1` and `0.2` of the same package in a single binary, cargo will link both versions into the executable. If you ever pulled your hair off trying to figure out why you get that “there is no reactor running” error, you know how painful these issues can be to debug.

Workspace dependencies and cargo update will help you keep your dependency graph in order.

You do not have to unify the feature sets for the same dependency across the workspace packages. Cargo compiles each dependency version once, thanks to the feature unification mechanism.



Put unit tests into separate files.

Rust allows you to write unit tests right next to your production code.

⊕

```
pub fn frobnicate(x: &Foo) → u32 {
    todo!("implement frobnication")
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_frobnication() {
        assert!(frobnicate(&Foo::new()), 5);
    }
}
```

This feature is very convenient, but it can slow down test compilation time. Cargo build cache can get confused when you modify the file, tricking cargo into re-compiling the crate under both dev and test profiles, even if you touched only the test part. By trial and error, we discovered that the issue does not occur if the tests live in a separate file.

⊕

```
pub fn frobnicate(x: &Foo) → u32 {
    todo!("implement frobnication")
}

// The contents of the module moved to foo/tests.rs.
#[cfg(test)]
```

```
mod tests;
```

This technique tightened our edit-check-test loop and made the code easier to navigate.

Common pitfalls

This section describes common issues Rust newcomers might run into. I experienced these issues myself and saw several colleagues struggling with them.

Confusing crates and packages

Imagine you have package `image-magic` defining a library for image processing and providing a command-line utility for image transformation called `transmogrify`. Naturally, you want to use the library to implement `transmogrify`.

⊕

```
[package]
name = "image-magic"
version = "1.0.0"
edition = 2018

[lib]

[[bin]]
name = "transmogrify"
path = "src/transmogrify.rs"

# dependencies...
```

Now you open `transmogrify.rs` and write something like the following:

```
use crate::{Image, transform_image}; //< Compile error.
```

The compiler will become upset and tell you something like

```
error[E0432]: unresolved imports `crate::Image`, `crate::transform_image`
  → src/transmogrify.rs:1:13
   |
1 | use crate::{Image, transform_image};
   |             ^^^^^^  ^^^^^^^^^^^^^^^^^^^^^ no `transform_image` in the root
   |             |
   |             no `Image` in the root
```

Oh, how is that? Aren't `lib.rs` and `transmogrify.rs` in the same *crate*? No, they are not. The `image-magic` *package* defines two *crates*: a *library crate* named `image_magic` (note that cargo replaced the dash in the package name with an underscore) and a *binary crate* named `transmogrify`.

So when you write `use crate::Image` in `transmogrify.rs`, you tell the compiler to look for the type defined in the same binary. The `image_magic` *crate* is just as external to `transmogrify` as any other library would be, so we have to specify the library name in the use declaration:

```
use image_magic::{Image, transform_image}; //< OK.
```

Quasi-circular dependencies

To understand this issue, we'll first learn about Cargo build profiles. Build profiles are named compiler configurations. For example:

release

The profile for production binaries. Highest optimization level, disabled debug assertions, long

compile times. Cargo uses this profile when you run `cargo build --release`.

dev

The profile for the normal development cycle. Debug asserts and overflow checks are enabled, optimizations are disabled for faster compile times. Cargo uses this profile when you run `cargo build`.

test

Mostly the same as the *dev* profile. When you test a library crate, cargo builds the library with the test profile and injects the main function executing the test harness. This profile is enabled when you run `cargo test`. Cargo builds dependencies of the crate under test using the *dev* profile.

Imagine now that you have a package with a library `foo`. You want good test coverage and the tests to be easy to write. So you introduce another package with many test utilities for `foo`, `foo-test-utils`.

It also feels natural to use `foo-test-utils` for testing the `foo` itself. Let's add `foo-test-utils` as a dev dependency of `foo`.

⊕

```
[package]
name = "foo"
version = "1.0.0"
edition = "2018"

[lib]

[dev-dependencies]
foo-test-utils = { path = "../foo-test-utils" }
```

⊕

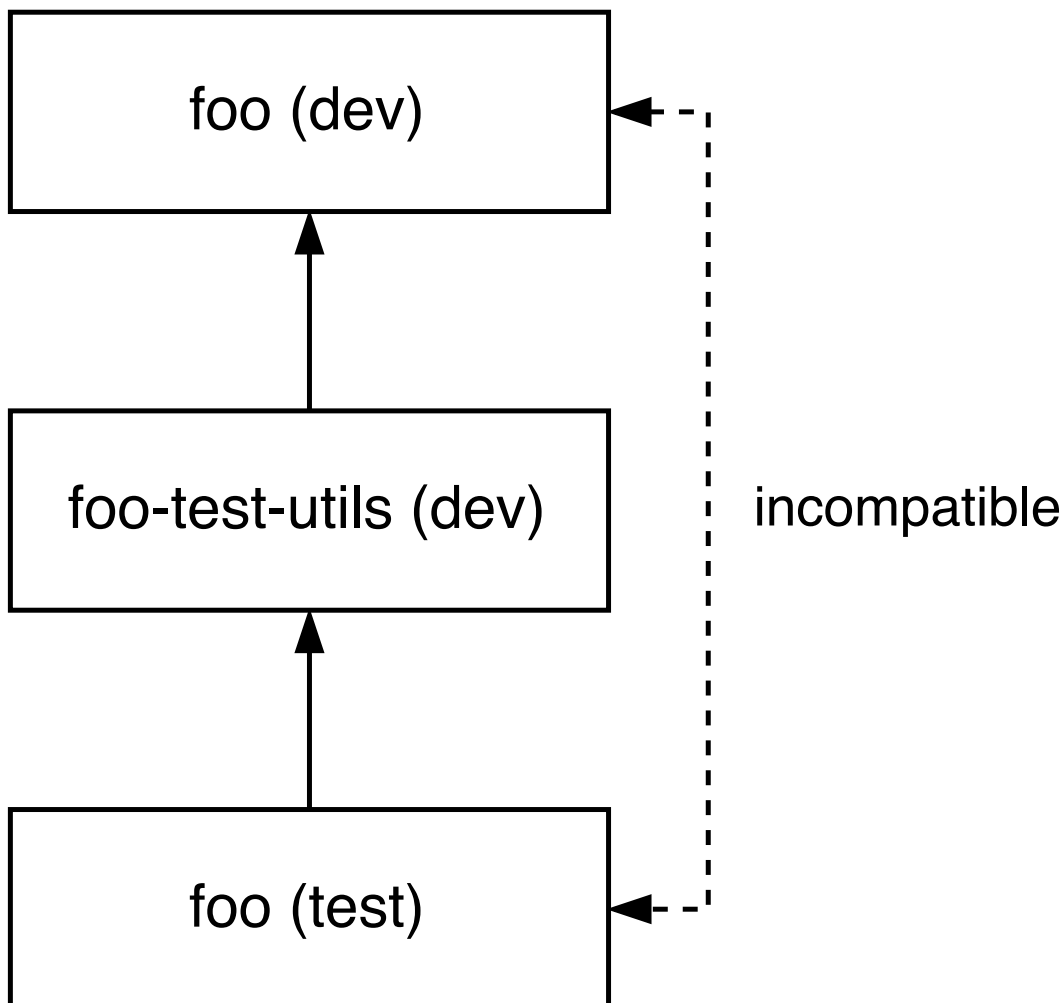
```
[package]
```

```
name = "foo-test-utils"  
version = "1.0.0"  
edition = "2018"  
  
[lib]  
  
[dependencies]  
foo = { path = "../foo" }
```

Wait, didn't we create a dependency cycle? `foo` depends on `foo-test-utils` that depends on `foo`, right?

There is no circular dependency because cargo compiles `foo` twice: once with *dev* profile to link with `foo-test-utils` and once with *test* profile to add the test harness.

⊕



Time to write some tests!

⊕

```
use foo::Foo;

pub fn make_test_foo() → Foo {
    Foo {
        name: "John Doe".to_string(),
        age: 32,
    }
}
```

⊕

```
#[derive(Debug)]
pub struct Foo {
    pub name: String,
    pub age: u32,
}

fn private_fun(x: &Foo) → u32 {
    x.age / 2
}

pub fn frobnicate(x: &Foo) → u32 {
    todo!("complete frobnication")
}

#[test]
fn test_private_fun() {
    let x = foo_test_utils::make_test_foo();
    private_fun(&x);
}
```

However, when we try to run `cargo test -p foo`, we get a cryptic compile error:

```
error[E0308]: mismatched types
  → src/lib.rs:14:17
   |
14 |     private_fun(&x);
   |                  ^^ expected struct `Foo`, found struct `foo`
   |
   = note: expected reference `&Foo`
```



```
found reference `&foo::Foo`
```

What could that mean? The compiler tells us that type definitions in the *test* and the *dev* versions of `foo` are incompatible. Technically, these are different, incompatible crates even though these crates share the name.

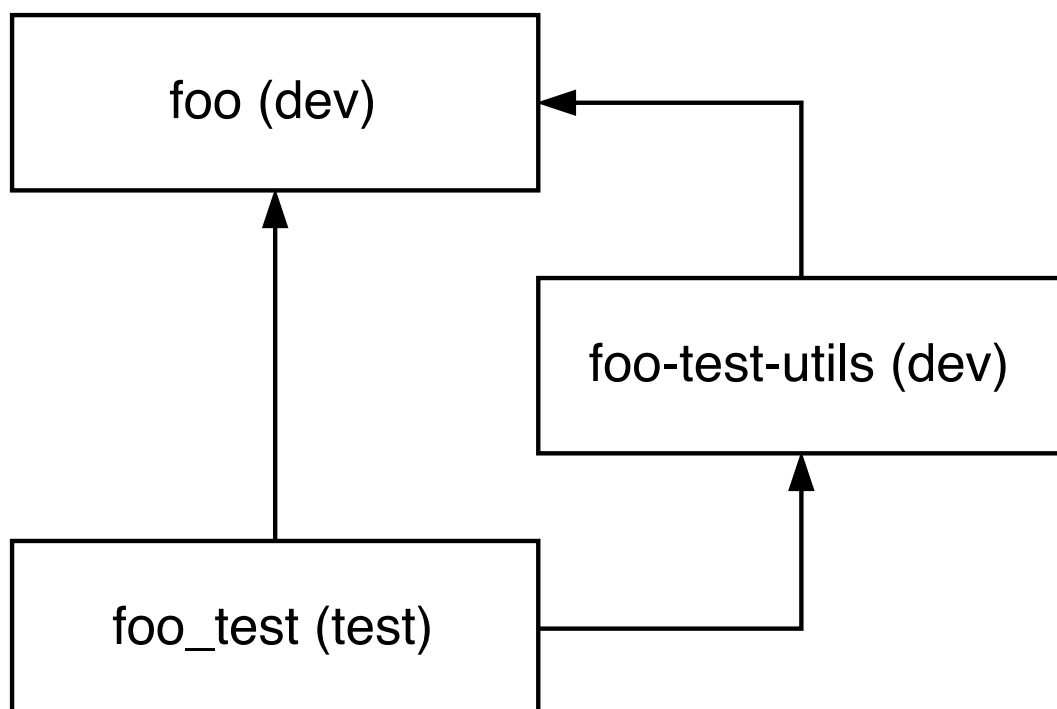
The way out of trouble is to define a separate integration test crate in the `foo` package and move the tests there. This approach allows you to test only the public interface of the `foo` library.

⊕

```
#[test]
fn test_foo_frobnication() {
    let foo = foo_test_utils::make_test_foo();
    assert_eq!(foo::frobnicate(&foo), 2);
}
```

The test above compiles fine because cargo links the test and `foo_test_utils` with the *dev* version of `foo`.

⊕



Quasi-circular dependencies are confusing. They also increase the incremental compilation time considerably. My advice is to avoid them when possible.

Conclusion

In this article, we looked at Rust's code organization tools. The key takeaways:

- Understand the difference between modules, crates, and packages.

- Rust's module system is convenient, but packing many modules into a single crate degrades the build time.

- Factoring the code into many cohesive packages is the most scalable approach.

- All implicit state is nasty.

Further reading

- Discuss this article on [r/rust](#).

- Alexey Kladov wrote a fantastic blog post series on the same topic, [One Hundred Thousand Lines of Rust](#).

Similar articles


- [Designing error types in Rust](#)

- [When Rust hurts](#)

- [Tutorial: stable-structures](#)

- [Scaling Rust builds with Bazel](#)

←Square joy: trapped rainwater
A swarm of replicated state machines→

©Roman Kashitsyn 
Source Code