Dependency Injection in Rust

Dennis Sprokholt

November 2022

1 Introduction

For my recent Rust programs, I use a variant of *dependency injection*[1] to achieve *inversion of control*. In this note, I describe my application of that *design pattern* in Rust.

As an illustrative example, consider a program that performs the "heavy computation" of finding the *next* prime number:

- We load our previous prime p_n from cache
- We compute the next prime p_{n+1}
- We store p_{n+1} to the cache

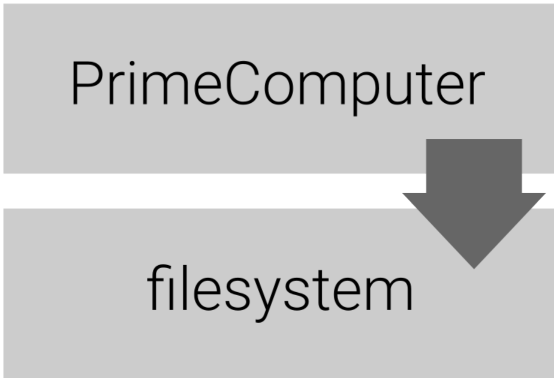
Our program could look as follows:

```
fn main( ) {
  let c = PrimeComputer;
  c.run( );
}

struct PrimeComputer;

impl PrimeComputer {
  fn run( &self ) {
    let p0 = read_prime_from_cache( );
    let p1 = compute_next_prime( p0 );
    store_prime_to_cache( p1 );
  }
}

fn read_prime_from_cache( ) -> u64 {
  fs::read_to_string( "cache.txt" ).ok( )
    .and_then( |s| s.parse::<u64>( ).ok( ) )
    .unwrap_or( 2 ) // reads 2 if empty
}

fn store_prime_to_cache( n: u64 ) {
  let mut file = File::create( "cache.txt" ).unwrap( );
  write!( &mut file, "{}", n ).unwrap( );
}
```

[1] specifically, *constructor injection*

We decided to use the file `cache.txt` as our cache. Simultaneously, we established a *strong coupling* between our *domain logic* (i.e., computing primes) and filesystem. We could depict this as:

Figure 1: Problematic Structure

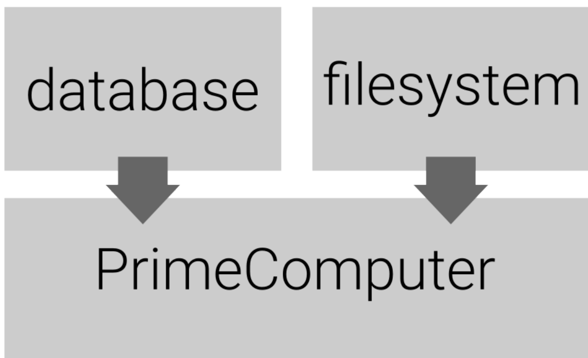


Problem – The `PrimeComputer` is *aware* of its *dependency* on the filesystem as cache.

2 Solution

Instead, we prefer our domain logic to be uncorrupted by interaction with *peripheral systems*. We *don't* want to build our application around the filesystem; So we *invert the structure*. Intuitively, we give the filesystem as a *plugin* to our domain logic. That also enables us to "swap out" our filesystem for another caching mechanism; For instance, a database. Consider this alternative structure:

Figure 2: Inverted Structure



There, `PrimeComputer` is unaware of the specific caching mechanism used. Instead, our filesystem (or rather, "filesystem object") *depends on* `PrimeComputer`. Let's go through an implementation in Rust. We define our `PrimeComputer` as follows:

prime_computer.rs

```
trait Cache {
  fn read_prime( &self ) -> u64;
  fn store_prime( &self, p: u64 );
}

struct PrimeComputer< C: Cache > { cache: C }

impl< C: Cache > PrimeComputer< C > {
  pub fn new( cache: C ) -> Self {
    PrimeComputer { cache }
  }

  pub fn run( &self ) {
    let p0 = self.cache.read_prime( );
    let p1 = compute_next_prime( p0 );
    self.cache.store_prime( p1 );
  }
}
```

Here, `PrimeComputer` is aware it has *some* cache. However, it is unaware *what* our cache is; It could be a file, a database, or something entirely different. We *decouple* our dependency on *a specific* caching mechanism from our domain logic. We *inject our dependency* (i.e., the filesystem cache) by passing it as an *argument* to our domain logic. In our `main.rs` we do that as follows:

main.rs

```
fn main( ) {
  let fs_cache = FilesystemCache;
  let c = PrimeComputer::new( fs_cache );
  c.run( );
}

struct FilesystemCache;

impl Cache for FilesystemCache {
  fn read_prime( &self ) -> u64 {
    fs::read_to_string( "cache.txt" ).ok( )
      .and_then( |s| s.parse::<u64>( ).ok( ) )
      .unwrap_or( 2 ) // reads 2 if empty
  }

  fn store_prime( &self, n: u64 ) {
    let mut file = File::create( "cache.txt" ).unwrap( );
    write!( &mut file, "{}", n ).unwrap( );
  }
}
```

Specific to Rust, we use *static dispatch* for `Cache`. That monomorphs (i.e., specializes) our `PrimeComputer` with `FilesystemCache`. Effectively, we end up with an executable that has performance identical to our original program (at least, it should).