An Exploration of Rust as an Object-Oriented Programming Language

Jonah Gutenberg, Jacob Burton
University of Colorado Boulder
{Jonah.Gutenberg, Jacob.Burton-1}@colorado.edu

Abstract

Object-oriented programming (OOP) is a popular programming paradigm based around concepts such as encapsulation, abstraction, polymorphism, and inheritance (Black 2013). The Rust language, in contrast, is a functional programming language that has gained widespread popularity for its safety and overall usability in a variety of situations (Perkel 2020). In its inception, the language was influenced by functional programming principles like immutability, higher-order functions, algebraic data types, and pattern matching. These factors, among others, have led to a significant rise in Rust's use in general and professional contexts over recent years. Despite Rust being a functional programming language, OOP techniques can be implemented in Rust, and this paper will examine how general OOP patterns and principles adapt into the Rust language. To accomplish this, we will build a version of the Polymorphia game, a simple adventure simulator, in Rust.

Introduction

Since its initial release in 2015, Rust has seen a significant rise in recognition and overall popularity. According to the 2022 Stack Overflow Developer Survey, "Rust has been the most loved programming language for six years in a row (since 2016)." (Bugden & Alahmar, 2022). This is in large part due to the various benefits Rust provides over many other popular programming languages. Rust is often praised for enforcing memory safety and having exceptional performance. According to the same paper, "Rust was found to be the safest programming language compared to C, C++, Java, Go, and Python." (Bugden & Alahmar, 2022). Its advantages go beyond safety though, excelling in other capabilities, such as code readability and complexity. In a study that compared Rust to a set of OOP languages including C++, Python, and JavaScript on a variety of common software metrics, Rust exhibited the lowest cognitive complexity among all languages tested (Ardito, Luca, et al., 2021). Further, in a paper about the systems of Rust, three areas of capabilities were identified: isolation, analysis, and automation. The key capability of isolation is described as enforcing "process-like boundaries around program modules in software, without relying on hardware protection" (Balasubramanian et al., 2017). For analysis, they also describe how Rust removes the complication that usually comes with aliasing data by limiting aliasing, whereas most programs sacrifice precision. So Rust maintains both precision and efficiency. And lastly, automation refers to how security techniques need to checkpoint data, but aliasing makes automating this process complicated due to multiple reference points. They make the point that Rust "adds the checkpointing capability to arbitrary data structures in an efficient and thread-safe way" (Balasubramanian et al., 2017), in part because of their restriction on aliasing.

Methods

To support our argument that Rust can allow for the implementation of OOP principles and design patterns, we built a version of the Polymorphia game in the Rust language. Polymorphia is a turn-based, self-playing adventure game developed in the CSCI 5448 course at CU Boulder that implements several OOP design patterns and focuses on encapsulation, abstraction, and polymorphism. We will demonstrate how Rust's features can be utilized to build an outwardly identical version of the game, named "Rusty Polymorphia," while implementing the same OOP design patterns and adhering to the same OOP principles. Specifically, we will demonstrate how the Factory and Builder patterns can be implemented in Rust. The version of polymorphia that we adapted is the very first one, including just one Creature and one Adventurer, and a simple maze design of just four rooms all interconnected. This is a good base for gaining familiarity with the Rust code, and for being able to adapt the patterns, without being overwhelmed by necessary inputs. The downside to this is that some patterns are not feasible due to a lack of objects, however the Factory and Builder still demonstrate a range of the capabilities that patterns try to access. The program was built using the Visual Studio Code IDE, mainly because it is capable of running a wide range of languages reliably, and due to general familiarity with its interface from previous use. Version control for Rusty Poly-

morphia was managed through a GitHub repository.

Functional Aspects of Rust

Rust is typically thought of as a functional programming language, and its influences were primarily functional. One main point of distinction between Rust and typical OOP languages is that while inheritance is one of the main principles of OOP, Rust does not allow objects to inherit from other objects (Klabnik & Nichols 2019). Rust does, in contrast, contain many functional programming features, including closures, iterators, and pattern matching. Closures are a unique aspect of functional programming languages that are "functions that can capture the enclosing environment" (Klabnik & Nichols 2019). This refers to the ability to call a variable out of the usual scope of the program, by using a special syntax. Normally these variables and their environments cease to exist after use (in a sense), but in Rust there are ways of maintaining them. Iterators are a function that is not inherently functional, but is a trait that is typical of functional programming's linear design. In Rust, this is manifested via iterators, which are applied to lists and used whenever the list needs to be iterated. As described, "iterators are lazy, meaning they have no effect until you call methods that consume the iterator to use it up" (Klabnik & Nichols 2019). By this method, when a list is put into a for loop, if the iterator is already applied it makes iteration convenient. Lastly pattern matching is a function that is very characteristic of functional programming languages, and Rust is no exception. This involves checking the equivalence of patterns and data, to check that not only the values are similar, but the "shape" is as well. Rust does this for a number of types including variables, literals, and patterns. (Klabnik & Nichols 2019).

OOP Principles in Rust

Although the language was primarily influenced by functional programming, Rust shares a number of features that are typical of OOP – namely encapsulation, polymorphism, and object creation. Even inheritance, a feature that the language ostensibly lacks, can be achieved in Rust via default trait method implementations, among other solutions (Klabnik & Nichols 2019). For encapsulation, Rust has two types of visibility: public, designated by the *pub* keyword, and private, which is the default privacy setting without any associated keyword. Items that are public can be accessed from external modules, while private items can only be accessed by the current module and its descendants (`https://doc.rust-lang.org/reference/`).
The OOP principle of polymorphism can also be achieved in Rust. One way of accomplishing polymorphism is by leveraging traits, which are collections of methods defined for an unknown type. Rust allows for two types of function calls, called "static dispatch" and "dynamic dispatch." With dynamic dispatch, trait methods can be called without specifying the object type at compile time, but rather at runtime (`https://softwaremill.com/rust-static-vs-dynamic-dispatch/`). By doing so, the compiler's knowledge of the type is erased; this allows for more reusable, polymorphic code. Further, while Rust does not have an "Object" class, the language does have the infrastructure to create items that "package data and the procedures that operate on that data," which is the typical definition of an "object" in programming as defined by the seminal catalog of OOP patterns, *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (1994). The primary way of creating complex objects in Rust is the *struct* type, which is itself composed of other types. One notable difference between a Rust *struct* and a Java *class* – a typical example of the OOP "object" – is its lack of a constructor. Instead of relying on a constructor to create an object, Rust relies on the creation of an instance by specifying all concrete values for each data field. In addition, Rust does not allow for certain fields to be mutable; either the entire instance and all related data fields are mutable, or no fields are mutable.

OOP Design Patterns in Rust

In addition to allowing for OOP principles such as encapsulation and polymorphism, and despite lacking key features of the OOP paradigm such as object inheritance, Rust does allow for the implementation of OOP design patterns. For example, the creational Factory and Builder patterns can both be implemented. Because Rust is a functional language rather than an object-oriented language, though, some aspects of these patterns must be implemented differently in Rust. While creating Rusty Polymorphia, we noted several differences – both drawbacks and benefits – in how Rust handled these design patterns. In some cases, Rust's features allowed for design patterns to be simplified greatly, while in other instances, design patterns were difficult to implement. These differences are outlined in the following sections.

Factory Pattern

Our Rusty Polymorphia implementation contains two characters: an Adventurer and a Creature. To demonstrate how the Factory Pattern can be executed in Rust, we defined a *CharacterFactory* struct, as well as its implementation. This can be referenced in Figure 1.

By using this method, character creation is decoupled from the main code. This has several benefits. First, character creation is made to be polymorphic, as multiple character types can be created using the same method. This not only allows for Adventurers and Creatures to be created using a single method, but it also can be extended to new character types in future versions of the game. In addition to increased flexibility, the Factory Pattern also increases the code's readability, as code duplication can be eliminated. The Factory

Figure 1: CharacterFactory struct and impl

```rust
pub struct CharacterFactory;

impl CharacterFactory {
    pub fn create_adventurer(name: &str) -> Adventurer {
        Adventurer::new(name)
    }

    pub fn create_creature(name: &str) -> Creature {
        Creature::new(name)
    }
}
```

Pattern approach described above involves static dispatch, in that types are resolved at compile time. However, this pattern could also be implemented using dynamic dispatch. In the case of Rusty Polymorphia, this would involve first defining a *CharacterFactory* trait which returns a dynamic type pointer. Then, concrete implementations of the factory could be defined, for each of the characters in the game. Each implementation's return type would again be dynamic and resolved only at runtime. By using this dynamic dispatch approach, the polymorphism of the code is further enhanced, as character creation can be handled solely with one method and would not require explicit type definitions.

Builder Pattern

The Builder Pattern, another creational design pattern, can also be executed in Rust. The Builder pattern is often utilized in OOP languages to construct objects step-by-step, and it allows for different representations of the same object to be created using the same construction code (`https://refactoring.guru/design-patterns/builder`). To accomplish this in our Rusty Polymorphia game, a *Maze* struct was first constructed. Inside its implementation, a *builder* function was created:

Figure 2: Maze::builder function

```rust
    pub fn builder() -> MazeBuilder {
        MazeBuilder::new()
    }
```

This function is used to create the Maze, and it achieves this by returning a *MazeBuilder*. The MazeBuilder, accordingly, contains functions that initialize and set up the maze. Its functions take in as input the number of rooms in a maze, the Adventurer and Creature in the maze, and the cost for an Adventurer to move throughout the maze, and each function returns the maze itself. Finally and most crucially, the MazeBuilder contains a *build* function that builds and returns the maze. The build method itself creates a maze in the starting state, placing the characters in random rooms. One thing that we had to be considerate of during this was the '.clone()' method in Rust, which is useful for changing an object's (struct's) components, but makes it easy to disconnect from itself. When the maze is ready to be built in the main code, all parameters are initialized using the MazeBuilder and its associated functions:

Figure 3: MazeBuilder::build function

```rust
    let mut maze: Maze = Maze::builder() MazeBuilder
    .num_rooms(4) MazeBuilder
    .moving_cost(0.25) MazeBuilder
    .add_adventurer(adventurer) MazeBuilder
    .add_creature(creature) MazeBuilder
    .build();
```

This is very similar to the structure we used in the Java, and if we were to do this with a later version of plymorphia, more inputs would certainly be required. Using the Builder Pattern in this way allows for better extensibility, in that if requirements were changed and the maze needed additional parameters, a function could simply be added to the MazeBuilder. In the case that the maze constructor required a different number of inputs to fulfill various uses, only one initialization function, *MazeBuilder::build*, is required. This eliminates the telescoping constructor antipattern, in which many different constructors are required in order to satisfy various combinations of inputs.

Other Design Patterns

In Rust, if you really want to, theoretically there are ways to implement just about every design pattern. The resource (`https://refactoring.guru/design-patterns`), which was a great source of inspiration throughout this project, implies that there is always some method of implementing these common patterns not just in Rust, but in most programming languages period. That's why in addition to creational patterns, things like behavioral and structural design patterns can also be achieved in Rust. One example of a behavioral pattern that can adapt well into Rust is the State Pattern. The State Pattern allows an object to change its behavior when its internal state changes. A real world example of this is the power off button on a cellphone, which can change function depending on the phone's current state (turning it on/off, confirming payments, etc). This pattern typically relies on both objects and inheritance, but a Rust implementation can make use of traits and structs instead (Klabnik & Nichols 2019). In this case the structs act similarly to an object, while the traits are interchangeable between states, and easy to be molded to fit the different internal states. Other OOP design patterns may be executed in Rust as well, including the Command Pattern, using either function pointers or trait objects; the Composite Pattern, using structs and implementations; and the Observer Pattern, which can be implemented in Rust by defining subscribers as functions that are callable objects. Despite the possibilities that we've mentioned, not all design patterns, however, can be seamlessly adapted into Rust, nor should they. For example, the Strategy Pattern, which presents behaviors as objects that can be set or changed at runtime, is largely

made obsolete in Rust due to the existence of traits. Like a foreign language, you can translate something literally, or you can dig deeper and bring out the true intention of a phrase. In this way, Rust has places where its own built in systems are more effective than the OOP patterns we learned in class, and to force a pattern could be sacrificing brevity and efficiency.

Results and Findings

Throughout the process of building Rusty Polymorphia, we observed many different ways in which Rust's design interacts with OOP principles and strategies. Overall, we noticed that many of the principles of OOP, such as abstraction and encapsulation, translated to Rust. While Rust does not have explicit Objects like a typical OOP language such as Java does, Rust allows for similar behavior by making use of structs and implementations. This makes it mostly seamless to integrate creational design patterns such as the Factory Pattern and the Builder Pattern. However, it was noticeable while building our game implementation that inheritance is not easily translatable to Rust. In an OOP language like Java, the Factory Pattern typically returns an abstract object in order to allow for polymorphic code. In Rust, on the other hand, this is not so simple. While it is possible to return a dynamic type in a function, this comes with its drawbacks: this dynamic dispatch strategy prevents the compiler from inlining the code, and it also entails an additional runtime cost. In our implementation, described under the Factory Pattern heading above, we opted to create two separate creation functions, one for each character in the game. We chose to do this in order to prevent the aforementioned inlining and runtime drawbacks. In making this design choice, we were also able to limit the complexity of code and improve its readability.

Conclusion

In this paper, we have discussed how object-oriented programming (OOP) techniques can be applied in the Rust language, despite the fact that Rust is typically thought of as a functional programming language. Motivated by Rust's continued increase in popularity and use, we developed a turn-based adventure simulation game, "Rusty Polymorphia," in which we implemented design patterns and principles based on the OOP paradigm. We discussed the various similarities and differences between Rust and typical OOP languages, and we made note of our workarounds and adaptations that allowed for OOP patterns to be implemented in Rust. Our aim is that the findings in this paper can provide insight for prospective developers who are deciding on a programming language for their project. Future work can be done to look into how other aspects of OOP can be implemented using Rust, such as additional design patterns. This paper primarily focused on creational patterns such as the Factory and Builder patterns, but further research into how behavioral and structural design patterns are handled by Rust may yield valuable insights.

References

Bugden, W., & Alahmar, A. (2022). Rust: The Programming Language for Safety and Performance. 2nd International Graduate Studies Congress (IGSCONG'22). https://doi.org/10.48550/arXiv.2206.05503

Perkel, Jeffrey M. "Why Scientists Are Turning to Rust." Nature, vol. 588, no. 7836, 1 Dec. 2020, pp. 185–186 https://doi.org/10.1038/d41586-020-03382-2.

Klabnik, S., & Nichols, C. (2019). The Rust Programming Language (Covers Rust 2018). No Starch Press.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: elements of reusable Object-Oriented software.

Ardito, Luca, et al. "Evaluation of Rust Code Verbosity, Understandability and Complexity." PeerJ Computer Science, vol. 7, 26 Feb. 2021, p. e406, peerj.com/articles/cs-406.pdf, https://doi.org/10.7717/peerj-cs.406.

Balasubramanian, Abhiram, et al. "System programming in rust." ACM SIGOPS Operating Systems Review, vol. 51, no. 1, 11 Sept. 2017, pp. 94–99, https://doi.org/10.1145/3139645.3139660.

Grajek, Krzysztof. "Rust Static vs. Dynamic Dispatch." SoftwareMill, SoftwareMill, 7 Nov. 2025, softwaremill.com/rust-static-vs-dynamic-dispatch/.

S, Klabnik, and Nichols C. Rust Documentation, doc.rust-lang.org/stable/. Accessed 8 Nov. 2025.

Black, Andrew P. "Object-oriented programming: Some history, and challenges for the next fifty years." Information and Computation, vol. 231, Oct. 2013, pp. 3–20, https://doi.org/10.1016/j.ic.2013.08.002.

"Design Patterns in Rust." Refactoring.Guru, 2014, refactoring.guru/design-patterns/rust.