Exemplary by Design: Building and Maintaining Rust at Scale

Intro

Hello! In the past, a handful of individuals have referred to some of the Rust code I've written as "exemplary" or something similar, stating that they show it to others to learn how Rust code should look.

In the coming minutes, I'll share some of the principles, practices, and patterns that I believe have led to this perception. Note that is descriptive, not prescriptive; idioms are what people make of them, and I don't expect everyone to agree with or adopt all of these ideas. A number of these I have violated myself, though I certainly regret doing so on a number of occasions.

■ Why care?

■ Poorly written code is hard to read, maintain, and extend. By following best practices, standard naming conventions, implementing proper testing, etc., ■ the number of bugs and ■ security vulnerabilities is reduced, particularly for code written in the future. Quality code also has ■ the potential for better performance as you can leverage zero-cost abstractions and the compiler is able to take advantage of known patterns to elide unnecessary checks, among other things.

Even if performance is not important to your use case, I sure hope that avoiding bugs and security vulnerabilities is. Fixing bugs is difficult and time-consuming, not to mention the potential for lost user trust, reputation damage, and legal liability if a security vulnerability is exploited. ■ All of this costs money, whether in developer time, lost customers, or legal fees.

Finally, writing quality code reduces cognitive load, making it easier to focus on the problem at hand rather than the quirks of the code. You will be able to have developers (both new and old) focus on the problem rather than struggle to understand the code itself.

Project Structure

■ module organization

To start off with project structure, I want to note that there are two ways to organize modules. Assuming you use the defaults (which you *absolutely* should), your entry point to a crate is `lib.rs` or `main.rs`, depending on whether you're writing a library or a binary, respectively. From there, each submodule can either be a file or a directory with a `mod.rs` file inside of it.

If the module you're adding, like `alpha`, has no submodules of its own, it should *always* be a standalone file, not nested. That's not a technical restriction, but frankly I've never seen it, nor did I even consider it as a possibility until writing this.

If it *does* have submodules, like `beta` which has submodule `gamma`, then you can choose between having a `beta.rs` file alongside a `beta/` directory *or* having a `beta/` directory with a `mod.rs` file inside it. Aside from placement and name, the `mod.rs` file here is otherwise identical to `beta.rs`. Depending in your editor, using `mod.rs` may not be quite as clear, but I'll cover a workaround for that later on.

Ultimately there is zero difference in functionality between these two approaches, and either is perfectly valid. My key suggestion is to pick one and stick with it for consistency.

■ system APIs

If you are writing code that abstracts over system APIs and want a common API (either internally or externally), it is generally cleaner to have separate files for each platform to avoid cluttering the code with `cfg` attributes.

Personally, I tend to use `cfg_attr` on a module declaration to select the appropriate file for the platform while always calling the module the same thing. I can then re-export everything, as each OS should have the same API exposed. While not explicitly declared, if the attributes all fall through then the module will load `imp.rs`, as the path is not specified.

Not everyone likes this approach due to its use of the `path` attribute, ■ so there is

an alternative that is currently available on nightly. That's the `cfg_select!` macro, which works similarly to a match statement. You specify the config the same way you would declare a match arm, and the relevant block is selected by the compiler. Once this macro is stabilized, I expect that to become the preferred way of handling this sort of situation.

■ workspaces

For larger projects, it is a near certainty that you will have multiple crates. When that happens, you should consider using a workspace to manage them. A workspace is a set of packages that share the same `Cargo.lock` and output directory, making it easier to ensure dependencies and lints are kept in sync, as well as simplifying commands like `cargo check` that can be configured to operate on the entire workspace at once.

Workspaces have other benefits as well, such as existing within a single repository, ensuring that code between sibling crates is always compatible, even if a rollback is necessary. Refactoring is also easier, as rust-analyzer is able to change code across multiple crates in a single operation.

■ crate graphs

Aside from cleaner code organization, workspaces also enable faster compilation. Adding some more crates to the previous example to be more realistic, we can visualize the dependency graph. Both `cli` and `web` depend on `models`, `utils`, and `core`, with `models` and `utils` also depending on `core`.

But when looking at compilation, we actually want to compile `core` first, so we should reverse ■ the arrows to visualize compilation order. You'll note that the arrows from `core` to `cli` and `web` skip over an intermediate crate, so we can ■ get rid of them as there will always be something else to compile in between.

Now we can see how splitting one crate into multiple enables faster compilation. `core` is compiled first, followed by `models` and `utils`. Note that these two can be compiled in parallel since they don't depend on each other. After both of those are done, `cli` and `web` can also be compiled in parallel. The more crates there are, the more opportunities for parallelism exist, which is particularly beneficial on

machines with many CPU cores.

This doesn't only affect full builds, but also incremental builds. If you change something in `core`, everything will need to be recompiled, but if all you change is `cli`, the compiler will recognize that `core`, `models`, and `utils` are unchanged and only recompile `cli`. Given that, in my experience, most work tends to happen at or near the top of the crate graph, this can lead to significant time savings.

■ rust-analyzer

On to editor configuration. I *highly* recommend the rust-analyzer extension, regardless of your choice of editor, as it provides countless features that make developing Rust code significantly more efficient.

For example, if offers ■ inline type hints, ■ the ability to look up all implementations and even references of a type, ■ viewing the expansion of macros, ■ test debugging, plus some features that I wanted, went to file a feature request for, all before searching through settings and ■ realizing that it already existed.

Additionally, rust-analyzer supports ■ semantic token color customization. If you want, mutable variables can be underlined to make them visually distinct. There are many categorizations like this one, some of which are incredibly useful. I personally have the `unsafe` keyword marked differently to highlight its usage, along with all unsafe operations, which can be a lifesaver when working with (and particularly when debugging) unsafe code.

I pulled up the complete list of what rust-analyzer emits, and it includes **61** tags and **20** modifiers. Needless to say you can do a crazy amount of customization on this front if you have the time and inclination to do so.

■ VS code specifics

I use VS Code, which from various surveys seems to be the most popular editor. For VS Code specifically, I suggest adding something like the snippet on screen to your global settings. This will change the title of tabs to ensure that files like `Cargo.toml` in a workspace or your various `mod.rs` files are identifiable at first

glance without needing to open them.

And if you work on the Rust standard library or compiler, feel free to reach out for additional project-specific configurations that I have found useful.

Design Patterns

■ clippy lints

If you're not familiar with it, clippy is a linter built into cargo that provides hundreds of lints to catch common mistakes. It can be run with `cargo clippy`, and I can't recommend it strongly enough. Even if you don't configure anything, the default set of lints is great.

As an example of the type of lints that clippy provides, there is one to catch the example on the screen. I have seen this written on two separate occasions in code reviews, wherein there is an `Arc` of a `Mutex` around a boolean flag. There is literally zero reason to do this. You're introducing reference counting, thread synchronization, locking, and the overhead of checking for a poisoned lock, every single time you want to read or write the flag. Instead, ■ you can use an `AtomicBool`, which handles all of this and is far more efficient. The `Arc<Mutex<bool>>` is eight bytes, while the `AtomicBool` is just one. The atomic operation also likely optimizes to nothing more than a single CPU instruction, while the version I've encountered has nontrivial overhead.

While not the most exciting example, it is a good illustration of the thoroughness of clippy's lints, even for something as small as a boolean flag. There are many more lints that catch potential performance pitfalls, as well as lints to catch common mistakes, stylistic issues, and even some security issues. If you haven't already, definitely give it a try.

■ iterators

In this admittedly contrived example, we're taking a slice of integers and computing the sum of the squares of the even numbers. In most programming languages, this is 100% how things would and *should* be written. In Rust, this will

work, but there is a better way.

■ By using iterators, we can express the same logic in a single expression. We turn the `if` statement into `filter`, the right hand side of the assignment into `map`, and the `for` loop itself into a method call. At the end, we simply call `sum` to consume the iterator and produce the final result, no temporary variables needed.

It's also worth noting that iterators are *lazy*, so all odd values do not exist as far as `map` is concerned. If we need to change the code, it's faster to see what's going on and to make the change without worrying about messing anything else up.

In this specific example, the iterator version is more concise, easier to read, and generates better assembly, as the `for` loop version has to do bounds checks that are not optimized away by the compiler.

■ `Result` vs. `Option`

When dealing with operations that can fail, there are a handful of options. One is to panic, but that is generally reserved for errors caused by the *programmer*, resulting an unexpected situation that shouldn't need to be handled in other parts of the code.

For *anticipated* errors, such as a network request failing, or a file not being found, the idiomatic way to handle these is with the `Result` type, which can represent either success or failure.

However, it's important to note the difference between an operation *failing* and an operation not producing a value. As in this example, if the database query completes successfully, but there is no row matching the query, that is not a failure; it is a successful operation that found nothing. Given this, the `Option` type should be used to indicate the potential absence of a value, while `Result` is wrapped around *that* to indicate the possibility of the query itself failing.

Having a `Result` of an `Option` is not uncommon to see, to the extent that there is a `transpose` method on both types to switch the order if needed.

■ trait usage

Here is an example of what many newcomers to Rust attempt to do, mimicking patterns from other languages. There is a struct `Dog` that contains an `Animal` struct, essentially inheriting its behavior. This leads to very tight coupling of the two types, not to mention the difficulty of writing a method that accepts any kind of `Animal`.

After all, the only way to know what type of animal you have is to look at the `Dog` struct, which doesn't exist as far as `Animal` is concerned. This inevitably leads to significant confusion, boilerplate, frustration, and unidiomatic code that is difficult to maintain.

■ Instead, we get rid of the `Animal` struct entirely. We keep `Dog` around, as that's something we want to represent. For `Animal`, it is now a trait that describes behavior, rather than implementing it concretely. So long as the `Animal` trait is in scope, we can call `make_noise` on any type that implements it.

In this manner, traits serve a similar purpose as interfaces in other languages, but with more flexibility and explicitness. If a function wants to accept an `Animal`, it can do so either by monomorphization, which is when the compiler generates a specific version of the function for each type, or by accepting a `dyn Animal`, where there is a small amount of runtime overhead from dynamic dispatch. Either way, the function can accept any type that implements the `Animal` trait, rather than accepting an `Animal` struct and managing dispatch manually.

Traits are used in essentially every program, so understanding them is critical to writing idiomatic Rust code.

■ extension traits

Extension traits are a great way to add a method to a pre-existing type that you are unable to add an inherent implementation for, generally because the type is defined in another crate. In this example lifted from the `time` crate, we add a `seconds` method to `i64` so that integer literals have a `seconds` method that returns a `Duration`. Extension traits can have any number of methods, though they are commonly used for only one.

■ To use the new method, the trait has to be in scope, which is trivially done by

`use`ing the trait as `_`, as shown in the example. This is so that the trait does not pollute the namespace with its name, but is still available for method resolution.

■ newtype

In this *purely hypothetical* example, we have a contractor using miles and the government using kilometers. Somewhere else in the code, these two values get added together, resulting in the loss of hundreds of millions of dollars.

How can we avoid this? Sure, we could be careful, but humans are stupid. They're using miles, after all. Instead, ■ we can use the **newtype pattern** to create small tuple structs that wrap the underlying value. ■ By defining the allowed operations, we can ensure that the compiler will prevent us from mixing up units. As a bonus, it will perform conversions when necessary, ensuring that our *purely hypothetical* satellite will not turn into fireworks for the native Martians.

And just as with with extension traits, you can add or even override methods on the newtype by using an inherent `impl`. If a method of the same name exists on the inner type, the newtype's method will take precedence.

■ adding additional field to existing struct

Next up is a relatively niche but useful pattern I have found for deserializing data that shares much in common but has an additional field or two. To handle this, we can create a wrapper struct that includes the original type — here the generic `T` — along with the additional fields we want. By using the `#[serde(flatten)]` attribute, `serde` effectively treats the fields as if they are part of a single struct, not nested.

To be a bit more ergonomic, ■ we can also implement `Deref` and `DerefMut` to allow easy access to the inner type's fields and methods. All we have to do now is ■ use the wrapper struct where it's needed.

When done correctly, this pattern can result in trivial boilerplate in a centralized location while only requiring minimal changes to existing code to support the additional field, here just replacing the type of the parameter. Without this pattern, you would likely be stuck with duplicating the `struct` or writing a custom deserializer, neither of which is particularly appealing.

■ `enum` within `struct`

Here we have a pattern that probably isn't used as much as it should be. It's very common to see `enum`s used directly in public APIs without much thought about the implications. `enums`, by design, do not have private variants or fields. This means that if you expose an `enum` in your public API, you are committing to having that representation for the lifetime of the API.

By wrapping an `enum` in a `struct`, ■ you gain complete control over your public API. ■ None of the implementation details are exposed unless you explicitly choose to do so. As a result, ■ you can remove a no-longer-needed variant without breaking downstream code, or ■ even refactor the implementation details entirely. If you were considering having a number of types instead of an `enum` to begin with, this pattern allows you to expose only a single type, ■ simplifying the mental model for downstream users.

In the standard library, this is used for the `io::Error` type, which is internally an `enum`. The standard library then exposes methods on the encompassing `struct` to allow users to determine the variant, though there is some additional categorization that happens to make things more logical on the user's end. This is a great example of where the pattern is used effectively and where people likely don't even realize that it is being used.

I have personally found this pattern to be incredibly useful, and not just for error handling, though that is certainly a common use case. If I had followed this pattern in the past, I would have been able to simplify the public API of a crate that I maintain, eliminating factors that extremely few if any users ever care about. Instead of exposing two `enum`s with five variants each *and* having there be differences in the API depending on which feature flags are enabled, all of this could have been abstracted away behind a single `struct` with no loss of functionality for users. For the rare user that *did* need the full power of the current API, I could easily add methods to provide that power while still simplifying the common case. Needless to say, next time I make a breaking change for this crate, moving to have an `enum` within a `struct` for this situation is a near-certainty to happen.

■ `#[non_exhaustive]`

There is a `#[non_exhaustive]` attribute that can be applied to `struct` s, `enum` s, and `enum` variants. ■ The attribute prevents pattern matching in other crates unless you include a catch-all arm for `enum` s or use `..` when destructuring `struct` s or `enum` variants.

Use it any time you might want the ability to add a field or variant in the future without it being a breaking change. That's pretty much all there is to say about this attribute; it really is that straightforward.

■ builder methods

Another common pattern is the builder pattern, which lets you construct large objects step-by-step. There are a few variations regarding the specifics, so I'll cover one of them here. Suppose you had a struct `Foo` with two fields, `alpha` and `beta` . We then define a separate struct, `FooBuilder` , which has the same fields, but wrapped in `Option` to indicate that they may or may not be set.

The builder struct has methods for setting each field, ■ which take `self` by value and return `Self` to allow for method chaining. Finally, there is a `build` method that consumes the builder and constructs the final `Foo` object. In this example, the `build` method would generally panic as any unset field would be the fault of the programmer, but it could also return a `Result` indicating success or failure.

If any of the fields in `Foo` are optional, the corresponding builder methods can be omitted, as is the case with `beta` here. The `build` method can provide default values for those fields if they were not set. It can also perform validation to ensure that the constructed object meets certain criteria, or any other logic that needs to be applied before the object is used.

■ builder methods: replacement?

Among the various design patterns being discussed, the builder pattern stands out as one that is likely to die off relatively soon, at least for most cases. This is because there is an upcoming feature — default field values — that will allow you to specify default values for a given field directly in the struct definition. What's on

screen is the actual syntax that is proposed and implemented on nightly. It's already been through the RFC process and its implementation has been making good progress.

With this feature, you can omit a field when constructing an instance of the struct if and only if it has a default value in the definition. To indicate that this is the case, you use the `..` syntax, which is similar to the existing syntax for struct update. This means that if you have a struct with many fields, you can simply specify the ones you care about and let everything else fall back to its default value.

■ idioms change

To wrap up design patterns, I want to highlight the fact that idioms change!

In the past, people would use the `maplit` crate to create a `HashMap` with a macro. ■ Once `FromIterator` was added to the prelude, the general suggestion was to call `HashMap::from_iter` with an array of key-value tuples instead. More recently, ■ there was an unstable `hash_map` macro added to the standard library, bringing us back to what the `maplit` crate provided.

Even farther back to when I started using Rust, ■ there used to be a `try` macro! ■ This is now deprecated in favor of the question mark operator that replaced it.

The takeaway here is that while your code will remain clear as the language evolves, there may be a more idiomatic way in the future to express the same intent. This isn't a bad thing, though if you're refactoring code anyway it is definitely something to keep in mind.

■ Error Handling

There are a number of ways to handle errors in Rust, but generally speaking, `anyhow` or `eyre` are good choices for binaries where you don't care about the specific error type, just that *some* error exists. This isn't recommended for libraries since it forces your users to work with an opaque error type containing no useful information beyond the error message and a backtrace. `anyhow` and `eyre` are very similar; `eyre` provides a bit more functionality if you need it, but you're not going to go wrong with either.

■ For libraries, `thiserror` is notable because it doesn't alter your public API in any way. It provides a derive macro to implement the `Error` trait for your custom error types, while at the same time generating code for `Display` and `From` implementations. The main (and realistically only) downside is that this will increase your compile times, but you're not likely to notice the difference unless your dependency tree doesn't already include `syn` and `quote`.

■ Finally, implementing `From` for your error types allows you to use the `?` operator to convert between them. This is especially useful because you're likely to call functions that return differing error types, and being able to convert into a common type with only a single character is a huge win for ergonomics. If you use `thiserror`, you can opt into this with a trivial attribute, and you should certainly take advantage of it.

If you combine these approaches with other design patterns such as using `Result` when appropriate, and wrapping `enums` into a `struct`, you'll find that error handling in Rust is not as painful as it may initially seem, and can be quite ergonomic with a little effort.

■ Testing

Everyone's favorite thing to do — write tests. While it may not be the most exciting topic, it's important to ensure that your code works as expected!

In Rust, your unit tests typically go into the same file as the code they're testing, though it's not uncommon to have them in a sibling file. Integration tests, on the other hand, *must* be in a separate file and are located in the `tests` directory by default. One key difference is that integration tests can only access the public API of your crate, while unit tests can access private items as well. For this reason, try to write integration tests to cover everything you can, as they shouldn't depend on implementation details and can survive even major refactorings. Other than this, unit and integration tests are largely the same as in other languages, so I won't dwell on them here.

■ Property testing, though, is a great way to test edge cases automatically. You can write tests that ensure that certain properties are always held true, regardless

of input. As an example, you can accept a string and ensure that base64 encoding it followed by decoding returns the original string. The `quickcheck` crate makes this very easy to do, and it's quite configurable, even if you have some restrictions on your input. The best part is that if it finds a counter-example, it will try to reduce the input as much as possible to eliminate unnecessary complexity, which can help you identify the root cause of the failure.

■ If you have any `unsafe` code, property testing may not be enough. You should also run your code under `miri`, which can catch undefined behavior such as out of bounds reads, dereferencing misaligned pointers, and more. If `miri` tells you that your code is exhibiting undefined behavior, that's not a "maybe" — it's definitely wrong, and you need to fix it.

■ Something else you might want to track is code coverage, though it shouldn't be your only goal. The `cargo-llvm-cov` tool makes it easy to generate coverage reports, and it can even be run in CI to ensure that your coverage doesn't decrease over time. Just remember, 100% coverage doesn't mean your code is perfect — it just means that every line, function, or whatever you're tracking has been executed at least once. You still need to ensure that your tests are meaningful and cover edge cases.

■ Maintenance

Great! Now you can write code that is idiomatic, efficient, and correct. But how do you keep it that way? Maintenance is a big part of software development, and Rust is no exception.

First up is simply keeping your dependencies up to date. There are numerous bots that can open PRs for this, and you can always use `cargo update` and `cargo upgrade` to do it manually. The difference here is `update` is semver-compatible changes, while `upgrade` includes breaking changes that may require action on your end.

■ For documentation, `rustdoc` is your friend! Rust's tooling is unmatched here — take advantage of it! You can include examples in doc comments, and `rustdoc` will even run library examples as tests to make sure they work as expected.

■ Perhaps not as much for corporate code that only you are using, but changelogs are a critical part of communicating to your users what has changed between versions. Anything is better than nothing here, though I've never seen anyone complain for having *too much info* in a changelog.

■ Last for maintenance is CI. Basic checks like `clippy` and `rustfmt` are great to see, with tests being key for acceptance. If your tests take a while to run, consider splitting them into separate jobs or only running a critical subset, reserving the full test suite for releases.

■ Resources

Without going into detail on these due to time constraints, here are some additional resources that can help you maintain a large Rust codebase and write idiomatic code.

For license enforcement and security advisories, check out `cargo-deny` which can be configured to your needs and can be run in CI.

■ For checking semver compatibility, the cargo book has a chapter detailing many types of changes, whether they're major or minor changes, and the `cargo-semver-checks` tool can help automate checking weird edge cases that are easy to miss.

■ To help with performance, `criterion` is my go-to crate for benchmarking, while `flamegraph` is amazing for visualizing where time is being spent in your code.

■ The Rust API guidelines are a solid resource for naming conventions, interoperability, and more, and are closely followed by most major crates.

■ Finally, the Rustonomicon is a deep dive into unsafe Rust, and is a must-read if you're writing unsafe code or interfacing with C code.

■ Outro

And that's what I have for today! Hopefully you found something useful to take away, and hopefully you'll be able to apply some of these ideas to your own Rust projects.

I know it was a lot to take in, but if you remember just a few things and can apply them, that's a win in my book. The slides are online now, so you can refer back to them at your own convenience.

My username on most platforms, including GitHub and Mastodon, is jhpratt. Feel free to ask any questions you may have, whether it be on Mastodon, via email, or elsewhere. Thank you!