How does the implementation of the Adapter pattern affect maintainability as measured by DCS, DIT and CBO?

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

***Abstract*—In this paper the implementation of the Adapter design pattern in two different languages: Rust and Java is compared with a focus on maintainability. More specifically this paper seeks to find how the Trait paradigm in Rust, which has no concept of classes nor inheritance, affects the implementation of a traditional object oriented design pattern. In this paper three major conclusions is drawn from the results. First, that the different implementations only differ marginally in terms of the software metrics used. Second, that the language constructs in Rust, specifically *Traits*, has some decisive advantages to the traditional object oriented model in Java. Third, that traditional software metrics as used here are problematic when comparing languages which differ considerably in the paradigms they support.**

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 paradigm (*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, only differing in the details. 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 the Adapter design pattern is affected by being implemented in the Rust by comparing it to Java. The focus in the comparison 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*). As Rust is not traditionally a OOP language some interpretations will have to be done to translate them appropriately. See Section II-A for details.

II. Background

A. Software Metrics

B. Adapter Design Pattern

C. Rust

The Rust programming language is a relatively new language. It was started back in 2006 by Gradon Hoare. Since then the project has been primarily backed by the Mozilla Foundation [1]. Rust is a system level programming aimed at performance critical systems. It's main feature it boosts is that it guarantees memory safety without overhead, even over multiple threads [2]. 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 [3]:

- 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 Section II-C2 for details on how this is done.

See Figure 1 for an example of Traits in Rust.

```
trait Animal {
    // Instance method signatures; to be
       implemented by concrete types.
    fn noise(&self) -> &'static str;

    // Traits can provide default method
       definitions.
    fn what_does_it_say(&self) {
        println!("I go {}!", self.noise()
            );
    }
}

// New struct/type
struct Sheep {}

// Implement concrete methods on Sheep
   type
impl Sheep {
    fn name(&self) -> &'static str {
        "Dolly"
    }
}

impl Animal for Sheep {
    // Concrete implementation of noise
    fn noise(&self) -> &'static str {
        "baaaaah!"
    }
}
```

Fig. 1. Trait syntax in Rust

2) Polymorphism:

III. Implementation

TODO: The implementation goes here.

A. Java

B. Rust

IV. Results

TODO: The results goes here.

V. Discussion

TODO: The discussion goes here.

VI. Conclusion

TODO: The conclusion goes here.

References

[1] Rust-lang.org. Faq - the rust project. [Online]. Available: https://www.rust-lang.org/en-US/faq.html

[2] N. D. Matsakis and F. S. Klock, II, "The rust language," *Ada Lett.*, vol. 34, no. 3, pp. 103–104, Oct. 2014. [Online]. Available: http://doi.acm.org.e.bibl.liu.se/10.1145/2692956.2663188

[3] R. B. A. Turon. (2015) Abstraction without overhead: traits in rust. [Online]. Available: https://blog.rust-lang.org/2015/05/11/traits.html