The Effect of Rust's Abstraction Language Constructs on Traditional Object Oriented Design Patterns

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

***Abstract*—In this paper the effects of implementing traditional object oriented design patterns in Rust are evaluated with a focus on maintainability. More specifically, this paper seeks to find how the Rust language, which has no concept of classes nor inheritance, affect the implementation of the Adapter, Template method and Builder patterns as measured by a set of software quality metrics. The results are three fold: First, that the implementations only differ marginally compared to those written in the traditional object oriented language Java in terms of the software metrics used. Second, that the abstraction language constructs in Rust, specifically *Traits*, have some decisive advantages and disadvantages over traditional object oriented constructs. Third, that the software metrics as used in this paper, are problematic when comparing languages which differ considerably in their fundamental design and paradigm support.**

I. Introduction

As software systems grows, so does the need for appropriate design which allows for easy handling 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 actual effect of design patterns on maintainability of software are still unclear according to Zhang and Budgen [1] as it depends on a number of different factors. One factor which should not be overlooked is the language features which are used in implementation.

Many design patterns revolve around the Object Oriented Programming (*OOP*) paradigm. That is not surprising since the OOP paradigm sees extremely wide spread usage, with many of the most popular languages around using it as their main paradigm. In fact, four of the five most popular programming languages in 2017 support OOP in the traditional sense, with classes and class based inheritance [2].

As new languages emerge there is a interest for transferring knowledge, experience and even design patterns from existing languages to new ones. Therefor one might ask one self how the utility of design patterns is impacted when they are 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 language features but with different terminology? Can the different language constructs be combined to achieve an equivalent result? Does entire new patterns emerge from the specific problems faced in the language? Or does some patterns have no real equivalent, are there language features which remove the need for them entirely?

This paper seeks to answer how the Adapter, Template method, and Builder design patterns are affected by being implemented in the programming language Rust as measured by the software quality metrics: Design Size in Classes and Interfaces (*DSCI*), Depth of Inheritance Tree (*DIT*) and Coupling Between Objects (*CBO*). The Java programming language will serve as a base line for comparison of the implementations, as well as a basis for a general discussion on the maintainability effects of the language constructs available in Rust. The reason behind choosing Adapter, Template method and Builder respectively were that: First, Rust allows for implementation of interfaces for existing types. Second, Rust has no concept of class based inheritance. Third, Rust does not support function or method overloading.

II. Background

A. Software Metrics

To make it possible to evaluate the implementations in Rust using the object oriented software metrics described, the equivalent of a class and inheritance in Rust needs to be defined since those concepts are not modeled directly.

A class as defined by Gamma et al. [3] 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, sometimes called trait bounds, which fill a very similar role

to an interface in most languages. They allow for a trait to specify that another trait must also be implemented. Therefor hierarchies can be modeled, but only leaves in a hierarchy tree can be concrete types that can instantiated. For this paper inheritance of default methods from a trait will be considered equivalent to class inheritance as it is the closest to class based inheritance that is possible to achieve. This does not make a trait with default methods a class however, since it neither specifes any internal data nor defines operations, it only specifies them.

The three software quality metrics that will be used in this paper are:

- *Design Size in Classes and Interfaces* (DSCI): which is the total number of classes and interfaces present in the design. Lower values are generally preferred since larger software size often implies increased effort when maintaining the software [4].
- *Depth of Inheritance Tree* (DIT): the level number for a class in an inheritance hierarchy. Base classes are defined to have a DIT of 0. Lower values are preferred, since deeper trees implies greater design complexity. Furthermore, classes deep in class hierarchies will likely inherit more methods thus making their behavior more unpredictable for the reader [5].
- *Coupling Between Objects* (CBO): is the number of other classes to which a class is coupled. High coupling between classes is unwanted, and in turn also high CBO values, because it works against modular design and reuse of code. Also, a high CBO value would indicate that the class is sensitive to changes made in related classes, resulting in reduced maintainability [5].

The DSCI value is calculated on the entire implementation, whereas DIT and CBO are calculated on class level. As implementation wide values are needed, averages over all classes will be used for DIT and CBO.

B. Adapter Pattern

The Adapter design pattern as defined by Gamma et al. [3] is a pattern which seeks to solve the problem of converting the interface of one or more classes into another interface which a client expects. It can be seen as wrapping one or more existing classes with a new interface. One common use case for this pattern is when there is a need to reuse some type of legacy component but that component does not satisfy the interface needed.

The very simple example which will be considered in this report is the following. Suppose there is a legacy class *Rectangle* which can be displayed using coordinates for the upper left corner, width and height, but the client wishes to handle a Rectangle as an instance of Shape, where Shape is an interface containing a more general display method which instead takes coordinates for the upper left corner and the bottom right. See Figure 1 for an UML diagram of the Adapter pattern applied to the example.

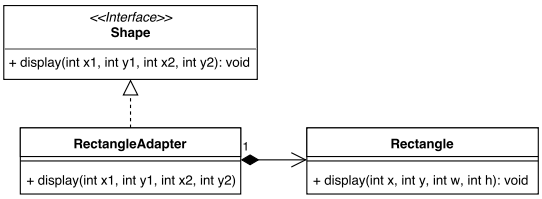

Fig. 1. UML diagram over the Adapter example used

C. Template method Pattern

In the Template method pattern one defines a skeleton of an algorithm, but deferring some steps to subclasses. It solves a similar problem as the Strategy pattern, although Strategy relies on composition rather than inheritance. The primary use case for the Template method pattern is to reduce duplicated code between two or more classes that have much in common in terms of source code, but differ in some details [3].

Consider the following example in which there exists several different animal classes: Cat, Sheep, et cetera, which all have a unique name and make a unique noise. Suppose that all these animals share a *what_does_it_say* method which prints the animal's name and the noise they make. The client does not want to have to care of the exact animal on which it calles the *what_does_it_say* method. For simplification, only the Sheep animal will be considered. See Figure 2 for an UML diagram of Template method applied to the example.

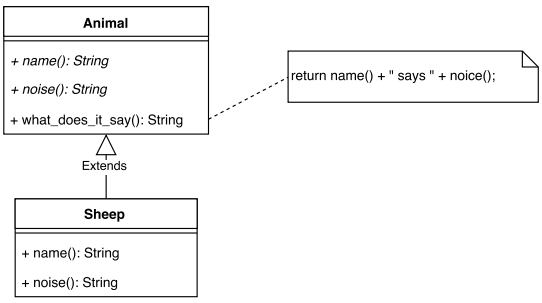

Fig. 2. UML diagram over the Template method example used

D. Builder Pattern

The Builder design pattern's primary intent is to reuse the same construction process to create multiple different representations by separating the construction of a complex object from its representation. It can for example be used to avoid a rapidly expanding set of constructors on a complex object. By separating the creation to separate class, a Builder, from the object itself the complexity of the original class is reduced [3].

The example which will be used in this paper is the following. Suppose there exists some Process class which is rather complex, especially its constructor. Furthermore, the client wishes to use some simplifyed method for creating Processes with many different values. For an UML

diagram depicting the Builder pattern applied to the example, see Figure 3.

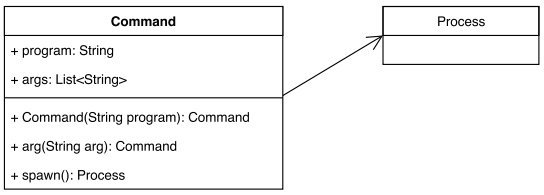

Fig. 3. UML diagram over the Builder example used

E. 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 [6]. Rust is a system level programming crafted for performance and critical systems. One of its primary features it is that it guarantees memory safety without any run-time overhead, even over multiple threads [7], [8]. Otherwise the language tries to solve problems where languages like C++ currently are most common.

Traits are 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 specify methods with arguments and return values for yet to be implemented types. When those types are later created, they must implement all methods with the given signatures in order to satisfy the interface. There are however some key differences between traits and typical interfaces in other languages [9]:

- 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. Note that Java added support for default methods in interfaces as well in version 8 [10].
- Traits can not only be implemented for new user level types, but also for existing types as trait implementation is completely separate from the type declaration.
- Traits can require constants to be specified as well as methods.
- Traits can both be statically and dynamically dispatched.

Polymorphism is the language construct which allows for a single interface to represent multiple different types [11]. In Rust polymorphism is achieved using traits, and it comes both in form of static dispatch and dynamic dispatch.

For static dispatch Rust uses generics which are very similar to the way C++ templates work, i.e the compiler will instantiate multiple versions of the function for each unique type satisfying the generic bounds. This means that generics results in zero abstraction overhead.

There is also support for dynamic dispatch, i.e dispatch at run-time when indirection is really needed, for instance when operating on a list of elements of one trait type but of varying concrete types. Generics would not work in this case since the concrete types are different. It is also not possible to pass traits as values to methods as the compiler generally need to know the exact size of parameter list for functions/methods and the size of a trait is unknown. To enable dynamic dispatch values must either be *Boxed*, which is done by wrapping the value in an unsized reference called a *Box*, or by passing the value as a reference [9]. See Figure 4 for an example on traits in Rust.

```
trait Animal {
    /// Methods to be implemented
    fn name(&self) -> &'static str;
    fn noise(&self) -> &'static str;
}

// All FlyingAnimals must implement Animal,
↪  inheriting of traits
trait FlyingAnimal: Animal {};

/// Static dispatch: generic function, compiler
↪  will instantiate concrete functions for each
↪  caller, e.g. Sheep, Cat etc
fn what_does_it_say<T: Animal>(animal: &T) {
    println!("{} goes {}", animal.name(),
↪  animal.noise());
}
/// Dynamic dispatch: Use Box pointer reference
fn what_does_it_say2(animal: &Box<Animal>) {
    println!("{} goes {}", animal.name(),
↪  animal.noise());
}
```

Fig. 4. Trait examples in Rust

III. Results

The *Adapter* implementation for the example used in Section II-B in the Rust programming language can be seen in Figure 5. Here the implementation uses the fact that Rust allows for the implementation of traits for existing types, meaning that the *Shape* trait can be implemented directly onto the *Rectangle* type after it is defined. In contrast, our Java implmenetaion requires a wrapper class to be used, *RectangleAdaper*, which implements the *Shape* interface and also contains a *Rectangle* instance to which it forwards calls to.

The Java implementation, see Figure 8, resembles the UML diagram presented in Section II-B very closely, unlike the Rust implementation which forgoes the need for an actual adapter class entirely. As such the Rust introduced one new interface (trait) but no new classes unlike Java which introduced both a new interface, and a new class. The use of an internal reference needed in the adapter class to forward requests in the Java implementation results in a

higher a CBO value of 1 instead of 0. The derived software metrics for the implementations can be seen in Table I.

TABLE I
SOFTWARE METRICS FOR ADAPTER IMPLEMENTATION

	DSCI	DIT	CBO
Rust	2	0	0
Java	3	0	1

For the *Template method* implementation in Rust, seen in Figure 6, Rust's notion of default methods in traits are used to establish the shared behavior between different concrete Animals. The other methods, *name* and *noise*, are not given any default implementation and therefor has to be implemented by the concrete types such as *Sheep*. In Java, see Figure 9, an abstract class is used to represent the Animal trait, letting the *name* and *noise* be abstract methods to be implemented by *Sheep* which is a concrete class extending *Animal*. The metrics calculated from the implementations can be seen in Table II. Note that the Rust implementation has a higher DIT value since the trait having the default implementation does not count as a class, i.e only the leaves in the hierarchy gets counted.

TABLE II
SOFTWARE METRICS FOR TEMPLATE METHOD IMPLEMENTATION

	DSCI	DIT	CBO
Rust	2	1	1
Java	2	0.5	1

In the *Builder* implementation in Rust, seen in Figure 7, there are two concrete types: *Command* and *Process*. Using the *new* method instantiates a new *Command* and sets the default state needed for creating a *Process* instance. Then using the *arg* method the internal state can be altered. Note that the *arg* method returns a mutable *Command* reference, allowing for chaining of calls together like $Command :: new("ls").arg("l").spawn()$. In the Java implementation, Figure 8, there are also two concrete classes working in an almost identical way, returning *this* in *arg* also allows for chaining. From the implementations the metric values seen in Table III are calculated, and both implementations are indentical in terms of those metrics.

TABLE III
SOFTWARE METRICS FOR BUILDER IMPLEMENTATION

	DSCI	DIT	CBO
Rust	2	0	1
Java	2	0	1

IV. DISCUSSION

From the results above we see that the implementations in Rust and Java are quite similar which is also reflected in the metrics. Therefor there is lettle general support from the metrics in the idea of change of maintainability of any of the solutions apart from the adapter implementation

```rust
/// Legacy component to be adapted
pub struct Rectangle;
impl Rectangle {
    fn display(&self, x: i32, y: i32, w: i32, h:
↪   i32) {
        println!("Show rectangle at {},{} with
↪   dimensions: {}, {}", x, y, w, h);
    }
}

/// User facing interface to be used
trait Shape {
    fn display(&self, x1: i32, y1: i32, x2: i32,
↪   y2: i32);
}

/// Append trait implementation
impl Shape for Rectangle {
    fn display(&self, x1: i32, y1: i32, x2: i32,
↪   y2: i32) {
        Rectangle::display(self, x1, y1, x2-x1,
↪   y2-y1);
    }
}
```

Fig. 5. Adapter implementation in Rust

```rust
trait Animal {
    /// Methods deferred to subtypes
    fn name(&self) -> &'static str;
    fn noise(&self) -> &'static str;

    // Trait default implementation, algorithm
↪   skeleton
    fn what_does_it_say(&self) {
        println!("{} goes {}", self.name(),
↪   self.noise());
    }
}

/// "Subtype" Sheep, implements the Animal trait
struct Sheep;
impl Animal for Sheep {
    fn name(&self) -> &'static str {
        "Sheep"
    }

    fn noise(&self) -> &'static str {
        "baaaaah!"
    }
}
```

Fig. 6. Template method implementation in Rust

```rust
/// Builder to create a Process instance
pub struct Command {
    program: &str,
    args: Vec<&str>,
}

impl Command {
    /// Constructor, set default values
    pub fn new(program: &str) -> Command {
        Command {
            program: program,
            args: Vec::new()
        }
    }

    /// Add an argument to pass to the program.
    pub fn arg<'a>(&'a mut self, arg: &str) ->
↪   &'a mut Command {
        self.args.push(arg);
        self
    }

    /// Executes the command as a child process,
↪   which is returned.
    /// Actually builds the instance
    pub fn spawn(&self) -> IoResult<Process> {
        ...
    }
}
```

Fig. 7. Builder implementation in Rust

```java
interface Shape {
    void display(int x1, int y1, int x2, int
    ↪  y2);
}

class Rectangle {
    public void draw(int x, int y, int w, int h)
    ↪  {
        System.out.println("Show rectangle at "
        ↪  + x + ", " + y + " with dimensions:
        ↪  " + w + ", " + h);
    }
}

class RectangleAdapter implements Shape {
    private Rectangle adaptee;

    public RectangleAdapter() {
        this.adaptee = new Rectangle();
    }

    @Override
    public void draw(int x1, int y1, int x2, int
    ↪  y2) {
        adaptee.display(x1, y1, x2-x1, y2-y1);
    }
}
```

Fig. 8. Adapter implementation in Java

which is quite different. There are however some key points to be discussed.

From the Adapter pattern example we see that Rust's support for implementation of traits for existing types eliminates the need for the pattern entirely. By having trait behavior for a type be separate from the data of that type we are able to extend the functionality of existing types without effecting the existing code base. This feature allows for great maintainability out of the box since the *Open-Closed* principle is basically enforced by the langauge [3]. By allowing extention of behavior in this way we also remove the need for converting between adapters and adaptees in any mediating source code between the client, which wishes the use the adapter, and the rest of the code base, which may use the legacy component. In Java however we would practically need to introduce a facade between the existing code base and the client which translates input of legacy instances into adaptered instances and output of adaptered instances into legacy instances. This in turn has a great effect of maintainability since the cost of reducing existing components in Rust is minimized.

But suppose we had N number of legacy components

```java
abstract class Animal {
    public abstract String name();
    public abstract String noise();

    public void what_does_it_say() {
        System.out.println(this.name() + " goes
        ↪  " + this.noise());
    }
}

// Sheep subclass, extends Animal
class Sheep extends Animal {
    public String name() {
        return "Sheep";
    }

    public String noise() {
        return "baaaaah!";
    }
}
```

Fig. 9. Template method implementation in Java

```
class Command {
    private String program;
    private List<String> args;

    public Command(String program) {
        this.program = program;
        this.args = new ArrayList<String>();
    }

    // Add an argument to pass to the program.
    public Command arg(String arg) {
        this.args.add(arg);
        return this;
    }

    // Executes the command as a child process,
    ↪  which is returned.
    // Actually builds the instance
    public Process spawn() {
        ...
    }

}
```

Fig. 10. Builder implementation in Java

which we wanted to implement the *Shape* for. In Java we would have introduced N new adapter classes. Where as in Rust we would have added zero new classes according to the definition given in Section II-A, i.e the size of introducing N adapter patterns would be constant. This strange result seems to point to a flaw in the definition of a class in Rust as given in this paper. However there are only two possible options in defining a class, both yielding unreasonable results:

- The type and all implementations counts as one class.
- The type and default implementation counts as one class, each trait implementation constitutes an additional class.

The first definition is what is used in this paper. By the second definition common types would make up a very large number of classes, for example the standard vector collection *vec* in Rust would constitute over a hundred classes [12]. This I believe points to a more general problem of using specifically object oriented metrics for evaluating languages which are not traditionally object oriented. I see two possible solutions to this, either to use non object oriented metrics such as number of lines of code (LOC) with the downside of not being able to reason about higher level constructs and for instance coupling easily, or to only use object oriented metrics for which no interpretations would have to be made. This could however greatly reduce the number of available metrics when comparing certain languages. We see a similar problem in the DIT since it in Rust, using the definition of class, only counts leaves as classes but includes traits as part of the hierarchy, unlike in Java where every node in the hierarchy is used.

Regarding the template method implementations we see strikingly similar solutions. Traits with default implementations fills a role similar to that of an abstract class in Java and many other languages. However, let there be known that it is not equivalent to an abstract class since a Rust trait cannot hold any state. Therefor it becomes difficult to replicate the behavior of a more complex case of the template method, specifically when the algorithm skeleton and surrounding methods rely on state between method calls. In this case one should probably look to the similar *Strategy* pattern instead when developing in Rust, since it does not require any class based inheritance. Furthermore one should probably prefer strategy generally over template method any way since it works based on composition rather than inheritance [3, p.32].

Finally, the builder pattern is practically identical between the two languages, disregarding syntax differences. Because Rust does not support function/method overloading the builder pattern becomes very useful. Not having support for multiple constructors means that a programmer in Rust would have to create and maintain a large number of uniquely named methods that constitute different constructors, leading to poor maintainability. Where as with a builder there is a separation of the construction and the behavior of a type thus resolving the programmer for creating variants of the ”new” method and instead only adding methods when new parameters to the actual constructor is added. It is important to note however that overloading can in some sense be achieved through trait implementations since we can over different trait implementations name our methods identically, even with the same parameter list. Using traits does of course imply that those traits need to be created also, so it is not exactly equivalent. Still, the builder is very useful, so much in fact that it is used multiple times in the Rust standard library and the example used in this paper is actually a simplification the *Command* builder in the Rust standard library [13]. There even exists Rust libraries that are able to automatically derive, i.e let the compiler generate source code, builder patterns for arbitrary types [14].

References

[1] C. Zhang and D. Budgen, "What do we know about the effectiveness of software design patterns?" *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1213–1231, 2012.

[2] S. Cass. (2017) The 2017 top programming languages. [Online]. Available: https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: Elements of reusable object-oriented software*. Boston, Massachusetts: Addison-Wesley, 1995.

[4] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2009, pp. 367–377.

[5] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994. [Online]. Available: http://dx.doi.org/10.1109/32.295895

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

[7] 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

[8] E. Reed, "Patina: A formalization of the rust programming language," *Tech. Rep. UW-CSE-15–03-02*, 2015.

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

[10] Oracle. (2017) Default methods. [Online]. Available: https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

[11] B. Stroustrup. (2017) Bjarne stroustrup's c++ glossary. [Online]. Available: http://www.stroustrup.com/glossary.html#Gpolymorphism

[12] Rust.org. (2017) Struct collections::vec::vec. [Online]. Available: https://doc.rust-lang.org/collections/vec/struct.Vec.html

[13] ——. (2017) Struct std::process::command. [Online]. Available: https://doc.rust-lang.org/std/process/struct.Command.html

[14] Crates.io. (2017) Builder pattern derive. [Online]. Available: https://crates.io/crates/derive_builder