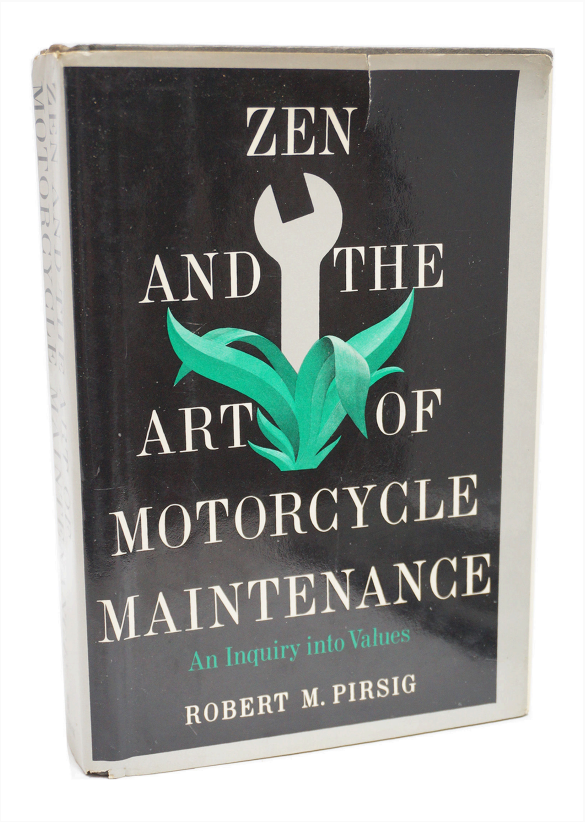
Zen and the art of library maintenance

Lukáš Hozda

2025-04-10

Braiins Systems s.r.o

Zen and the Art of Library Maintenance



Zen and the Art of Motorcycle Maintenance by Robert Pirsig - Good book about quality

Writing libraries and testing

- General principles
- Rust design
- Testing
- Maintenance

General principles

General principles

- There are some general points to consider when writing a library

General principles

- There are some general points to consider when writing a library
- Consistency

General principles

- There are some general points to consider when writing a library
- Consistency
- API stability

General principles

- There are some general points to consider when writing a library
- Consistency
- API stability
  - The best library in the world is useless if I cannot rely on it not breaking completely every release

General principles

- There are some general points to consider when writing a library
- Consistency
- API stability
  - The best library in the world is useless if I cannot rely on it not breaking completely every release
- Encapsulation

General principles

- There are some general points to consider when writing a library
- Consistency
- API stability
  - The best library in the world is useless if I cannot rely on it not breaking completely every release
- Encapsulation
  - Data Hiding

General principles

- There are some general points to consider when writing a library
- Consistency
- API stability
  - The best library in the world is useless if I cannot rely on it not breaking completely every release
- Encapsulation
  - Data Hiding
  - Modularity

General principles

- There are some general points to consider when writing a library
- Consistency
- API stability
  - The best library in the world is useless if I cannot rely on it not breaking completely every release
- Encapsulation
  - Data Hiding
  - Modularity
- Ease of use - use the right level of abstraction, provide sensible defaults

General principles

- There are some general points to consider when writing a library
- Consistency
- API stability
  - The best library in the world is useless if I cannot rely on it not breaking completely every release
- Encapsulation
  - Data Hiding
  - Modularity
- Ease of use - use the right level of abstraction, provide sensible defaults
- Technical aspects

General principles

- There are some general points to consider when writing a library
- Consistency
- API stability
  - The best library in the world is useless if I cannot rely on it not breaking completely every release
- Encapsulation
  - Data Hiding
  - Modularity
- Ease of use - use the right level of abstraction, provide sensible defaults
- Technical aspects
  - Performance

General principles

- There are some general points to consider when writing a library
- Consistency
- API stability
  - The best library in the world is useless if I cannot rely on it not breaking completely every release
- Encapsulation
  - Data Hiding
  - Modularity
- Ease of use - use the right level of abstraction, provide sensible defaults
- Technical aspects
  - Performance
  - Achievement of its goals

General principles

- There are some general points to consider when writing a library
- Consistency
- API stability
  - The best library in the world is useless if I cannot rely on it not breaking completely every release
- Encapsulation
  - Data Hiding
  - Modularity
- Ease of use - use the right level of abstraction, provide sensible defaults
- Technical aspects
  - Performance
  - Achievement of its goals
- Licensing

General principles

- There are some general points to consider when writing a library
- Consistency
- API stability
  - The best library in the world is useless if I cannot rely on it not breaking completely every release
- Encapsulation
  - Data Hiding
  - Modularity
- Ease of use - use the right level of abstraction, provide sensible defaults
- Technical aspects
  - Performance
  - Achievement of its goals
- Licensing
  - No serious developer will touch your “OSS” library if you forget a license

Consistency

- Similar things should be done the same way as much as possible

Consistency

Consistency

- Similar things should be done the same way as much as possible
- Suppose we are writing a library for rational arithmetic (with fractions)

Consistency

Consistency

- Similar things should be done the same way as much as possible
- Suppose we are writing a library for rational arithmetic (with fractions)
  - We have a Fraction type

Consistency

Consistency

- Similar things should be done the same way as much as possible
- Suppose we are writing a library for rational arithmetic (with fractions)
  - We have a Fraction type
  - Methods:

Consistency

Consistency

- Similar things should be done the same way as much as possible
- Suppose we are writing a library for rational arithmetic (with fractions)
  - We have a Fraction type
  - Methods:
    - `add(other_frac)`
    - `subtract(other_frac)`
    - `multiply(other_frac)`
    - `divide(other_frac)`

Consistency

Consistency

- Similar things should be done the same way as much as possible
- Suppose we are writing a library for rational arithmetic (with fractions)
  - We have a Fraction type
  - Methods:
    - `add(other_frac)`
    - `subtract(other_frac)`
    - `multiply(other_frac)`
    - `divide(other_frac)`
  - Bad: divide() and multiply() mutate the original, add() and subtract() return new Fraction

Consistency

Consistency

- Similar things should be done the same way as much as possible
- Suppose we are writing a library for rational arithmetic (with fractions)
  - We have a Fraction type
  - Methods:
    - `add(other_frac)`
    - `subtract(other_frac)`
    - `multiply(other_frac)`
    - `divide(other_frac)`
  - Bad: divide() and multiply() mutate the original, add() and subtract() return new Fraction
  - Good: they all return a new Fraction and are **consistent** with primitive numbers' operators

Consistency

Consistency

- Similar things should be done the same way as much as possible
- Suppose we are writing a library for rational arithmetic (with fractions)
  - We have a Fraction type
  - Methods:
    - `add(other_frac)`
    - `subtract(other_frac)`
    - `multiply(other_frac)`
    - `divide(other_frac)`
  - Bad: divide() and multiply() mutate the original, add() and subtract() return new Fraction
  - Good: they all return a new Fraction and are **consistent** with primitive numbers' operators
  - You laugh, but I have seen this exact thing

Consistency

Consistency

- Similar things should be done the same way as much as possible
- Suppose we are writing a library for rational arithmetic (with fractions)
  - We have a Fraction type
  - Methods:
    - `add(other_frac)`
    - `subtract(other_frac)`
    - `multiply(other_frac)`
    - `divide(other_frac)`
  - Bad: divide() and multiply() mutate the original, add() and subtract() return new Fraction
  - Good: they all return a new Fraction and are **consistent** with primitive numbers' operators
  - You laugh, but I have seen this exact thing
- Also, be consistent across things like generics

Consistency

- Similar things should be done the same way as much as possible
- Suppose we are writing a library for rational arithmetic (with fractions)
  - We have a Fraction type
  - Methods:
    - `add(other_frac)`
    - `subtract(other_frac)`
    - `multiply(other_frac)`
    - `divide(other_frac)`
  - Bad: divide() and multiply() mutate the original, add() and subtract() return new Fraction
  - Good: they all return a new Fraction and are **consistent** with primitive numbers' operators
  - You laugh, but I have seen this exact thing
- Also, be consistent across things like generics
  - If my type can work with "anything that can be turned into a Path," the constructor should not take only a hardcoded string type

API stability

- API (Application Programming Interface)

API stability

API stability

- API (Application Programming Interface)
  - You know this

API stability

API stability

- API (Application Programming Interface)
  - You know this
  - Types, methods, functions, constants, global variables (DONT), publicly exposed by a library

API stability

API stability

- API (Application Programming Interface)
  - You know this
  - Types, methods, functions, constants, global variables (DONT), publicly exposed by a library
- Naturally, for new libraries and SW in general, some breaking changes are unavoidable

API stability

API stability

- API (Application Programming Interface)
  - You know this
  - Types, methods, functions, constants, global variables (DONT), publicly exposed by a library
- Naturally, for new libraries and SW in general, some breaking changes are unavoidable
  - It would be pretty bad to be forever shackled to the poor decisions of your youth

API stability

API stability

- API (Application Programming Interface)
  - You know this
  - Types, methods, functions, constants, global variables (DONT), publicly exposed by a library
- Naturally, for new libraries and SW in general, some breaking changes are unavoidable
  - It would be pretty bad to be forever shackled to the poor decisions of your youth
- What we need is a good API design and versioning

API stability

API stability

- API (Application Programming Interface)
  - You know this
  - Types, methods, functions, constants, global variables (DONT), publicly exposed by a library
- Naturally, for new libraries and SW in general, some breaking changes are unavoidable
  - It would be pretty bad to be forever shackled to the poor decisions of your youth
- What we need is a good API design and versioning
- Good API design

API stability

API stability

- API (Application Programming Interface)
  - You know this
  - Types, methods, functions, constants, global variables (DONT), publicly exposed by a library
- Naturally, for new libraries and SW in general, some breaking changes are unavoidable
  - It would be pretty bad to be forever shackled to the poor decisions of your youth
- What we need is a good API design and versioning
- Good API design
  - Maximum flexibility, minimal surface (if you expose the entirety of your library, every change is a breaking API change)

API stability

API stability

- API (Application Programming Interface)
  - You know this
  - Types, methods, functions, constants, global variables (DONT), publicly exposed by a library
- Naturally, for new libraries and SW in general, some breaking changes are unavoidable
  - It would be pretty bad to be forever shackled to the poor decisions of your youth
- What we need is a good API design and versioning
- Good API design
  - Maximum flexibility, minimal surface (if you expose the entirety of your library, every change is a breaking API change)
    - Allows me to do extensive rewrites of the underlying logic while maintaining the same API

API stability

API stability

- API (Application Programming Interface)
  - You know this
  - Types, methods, functions, constants, global variables (DONT), publicly exposed by a library
- Naturally, for new libraries and SW in general, some breaking changes are unavoidable
  - It would be pretty bad to be forever shackled to the poor decisions of your youth
- What we need is a good API design and versioning
- Good API design
  - Maximum flexibility, minimal surface (if you expose the entirety of your library, every change is a breaking API change)
    - Allows me to do extensive rewrites of the underlying logic while maintaining the same API
- Versioning - SemVer (Semantic Versioning)

API stability

- API (Application Programming Interface)
  - You know this
  - Types, methods, functions, constants, global variables (DONT), publicly exposed by a library
- Naturally, for new libraries and SW in general, some breaking changes are unavoidable
  - It would be pretty bad to be forever shackled to the poor decisions of your youth
- What we need is a good API design and versioning
- Good API design
  - Maximum flexibility, minimal surface (if you expose the entirety of your library, every change is a breaking API change)
    - Allows me to do extensive rewrites of the underlying logic while maintaining the same API
- Versioning - SemVer (Semantic Versioning)
  - “I can depend on the latest version that will not break my stuff”

API stability

API stability

- API (Application Programming Interface)
  - You know this
  - Types, methods, functions, constants, global variables (DONT), publicly exposed by a library
- Naturally, for new libraries and SW in general, some breaking changes are unavoidable
  - It would be pretty bad to be forever shackled to the poor decisions of your youth
- What we need is a good API design and versioning
- Good API design
  - Maximum flexibility, minimal surface (if you expose the entirety of your library, every change is a breaking API change)
    - Allows me to do extensive rewrites of the underlying logic while maintaining the same API
- Versioning - SemVer (Semantic Versioning)
  - “I can depend on the latest version that will not break my stuff”
  - Built into Cargo with a quirk

API stability

- API (Application Programming Interface)
  - You know this
  - Types, methods, functions, constants, global variables (DONT), publicly exposed by a library
- Naturally, for new libraries and SW in general, some breaking changes are unavoidable
  - It would be pretty bad to be forever shackled to the poor decisions of your youth
- What we need is a good API design and versioning
- Good API design
  - Maximum flexibility, minimal surface (if you expose the entirety of your library, every change is a breaking API change)
    - Allows me to do extensive rewrites of the underlying logic while maintaining the same API
- Versioning - SemVer (Semantic Versioning)
  - “I can depend on the latest version that will not break my stuff”
  - Built into Cargo with a quirk => “1.0.0” considered to be “^1.0.0”

Encapsulation

- Data hiding

Encapsulation

- Data hiding
  - Your library should only expose what needs to be exposed

Encapsulation

- Data hiding
  - Your library should only expose what needs to be exposed
  - Consider private visibility the default, and mark stuff as public when needed

Encapsulation

- Data hiding
  - Your library should only expose what needs to be exposed
  - Consider private visibility the default, and mark stuff as public when needed
    - In the context of Rust, only mark the actual API as `pub`, use `pub(crate)` if something needs to be accessible everywhere in your lib

Encapsulation

- Data hiding
  - Your library should only expose what needs to be exposed
  - Consider private visibility the default, and mark stuff as public when needed
    - In the context of Rust, only mark the actual API as `pub`, use `pub(crate)` if something needs to be accessible everywhere in your lib
  - Hiding the internal state of your objects is good for both rewrites and security

Encapsulation

- Data hiding
  - Your library should only expose what needs to be exposed
  - Consider private visibility the default, and mark stuff as public when needed
    - In the context of Rust, only mark the actual API as `pub`, use `pub(crate)` if something needs to be accessible everywhere in your lib
  - Hiding the internal state of your objects is good for both rewrites and security (You do not need to account for the possibility that someone writes invalid values into your `struct`'s fields)

Encapsulation

- Data hiding
  - Your library should only expose what needs to be exposed
  - Consider private visibility the default, and mark stuff as public when needed
    - In the context of Rust, only mark the actual API as `pub`, use `pub(crate)` if something needs to be accessible everywhere in your lib
  - Hiding the internal state of your objects is good for both rewrites and security (You do not need to account for the possibility that someone writes invalid values into your `struct`'s fields)
- Modularity

Encapsulation

- Data hiding
  - Your library should only expose what needs to be exposed
  - Consider private visibility the default, and mark stuff as public when needed
    - In the context of Rust, only mark the actual API as `pub`, use `pub(crate)` if something needs to be accessible everywhere in your lib
  - Hiding the internal state of your objects is good for both rewrites and security (You do not need to account for the possibility that someone writes invalid values into your `struct`'s fields)
- Modularity
  - Types understood through their interfaces as distinct units

Ease of use

- Quick to get started with

Ease of use

- Quick to get started with
- Easy to use in the correct way

Ease of use

- Quick to get started with
- Easy to use in the correct way
- Performant, achieve stated goal, flexible enough

Ease of use

- Quick to get started with
- Easy to use in the correct way
- Performant, achieve stated goal, flexible enough
- Well tested and documented

Ease of use

- Quick to get started with
- Easy to use in the correct way
- Performant, achieve stated goal, flexible enough
- Well tested and documented
  - This actually helps with design, since it forces you to look at your library from a user perspective

Ease of use

- Quick to get started with
- Easy to use in the correct way
- Performant, achieve stated goal, flexible enough
- Well tested and documented
  - This actually helps with design, since it forces you to look at your library from a user perspective
- Small API

Ease of use

- Quick to get started with
- Easy to use in the correct way
- Performant, achieve stated goal, flexible enough
- Well tested and documented
  - This actually helps with design, since it forces you to look at your library from a user perspective
- Small API
  - Less to learn, maintain, less opportunity to make breaking changes

Rust library design

Rust library design

- Write idiomatic code (code that looks nice)

Rust library design

- Write idiomatic code (code that looks nice)
  - Typically also happens to be easy to write (in terms of the language not putting obstacles in your way)

Rust library design

- Write idiomatic code (code that looks nice)
  - Typically also happens to be easy to write (in terms of the language not putting obstacles in your way)
- Rust railroads you into idiomatic code pretty hard

Rust library design

- Write idiomatic code (code that looks nice)
  - Typically also happens to be easy to write (in terms of the language not putting obstacles in your way)
- Rust railroads you into idiomatic code pretty hard
  - Everyone fought the borrow-checker and various APIs at one time

Going back to the “well tested” point

- In Rust, we have doc-tests

Going back to the “well tested” point

Going back to the “well tested” point

- In Rust, we have doc-tests
  - Documentation examples become tests:

Going back to the “well tested” point

Going back to the “well tested” point

- In Rust, we have doc-tests
  - Documentation examples become tests:

```
/// # Examples
/// ```
/// let mut vec = Vec::with_capacity(10);
/// assert_eq!(vec.len(), 0);
/// assert!(vec.capacity() >= 10);
/// for i in 0..10 {
///     vec.push(i);
/// }
/// assert_eq!(vec.len(), 10);
/// assert!(vec.capacity() >= 10);
/// ```
```

Going back to the “well tested” point

- Rust highlighting by default, use “#” to hide lines, use `no_run` (after opening backticks) to prevent

Going back to the “well tested” point

- In Rust, we have doc-tests
  - Documentation examples become tests:

```
/// # Examples
/// ```
/// let mut vec = Vec::with_capacity(10);
/// assert_eq!(vec.len(), 0);
/// assert!(vec.capacity() >= 10);
/// for i in 0..10 {
///     vec.push(i);
/// }
/// assert_eq!(vec.len(), 10);
/// assert!(vec.capacity() >= 10);
/// ```
```

Going back to the “well tested” point

- Rust highlighting by default, use “#” to hide lines, use `no_run` (after opening backticks) to prevent snippet from being a doctest

Going back to the “well tested” point

- In Rust, we have doc-tests
  - Documentation examples become tests:

```
/// # Examples
/// ```
/// let mut vec = Vec::with_capacity(10);
/// assert_eq!(vec.len(), 0);
/// assert!(vec.capacity() >= 10);
/// for i in 0..10 {
///     vec.push(i);
/// }
/// assert_eq!(vec.len(), 10);
/// assert!(vec.capacity() >= 10);
/// ```
```

Going back to the “well tested” point

- Rust highlighting by default, use “#” to hide lines, use `no_run` (after opening backticks) to prevent snippet from being a doctest
- `cargo test` will run doctests

Going back to the “well tested” point

- In Rust, we have doc-tests
  - Documentation examples become tests:

```
/// # Examples
/// ```
/// let mut vec = Vec::with_capacity(10);
/// assert_eq!(vec.len(), 0);
/// assert!(vec.capacity() >= 10);
/// for i in 0..10 {
///     vec.push(i);
/// }
/// assert_eq!(vec.len(), 10);
/// assert!(vec.capacity() >= 10);
/// ```
```

Going back to the “well tested” point

- Rust highlighting by default, use “#” to hide lines, use `no_run` (after opening backticks) to prevent snippet from being a doctest
- `cargo test` will run doctests
- Document public items in any case

Going back to the “well tested” point

- In Rust, we have doc-tests
  - Documentation examples become tests:

```
/// # Examples
/// ```
/// let mut vec = Vec::with_capacity(10);
/// assert_eq!(vec.len(), 0);
/// assert!(vec.capacity() >= 10);
/// for i in 0..10 {
///     vec.push(i);
/// }
/// assert_eq!(vec.len(), 10);
/// assert!(vec.capacity() >= 10);
/// ```
```

Going back to the “well tested” point

- Rust highlighting by default, use “#” to hide lines, use `no_run` (after opening backticks) to prevent snippet from being a doctest
- `cargo test` will run doctests
- Document public items in any case
  - You can use `#![deny(missing_docs)]` in your crate root

Standard layout

```
$ tree
.
|-- Cargo.toml
|-- benches                 - benchmarks
|-- examples                - independent examples (binaries)
|-- src                     ↓
|   |-- bin                 ↓
|   |   `-- something.rs    - other binaries (if any)
|   |-- lib.rs              - library entrypoint
|   `-- main.rs             - if your crate also has an executable (eg. CLI)
`-- tests                   - "integration" tests
```

API design

- Make illegal states unrepresentable (Haskell proverb)

API design

- Make illegal states unrepresentable (Haskell proverb)
  - Meaning - only correct usage, within reason, should compile

API design

- Make illegal states unrepresentable (Haskell proverb)
  - Meaning - only correct usage, within reason, should compile
- Avoid “stringly-typed” APIs (Pascal Hertleif quote)

```rust
use chrono::{DateTime, Utc, Weekday};

fn is_matching_day(datetime: DateTime<Utc>, day: &str) -> bool {
    let weekday = datetime.weekday();
    match day.to_lowercase().as_str() {
        "monday" => weekday == Weekday::Mon,
        "tuesday" => weekday == Weekday::Tue,
        "wednesday" => weekday == Weekday::Wed,
        "thursday" => weekday == Weekday::Thu,
        "friday" => weekday == Weekday::Fri,
        "saturday" => weekday == Weekday::Sat,
        "sunday" => weekday == Weekday::Sun,
        _ => unreachable!("there is only 7 days in a week, no?"),
    }
}
```

Oops

```
let is_tuesday = is_matching_day(
  some_date,
  "If you ask Rick Astley for a copy of the movie “UP”, he cannot give you it
as he can never give you up. But, by doing that, he is letting you down, and
thus, is creating something known as the Astley Paradox.",
);
```

- The correct thing to do is use a more concrete type (e.g. an enum for all the days in this case)

Oops

```
let is_tuesday = is_matching_day(
  some_date,
  "If you ask Rick Astley for a copy of the movie “UP”, he cannot give you it
as he can never give you up. But, by doing that, he is letting you down, and
thus, is creating something known as the Astley Paradox.",
);
```

- The correct thing to do is use a more concrete type (e.g. an enum for all the days in this case)
  - Enums are great for representing states in general

Oops

```
let is_tuesday = is_matching_day(
  some_date,
  "If you ask Rick Astley for a copy of the movie “UP”, he cannot give you it
as he can never give you up. But, by doing that, he is letting you down, and
thus, is creating something known as the Astley Paradox.",
);
```

- The correct thing to do is use a more concrete type (e.g. an enum for all the days in this case)
  - Enums are great for representing states in general
  - Unlike random strings, you can document enums

Builder pattern

- Extremely popular in Rust (because we have no default parameters and variadic functions)

Builder pattern

Builder pattern

- Extremely popular in Rust (because we have no default parameters and variadic functions)

```
fn main() {
    let car = CarBuilder::new("Toyota", "Corolla")
        .year(2020)
        .color(Color::Blue)
        .automatic(true)
        .build();

    println!("{:#?}", car);
}
```

- Essentially: Constructor (`T::new()`) -> methods that modify the instance -> `build()`/`finish()`/`whatever()`

Builder pattern

Builder pattern

- Extremely popular in Rust (because we have no default parameters and variadic functions)

```
fn main() {
    let car = CarBuilder::new("Toyota", "Corolla")
        .year(2020)
        .color(Color::Blue)
        .automatic(true)
        .build();

    println!("{:#?}", car);
}
```

- Essentially: Constructor (`T::new()`) -> methods that modify the instance -> `build()`/`finish()`/`whatever()`
- Generally, the builder methods return either **T** or **&mut T** (and rarely **&T** if doing interior mutability magic)

Builder pattern

Builder pattern

- Extremely popular in Rust (because we have no default parameters and variadic functions)

```
fn main() {
    let car = CarBuilder::new("Toyota", "Corolla")
        .year(2020)
        .color(Color::Blue)
        .automatic(true)
        .build();

    println!("{:#?}", car);
}
```

- Essentially: Constructor (`T::new()`) -> methods that modify the instance -> `build()`/`finish()`/`whatever()`
- Generally, the builder methods return either **T** or **&mut T** (and rarely **&T** if doing interior mutability magic)

Builder pattern

- Let's you validate and convert parameters implicitly, use defaults, and keep internal structure hidden

Builder pattern

- Extremely popular in Rust (because we have no default parameters and variadic functions)

```
fn main() {
    let car = CarBuilder::new("Toyota", "Corolla")
        .year(2020)
        .color(Color::Blue)
        .automatic(true)
        .build();

    println!("{:#?}", car);
}
```

- Essentially: Constructor (`T::new()`) -> methods that modify the instance -> `build()`/`finish()`/`whatever()`
- Generally, the builder methods return either **T** or **&mut T** (and rarely **&T** if doing interior mutability magic)

Builder pattern

- Let's you validate and convert parameters implicitly, use defaults, and keep internal structure hidden
  - Forward compatibility - you can change the struct fields however you want

Builder pattern

- Extremely popular in Rust (because we have no default parameters and variadic functions)

```
fn main() {
    let car = CarBuilder::new("Toyota", "Corolla")
        .year(2020)
        .color(Color::Blue)
        .automatic(true)
        .build();

    println!("{:#?}", car);
}
```

- Essentially: Constructor (`T::new()`) -> methods that modify the instance -> `build()`/`finish()`/`whatever()`
- Generally, the builder methods return either **T** or **&mut T** (and rarely **&T** if doing interior mutability magic)

Builder pattern

- Let's you validate and convert parameters implicitly, use defaults, and keep internal structure hidden
  - Forward compatibility - you can change the struct fields however you want
- In std: e.g. `std::process::Command`

Builder pattern

- Extremely popular in Rust (because we have no default parameters and variadic functions)

```
fn main() {
    let car = CarBuilder::new("Toyota", "Corolla")
        .year(2020)
        .color(Color::Blue)
        .automatic(true)
        .build();

    println!("{:#?}", car);
}
```

- Essentially: Constructor (`T::new()`) -> methods that modify the instance -> `build()`/`finish()`/`whatever()`
- Generally, the builder methods return either **T** or **&mut T** (and rarely **&T** if doing interior mutability magic)

Builder pattern

- Let's you validate and convert parameters implicitly, use defaults, and keep internal structure hidden
  - Forward compatibility - you can change the struct fields however you want
- In std: e.g. `std::process::Command`
- You can also do **session types** where your builder goes through several types with different methods

Builder pattern

- Extremely popular in Rust (because we have no default parameters and variadic functions)

```
fn main() {
    let car = CarBuilder::new("Toyota", "Corolla")
        .year(2020)
        .color(Color::Blue)
        .automatic(true)
        .build();

    println!("{:#?}", car);
}
```

- Essentially: Constructor (`T::new()`) -> methods that modify the instance -> `build()`/`finish()`/`whatever()`
- Generally, the builder methods return either **T** or **&mut T** (and rarely **&T** if doing interior mutability magic)

Builder pattern

- Let's you validate and convert parameters implicitly, use defaults, and keep internal structure hidden
  - Forward compatibility - you can change the struct fields however you want
- In std: e.g. `std::process::Command`
- You can also do **session types** where your builder goes through several types with different methods

(Meaning you can force an order of operations - useful with protocols, e.g. HTTP requests)

Builder pattern

- Extremely popular in Rust (because we have no default parameters and variadic functions)

```
fn main() {
    let car = CarBuilder::new("Toyota", "Corolla")
        .year(2020)
        .color(Color::Blue)
        .automatic(true)
        .build();

    println!("{:#?}", car);
}
```

- Essentially: Constructor (`T::new()`) -> methods that modify the instance -> `build()`/`finish()`/`whatever()`
- Generally, the builder methods return either **T** or **&mut T** (and rarely **&T** if doing interior mutability magic)

Builder pattern

- Let's you validate and convert parameters implicitly, use defaults, and keep internal structure hidden
  - Forward compatibility - you can change the struct fields however you want
- In std: e.g. `std::process::Command`
- You can also do **session types** where your builder goes through several types with different methods

(Meaning you can force an order of operations - useful with protocols, e.g. HTTP requests)
(starts to smell like substructural type systems :))

Type conversion ergonomics

```rust
use std::path::Path; // let's pretend Path::exists() doesn't exist :)
use std::fs;

fn file_exists(path: &Path) -> bool {
    fs::metadata(path).is_ok()
}

fn main() {
    // Example usage
    let path = Path::new("./example.txt");
    println!("Does the file exist? {}", file_exists(path));
}
```

- Not ideal, since we now require user to construct a Path directly. Less flexible

```rust
use std::path::Path; // let's pretend Path::exists() doesn't exist :)
use std::fs;

fn file_exists<P: AsRef<Path>>(path: P) -> bool {
    fs::metadata(path.as_ref()).is_ok()
}

fn main() {
    // Example usage with a &str
    let path_str = "./example.txt";
    println!("Does the file exist? {}", file_exists(path_str));

    // Example usage with a PathBuf
    let path_buf = Path::new("./example.txt").to_path_buf();
    println!("Does the file exist? {}", file_exists(path_buf));
}
```

Nice traits to use:

Nice traits to use:

- AsRef

Nice traits to use:

- AsRef
- AsMut

Nice traits to use:

- AsRef
- AsMut
- From

Nice traits to use:

- AsRef
- AsMut
- From
- Into - Do not implement directly unless needed (blanket impl on From<T>)

Nice traits to use:

- AsRef
- AsMut
- From
- Into - Do not implement directly unless needed (blanket impl on From<T>)
- TryFrom

Nice traits to use:

- AsRef
- AsMut
- From
- Into - Do not implement directly unless needed (blanket impl on From<T>)
- TryFrom
- TryInto

Nice traits to use:

- AsRef
- AsMut
- From
- Into - Do not implement directly unless needed (blanket impl on From<T>)
- TryFrom
- TryInto
- FromStr (gives you a free `.parse()` on string types)

Nice traits to use:

- AsRef
- AsMut
- From
- Into - Do not implement directly unless needed (blanket impl on From<T>)
- TryFrom
- TryInto
- FromStr (gives you a free `.parse()` on string types)

Naturally, you can implement these on your types wherever applicable

Nice traits to implement

- Debug and optionally Display

Nice traits to implement

- Debug and optionally Display
- Display & Error (for your Error types)

Nice traits to implement

- Debug and optionally Display
- Display & Error (for your Error types)
- (Partial)Ord, (Partial)Eq

Nice traits to implement

- Debug and optionally Display
- Display & Error (for your Error types)
- (Partial)Ord, (Partial)Eq
- Clone, and Copy if it makes sense

Nice traits to implement

- Debug and optionally Display
- Display & Error (for your Error types)
- (Partial)Ord, (Partial)Eq
- Clone, and Copy if it makes sense
- Hash (if applicable)

Nice traits to implement

- Debug and optionally Display
- Display & Error (for your Error types)
- (Partial)Ord, (Partial)Eq
- Clone, and Copy if it makes sense
- Hash (if applicable)
- Serde Serialize & Deserialize

Nice traits to implement

- Debug and optionally Display
- Display & Error (for your Error types)
- (Partial)Ord, (Partial)Eq
- Clone, and Copy if it makes sense
- Hash (if applicable)
- Serde Serialize & Deserialize
- **Default**

Nice traits to implement

- Debug and optionally Display
- Display & Error (for your Error types)
- (Partial)Ord, (Partial)Eq
- Clone, and Copy if it makes sense
- Hash (if applicable)
- Serde Serialize & Deserialize
- **Default**
- Also implement Iterator if your type is a collection or a stream

Nice traits to implement

- Debug and optionally Display
- Display & Error (for your Error types)
- (Partial)Ord, (Partial)Eq
- Clone, and Copy if it makes sense
- Hash (if applicable)
- Serde Serialize & Deserialize
- **Default**
- Also implement Iterator if your type is a collection or a stream
- And FromIterator if you want to use `.collect()`

Nice traits to implement

- Debug and optionally Display
- Display & Error (for your Error types)
- (Partial)Ord, (Partial)Eq
- Clone, and Copy if it makes sense
- Hash (if applicable)
- Serde Serialize & Deserialize
- **Default**
- Also implement Iterator if your type is a collection or a stream
- And FromIterator if you want to use `.collect()`
- TIP: Prefer taking a slice to taking a Vec, and if possible, just take a generic Iterator

Nice traits to implement

- Debug and optionally Display
- Display & Error (for your Error types)
- (Partial)Ord, (Partial)Eq
- Clone, and Copy if it makes sense
- Hash (if applicable)
- Serde Serialize & Deserialize
- **Default**
- Also implement Iterator if your type is a collection or a stream
- And FromIterator if you want to use `.collect()`
- TIP: Prefer taking a slice to taking a Vec, and if possible, just take a generic Iterator
  - e.g `fn hi<T: Iterator<Item=u32>>(numbers: T)`

```rust
struct MyCollection<T> { // simple example
    elements: Vec<T>,    // consider e.g. address book or any tree ADT
}

// Implementing the FromIterator trait for MyCollection
impl<T> FromIterator<T> for MyCollection<T> {
    fn from_iter<I: IntoIterator<Item = T>>(iter: I) -> Self {
        let mut c = MyCollection { elements: Vec::new() };

        for i in iter { c.elements.push(i); }

        c
    }
}

let collected: MyCollection<i32> = vec![1, 2, 3, 4, 5].into_iter().collect();
println!("{:?}", collected.elements); // Prints: [1, 2, 3, 4, 5]
```

Extension traits and orphan rules

- It is handy to implement foreign traits, sometimes for foreign types

Extension traits and orphan rules

- It is handy to implement foreign traits, sometimes for foreign types
  - Forbidden to do ForeignTrait on ForeignType -> Use `newtype` pattern

Extension traits and orphan rules

- It is handy to implement foreign traits, sometimes for foreign types
  - Forbidden to do ForeignTrait on ForeignType -> Use `newtype` pattern
- You can also write extension traits to provide additional functionality to existing items:

Extension traits and orphan rules

- It is handy to implement foreign traits, sometimes for foreign types
  - Forbidden to do ForeignTrait on ForeignType -> Use `newtype` pattern
- You can also write extension traits to provide additional functionality to existing items:
  - By convention name is `TraitOrTypeNameExt`

Extension traits and orphan rules

- It is handy to implement foreign traits, sometimes for foreign types
  - Forbidden to do ForeignTrait on ForeignType -> Use `newtype` pattern
- You can also write extension traits to provide additional functionality to existing items:
  - By convention name is `TraitOrTypeNameExt`
  - In std for example `std::ascii::AsciiExt` (deprecated)
- https://github.com/Ixrec/rust-orphan-rules

```rust
use std::fmt;
pub struct DisplayVec<T>(pub Vec<T>);

impl<T: fmt::Display> fmt::Display for DisplayVec<T> {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        let elements_as_strings: Vec<String> = self.0.iter().map(|e|
e.to_string()).collect();
        write!(f, "[{}]", elements_as_strings.join(", "))
    }
}

let numbers = DisplayVec(vec![1, 2, 3, 4, 5]);
println!("{}", numbers); // Prints: "[1, 2, 3, 4, 5]"
```

Testing

- Use `[dev-dependencies]` in Cargo.toml for things you only need for examples

Testing

- Use `[dev-dependencies]` in Cargo.toml for things you only need for examples and testing

Testing

- Use `[dev-dependencies]` in Cargo.toml for things you only need for examples and testing
- e.g. criterion for writing benchmarks

Testing

- Use `[dev-dependencies]` in Cargo.toml for things you only need for examples and testing
- e.g. criterion for writing benchmarks
- In Rust:

Testing

- Use `[dev-dependencies]` in Cargo.toml for things you only need for examples and testing
- e.g. criterion for writing benchmarks
- In Rust:
  - Unified built-in syntax for writing tests

Testing

- Use `[dev-dependencies]` in Cargo.toml for things you only need for examples and testing
- e.g. criterion for writing benchmarks
- In Rust:
  - Unified built-in syntax for writing tests
  - Types of tests:

Testing

- Use `[dev-dependencies]` in Cargo.toml for things you only need for examples and testing
- e.g. criterion for writing benchmarks
- In Rust:
  - Unified built-in syntax for writing tests
  - Types of tests:
    - Unit - next to your code or `src/tests.rs`

Testing

- Use `[dev-dependencies]` in Cargo.toml for things you only need for examples and testing
- e.g. criterion for writing benchmarks
- In Rust:
  - Unified built-in syntax for writing tests
  - Types of tests:
    - Unit - next to your code or `src/tests.rs`
    - Integration - in `tests/`, view your code as a dependency

Testing

- Use `[dev-dependencies]` in Cargo.toml for things you only need for examples and testing
- e.g. criterion for writing benchmarks
- In Rust:
  - Unified built-in syntax for writing tests
  - Types of tests:
    - Unit - next to your code or `src/tests.rs`
    - Integration - in `tests/`, view your code as a dependency
      - -> you cannot touch internal states

Testing

- Use `[dev-dependencies]` in Cargo.toml for things you only need for examples and testing
- e.g. criterion for writing benchmarks
- In Rust:
  - Unified built-in syntax for writing tests
  - Types of tests:
    - Unit - next to your code or `src/tests.rs`
    - Integration - in `tests/`, view your code as a dependency
      - -> you cannot touch internal states
    - Benchmarks - similar to integration tests, but for making perf statistics

Testing

- Use `[dev-dependencies]` in Cargo.toml for things you only need for examples and testing
- e.g. criterion for writing benchmarks
- In Rust:
  - Unified built-in syntax for writing tests
  - Types of tests:
    - Unit - next to your code or `src/tests.rs`
    - Integration - in `tests/`, view your code as a dependency
      - -> you cannot touch internal states
    - Benchmarks - similar to integration tests, but for making perf statistics
      - Some parts of support are nightly-only

Test syntax in Rust

```rust
#[cfg(test)]
mod tests {
    #[test]
    fn test_addition() {
        let sum = 2 + 2;
        assert_eq!(sum, 4);
    }

    #[test]
    #[should_panic(expected = "assertion failed")]
    fn test_failure_scenario() {
        assert!(false, "This test will panic!");
    }
}
```

Run with `cargo test`

Result<T> tests

```rust
#[cfg(test)]
mod tests {
    #[test]
    fn test_division() -> Result<(), String> {
        let result = 10 / 2;
        if result == 5 {
            Ok(())
        } else {
            Err(String::from("Division result was not as expected."))
        }
    }
}
```

Benchmark

```rust
use criterion::{black_box, criterion_group, criterion_main, Criterion};

fn fibonacci(n: u64) -> u64 {
    match n {
        0 => 0,
        1 => 1,
        _ => fibonacci(n - 1) + fibonacci(n - 2),
    }
}

fn criterion_benchmark(c: &mut Criterion) {
    c.bench_function("fibonacci 20", |b| b.iter(|| fibonacci(black_box(20))));
}

criterion_group!(benches, criterion_benchmark); criterion_main!(benches);
```

Run with `cargo bench`

Maintenance

Maintenance

- Use `rustfmt` (formatting) and `clippy` (linter)

Maintenance

- Use `rustfmt` (formatting) and `clippy` (linter)
  - VS Code and others can do format-on-save, handy

Maintenance

- Use `rustfmt` (formatting) and `clippy` (linter)
  - VS Code and others can do format-on-save, handy
- You can add them as automatic checks to your CI pipeline

Maintenance

- Use `rustfmt` (formatting) and `clippy` (linter)
  - VS Code and others can do format-on-save, handy
- You can add them as automatic checks to your CI pipeline
  - E.g. GitHub Actions, Gitlab CI

Maintenance

- Use `rustfmt` (formatting) and `clippy` (linter)
  - VS Code and others can do format-on-save, handy
- You can add them as automatic checks to your CI pipeline
  - E.g. GitHub Actions, Gitlab CI
- `#![deny(warnings)]` in crate root

Maintenance

- Use `rustfmt` (formatting) and `clippy` (linter)
  - VS Code and others can do format-on-save, handy
- You can add them as automatic checks to your CI pipeline
  - E.g. GitHub Actions, Gitlab CI
- `#![deny(warnings)]` in crate root
  - In Rust, it is a standard practice not to publish code with warnings

Maintenance

- Use `rustfmt` (formatting) and `clippy` (linter)
  - VS Code and others can do format-on-save, handy
- You can add them as automatic checks to your CI pipeline
  - E.g. GitHub Actions, Gitlab CI
- `#![deny(warnings)]` in crate root
  - In Rust, it is a standard practice not to publish code with warnings
  - You can enable additional lints and additional clippy lints

Maintenance

- Use `rustfmt` (formatting) and `clippy` (linter)
  - VS Code and others can do format-on-save, handy
- You can add them as automatic checks to your CI pipeline
  - E.g. GitHub Actions, Gitlab CI
- `#![deny(warnings)]` in crate root
  - In Rust, it is a standard practice not to publish code with warnings
  - You can enable additional lints and additional clippy lints
- Other nice checks:

Maintenance

- Use `rustfmt` (formatting) and `clippy` (linter)
  - VS Code and others can do format-on-save, handy
- You can add them as automatic checks to your CI pipeline
  - E.g. GitHub Actions, Gitlab CI
- `#![deny(warnings)]` in crate root
  - In Rust, it is a standard practice not to publish code with warnings
  - You can enable additional lints and additional clippy lints
- Other nice checks:
  - `cargo-machete` - detect unused deps

Maintenance

- Use `rustfmt` (formatting) and `clippy` (linter)
  - VS Code and others can do format-on-save, handy
- You can add them as automatic checks to your CI pipeline
  - E.g. GitHub Actions, Gitlab CI
- `#![deny(warnings)]` in crate root
  - In Rust, it is a standard practice not to publish code with warnings
  - You can enable additional lints and additional clippy lints
- Other nice checks:
  - `cargo-machete` - detect unused deps
  - `cargo-outdated` - outdated deps

Maintenance

- Use `rustfmt` (formatting) and `clippy` (linter)
  - VS Code and others can do format-on-save, handy
- You can add them as automatic checks to your CI pipeline
  - E.g. GitHub Actions, Gitlab CI
- `#![deny(warnings)]` in crate root
  - In Rust, it is a standard practice not to publish code with warnings
  - You can enable additional lints and additional clippy lints
- Other nice checks:
  - `cargo-machete` - detect unused deps
  - `cargo-outdated` - outdated deps
  - `cargo-tarpaulin/cargo-llvm-cov` - code coverage

Conclusion

- Make your API minimal and consistent

Conclusion

- Make your API minimal and consistent
- Don’t hardcode types where you don’t need to

Conclusion

- Make your API minimal and consistent
- Don’t hardcode types where you don’t need to
  - Use generics with trait bounds

Conclusion

- Make your API minimal and consistent
- Don't hardcode types where you don't need to
  - Use generics with trait bounds
- Integrate with Rust std traits

Conclusion

- Make your API minimal and consistent
- Don't hardcode types where you don't need to
  - Use generics with trait bounds
- Integrate with Rust std traits
- Write tests and documentation

Conclusion

- Make your API minimal and consistent
- Don't hardcode types where you don't need to
  - Use generics with trait bounds
- Integrate with Rust std traits
- Write tests and documentation
- Use tools to perform automatic checks