The effect of Rust's abstraction langauge constructs on traditional object oriented design patterns

Viktor Holmgren
Linköping University
Linköping, Sweden
Email: vikho394@student.liu.se

***Abstract*—In this paper the implementation of the Adapter, Template method and Builder design patterns in Rust is evaluated with a focus on maintainability. More specifically this apper seeks to find how the Rust language, which has no concept of classes nor inheritance, affects the implementation of traditinal object oriented design patterns.**

I. Introduction

As software systems grows, so does the need for appropriate design which allows for easy maintenance of existing code, and the flexibility to extend the code base with new features. Over the years similar problems have been met by different people, and different solutions have been implemented to varying degrees of success. Those solutions which have have stuck around and have been codified, are often referred to as, *design patterns*. The result of a design pattern implementation depends on a number of different factors, not least the language features which are used for its implementation.

The most common types of design patterns revolve around the Object Oriented Programming (*OOP*). That is not surprising since the OOP paradigm sees extremely wide spread usage, with many of the most popular languages around use it as their main paradigm[INSERT CITE ON LANGAUGE USAGE]. One question one may ask one self is how the gain of using these patterns is impacted when implemented in a language which lacks any traditional OOP language constructs, such as inheritance or even classes? Is it simply the case that these languages provide similar functionality only with a different facade? Does the different language constructs result in a somewhat different structure and behavior but with the same result? Or does the pattern as such have no real equivalent, perhaps that language removes the need for some patterns entirely?

This paper will examine how the implementation of three different design patterns in Rust compares to a standard implementation in Java [CITE AND NOTE WHICH STANDARD IMPLEMENTATIONS WILL BE USED]. The three patterns chosen for examination are Adapter, due to the fact that Rust allo ws for implementation of interfaces for exisitng types, Template method, since Rust has no concept of class inheritance, and Builder, because Rust does not support overloading.

The focus in the examination will be on the maintainability of the source code as measured by a number of different software quality metrics: Design size in Classes (*DCS*), Depth of Inheritance tree (*DIT*) and Coupling between Objects (*CBO*) and a general evaluation on the impact of the langauge constructs used. As Rust is not traditionally a OOP language some interpretations will have to be done to translate them appropriately, see **??** for details.

II. Background

A. Terminology

B. Software Metrics

To make it possible to evaluate the implementations in Rust using the object oriented software metrics described we need to define the equivalent of a class, and inheritance in Rust since those concepts are not modelled directly.

A class as defined by **?**] is a construct which "...specifies the object's internal data and representation and defines the operations the object can perform." In Rust this would be equivalent to a struct or enum and all implementations for that data, i.e all implementation blocks. This would include default implementations for the concrete type as well as all Trait implementations.

Rust does not have the traditional notion of class based inheritance. It does however support inheritance for Traits which fill a very similar role to a interface in most langauges. Therefor hierarchies can be modeled, but only leaves in this hierarchy tree can be concrete types that can instansiated.

C. Adapter Pattern

D. Template method Pattern

E. Builder Pattern

F. Rust

The Rust programming language is a relatively new language, only started in 2006 by Gradon Hoare. Since then the project has been primarily backed by the Mozilla Foundation [**?**]. Rust is a system level programming aimed at performance and critical systems. It's main feature it boosts is that it guarantees memory safety without overhead, even over multiple threads [**?**]. Otherwise the

language tries to solve problems where languages like C++ currently are most common. [**?**]

1) Traits: Traits is the main way to express abstractions in Rust, and on the surface they are very similar to what would be called an *Interface* in most other languages. Similar to interfaces they allow the programmer to declare methods with arguments and return values for yet to be implemented types. Those types when they are later created must implement methods with the given signature to satisfy the interface. There are however some key differences between traits and interfaces [**?**]:

- Traits can not only require methods to be implemented, but can also provide default implementations. This allows Traits to fill a role similar to abstract classes.
- Traits can not only be implemented for new user level types, but also for existing types as the Trait implementation is completely separate from the type declaration.
- Traits can both be statically and dynamically dispatched. Static dispatch is done by the use of generics which results in zero abstraction overhead. Dynamic dispatch is done by the use of unsized references, i.e a *Box*. This allows dispatch at run-time, good for when indirection is really needed. See **??** for details on how this is done.

2) Polymorphism: Polymorphism is the language construct which allows for a single interface to represent multiple different types [INSERT CITE]. In Rust polymorphism is achived using Traits, and it comes both in form of static dispatch and dynamic dispatch.

For static dispatch Rust uses genrics which are very similar to the way C++ templates work, i.e the compiler will instanciate multiple versions of the function for each unique type which implement the Trait [INSERT CITE on Templates C++ and Rust generics].

Dynamic dispatch is a bit more complex since Rust generally need to know the exact size of the argument a function/method is called with and the size of a Trait implementation is unknown [INSERT CITE for Box is needed]. It is therefor neccessary to use the Box type, which is basically a pointer, as the argument. See **??** for an example on the synatx for both static and dynamic dispatch.

Fig. 1. Static and dynamic dispatch in Rust

III. Implementation

TODO: The implementation goes here.

A. Adapter implementation

Fig. 2. Adapter implementation in Rust

Fig. 3. Template method implementation in Rust

B. Template method implementation

C. Builder implementation

Fig. 4. Builder implementation in Rust

D. Rust

IV. Results

TODO: The results goes here.

V. Discussion

TODO: The discussion goes here.

VI. Conclusion

TODO: The conclusion goes here.