UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

COS700 Research

Design Pattern Metaprogramming Foundations in Rust

Abstract Factory and Visitor

Student number: u19239395

Supervisor:
Dr. Linda Marshall

??? 2020

Abstract

Keywords:

1 Introduction

2 Rust

Rust is a relatively new language created by Mozilla to be memory safe yet have low level like performance [KN19]. Traditionally, memory safe languages will make use of a garbage collector which slows performance [HB05]. Garbage collector languages include C# [RNW+04], Java [GJS96], Python [Mar06], Golang [Tso18] and Javascript [Fla06]. Languages that perform well use manual memory management, which is not memory safe whenever the programmer is not careful. Dangling pointers [CGMN12], memory leaks [Wil92], and double freeing [Sha13] in languages like C and C++ are prime examples of manual memory management problems [Kok18]. Few languages have both memory safety and performance. However, Rust achieves both by using a less popular model known as ownership [MK14].

2.1 Ownership

In the ownership model, the compiler uses statical analysis [RL19] to track which variable owns a piece of heap data – this does not apply to stack data. Each data piece can only be owned by one variable at a time. The owning variable is called the *owner* [KN19].

A variable also has a scope. The scope starts at the variable declaration and ends at the closing curly bracket of the code block containing the variable. When the owner goes out of scope, Rust returns the memory by calling the *drop* method at the end of the scope. Ownership is manifested in two forms – moving and borrowing. These two forms are explained next [KN19].

2.1.1 Moving

Moving happens when one variable is assigned to another. The compiler's analysis moves ownership of the data to the new variable from the initial variable. The initial variable's access is then invalidated [KN19]. An analogy example would be to give a book to a friend. The friend can do anything from annotating to burning the book as they feel fit since the friend is the book's owner.

Listing 1: Example of ownership transfer

```
1 {
2     let s = String::from("string");
3     let t = s;
4
5     println!("String len: {}", s.len()); - borrow of moved value: `s`
6 } // Compiler will 'drop' t here
```

In Listing 1, on line 2, a heap data object is created and assigned to variable *s*. Line 3 assigns *s* to *t*. However, because *s* is a heap object, the compiler transfers ownership of the data from *s* to *t* and marks *s* as invalid.

When trying to use the data on line 5, via s, the compiler throws an error saying s was moved. Any reference to s after line 3 will always give a compiler error.

Finally, the scope of t ends on line 6. Since the compiler can guarantee t is the only variable owning the data, the compiler can free the memory on line 6.

Listing 2: Function taking ownership

```
1  fn main() {
2      let s = String::from("string");
3      take_ownership(s);
4
5      println!("String len is {}", s.len()); - borrow of moved value: `s`
6  }
7
8  fn take_ownership(a: String) {
9      // some code working on a
10 } // Compiler will 'drop' a here
```

Having ownership moving makes excellent memory guarantees within a function; however, it is annoying when calling another function, as seen in Listing 2. The *take_ownership* function takes ownership of the heap data resulting in memory cleanup code correctly being inserted at the end of *take_ownership*'s scope on line 10. When *main* calls *take_ownership*, *a* becomes the new owner of *s*'s data, making the call on line 5 invalid. When taking ownership is not desired, the second form of ownership, borrowing, should be used instead.

2.1.2 Borrowing

Borrowing has a new variable take a reference to data rather than becoming its new owner [KN19]. An analogy is borrowing a book from a friend with a promise of returning the book to its owner once done with it.

Listing 3: Function taking borrow

```
1  fn main() {
2      let s = String::from("string");
3      take_borrow(&s);
4
5      println!("String len is {}", s.len());
6  } // Compiler will 'drop' s here
7
8  fn take_borrow(a: &String) {
9      a.push_str("suffix"); - cannot borrow *a as mutable
10 } // a is borrowed and wiil therefore not be dropped
```

As seen in Listing 3, borrowing makes the function *take_borrow* take a reference to the data. References are activated with an ampersand (&) before the type. Once *take_borrow* has ended, control goes back to *main* - where the

cleanup code will be inserted. Having references as function parameters is called borrowing [KN19]. The ampersand is also used in the call argument on line 3 to signal the called function will borrow the data.

However, in Rust, all variables are immutable by default [KN19]. Hence changing the data in *take_borrow* causes an error stating the borrow is not mutable on line 9. Returning to the borrowed book analogy. One would not make highlights and notes in a book one borrowed unless the owner gave explicit permission.

2.2 Immutable by default

Mutable borrows are an explicit indication that a function/variable is allowed to change the data.

Listing 4: Function taking mutable borrow

```
1  fn main() {
2      let mut s = String::from("string");
3      take_borrow(&mut s);
4
5      println!("String len is {}", s.len());
6  } // Compiler will add memory clean up code for s here
7
8  fn take_borrow(a: &mut String) {
9      a.push_str("suffix");
10 }
```

As seen in Listing 4, line 8, mutable borrows are activated using *&mut* on the type. Again, *mut* is also used in the call on line 3 to make it explicit the function will modify the data. Variables – on the stack or heap – also need to be declared *mut* to use them as mutable [KN19] as seen on line 2.

The two ownership forms - moving and borrowing - together with mutable variables put some constraints on the code for variables and their calls: [KN19]

- Moving will always invalidate the variable.
- Borrowed variables cannot be mutated. However, more than one function can borrow the data simultaneously in parallel and concurrent code.
- Mutable borrowing does allow mutations. But only one function can hold a mutable borrow at a time, and no other immutable borrows can exist.

The constraints will always be enforced by the compiler, thus requiring all code - written by hand or a macro - to meet them. Meeting these constraints also requires some shift in thinking. Another shift is required because Rust may not classify as an Object-Oriented Programming (OOP) language.

2.3 Not quite OOP

No single definition exists to qualify a language as Object-Oriented [Mey97, SB85, GHJV94, KN19]. Three Object-Oriented definitions will be explored to

understand Rust better. These three are Objects as Data and their Behaviour, Encapsulation, and Inheritance.

2.3.1 Objects as Data and their Behaviour

The first definition of Object-Oriented design is a language using objects. An object, in turn, holds both data and procedures operating on the data [Mey97, SB85, GHJV94]. Rust meets the data with their operations requirement by holding the data in *struct*s and having the operations defined in *impl* blocks [KN19].

Struct A *struct* is the same as a *struct* in C [Str13] and other C like languages [RNW+04, WS15, Mal09]. Structs are used to define objects with named data pieces, as shown in Listing 5 lines 1 to 5. Each of the struct properties is named followed by a type.

Listing 5: Example of a Struct

```
1  pub struct Foo {
2      pub name: String,
3      age: u8,
4      gender: Gender,
5  }
6
7  impl Foo {
8      pub fn get_age(&self) -> u8 {
9          self.age
10     }
11
12     pub fn have_birthday(&mut self) {
13         self.age += 1;
14     }
15
16     fn have_burial(self) {
17         unimplemented!();
18     } // `self` will be dropped here
19 }
20
21 fn main() {
22     let mut bar = Foo {
23         name: String::from("bar"),
24         age: 1,
25         gender: Gender::Male,
26     };
27
28     bar.have_birthday();
```

```
29
30      // Party was crazy
31      bar.have_burial();
32
33      println!("Bar was {} years old", bar.get_age()); - borrow of moved
    ↪   value: 'bar'
34  }
35
36  impl Foo {
37      pub fn be_born(name: String) -> Self {
38          Foo {
39              name,
40              age: 0,
41              gender: Gender:Female,
42          }
43      }
44  }
```

Methods The operations to perform on a struct are defined in *impl* blocks, as seen in lines 7 - 19 in Listing 5. Notice how the ownership rules apply to the struct.

The methods *get_age* and *have_birthday* will take a borrow of the struct object. To age, while having a birthday, *have_birthday* needs to take a mutable borrow of *self* - also why *bar* needs to be *mut* in *main*. The method *have_burial* moves *self*, thus invalidating any objects of *Foo* after *have_burial* is called - resulting in a compile error on line 33. The same error happened in Listing 2.

Note

Rust does not always use the *return* keyword. The missing semi-colon on line 9 is deliberate, and the Rust way to say we want to return the age for the Foo instance.

2.3.2 Encapsulation

The next definition deals with hiding implementation details from the client - known as encapsulation [KN19, Mey97]. Encapsulation allows the struct creator to change the internal procedures of the struct without affecting the public interface used by clients. Rust also meets the encapsulation definition by using the *pub* keyword.

pub As can be seen in Listing 5, the *Foo* struct and its *name* data member is made public explicitly. The methods *get_age* and *have_birthday* are also made public explicitly. All the other data members and methods are private - Rust's default - and unreachable by client code. A curious question then is why can *main* access the private variables and methods. The answer being: *main* is located in the same module/file as the struct and therefore has full access to it.

Note

Rust does not provide constructors like other OOP languages. Instead, Rust has what it calls associate methods. An associate method is a method definition not containing *self* in the parameter list [KN19]. A constructor like associate-method will return an owned instance of the struct being constructed. Listing 5, lines 36 to 44 shows an example of an associate method for Foo defined in the same file as Foo's definition. Yes, Rust code can have multiple *impl* blocks for a single struct.

2.3.3 Inheritance

The last definition being looked at is inheritance. Inheritance has an object inherit some of its data members and procedures from a parent object [Mey97, SB85, GHJV94]. Inheritance is mostly used to reduce code duplication. Rust does not meet the inheritance definition for OOP.

The GoF made their design patterns for Object-Oriented languages. They also define two principles for Object-Oriented design [GHJV94].

- "Program to an interface, not an implementation." Rust meets this requirement by using *traits* - discussed next.
- "Favor object composition over class inheritance." Not having inheritance causes Rust code to use composition naturally.

Rust *traits* cause it to meet both requirements allowing Rust to implement the GoF design patterns.

2.4 Traits

Traits are similar [KN19] to *interfaces* in other languages like C# [RNW+04] and Java [GJS96]. In C++, *abstract classes* are the equivalent of *interfaces* [Mal09, Str13, Ale01]. Thus, *traits* allow the definition of abstract behavior as seen in Listing 6, lines 1 to 7.

Listing 6: Working with traits

```
1  trait Show {
2      fn show(&self) -> String;
3
4      fn show_size(&self) -> usize {
5          self.show().chars().count()
6      }
7  }
8
9  fn work<T: Show>(object: T) {
10     println!("{}", object.show());
11 }
12
```

```
13 fn complex<T: Show + Display + PartialEq<U>, U: Display>(first: T,
   ↪  second: U) {
14     if first == second {
15         println!("{}({}) is equals {}", first, first.show(),
   ↪  second);
16     }
17 }
18
19 fn complex_where<T, U>(first: T, second: U)
20 where
21     T: Show + Display + PartialEq<U>,
22     U: Display,
23 {
24     // Same as complex
25 }
26
27 struct Tester {}
28
29 impl Show for Tester {
30     fn show(&self) -> String {
31         String::from("Tester")
32     }
33 }
```

The compiler uses *traits* at compile time to guarantee an object implements a set of methods using *trait bounds*. Line 9 shows the *work* function having a *trait bound* on the generic T type. Thus, any object choosing to implement the *Show* trait can be passed to *work*. In turn, *work* knows it is safe to call *show()* on the object for type T.

More complex trait bounds can be constructed as shown on line 13. Here T (*first*) has to implement the traits: *Show* to be able to call the *show()* method on *first*; *Display* to pass it to *println!*; *PartialEq* to compare it with U (*second*). Trait bounds this complex become hard to read, so Rust offers an alternative syntax for placing trait bounds on generics as seen in lines 19 to 22.

Traits are implemented using the *impl* keyword followed by the trait name as seen on line 29. The *Show* trait has a default implementation for *show_size*. Thus, *Tester* does not need to implement *show_size* and chooses only to implement the *show* method.

Generics and traits are a good match since they result in static dispatch function calls.

2.4.1 Static Dispatch

When Rust sees generics on a function or type, Rust uses what it calls *monomorphization* [KN19]. *Monomorphization* creates a new function or type at compile time for each concrete object passed into the generic placeholders. Thus,

at compile-time, the compiler knows exactly which version of the expanded function to call. Knowing which function to call at compile-time is known as *compile-time binding* [Mal09] or *static dispatch* [KN19, Ale01]. Using just *trait objects* - rather than generics - will result in dynamic dispatch.

2.4.2 Dynamic Dispatch

Dynamic dispatch [Ale01, KN19] happens when the compiler cannot determine which method to call because the *self* object is not fixed. Instead, at run-time, pointers inside the trait objects are used to determine which method to call [KN19], hence why it is also known as *run-time binding* [Mal09].

Listing 7 shows *work* with dynamic dispatch rather than generics.

Listing 7: Dynamic Dispatch with *dyn*

```
1 fn work(object: &dyn Show) {
2     println!("{}", object.show());
3 }
```

Three changes need to be made to the function signature.

- The generic *T* is removed and replaced with *Show*.
- The *dyn* keyword is added to the type to indicate dynamic dispatch will take place explicitly [KN19].
- Borrowing now takes place.

The static dispatch generic trait examples can also be made to take a borrow, but taking ownership gives a compile-time error with dynamic dispatch. The special *Sized* trait is the cause of the error.

2.4.3 Sized trait

Rust keeps all local variables and function arguments on the stack. Having values on the stack requires their size to be known at compile-time. A special trait called *Sized* is used by the compiler to make sure this requirement is met [KN19].

The size of an object is influenced by the data it holds. Also, any object can choose to implement *Show*. Ownership will want to pass each object on the stack, but each object will need a different stack size only known at run-time. However, pointers have a fixed size. Therefore, putting the dynamic object behind any pointer means the stack size will be fixed.

Note

Sized is a unique Rust trait. It is the only trait automatically added as a trait bound on all generics. For this reason, a special syntax is needed when the programmer wants to opt-out of the *Sized* trait bound. The syntax in question is the *?Sized* trait bound [KN19].

Rust offers many pointer options: [KN19]

- A reference - also called a borrow.
- The *Box<T>* used to hold heap objects. *Box* differs from a reference since *Box* owns its *T* [1].
- The *Rc<T>* reference counting pointer that is essentially a run-time immutable borrow. When the reference count reaches zero, the *T* is dropped.
- The special *RefCell<T>* which exposes mutable access behind an immutable object using the *Interior Mutability Pattern*.
- The combined *Rc<RefCell<T>>* - the reference counting pointer allows multiple objects to exist, while the *RefCell* allows mutable access to each existing object.
- *Arc<T>* - the thread-safe version of *Rc*.
- *Mutex<T>* - a thread-safe version of *RefCell*. However, a lock needs to be acquired first.
- *RwLock<T>* - like *Mutex*, but distinguishes between a reader and a writer.
- Finally, *Arc<RwLock<T>>* which will be used by our Abstract Factory. It will hold *T*s for use in multiple threads using *Arc*, while *RwLock* will guarantee only one writer.

One more Rust uniqueness is left to be covered. Rust treats enums differently than other languages.

2.5 Enums

In Rust, enums can also hold objects [KN19] as seen in Listing 8, lines 1 to 3. The *Option* enum, as defined here, is built into the standard Rust library [KN19] to replace *null* as used in other languages. An *Option* can either be *Some* object or *None*. This is yet another design Rust uses to be memory safe [2] by checking the "*null*" (*None*) option is handled at compile-time.

Listing 8: Enums holding objects in Rust

```
1 enum Option<T> {
2     Some(T),
3     None,
4 }
5
6 fn main() {
7     let age = Option::Some(5);
8
```

[1] https://joshleeb.com/blog/rust-traits-trait-objects/

[2] Tony Hoare, the inventor of the *null* reference, has called the *null* reference a billion-dollar mistake in his 2009 presentation "Null References: The Billion Dollar Mistake" (https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare/)

```
9       match age {
10          Option::Some(value) => println!("Age = {}", value),
11          Option::None => println!("Age is unknown"),
12      }
13
14      let valid = if let Option::Some(_) = age {
15          println!("Age is set");
16          true
17      } else {
18          false
19      };
20  }
```

Lines 9 to 12 show the use of *match* - called a *switch* in most languages - to match against each possible enum variant. Line 10 and 11 are each called a *match arm*. Line 10 shows how an object can be extracted on an arm and be assigned to a value variable. If any of the two arms are missing, the compiler will give an error stating not all the enum options are covered. Because matches need to be exhaustive - i.e. all variants need to be covered - in Rust. There are two options for getting around the exhaustive check [KN19].

Either adding the _ (underscore) catch-all arm to handle the default case for all missing enum options. Alternatively, using the *if let* pattern as seen on line 14. The *if let* pattern also allows extraction of the enum object. However, since the object is not used inside the if block, no extraction needs to occur. The _ is, therefore, used to not extract the enum object.

Both *if* and *match* blocks are considered expressions in Rust. Thus, lines 16 and 18 miss their ending semi-colon to return *true* and *false* from the *if*. The value returned from the *if* is assigned to *valid*. Returning from a *match* is quite common, especially with a particular enum used for error handling.

2.5.1 Result enum for error handling

While other languages use exceptions to propagate errors up to the caller, Rust uses the *Result* enum instead. The definition for *Result* can be seen in Listing 9 on lines 1 to 4.

Listing 9: The *Result* enum

```
1  enum Result<T, E> {
2      Ok(T),
3      Err(E),
4  }
5
6  fn may_error() -> Result<i32, String> {
7      Result::Err(String::from("Network down!"))
8  }
9
```

```rust
fn error_explicit_handle() {
    let r = may_error();

    let r = match r {
        Result::Ok(result) => result,
        Result::Err(error) => panic!("Operation failed: {}", error),
    };

    // Use r
}

fn error_short_handle() {
    let r = may_error().expect("Operation failed");

    // Use r
}

fn error_explicit_propogation() -> Result<i32, String> {
    let r = match may_error() {
        Result::Ok(result) => result,
        Result::Err(error) => return Err(error),
    };

    Ok(2)
}

fn error_short_propogation() -> Result<i32, String> {
    let r = may_error()?;

    Ok(2)
}
```

A function will return *Result* to indicate if it was successful with the *Ok* variant holding the successful value of type *T*. In the event of an error, the *Err* variant is returned with the error of type *E* - like *may_error* on line 7. Any calls to *may_error* have to handle the possible error. A few error handling options exist: trying an alternative, panicking, or propagating the error.

Alternative Trying an alternative is quite simple. If the *Err* enum is returned, then just run the alternative instead.

Panicking The caller will use a *match pattern* to extract the error and panic as seen on lines 13 to 16. However, writing matches all the time for possible errors breaks the flow of the code. So the *Result* enum has some helper methods

defined on it [3] - yes, Rust enums are like ordinary objects that can have methods.

The helper method *expect*, as seen on line 22, does the exact same as the match on lines 13 to 16.

Propagation The caller might decide more information is need to panic. So the caller's caller will need to handle the error instead.

Line 30 shows how to propagate the error up the stack - the *return* is to return from the function and not the match. Line 29 uses the result if it is fine - the lack of *return* returns from the match and assigns *result* to *r*. Again, the match is verbose and Rust offers a helper to make it shorter. The helper is the *?* (question mark) operator. The *?* operator can be used in any function returning a *Result* or *Option* - or type implementing the *std::ops::Try* trait [KN19]. Line 37 shows how to use *?* - doing precisely the same as lines 28 to 31.

3 Reporting

4 Conclusion

References

[Ale01] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[CGMN12] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, page 133âĂŞ143, New York, NY, USA, 2012. Association for Computing Machinery.

[Fla06] David Flanagan. *JavaScript: The Definitive Guide*, chapter 11, pages 171–172. OâĂŹReilly Media, Inc., 2006.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[GJS96] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*, chapter 9, 12, pages 199–208, 245. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1996.

[HB05] Matthew Hertz and Emery D. Berger. Quantifying the performance of garbage collection vs. explicit memory management. *SIGPLAN Not.*, 40(10):313âĂŞ326, October 2005.

[KN19] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[3] https://doc.rust-lang.org/std/result/enum.Result.html

[Kok18] Konrad Kokosa. *Pro .NET Memory Management: For Better Code, Performance, and Scalability*, chapter 1, pages 28–34. Apress, USA, 1st edition, 2018.

[Mal09] Davender S Malik. *Data structures using C++*, chapter 1, 3, pages 33, 170. Cengage Learning, 2009.

[Mar06] Alex Martelli. *Python in a Nutshell (In a Nutshell (OâĂŹReilly))*, chapter 13, pages 269–272. OâĂŹReilly Media, Inc., 2006.

[Mey97] Bertrand Meyer. *Object-oriented software construction*, volume 2. Prentice hall Englewood Cliffs, 1997.

[MK14] Nicholas D Matsakis and Felix S Klock. The rust language. *ACM SIGAda Ada Letters*, 34(3):103–104, 2014.

[RL19] Morten Meyer Rasmussen and Nikolaj Lepka. Investigating the benefits of ownership in static information flow security. 2019.

[RNW+04] Simon Robinson, Christian Nagel, Karli Watson, Jay Glynn, and Morgan Skinner. *Professional C#*, chapter 3, 4, 7, pages 101–104, 123–130, 192–193. John Wiley & Sons, Inc., Hoboken, NJ, USA, 2004.

[SB85] Mark Stefik and Daniel G. Bobrow. Object-oriented programming: Themes and variations. *AI Magazine*, 6(4):40, Dec. 1985.

[Sha13] John Sharp. *Microsoft Visual C# 2013 Step by Step*, chapter 14, pages 320–321. Microsoft Press, USA, 1st edition, 2013.

[Str13] B. Stroustrup. *The C++ Programming Language*, chapter 8, 20, pages 205, 598. Addison-Wesley, 2013.

[Tso18] Mihalis Tsoukalos. *Mastering Go: Create Golang production applications using network libraries, concurrency, and advanced Go data structures*, chapter 2, pages 47–49. Packt Publishing Ltd, 2018.

[Wil92] Paul R Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, pages 1–42. Springer, 1992.

[WS15] Kenrich Mock Walter Savitch. *Problem Solving in C++*, chapter 10, pages 572–583. Pearson Education Ltd, Edinburgh Gate, Harlow, England, 2015.