Exemplary by Design

Building and Maintaining Rust at Scale

Jacob Pratt
2025-10-09
Paris

EURO RUST

Why care?

Idiomatic code, generally, is:

Why care?

Idiomatic code, generally, is:

- easier to read, maintain, and extend

Why care?

Idiomatic code, generally, is:

- easier to read, maintain, and extend
- less buggy

Why care?

Idiomatic code, generally, is:

- easier to read, maintain, and extend
- less buggy
- contains fewer security vulnerabilities

Why care?

Idiomatic code, generally, is:

- easier to read, maintain, and extend
- less buggy
- contains fewer security vulnerabilities
- faster

Why care?

Idiomatic code, generally, is:

- easier to read, maintain, and extend
- less buggy
- contains fewer security vulnerabilities
- faster
- potentially cheaper

Project Structure

module organization

```
src/
├ lib.rs
├ alpha.rs
└ beta/
  ├ gamma.rs
  └ mod.rs
```

```
src/
├ lib.rs
├ alpha.rs
├ beta.rs
└ beta/
  └ gamma.rs
```

Project Structure

system APIs

```
#[cfg_attr(
    windows,
    path = "windows.rs"
)]
#[cfg_attr(
    unix,
    path = "unix.rs"
)]
// fall back to `imp.rs`
mod imp;
use imp::*;
```

Project Structure

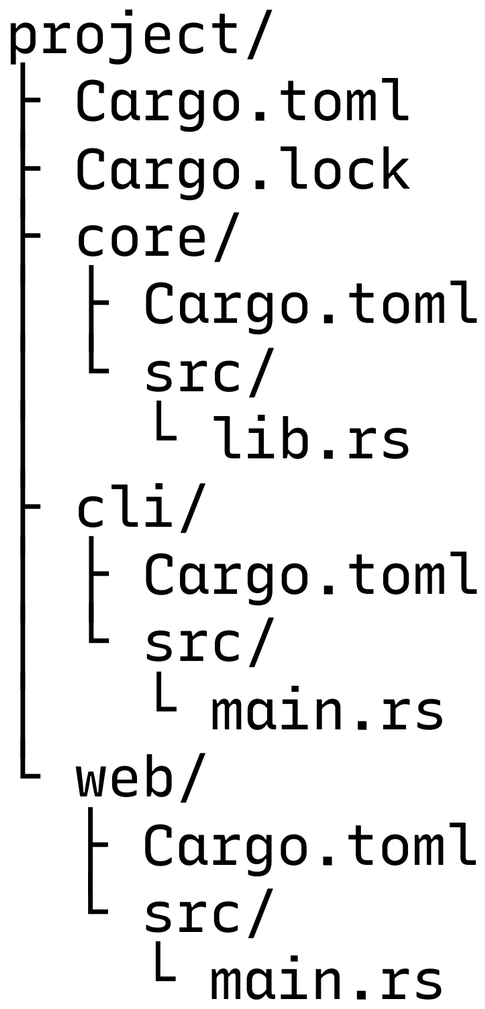
system APIs

```
#[cfg_attr(
    windows,
    path = "windows.rs"
)]
#[cfg_attr(
    unix,
    path = "unix.rs"
)]
// fall back to `imp.rs`
mod imp;
use imp::*;
```

```
cfg_select! {
    windows => {
        mod windows;
        use windows::*;
    },
    unix => {
        mod unix;
        use unix::*;
    },
    _ => {
        mod fallback;
        use fallback::*;
    },
}
```

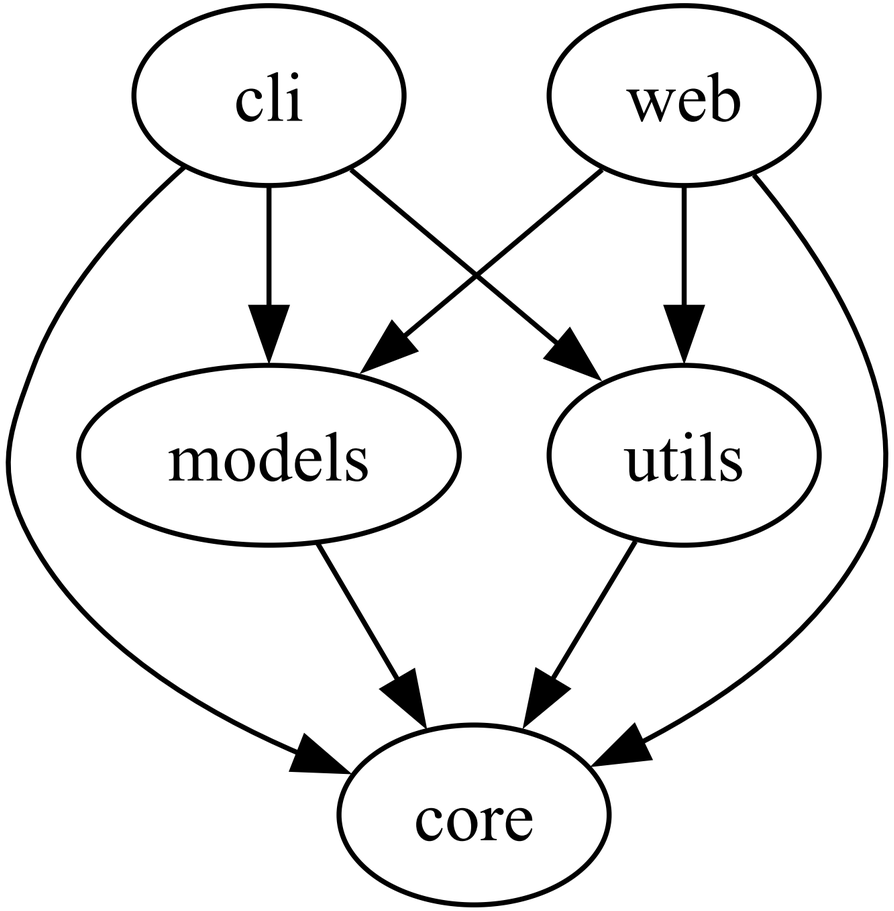
Project Structure

workspaces



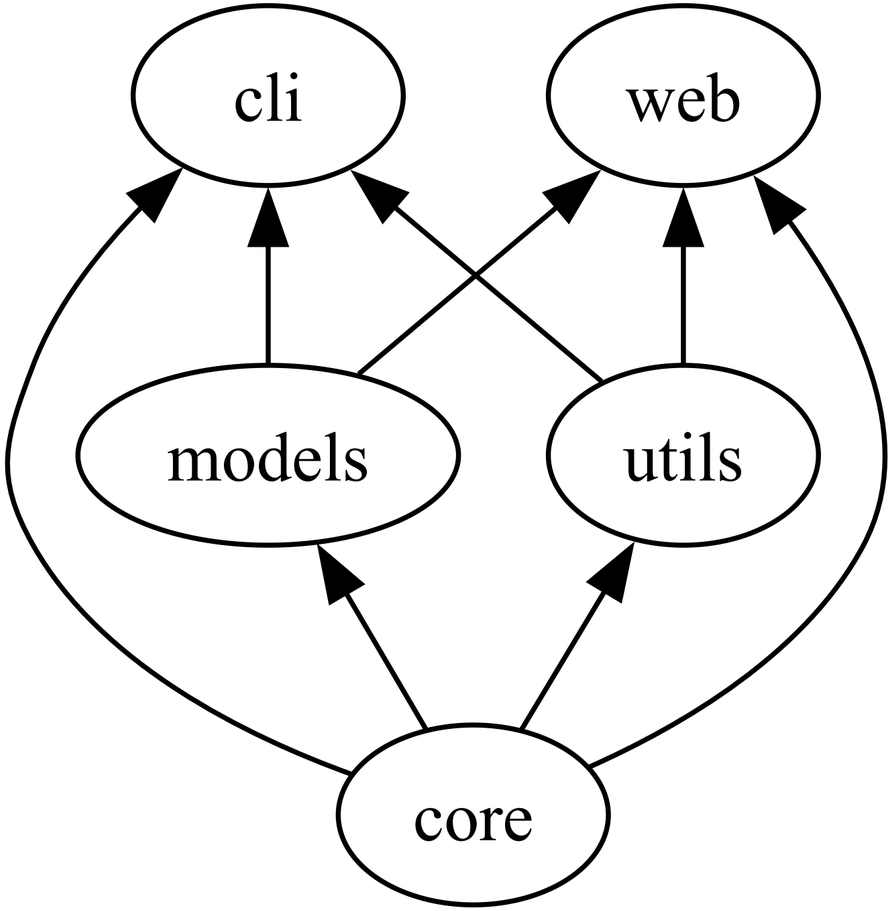
Project Structure

crate graphs



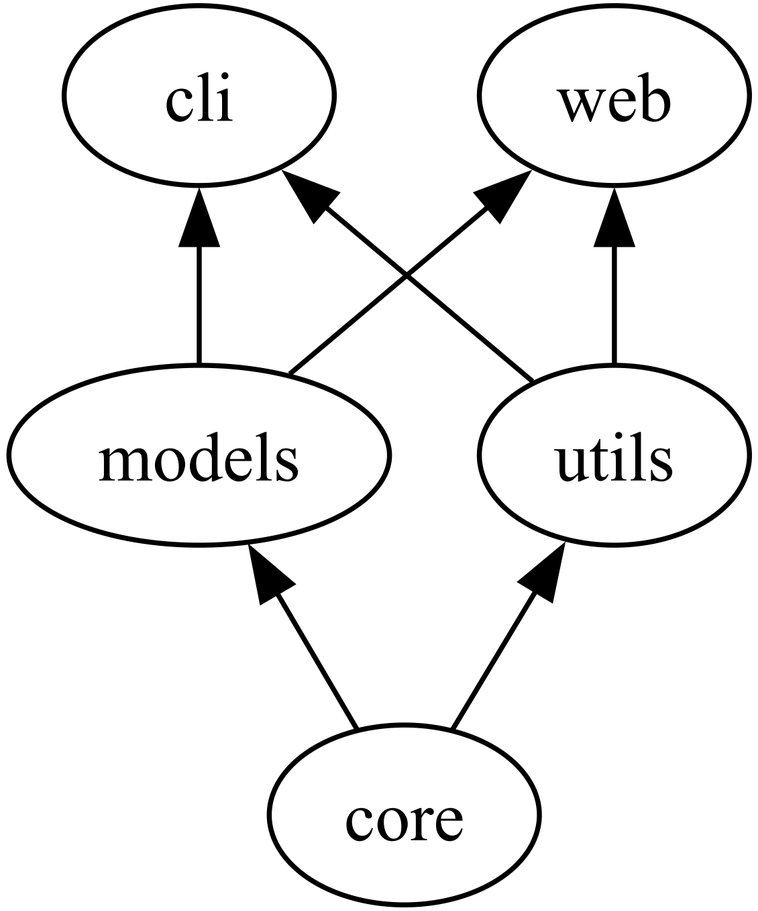
Project Structure

crate graphs



Project Structure

crate graphs



Editor Configuration

rust-analyzer

Editor Configuration
rust-analyzer

- inline type hints

Editor Configuration

rust-analyzer

- inline type hints
- look up implementations & references of a type

Editor Configuration

rust-analyzer

- inline type hints
- look up implementations & references of a type
- view macro expansion

Editor Configuration

rust-analyzer

- inline type hints
- look up implementations & references of a type
- view macro expansion
- test running and debugging

Editor Configuration

rust-analyzer

- inline type hints
- look up implementations & references of a type
- view macro expansion
- test running and debugging
- about a million other things

Editor Configuration

rust-analyzer

- inline type hints
- look up implementations & references of a type
- view macro expansion
- test running and debugging
- about a million other things

```
"editor.semanticTokenColorCustomizations": {
    "rules": {
        "*.mutable:rust": {
            "underline": true
        },
    }
},
```

Editor Configuration

VS Code specifics

```
"workbench.editor.customLabels.patterns": {
    "**/Cargo.toml":  "${dirname}/Cargo.toml",
    "**/build.rs":    "${dirname}/build.rs",
    "**/mod.rs":      "${dirname}/mod.rs",
    "*/*/**/bin.rs":  "${dirname(1)}/bin.rs",
    "*/*/**/lib.rs":  "${dirname(1)}/lib.rs"
    "*/*/**/main.rs": "${dirname(1)}/main.rs",
},
```

Design Patterns

clippy lints

```
struct Foo {
    alpha: i32,
    some_flag: Arc<Mutex<bool>>,
}
```

Design Patterns

clippy lints

```
struct Foo {
    alpha: i32,
    some_flag: Arc<Mutex<bool>>,
}

struct Foo {
    alpha: i32,
    some_flag: AtomicBool,
}
```

Design Patterns

iterators

```
fn sum_even_squares(nums: &[i32]) -> i32 {
    let mut total = 0;
    for idx in 0..=nums.len() - 1 {
        if nums[idx] % 2 == 0 {
            total += nums[idx] * nums[idx];
        }
    }
    total
}
```

Design Patterns

iterators

```rust
fn sum_even_squares(nums: &[i32]) -> i32 {
    let mut total = 0;
    for idx in 0..=nums.len() - 1 {
        if nums[idx] % 2 == 0 {
            total += nums[idx] * nums[idx];
        }
    }
    total
}

fn sum_even_squares(nums: &[i32]) -> i32 {
    nums.iter()
        .filter(|&&x| x % 2 == 0)
        .map(|&x| x * x)
        .sum()
}
```

Design Patterns

Result vs. *Option*

```
match read_db_row() { // -> Result<Option<Row>, DbError>
    Ok(Some(row)) => println!("data: {row}"),
    Ok(None) => println!("no row"),
    Err(e) => eprintln!("database error: {e}"),
}
```

Design Patterns

trait usage

```
struct Dog {
    animal: Animal,
    /* … */
}
struct Animal {
    /* … */
}

impl Animal {
    fn make_noise(&self) {
        /* … */
    }
}
```

Design Patterns

trait usage

```
struct Dog {
    animal: Animal,
    /* … */
}
struct Animal {
    /* … */
}

impl Animal {
    fn make_noise(&self) {
        /* … */
    }
}
```

```
struct Dog {
    /* … */
}

trait Animal {
    fn make_noise(&self);
}

impl Animal for Dog {
    fn make_noise(&self) {
        /* … */
    }
}
```

Design Patterns

extension traits

```
trait NumericalDuration {
    fn seconds(self) -> Duration;
}

impl NumericalDuration for i64 {
    fn seconds(self) -> Duration {
        Duration::seconds(self)
    }
}
```

Design Patterns

extension traits

```
trait NumericalDuration {
    fn seconds(self) -> Duration;
}

impl NumericalDuration for i64 {
    fn seconds(self) -> Duration {
        Duration::seconds(self)
    }
}

// another module/crate
use crate::NumericalDuration as _;

// `5.seconds()` now works!
```

Design Patterns

newtype

```
let miles = 100.;                    // contractor
let kilometers = 100.;               // government

// satellite go 💥
let distance = miles + kilometers;
```

Design Patterns

newtype

```
struct Miles(f32);
struct Kilometers(f32);



let miles = Miles(100.);              // contractor
let kilometers = Kilometers(100.); // government

// satellite go ⛔
let distance = miles + kilometers; // compile error
```

Design Patterns

newtype

```
struct Miles(f32);
struct Kilometers(f32);

impl Add for Miles { /* … */ }

let miles = Miles(100.);              // contractor
let kilometers = Kilometers(100.); // government

// satellite go 🛰️
let distance = miles + kilometers;
```

Design Patterns

adding field to existing struct

```rust
#[derive(Serialize, Deserialize)]
struct WithFoo<T> {
    foo: i32,
    #[serde(flatten)]
    inner: T,
}
```

Design Patterns

adding field to existing struct

```rust
#[derive(Serialize, Deserialize)]
struct WithFoo<T> {
    foo: i32,
    #[serde(flatten)]
    inner: T,
}

impl<T> Deref for WithFoo<T> {
    type Target = T;
    fn deref(&self) -> &T { &self.inner }
}

impl<T> DerefMut for WithFoo<T> {
    fn deref_mut(&mut self) -> &mut T { &mut self.inner }
}
```

Design Patterns

adding field to existing struct

```rust
#[derive(Serialize, Deserialize)]
struct WithFoo<T> {
    foo: i32,
    #[serde(flatten)]
    inner: T,
}

impl<T> Deref for WithFoo<T> {
    type Target = T;
    fn deref(&self) -> &T { &self.inner }
}

impl<T> DerefMut for WithFoo<T> {
    fn deref_mut(&mut self) -> &mut T { &mut self.inner }
}

fn bar(x: WithFoo<T>) { /* … */ }
```

Design Patterns

enum within struct

```
pub struct Error(ErrorInner);
enum ErrorInner {
    /* … */
}
```

Design Patterns

enum within struct

```
pub struct Error(ErrorInner);
enum ErrorInner {
    /* … */
}
```

- complete control over public API

Design Patterns

enum within struct

```
pub struct Error(ErrorInner);
enum ErrorInner {
    /* … */
}
```

- complete control over public API
- avoid exposing implementation details

Design Patterns

enum within struct

```
pub struct Error(ErrorInner);
enum ErrorInner {
    /* … */
}
```

- complete control over public API
- avoid exposing implementation details
- can remove variants without breaking changes

Design Patterns

enum within struct

```
pub struct Error(ErrorInner);
enum ErrorInner {
    /* … */
}
```

- complete control over public API
- avoid exposing implementation details
- can remove variants without breaking changes
- allows refactoring of implementation details

Design Patterns

enum within struct

```
pub struct Error(ErrorInner);
enum ErrorInner {
    /* … */
}
```

- complete control over public API
- avoid exposing implementation details
- can remove variants without breaking changes
- allows refactoring of implementation details
- easier for downstream users to handle one struct

Design Patterns

#[non_exhaustive]

```
#[non_exhaustive]
pub struct Config {
    pub foo: String,
}
#[non_exhaustive]
enum Message {
    Start,
    Stop,
}
```

Design Patterns

#[non_exhaustive]

```rust
#[non_exhaustive]
pub struct Config {
    pub foo: String,
}
#[non_exhaustive]
enum Message {
    Start,
    Stop,
}

let Config { foo } = cfg; // error! `..` is needed
match msg { // error! `_` arm is needed
    Message::Start => { /* … */ }
    Message::Stop => { /* … */ }
}
```

Design Patterns

builder methods

```
struct FooBuilder {
    alpha: Option<String>,
    beta: Option<u32>,
    /* … */
}

impl FooBuilder {
    fn alpha(mut self, value: String) -> Self { /* … */ }
    fn beta(mut self, value: u32) -> Self { /* … */ }
    fn build(self) -> Foo { /* … */ }
}
```

Design Patterns

builder methods

```
struct FooBuilder {
    alpha: Option<String>,
    beta: Option<u32>,
    /* … */
}

impl FooBuilder {
    fn alpha(mut self, value: String) -> Self { /* … */ }
    fn beta(mut self, value: u32) -> Self { /* … */ }
    fn build(self) -> Foo { /* … */ }
}

let foo = FooBuilder::new()
    .alpha("example".to_owned())
    .build(); // `beta` is a default value
```

Design Patterns

builder methods: replacement?

```
struct Foo {
    alpha: String,
    beta: u32 = 42,
    /* … */
}

let foo = Foo {
    alpha: "example".to_owned(),
    ..
}
```

Design Patterns

string concatenation

```
let s1 = "Hello, ".to_owned();
let s2 = "world!";
let s3 = s1 + s2;
// `s1` is moved above and can no longer be used
```

Design Patterns

string concatenation

```
let s1 = "Hello, ".to_owned();
let s2 = "world!";
let s3 = s1 + s2;
// `s1` is moved above and can no longer be used

let s1 = "Hello, ".to_owned();
let s2 = "world!";
let s3 = format!("{s1}{s2}");
// `s1` can still be used after this line
```

Design Patterns

idioms change!

```
maplit::hashmap! {}
```

Design Patterns

idioms change!

```
maplit::hashmap! {}
↳ HashMap::from_iter()
```

Design Patterns

idioms change!

```
maplit::hashmap! {}
↳ HashMap::from_iter()
↳ hash_map! {}  (unstable)
```

Design Patterns

idioms change!

```
maplit::hashmap! {}
↳ HashMap::from_iter()
↳ hash_map! {}  (unstable)

try!()
```

Design Patterns

idioms change!

```
maplit::hashmap! {}
↳ HashMap::from_iter()
↳ hash_map! {}  (unstable)

try!()
↳ ?
```

Error Handling

- `anyhow` or `eyre` for binaries

Error Handling

- `anyhow` or `eyre` for binaries
- `thiserror` for libraries

Error Handling

- `anyhow` or `eyre` for binaries
- `thiserror` for libraries
- implement `From` for error conversions with `?`

Testing

- unit vs. integration tests

Testing

- unit vs. integration tests
- property testing: quickcheck

Testing

- unit vs. integration tests
- property testing: `quickcheck`
- undefined behavior checks: `miri`

Testing

- unit vs. integration tests
- property testing: `quickcheck`
- undefined behavior checks: `miri`
- code coverage: `cargo-llvm-cov`

Maintenance

- dependency updates, upgrades

Maintenance

- dependency updates, upgrades
- documentation

Maintenance

- dependency updates, upgrades
- documentation
- changelogs

Maintenance

- dependency updates, upgrades
- documentation
- changelogs
- CI

Resources

- license enforcement & advisories: `cargo-deny`

Resources

- license enforcement & advisories: `cargo-deny`
- semver: `cargo book` & `cargo-semver-checks`

Resources

- license enforcement & advisories: `cargo-deny`
- semver: `cargo book` & `cargo-semver-checks`
- performance: `criterion` & `flamegraph`

Resources

- license enforcement & advisories: `cargo-deny`
- semver: `cargo book` & `cargo-semver-checks`
- performance: `criterion` & `flamegraph`
- Rust API guidelines

Resources

- license enforcement & advisories: `cargo-deny`
- semver: `cargo book` & `cargo-semver-checks`
- performance: `criterion` & `flamegraph`
- Rust API guidelines
- Rustonomicon

Exemplary by Design

Building and Maintaining Rust at Scale

Jacob Pratt

@jhpratt

@jhpratt@mastodon.social

jhpratt.dev

sponsor.jhpratt.dev

2025-10-09
Paris

EURO RUST