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

COS700 Research

Design Pattern Metaprogramming Foundations in Rust

A Study of Abstract Factory and Visitor

Student number: u19239395

Supervisor:
Dr. Linda Marshall

6 November 2020

Abstract

Design patterns are generic solutions to common software design problems. This report focuses on a foundation for Rust metaprogramming macros to automatically create implementations for design patterns at compile-time. The foundation is derived by creating a macro to implement the Abstract Factory design pattern. Thereafter, the foundation is applied to a macro implementing the Visitor design pattern.

Both macro implementations need to know what a manual implementation for each design pattern will look like in Rust. Thus, a manual implementation for each design pattern is also presented after exploring the Rust language. The macro implementations will create code that is identical to the manual implementations. The metaprogramming style used by Rust is also explored in this report to understand how Rust metaprogramming compares to metaprogramming in other languages.

Keywords:

Metaprogramming, Rust, Design Patterns, Procedural Macros, Abstract Factory, Visitor

1 Introduction

This report will focus on the possibility and foundations needed to implement design patterns using metaprogramming in the Rust language. Software design is constantly faced with solving the same problems. The design solutions to these problems became known as design patterns. However, manually programming a design pattern is a tedious and error-prone task that can be solved using metaprogramming.

Rust is a new programming language sponsored by Mozilla Research and was announced the most loved language for the 5th consecutive year, according to the 2020 Developer Survey by StackOverflow[1]. Rust also has metaprogramming support. Thus, this report's first objective is to see if Rust's metaprogramming has the capacity to implement design patterns. If Rust's metaprogramming is capable enough, this report will propose a foundation for implementing any design pattern using Rust's metaprogramming.

In Section 2, this report will explore the need for design patterns and how they came about. Next, the definition and design of the Abstract Factory pattern and the Visitor pattern will be created.

Section 3 will identify different types of metaprogramming and classify the types Rust uses. Rust basics and some lesser-used programming concepts present in Rust will be explored in Section 4.

Finally, Section 5 will create implementations for the Abstract Factory and Visitor pattern in the same way a programmer will do manually. The section will then create a Rust macro to write the same Abstract Factory implementation as the manual implementation. Using the foundation identified for the Abstract Factory implementation, the section will create a Rust macro for the Visitor implementation.

2 Design Patterns

Software design focuses on designing and implementing software to solve a particular problem [iee09, SJB15]. Some problems repeat themselves over time [GY11] with the solutions remaining the same each time. But a novice designer facing any of these repeated problems for the first time will attempt to solve them from first principles [GHJV94, Son98]. When the solution proves flawed or misunderstood some weeks later, a small improvement will be made to the solution [Zhu05, iee09]. These improvements are repeated until all the flaws are removed from the solution [Ste15, SJB15].

On the other hand, seasoned designers create good designs from their own or their colleagues' past experiences [Son98]. These designs focus not only on immediate development but also on the development needed during maintenance [Ker05, GHJV94]. These solutions are easy to find in mature libraries and projects [GHJV94]. Unfortunately, novices are unlikely to get exposure to these projects [Zhu05] or are just overwhelmed by their size [HSG18]. Having exposure to these projects will allow novices to jump to the good design directly, saving time on the iteration process [SJB15].

But rather than taking novices to the projects, it might be possible to take the designs to the novices [CK03]. This is exactly what happened in the 90s. Gamma et al., which the rest of this report will refer to as the Gang of Four (GoF), took some of the repeated designs in projects and documented them in "Design Patterns: Elements of Reusable Object-Oriented Software" [GHJV94].

[1] https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages

Each pattern is documented with a name, the problem it is solving, the solution, and the consequences of using it. Thus, each pattern is an explicit specification for the solution's design while the name becomes a vocabulary encapsulating the specification [GHJV94, BJ12]. The GoF also groups the patterns into 3 categories: creational, structural, and behavioral. This report will focus on one creational pattern – *Abstract Factory* – and one behavioral pattern – *Visitor*. No Structural patterns will be discussed. Since the *Factory Method* pattern can be used to implement the *Abstract Factory* pattern [Nyk12, GHJV94], this section will cover the *Factory Method* too. A discussion of the *Factory Method*, *Abstract Factory*, and *Visitor* follows.

2.1 Factory Method

When an object is created, an isolated function/method may not care which concrete version gets created. It may only care about an abstract definition of the object to perform its duties. The Factory Method design pattern proposes to solve three variants of this problem [GHJV94].

2.1.1 Problems

The first problem is when a function does not have enough information to determine the concrete object it should create. The method/function only knows the abstract object it wants. An example of this is the button on a confirmation dialog. An abstract confirmation process only needs to create a button. Whether this is a blue button used during saving or a glossy button used when installing is not the abstract confirmation process's concern.

Problem two follows on from problem one. Needing to create more than one button means logic to decide on a button to create. Duplicating this logic at each instantiation will complicate maintenance. Assume that the blue button is decommissioned as part of a new facelift in favor of the red button. Updating every line instantiating a blue button will take maintenance time and is error-prone.

Having a superclass delegate the creation responsibility to a subclass is a third problem. Since all Graphical User Interface (GUI) dialogs follow the same process, the design may call for abstracting the common code into an abstract class. The abstract class will create the dialog, draw the needed elements on it, and destroy it once done. However, the concrete open dialog and concrete save dialog will need different buttons. The abstract class will use virtual methods on the concretes classes to instantiate the correct button for drawing.

2.1.2 Solution

The solution will focus on the first two problems since they relate to Abstract Factory. The first problem calls for an abstraction of the product being created. This will allow the confirmation process to function against an abstract button and not a blue or glossy one.

Problem two calls for the isolation of a button's instantiation from the decision logic. This means another abstract class – called *Factory* – to hold the instantiation of a concrete product. The logic will decide which concrete Factory to use at a later stage. The result is the design in Figure 1. The client code will mostly be working with the interfaces in white.

For problem one, functions can create objects from a *Factory<Button>* without worrying if it is working with the brand or fancy factory. The instantiation in *BrandFactory* is the only line needing to swap to a red button for problem two. The single logic decision point will be the only client code containing the Brand and Fancy factory classes. No client code will contain the blue or glossy button classes.

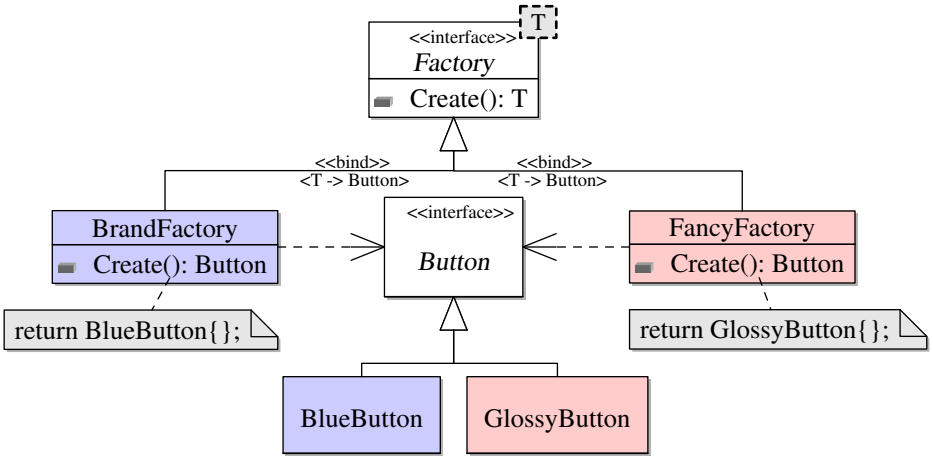

Figure 1: Factory Method design

2.1.3 Consequences

The client code is no longer bound to a concrete button. It now just works with an abstract button. Also, the logic to choose a factory appears once in the code.

However, this solution does require a new factory to be created for each button type. If a new transparent button is to be added, then a new opacity factory will be needed. Maintenance is not compromised since only the single logic point needs to be updated to introduce the new factory. The rest of the client code still does not care that it is now working with a transparent button since the white interfaces did not change.

2.2 Abstract Factory

During the instantiation of classes, four independent sets of problems might exist. The Abstract Factory design pattern proposes to be a solution to these four problems [GHJV94].

2.2.1 Problems

The first problem is when the instantiation and representation of classes need to be separate from the application code. Keeping data structures in a standard library and not the application code is an example of the first problem. It can be argued that using Abstract Factory for this problem might be over-engineering the solution [Ker05].

A second problem is the reverse of the first. When a designer wants to create a library of objects but only expose their interface and not their implementation. In a GUI library, only exposing the operations on a button and not the fact that the button is blue or glossy is an example of hiding the implementation. This should remind us of the Factory Method design just created.

The designer wanting to have a family of related objects to be used together is the third possible problem. Forcing the glossy button to appear with the glossy scroller is an example of wanting the object families together.

Lastly, wanting to swap a family of products for another family of products is the fourth possible problem. This is, swapping all the glossy GUI items to the flat blue items for the entire

application by changing one line will be nice. Again, reminding us of the Factory Method design. This time just for more products.

This report will only focus on problems two to fourth.

2.2.2 Solution

Problems two and four requires each product to have an abstract definition – called *Abstract Product*. Doing so will hide the implementation for problem two – effectively solving problem two. For problem four, all the application code will operate against the interface for a button, scroller, and any other product. The client code will never operate against concrete implementations. Thus, swapping from the glossy to the blue elements will not require any additional code changes at the method calls.

Problems three and four both need to control the instantiation of a family of products. Therefore, a class dedicated to products creation will be needed. Problem four needs this class to be abstract to swap one family for another – hence it being called *Abstract Factory*. Everything presented so far is the same as Factory Method's. However, more than one product needs to be created now. Figure 2 shows the Factory Method design extended to more than one product.

Problem three does not need *AbstractFactory* to be abstract since it has only one family. Figure 2 shows how the concrete brand GUI family maps to all the abstractions. The same mapping can be seen for a fancy family. Since both concrete designs have the same interfaces, swapping the one for the other is non-trivial since client code will again only operate against the white interfaces.

2.2.3 Consequences

Thus, the *Abstract Factory* pattern makes it easy to group a family of related products and swap one family for another. By having client code only work against the abstractions, the *Abstract Factory* pattern also isolated the concrete implementations from the client code. Again, adding a transparent family requires the creation of a transparent factory and each transparent product. However, only the single logic line in the client needs to be updated as a maintenance exercise.

But, adding a new abstract product to the family creates a drawback [GHJV94]. Each concrete family will have to add its own concrete form of the product too. So adding a new window product means creating one abstraction and updating the two families. Thus, the number of classes needing to change is $1 + n$, where n is the number of families [BJ12].

2.3 Visitor

Performing an operation on a set of objects can be quite difficult. The *Visitor* design pattern proposes a solution to three problems [GHJV94].

2.3.1 Problems

For the first problem, imagine classes all with different interfaces. But, an operation needs to be performed against each concrete class. For example, a button and a scroller have different interfaces. However, both have to be drawn. Alternatively, a need might exist to read both aloud for the screen-reader.

Doing unrelated operations with the classes is a second problem. For a study, a company might want to know the average screen surface area of its GUI elements. This is unrelated to a GUI library. Adding surface area methods to GUI classes will pollute the classes.

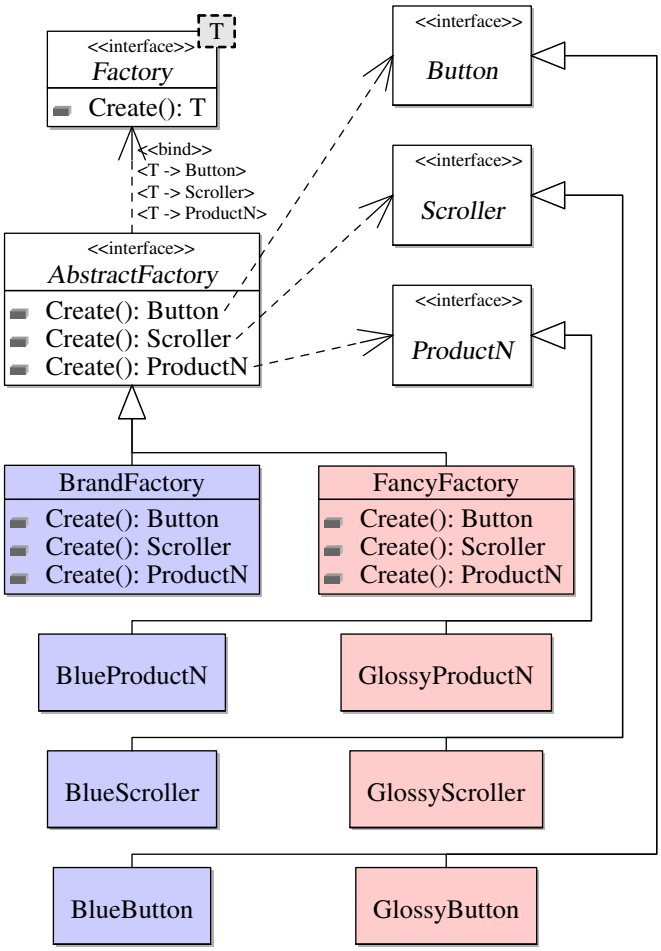

Figure 2: Interfaces needed for Abstract Factory

Lastly, the classes may rarely change as a third problem, but the operations performed on them change often. Coming back to the study, a week later, finding the most common element color might be needed. No new elements were added to the GUI library. Only the need for a new operation exists.

2.3.2 Solution

The solution is to look outside the classes. Thus, creating a new class which knows how to perform only a single operation. The new class will need to visit each of the classes in the problem space. This class is called *Visitor* and solves problems one and two.

However, problem three adds a new dimension. Creating a new operation means creating a new visitor type. Since they are both visiting the same classes, they are both the same in an abstract sense. It is only their implementations that differ. Thus, having an *Abstract Visitor* to represent both is needed.

Abstract Visitor will have a method for each class it needs to visit, as seen in Figure 3. Requiring the client to remember the method corresponding to each class will not be ideal when

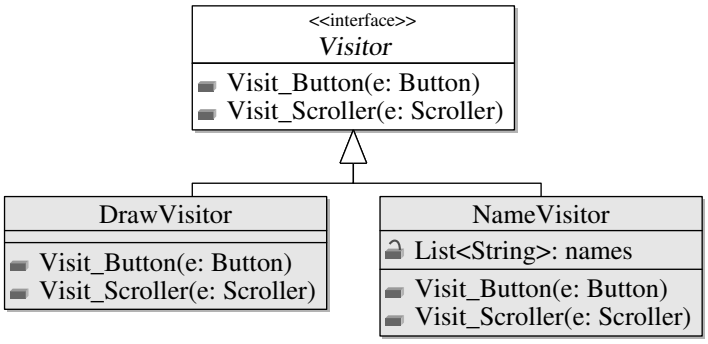

Figure 3: Interfaces needed for Abstract Visitor

the classes reach more than 30. It is also not ideal for generic pieces of code since the method names are not the same.

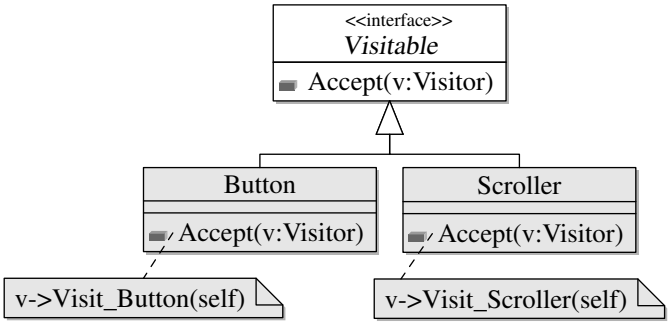

Figure 4: Accept on elements to visit

To solve this, each class has a method to *accept* a visitor, as seen in Figure 4. This method calls for another abstraction, called *Visitable*, with the *accept* method. In the *accept* method, each class can call the visitor operation corresponding to it.

2.3.3 Consequences

New operations (*Visitors*) can easily be added without touching the classes. Catering for next week's survey means creating a new visitor without touching button or scroller. Related operations are now also isolated to each visitor. Thus the classes are not polluted with unrelated methods. Visitors also store the state information they need rather than passing it to each function, as seen in *NameVisitor*.

However, there are two problems. First, adding a new class means updating all the visitors with a method for it. Thus, adding the window element will require an update for each visitor to visit it too. Second, it is assumed that each class exposes enough information through its public interface for visitors to perform their needed operations. The scroller may not expose its name. This will leave the name visitor not being able to get the name of scroll elements.

The code implementations for a design pattern follows the same structure. This means repeated coding each time a pattern is used. However, computers excel at following repetitive

instructions and can do so quickly. Writing these structures using instructions is made possible with metaprogramming.

3 Metaprogramming

Metaprogramming is using a program that writes another program. A program operates on data; a metaprogram treats a program as its data, as shown in Figure 5 [LS19, Ang17, She01]. Input to a metaprogram will be referred to as meta-code. This makes a metaprogram like any regular program. Therefore, a metaprogram can be refactored, abstracted, turned into library helpers, and tested [LS15]. Being able to write a program using code opens up many uses.

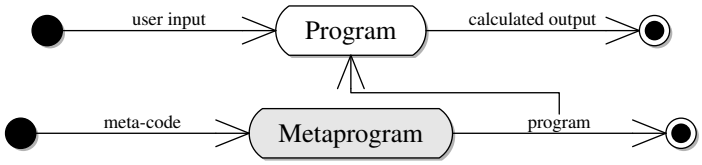

Figure 5: Programs operating on data

3.1 Uses

There are three main uses for metaprogramming: code optimization, code reuse, and analysis.

Code Optimization: A metaprogram can be used to improve the execution speed of the code. For example, with a Just-In-Time (JIT) compiler, a metaprogram can optimize blocks of code that are called more often than others [Hin13] or by caching the results of a method's call [MSD15]. Another example is the use of Domain-Specific Languages (DSL). Here a metaprogram has a better understanding of the code and can apply optimizations unknown to the compiler. Optimizations include knowing a value can never be negative [Hin13], simplifying an expression [She01], or offloading to the GPU [VPG+18].

Code Reuse: Metaprogramming can also be viewed as a code reuse tool. Repeated code – like design patterns [LS15, Ale01] – can be wrapped behind a metaprogram function that will write the reusable code [LS19, KN19]. Complex code can also be translated from a simpler language that end-users can understand [Hin13]. The generated code can be anything from one-liners to classes [LS19].

Analysis: Reading a program as input is the last use for metaprogramming. After reading a program, the metaprogram can analyze its control flow, check types on a dynamic language, or build a proof theorem [She01].

3.2 Dimensions

Metaprogramming has many dimensions. This section will briefly discuss these dimensions, as presented by Lilis and Savidis [LS19], by focusing on the relationship between the metalanguage and the object language, the model used for metaprogramming, when the metaprogram is executed, the location of the meta-code, and how the final program is represented.

3.2.1 Relationship to the Object Language

Metaprogramming will output code in some language. The output language is called the object language, while the metaprogram is written in the metalanguage. These two languages can be different. If they are different, the system is called heterogeneous. For a heterogeneous system, the metalanguage can extend the object language with extra features or be a completely new language. When they are the same, it is called a homogenous system [She01].

3.2.2 Model

Programming comes in different models, such as procedural, functional, and Object-Oriented. The same is true for metaprogramming, which includes the following.

Macro Systems: A macro system takes input and expands it to some output. The output can be another macro. Thus expansion continues until no more macros are left.

The input can come in two forms. First is *lexical macros*, where the input is a stream of tokens. These tokens can be anything and do not need to adhere to a syntax. The second form operates on a specific syntax and is called *syntactic macros*.

Parsing the input can be procedural or pattern-based. Procedural will use an algorithm to generate the output. Patterns will match an input pattern to its output.

Reflection Systems: Reflection is the process an object follows to look at its members and methods. The object can then modify its structure dynamically. This is commonly done at run-time but can also happen at compile-time.

Metaobject Protocol (MOP): Rather than modifying an object's structure, MOPs modify an object's behavior. This is done by inheriting from a metaclass that can modify its own behavior. The modification then affects the subclasses [LZ95]. MOPs can also be used to modify a language's behavior [MSD15].

Aspect-Oriented Programming (AOP): Assume one wants to add logging or performance metrics to all functions in a program. Modifying each function will add a responsibility that does not form part of the function's duties. There is also the cost in the time it will take to modify each function. AOP will *weave* the extra responsibility – called *advice* – into each function.

Generative Programming: This is like a macro system. The difference being that generative code is clearly not meta-code to be expanded by the macro system. They also typically represent their data as an Abstract Syntax Tree (AST).

Multistage Programming: Generating object code in stages is made possible by multistage programming. The generations can either be automatic or require manual annotations [She01, Tah04].

3.2.3 Metaprogram Execution

Three options exist for when the metaprogram can be executed.

Before Compilation: The metaprogram can be executed before compilation. This offers the option of using any language for metaprogramming. The metaprogram takes a source file with meta-code and outputs a file without meta-code.

During Compilation: This option runs the metaprogram as part of the compilation. This requires the compiler to support metaprogramming, or it needs to have a plugin for metaprogramming.

Run-time: Lastly, the metaprogram might execute at run-time. This will require the language execution system to support dynamic code generation and execution.

3.2.4 Meta-code Location

The meta-code location dimension takes into consideration the location of the meta-code to be used as input.

External: The meta-code can be in an external file and will result in a new file with the object code. This option is used with the before compilation option and generative model. Alternatively, suppose this option is used with compilation time execution. In that case, the file might need to be passed to the compiler with a flag.

Embedded: The meta-code can also be embedded within the program to be transformed. This means the source file has a mixture of regular code and meta-code. Embedded code can have three levels of context-awareness.

1. **Completely unaware**: The meta-code only relies on the inputs passed to it. The code immediately after the meta-code is not available to the metaprogram.
2. **Local awareness**: The meta-code relies not only on inputs but also on the code immediately after the meta-code.
3. **Global awareness**: The meta-code relies on inputs and is aware of all code in the file.

3.2.5 Data Representation

The final dimension to consider is the representation used to hold the final code. Since the final code is the metaprogram's data [B+99], it needs to be held in some type. Many systems use strings, graphs, or an algebraic data structure [She01]:

String: The final program is held in a string. This option is not desired since building a class may need hundreds of string append operations spanning hundreds of lines. The object code will be interleaved with the metaprogram making it hard to distinguish between them, thus making it easy to construct a string that is not syntactically correct.

Graphs: A graph type will add structure to the program being built. Furthermore, it makes a better separation between the object code and the metaprogram code. However, it still does not guarantee that the structure will be syntactically correct.

Algebraic: Storing the data as an algebraic expression with type encoding or an AST is the only guarantee of a syntactically correct program. However, building an AST by hand is hard. Lisp solved this problem by using *quasiquotes* [B+99].

Quasiquotes is a form of templating/string interpolation. It allows writing the data as a "string" (enclosed in backtick quotes) in the object language's syntactical form. This *quoted* "string" is then transformed into the desired data structure. Thus, it acts as a shortcut for constructing an AST [LS15]. Placeholders are placed in the *quoted* "string" to be replaced with variables from the metaprogram context. These placeholders need to be identifiable. Thus, the placeholders are preceded by some *unquote* character [B+99].

Each dimension has an option used by Rust to enable metaprogramming.

3.3 Metaprogramming in Rust

Rust has two metaprogramming functionalities built into the language. The first has been part of the language for some time and is meant for general metaprogramming. The second, called *Procedural Macros*, is a newer addition added in late 2018 [2] and is the focus of this report [KN19]. *Procedural Macros* use each dimension discussed in Section 3.2 as follow.

The metalanguage for *Procedural Macros* in Rust is coded in Rust syntax. Thus, *Procedural Macros* are homogenous. From the name *Procedural Macros*, it is also clear it follows the macro model, and the parsing is procedural. The input stream is also a lexical token stream.

Execution happens during compilation. This means the macros need to be available to the compiler and need to be precompiled. Therefore, the macros need to be isolated from client code in a library marked for macro use [3]. The macro invocation is embedded in the client code. *Procedural macros* have both local awareness or no awareness, depending on the flavor used. Flavors will be discussed in Section 3.3.1. The data representation is the same as the input – a lexical token stream.

The input lexical token stream does not need to follow a specific syntax. The metaprogram designer is free to choose this syntax. However, the output stream needs to be valid Rust code. Rust tries to keep its standard library as slim as possible while offloading features to libraries. Given the wide range of possible inputs, no standard library helpers exist for working with a token stream. However, two Rust libraries exist for working with token streams – the input and output of *Procedural Macros*.

The first is *syn*[4] for parsing Rust syntax to a syntax tree. Other parsers can also be built using *syn*.

The second is *quote*[5] for generating a token stream from Rust syntax. It is a macro that uses the quasiquotes concept from Lisp. Thus, anything in the *quoted* "string" is correctly highlighted, formatted, and autocompleted by an editor.

3.3.1 Procedural Macro Flavors

The three flavors of Procedural Macros are function-like, derive, and attribute macros.

Function-like Macros: These are the most straightforward flavor of procedural macros. They take an input stream and return an output stream – i.e., they are context unaware. Listing 1

[2] https://blog.rust-lang.org/2018/12/21/Procedural-Macros-in-Rust-2018.html
[3] https://doc.rust-lang.org/reference/procedural-macros.html
[4] https://docs.rs/syn/1.0.31/syn/index.html
[5] https://docs.rs/quote/1.0.7/quote/index.html

shows a reflective function-like macro that returns its input unaltered as output. Note, Section 4 will explain Rust syntax in detail.

Listing 1: Declaring a function-like macro

```
1 extern crate proc_macro;
2 use proc_macro::TokenStream;
3
4 #[proc_macro]
5 pub fn reflect(input: TokenStream) -> TokenStream {
6     input
7 }
```

Lines 1 and 2 import the *proc_macro* library and the *TokenStream* type in the library. These two lines will be needed for all macro definitions and will not appear in future examples. The *#[proc_macro]* attribute in line 4 marks the function that follows as a function-like macro. Line 5 shows it taking one input and returning one output – both of type *TokenStream*. In line 6, the input is returned unaltered.

Client code will have a call as follows to use the macro.

```
reflect!(2 + 3, 5);
```

This call has the same function name as the macro. All function macros are invoked using the *!* (exclamation) sign – called the *macro invocation operator* – to distinguish them from regular function calls. The call will be replaced with the output "2 + 3, 5". Since the output is invalid Rust code, the compiler will give an error on the call line. Notice how everything inside the parenthesis (2 + 3, 5) will be passed to input. A *TokenStream* can be thought of as a list of tokens. There are four possible token types [6]:

- An *Ident* to hold an identifier like a variable name or keyword.
- A *Punct* to hold a single punctuation mark.
- A *Literal* to hold a literal like an integer value, string, or literal character.
- A *Group* to hold an inner/nested *TokenStream* surrounded by brackets.

2, 3, and 5 in the above macro call will be literal tokens. The +(plus) sign and comma(,) will be punctuations in both streams. Again, parsing and generating the list will be hard. The next example shows how to use the *syn* and *quote* libraries to make parsing and generating easier.

Derive Macros: These macros are used to automatically implement interface methods on objects, as seen in Listing 2.

Listing 2: A derive macro using *syn* and *quote*

```
1 use syn::{parse_macro_input, DeriveInput};
2 use quote::quote;
3
4 #[proc_macro_derive(GetType)]
5 pub fn derive_answer_fn(tokens: TokenStream) -> TokenStream {
```

[6] https://doc.rust-lang.org/proc_macro/enum.TokenTree.html

```
6       // Parse the context tokens to a DeriveInput syntax tree
7       let context = parse_macro_input!(tokens as DeriveInput);
8       // Get the type name from the syntax tree
9       let name = &context.ident;
10      // Construct the output using the quote macro
11      let output = quote! {
12          impl GetType for #name {
13              fn get_type(&self) -> String {
14                  String::from(stringify!(#name))
15              }
16          }
17      };
18
19      output.into() // Convert the output to a TokenStream
20  }
```

Line 1 shows the import for the *syn* library to parse an unstructured input list of tokens to a syntax tree. This is followed by the *quote* macro in the *quote* library for generating a token stream. Derive macros have the *proc_macro_derive* attribute with an argument for the macro name, as seen in line 4. The function's input in line 5 is the context the macro is called on. Thus, derive macros have local context-awareness.

This macro is called by annotating a type with the derive attribute. The type being annotated will serve as the local context for the macro.

```
#[derive(GetType)]
struct SomeStruct;
```

Here, *SomeStruct* is the type being annotated, and it is the context passed to the macro. Line 7 parses the context to a *DerivedInput* syntax tree from *syn*. TokenStreams can be parsed to any syntax tree defined by the macro developer. Here *DerivedInput* is chosen since it is provided by *syn* for derive macros. *Syn* will give a compilation error if the parsing fails. Getting the struct's name happens in line 9.

Lines 11 to 17 show the use of the *quote* library for quasiquotes. Rather than a *quoted* "string", it uses the *quote* macro. Placeholders are *unquoted* with the # (pound) sign. Thus, *#name* will come from the metaprogram context – line 9. Notice the syntax highlighting being correct inside the quote macro. Finally, line 19 converts *output* to a *TokenStream*.

The compiler will append the macro's output below the annotated type for derive macros. A snippet of the output is as follows – it comes from line 12.

```
impl GetType for SomeStruct { ... }
```

Attribute Macros: A function-like macro with context-awareness results in attribute macros. Therefore, two token streams – the input and the context – are passed as arguments to it. This time the attribute above the function is *proc_macro_attribute*. Listing 3 shows another reflect macro that returns the context unaltered.

Listing 3: Declaring an attribute macro

```
#[proc_macro_attribute]
pub fn reflect_two(input: TokenStream, context: TokenStream) -> TokenStream {
    context
}
```

Client code will again annotate a type with an attribute to call the macro. However, the attribute is the macro's function name.

```
#[reflect_two(multiple => tokens)]
fn some_function() {}
```

The attribute's input – not following a specific syntax – is the first stream passed to the function, while the context is the second. Like function macros, the context will be replaced with the output.

Since the metalanguage is Rust code itself, it is time to learn more about Rust.

4 Rust

Rust is a relatively new language sponsored by Mozilla Research 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].

4.1 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, called the *owner* [KN19].

A variable also has 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].

4.1.1 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 4: Example of ownership transfer

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

In Listing 4, in 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 in line 4, 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 in line 5. Since the compiler can guarantee *t* is the only variable owning the data, the compiler can free the memory in line 5.

Listing 5: Function taking ownership

```
1 fn main() {
2     let s = String::from("string");
3     take_ownership(s);
4     println!("String len is {}", s.len()); - borrow of moved value: `s`
5 }
6 fn take_ownership(a: String) {
7     // some code working on a
8 } // 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 5. 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 in line 8. When *main* calls *take_ownership*, *a* becomes the new owner of *s*'s data, making the call in line 4 invalid. When taking ownership is not desired, the second form of ownership, borrowing, should be used instead.

4.1.2 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 6: Function taking borrow

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

As seen in Listing 6, 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 in 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 in line 7. Returning to the borrowed book analogy. One would not make highlights and notes in a book one borrowed unless the owner gave explicit permission.

4.2 Immutable by Default

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

Listing 7: Function taking mutable borrow

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

As seen in Listing 7, line 6, mutable borrows are activated using *&mut* on the type. Again, *mut* is also used in the call in line 3 to make it explicit that 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 in 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]. The compliler wll always enforce these constraints, requiring all code – written manually 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, which will now be discussed.

4.3 Not Quite OOP

No single definition exists to qualify a language as Object-Oriented [Mey97, SB85, GHJV94, KN19]. Three Object-Oriented concepts will be explored to understand Rust better. These three are *Objects as Data and their Behaviour*, *Encapsulation*, and *Inheritance*.

4.3.1 Objects as Data and Their Behaviour

An object holds both data and procedures operating on the data [Mey97, SB85, GHJV94, Mal09]. Rust holds data in *structs* with the operations being 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 followed by a type, as shown in Listing 8 lines 1 to 5.

Methods: The operations to perform on a struct are defined in *impl* blocks, as seen in lines 6 – 16 in Listing 8.

The ownership rules apply to the struct as follow:

- The method *have_burial* moves *self* and will result in cleanup code being inserted in line 15. Thus, any objects of *Foo* after *have_burial* is called will be invalidated. This happens in line 22, resulting in a compile error – the same error happened in Listing 5.
- The method *get_age* 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* in line 19.

Listing 8: Example of a Struct and Methods

```
1  pub struct Foo {
2      pub name: String,
3      age: u8,
4      gender: Gender,
5  }
6  impl Foo {
7      pub fn get_age(&self) -> u8 {
8          self.age // age is returned [7]
9      }
10     pub fn have_birthday(&mut self) {
11         self.age += 1;
12     }
13     fn have_burial(self) {
14         unimplemented!();
15     } // `self` will be dropped here
16 }
17
18 fn main() {
19     let mut bar = Foo { name: String::from("bar"), age: 1, gender: Gender::Male }; [8]
20     bar.have_birthday();
21     bar.have_burial(); // Party was crazy [8]
22     println!("Bar was {} years old", bar.get_age()); - borrow of moved value: 'bar'
23 }
24
25 // Second implementation block [9]
26 impl Foo {
27     // Construct a new `Foo` with a given name [10]
28     pub fn be_born(name: String) -> Self {
29         Foo { name, age: 0, gender: Gender:Female }
30     }
31 }
```

[7] Rust does not always use the *return* keyword. Lines without a semi-colon and that are the last line in a scope act as return statements. Thus, line 8 acts as a return statement.

[8] *main* can access the private members because main in the same file as them.

[9] Rust code can have multiple *impl* blocks for a single type.

[10] Rust does not provide constructors like other OOP languages. Instead, Rust has what it calls *associate*

4.3.2 Encapsulation

Encapsulation hides the implementation details from the client [KN19, Mey97, WS15]. In Rust, encapsulation is the default unless specified otherwise using the *pub* keyword, as seen in lines 2, 7, and 10. Encapsulation allows the struct creator to change the internal procedures of the struct without affecting the public interface used by clients. Therefore, Rust meets the encapsulation concept.

4.3.3 Inheritance

Inheritance allows an object to inherit some of its data members and procedures from a parent object [Mey97, SB85, GHJV94, WS15]. Inheritance is mostly used to reduce code duplication. Rust does not make provision for the inheritance concept.

Rust does make provision for "Program to an interface, not an implementation" and "Favor object composition over class inheritance", which the GoF includes as concepts for OOP design [GHJV94]. Both concepts are realized using *traits*, thereby allowing Rust to implement the GoF design patterns.

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 to program to an interface, as seen in Listing 9, lines 1 to 7.

Listing 9: Working with traits

```
1  trait Show {
2      fn show(&self) -> String;
3      // Method with a default implemetation
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 struct Tester {}
14 impl Show for Tester {
15     fn show(&self) -> String {
16         String::from("Tester")
17     }
18 }
```

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*.

Traits are implemented on structs using the *impl* keyword followed by the trait name, as seen in line 14. The *Show* trait has a default implementation for *show_size* in line 4. Thus, *Tester*

functions. An associate function is a method definition not containing *self* in the parameter list [KN19]. They are used as constructors by returning an owned instance of the struct, as seen in line 28.

does not need to implement *show_size*, but is only required to implement the *show* method.

Traits allow one to use composition to construct complex objects. Getting back to the abstract confirmation dialog presented in Section 2.1.1 for an open dialog and a safe dialog, using composition, the generic confirmation dialog will have to be composed of a button and text, as seen in Listing 10.

Listing 10: Using composition

```
1  pub trait Button {
2      fn draw(&self); // Button to draw itself
3  }
4  pub trait DisplayText {}
5
6  struct ConfirmationDialog {
7      button: Box<dyn Button>,
8      display_text: Box<dyn DisplayText>,
9  }
10 impl ConfirmationDialog {
11     // Constructor to build a new `ConfirmationDialog`
12     pub fn compose(button: Box<dyn Button>, display_text: Box<dyn DisplayText>) -> Self {
13         ConfirmationDialog { button, display_text }
14     }
15     pub fn show(&self) {
16         self.button.draw(); // Draw the button for the dialog
17     }
18 }
```

Using traits, an abstract button and display text is defined in lines 1 to 4. The confirmation dialog "has-a" [Mal09] button and display text in lines 7 to 8. Using associate functions[10], a composition constructor is created in line 12. To construct the open dialog, the open button will be passed to this constructor. The safe dialog will need the safe button to be passed in. Finally, the *show* method in line 15 will show the confirmation dialog, whether it is the open dialog, save dialog, or a new dialog definition. This is called polymorphism since a dialog will change form depending on the elements passed to the constructor at run-time [WS15, Mal09, GHJV94].

The *show* method will now work with more than one button. The open button will have a different implementation for the *draw* method, defined in line 2, than the save button. Since the buttons change at run-time, the compiler will be unable to determine the correct *draw* method to call in *show* in line 16. Dispatching, as discussed next, resolves this problem [PB02].

4.4 Dispatch

Dispatching is the matching of the correct method to an object at method invocations [DHV95]. Sometimes there is only one object to match against, making matching easy. However, it can also be the case that multiple objects have the same method name, making matching harder. The matching can happen at compile-time – called *static dispatching* – or run-time – called *dynamic dispatching*.

4.4.1 Static Dispatch

Matching the method call at compile-time is called *static dispatch* [KN19, Ale01, AC12]. When Rust compiles 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.

In Listing 9, line 9, the generic *work* method is defined. Suppose the *work* method is called with a *Tester*, defined in line 13, and later with a *String*. Then an example of the *monomorphization* process during compilations is shown in Listing 11.

Listing 11: Example of *monomorphization*

```
1 fn work_tester(object: Tester) {
2     println!("{}", object.show());
3 }
4 fn work_string(object: String) {
5     println!("{}", object.show());
6 }
```

Each *show* call in the expanded functions can match only one object – *Tester* and *String* respectively – making this a simple case. Using *trait objects* – rather than generics – will result in multiple objects having the same method name and the need for dynamic dispatch.

4.4.2 Dynamic Dispatch

When the compiler cannot determine which method to call at compile-time, because the object type is not fixed, dynamic dispatch [Ale01, KN19, AC12] occurs. At run-time, pointers held by the trait objects are used to determine which method to call [KN19] based on the object type the method is called on.

Listing 12 shows the *work* function implemented using dynamic dispatch rather than generics – refer to *work* in Listing 9 line 9 for the generic version. Since generics are not used, *monomorphization* will not happen. Calling this function with a *Tester* object means the *show* for *Tester* needs to match at the method invocation in line 2. But later, calling this function with a *String* object means *String*'s *show* needs to match. Since the objects passed in changes at run-time, knowing which object to match against can, therefore, only be known at run-time. Thus, pointers inside the *Show* trait object – line 1 of Listing 9 – are used at run-time to match each object against its method.

Listing 12: 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 to use trait objects rather than generics – compare to *work* in Listing 9, line 9.

1. The generic *T* is removed and replaced with *Show*.
2. The *dyn* keyword is added to the type to indicate that dynamic dispatch will take place explicitly [KN19].
3. Borrowing (& before *dyn*) now takes place.

The static dispatch generic trait examples in Listing 9 can also be made to take a borrow, but taking ownership gives a compile-time error with dynamic dispatch caused by the *Sized* trait.

4.4.3 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 mark that the size of a type is known at compile-time. This marking is the only use of the *Sized* trait, and it has no meaning or representation after compilation. However, Rust automatically/implicitly adds the *Sized* trait bound to all function arguments and local variables [KN19].

The size of an object is influenced by the data it holds. Also, any object can choose to implement *Show*. Thus, two different objects implementing *Show* can have different sizes. Ownership will want to pass each object on the stack, but with dynamic dispatch, each object will need a different stack size only known at run-time. Therefore, the *Show* dynamic trait's size cannot be determined at compile-time, thus it cannot implement the *Sized* trait.

Since all function arguments expect the type to implement the *Sized* trait but having the *Show* dynamic trait not implement it, a compile error will be given stating *dyn Show* does not implement *Sized* when trying to use it as a function argument. However, pointers do implement *Sized*. Therefore, putting the dynamic trait behind any pointer will allow it to be used as a function argument or local variable. The reference (taking a borrow) and the *Box<T>* type are two such pointers[KN19].

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

4.5 Enums

In Rust, enums can also hold objects [KN19] as seen in Listing 13, lines 1 to 4. The *Option* enum, as defined here, is built into the standard Rust library [KN19] to replace *null* as used in most languages. An *Option* can either be *Some* object or *None*. This is a design Rust uses to be memory safe [11] by handling the (*None*) option at compile-time.

Listing 13: 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
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
```

[11] 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/)

```
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 arms are missing, the compiler will provide an error stating not all the enum options are covered. Matches need to be exhaustive since all variants need to be covered in Rust. The exhaustive check is not always desired. Thus, there are two options for mitigating the exhaustive check [KN19].

- Adding the _ (underscore) catch-all arm to handle the default case for all missing enum options.
- Using the *if let* pattern as seen in line 14.

The *if let* pattern also allows extraction of the enum object. Here the extraction will not be used as indicated by the _ (underscore).

Match blocks and *if* conditions – which include *if let* – are considered expressions in Rust. Thus, lines 16 and 18 are not including their ending semi-colon to return *true* and *false* from the *if* expression which is assigned to the *valid* variable.

4.5.1 Result Enum for Error Handling

While most languages use exceptions to propagate errors back to the caller, Rust uses the *Result* enum instead. The definition for *Result* can be seen in Listing 14 in lines 1 to 4.

Listing 14: 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
10  fn error_explicit_handle() {
11      let r = may_error();
12      let r = match r {
13          Result::Ok(result) => result,
14          Result::Err(error) => panic!("Operation failed: {}", error),
15      };
16  }
17  fn error_short_handle() {
18      let r = may_error().expect("Operation failed");
19  }
20
21  fn error_explicit_propogation() -> Result<i32, String> {
```

```
22      let r = match may_error() {
23          Result::Ok(result) => result,
24          Result::Err(error) => return Err(error),
25      };
26
27      Ok(2)
28  }
29  fn error_short_propogation() -> Result<i32, String> {
30      let r = may_error()?;
31
32      Ok(2)
33  }
```

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* in line 7. Any calls to *may_error* have to handle the possible error by panicking or propogating the error.

Panicking: The caller will use a *match pattern* to extract the error and panic, as seen in lines 12 to 15. 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 [12]. The helper method *expect*, as seen in line 18, is the same as to lines 12 to 15.

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

Line 24 shows how to propagate the error up the stack – the *return* is to return from the function *error_explicit_propogation* and not the match. Line 23 uses the *result* if it is fine – the lack of *return* returns from the match and assigns *result* to *r*. Line 30 shows how to use the *?* (question mark) operator to do the same thing. The *?* can be used in functions that return *Result* [KN19].

5 Reporting

This section will discuss the implementations for an AF macro and a Visitor macro. It will first discuss the layout used for the macro library. Next, manual implementations, as written by a programmer for AF and Visitor will be presented. These implementations will be the goalposts for the macro outputs. Finally, the macros will be written to generate the same outputs.

5.1 Library Layout

Other macros/libraries can use parts of the framework being created. Thus, the macro implementations will be separated from the structures they will use. Another reason for this choice is that the *TokenStream*s, which were presented in Section 3.3, cannot be unit tested. For this reason, *Syn* and *quote* operate against a wrapper found in *proc_macro2* [13], which the helper structures in this section will also use. This leads to the libraries shown in Figure 6.

[12] https://doc.rust-lang.org/std/result/enum.Result.html
[13] https://docs.rs/proc-macro2/1.0.19/proc_macro2/index.html

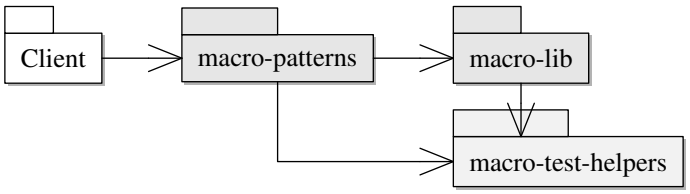

Figure 6: Layout of libraries

Client code will use the *macro-patterns* library. *Macro-patterns* will contain macro definitions, as was defined in Section 3.3, for Abstract Factory and Visitor. *Macro-lib* will provide syntax tree components that are missing from *syn* or are simpler than *syn*'s. Finally, all the code is tested with *macro-test-helpers* providing helpers dedicated to making tests easier. The tests will not be covered in this report. However, the reader should note that automated tests are used to ensure the macro outputs are identical to the manual implementations covered in the next section.

Some of the *client* code, all of the *macro-lib*, and all the *macro-patterns* code are shown in the appendix of this report in the order just listed. In each appendix section, the code is ordered by filename and not the order in which they are presented next.

5.2 Manual Implementations

Typically the design pattern implementations will be written by a programmer without reusing code. Even though this section creates macros to replace this manual process, the design patterns will be implemented here manually to know what the macro outputs should be.

5.2.1 Simple GUI

The design pattern implementations are built on the simple GUI library, shown in Listing 16, which defines:

- An *Element* that can create itself with a given name and return that name.
- A *Button* that is an *Element* to be clicked with text.
- An *Input* element that can hold text inputs.
- A *Child* enum that can be a *Button* or *Input*.
- A concrete *Window* struct that can hold *Child* elements.

The abstract *Button* and *Input* each have a concrete brand version. They are the *BrandButton* and *BrandInput* shown in Listing 15.

5.2.2 Manual Abstract Factory Implementation

Listing 17, lines 10 to 17, shows an implementation of AF for the GUI. This implementation maps directly to the UML presented for AF in Section 2.2.

Line 6 imports the concrete brand elements from Listing 15, with line 7 importing the abstract elements from Listing 16. A factory method is defined in lines 10 to 12. As explained in

Section 4.4.3, Rust defaults to *Sized* types. But, *Factory* will have to create dynamic types. Thus, to allow dynamic generics, the sized requirement is turned off using the *?Sized* syntax, in line 10. In line 14, an AF is defined using the factory method as super traits. The *Display* super trait is to show the macro can handle complex AFs.

Client code will create a concrete brand factory as follow:

```
struct BrandFactory {}
impl AbstractGuiFactory for BrandFactory {}
impl Factory<dyn Button> for BrandFactory {
    fn create(&self, name: String) -> Box<dyn Button> {
        Box::new(brand_elements::BrandButton::new(name))
    }
}
impl Factory<dyn Input> for BrandFactory {
    fn create(&self, name: String) -> Box<dyn Input> {
        Box::new(brand_elements::BrandInput::new(name))
    }
}
impl Factory<Window> for BrandFactory {
    fn create(&self, name: String) -> Box<Window> {
        Box::new(Window::new(name))
    }
}
```

5.2.3 Manual Visitor Implementation

A manual visitor implementation can be seen in Listing 18. Visitor consists of three parts:

1. The abstract visitor, as defined in lines 8 to 18, maps to the UML for visitor, as defined in Section 2.3.

2. Helper functions for traversing the object structure [GHJV94] in lines 20 to 41. This allows for default implementations on the abstract visitor to call its respective helper. Doing this allows the client to write less code when their visitor will not visit each element. It means client code does not need to repeat code to visit into an element's children since the client can call a helper with the traversal code – like *visit_window* in line 31.

3. Making each element visitable in lines 43 to 62 which maps to the *Visitable* UML, as defined in Section 2.3.

A client will write a concrete visitor, as shown below. This visitor collects the names of each element in a structure except for the names of windows to show the power of the default implementations delegating to the helpers. Because the default implementation in Listing 18 line 16 uses the helper in line 31, *NameVisitor* does not need to implement anything for *Window* to traverse into a *Window*'s children. This visitor implements the *Display* trait to be able to call *to_string()* on it. Calling *to_string()* will join all the names this visitor collected.

```
struct NameVisitor {
    names: Vec<String>,
```

```
}
impl NameVisitor {
    pub fn new() -> Self {
        NameVisitor { names: Vec::new() }
    }
}
impl Visitor for NameVisitor {
    fn visit_button(&mut self, button: &dyn Button) {
        self.names.push(button.get_name().to_string());
    }
    fn visit_input(&mut self, input: &dyn Input) {
        self.names
            .push(format!("{} ({})", input.get_name(), input.get_input()));
    }
}
impl fmt::Display for NameVisitor {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}", self.names.join(", "))
    }
}
```

The test function in Listing 18, lines 111 to 128, show how to use this visitor. First, a window, button, and input are created. A random name is set on the input before it, and the button, is added to the window. A *NameVisitor* is created and applied to the window. Lastly, a test in line 128 shows the visitor collected the correct names.

5.3 Macros

Rust's metaprogramming abilities will be used to create macros that can write the repeated sections in the manual implementations. The outputs of each macro should match exactly the manual implementations written by a programmer. Three macros will be created: one to create an AF; one to implement a concrete factory for an AF; one to create a Visitor.

5.3.1 Abstract Factory Macro

The implementation of the AF macro will be used as a foundation to implement the Visitor macro. The input passed to the macro – defined as meta-code in Section 3 – will be parsed to a model. This model will be able to expand itself into its pattern implementation, as defined in Section 5.2. A model will be composed of syntax elements. Some of the syntax elements will come from *syn*, and others will have to be created.

The client meta-code for AF is as follows – since it is the same as the manual implementation, it also maps directly to the AF UML given in Section 2.2:

```
use crate::gui::{
    brand_elements,
    elements::{Button, Element, Input, Window},
};
```

```rust
pub trait Factory<T: Element + ?Sized> {
    fn create(&self, name: String) -> Box<T>;
}

#[abstract_factory(Factory, dyn Button, dyn Input, Window)]
pub trait AbstractGuiFactory: Display {}

struct BrandFactory {}
impl AbstractGuiFactory for BrandFactory {}

#[interpolate_traits(
    Button => brand_elements::BrandButton,
    Input => brand_elements::BrandInput,
)]
impl Factory<dyn TRAIT> for BrandFactory {
    fn create(&self, name: String) -> Box<dyn TRAIT> {
        Box::new(CONCRETE::new(name))
    }
}

#[interpolate_traits(Window => Window)]
impl Factory<TRAIT> for BrandFactory {
    fn create(&self, name: String) -> Box<TRAIT> {
        Box::new(CONCRETE::new(name))
    }
}
```

The client needs to specify the factory method they will use. This factory method needs to take a generic element *T*. The *AbstractGuiFactory* is annotated with an attribute macro (see Section 3.3.1) named *abstract_factory*. The factory method and factory elements are passed to the macro.

The client will create their concrete *BrandFactory* and use the *interpolate_traits* attribute macro to implement each element's factory method. Here the client uses two invocations of *interpolate_traits* since *Window* is concrete and does not use the *dyn* keyword.

Models: Both *abstract_factory* and *interpolate_traits* take in a comma-separated list of inputs. The *syn* library provides the *Punctuated*[14] type to parse a list of elements separated by any punctuation marker. *Syn* also provides *Type*[15] for parsing Rust types that will be used by the *abstract_factory* macro. The elements passed to *interpolate_traits* need to be custom made. Two models need to be created for the AF macros:

1. A ***TraitSpecifier*** to hold an item passed to the *interpolate_traits* macro. Each item will map a trait to its corresponding concrete type.

2. ***AbstractFactoryAttribute*** to hold the input passed to the *abstract_factory* macro. The input will consist of a factory method and a list of elements the AF will create.

[14] https://docs.rs/syn/1.0.48/syn/punctuated/struct.Punctuated.html
[15] https://docs.rs/syn/1.0.48/syn/enum.Type.html

TraitSpecifier is defined in Listing 24. It will use the syntax *trait => concrete* to map a trait type to its concrete definition.

Lines 5 and 6 import the *syn* elements that will be used. The tests use line 13. The model is defined in lines 14 to 18 to hold the abstract trait, the arrow token, and the concrete. The *Token*[16] macro in line 16 is a helper from *syn* to expand Rust tokens and punctuations easily. Lines 21 to 29 implement the *Parse*[17] trait from *syn* for parsing a token stream to this model. Here parsing is simple, *parse* each stream token or propagate the errors. *Syn* will take care of converting the errors into compiler errors.

AbstractFactoryAttribute is defined in Listing 25. This will be the input passed to the *abstract_factory* macro.

The model takes the factory method trait as the first input, separated (*sep*) by a comma, followed by a comma-separated list of types the abstract factory will create as, was shown in the client meta-code.

An *expand* method for the *AbstractFactoryAttribute* model is also defined in Listing 25. The expand method takes in a trait definition syntax tree as *input_trait* in line 36. In lines 38 to 45, a factory method super trait is created for each type passed to the macro. Lines 39 and 40 create local variables to be passed to *quote*. Line 42 uses a *syn* helper function to turn a *quote* into a syntax tree. Since *types* defined in line 29 is a list, *quote* has to be told to expand each list element. The special *#(list-quote)<sep>** quasiquote is used to specify how to expand a list. The optional *sep* character is used as a separator for each item. In line 43, the factory method is expanded for each type using the + (plus) sign as a separator. Thus, if *MyFactory, Type1, Type2* is passed to the macro, then line 43 will create *MyFactory*<*Type1*> + *MyFactory*<*Type2*>.

Line 48 appends the factory super trait that was just constructed to the context input. The new context input is returned in line 51.

Definitions: The AF macro is shown in Listing 26 to be an attribute macro, as was defined in Section 3.3.1. Line 17 parses the input context – which is the *AbstractGuiFactory* definition in the meta-code – with line 18 parsing the macro inputs to *AbstractFactoryAttribute* as defined in Listing 25. Line 20 expands the inputs on the context as defined in Listing 25.

The *interpolate_traits* macro – also being an attribute macro – is also shown in Listing 26.

Line 25 parses the macro inputs to a comma-separated list of *TraitSpecifier*s defined in Listing 24. Rather than parsing the context input to a model, the context input is used as a template for each concrete factory implementation. *Quote* macro templates expand when the macro is compiled. However, these templates need to be expanded when the macro is run. Macros are run at the compile-time of the client code. Thus a string interpolator like *quote* is needed that can run at the macro's run-time. Listing 23 defines such a helper for a *proc_macro2* token stream.

Line 7 defines an *Interpolate* trait for types that will be interpolatable at macro run-time. Line 13 implements the *Interpolate* trait for *syn*'s *Punctuated* type if the punctuated tokens implement the *Interpolate* trait – the *TraitSpecifier* token will be made interpolatable soon.

The *interpolate* function in line 25 takes in a template *stream* and hash map of items to replace in the input stream. Thus, if the hash map has a key of *TRAIT* with the value of *Window*, then each *TRAIT* in the template will be replaced with *Window*. Line 29 creates a *new* token stream that will be returned from the function in line 62. Each token in *stream* will be copied to *new* if the token does not need to be replaced.

Line 33 starts looping through the tokens, and line 34 matches on the token type. Four token types were presented in Section 3.3.1. The *Literal*, *Punct*, and *Group* tokens will be copied as-is.

[16] https://docs.rs/syn/1.0.48/syn/macro.Token.html
[17] https://docs.rs/syn/1.0.48/syn/parse/trait.Parse.html

Since the *Group* token holds its own token stream, it needs to recursively call *interpolate* on its stream and create a new group from the result – the span copied in line 41 is to preserve the context for compilation errors. Only the *Ident* tokens are matched against the replacements. Thus, if the identifier matches any of the replacements in line 49, then the replacement *value* is copied to the *new* stream in line 51. Otherwise, the identifier is copied in line 57.

Listing 24, lines 32 to 44, shows interpolation being implemented for the *TraitSpecifier*. Lines 37 and 40 set up the hash map to replace *TRAIT* with the abstract trait and *CONCRETE* with the concrete type. Line 42 calls *interpolate* as defined in Listing 23, line 25.

Thus, line 28 in Listing 26 will use the context input as a template to interpolate each *TraitSpecifier* passed into the *interpolate_traits* macro.

5.3.2 Visitor Macro

The Visitor macro implementation will be a function-like macro – as was defined in section Section 3.3.1. Like the AF implementation, it will use *syn* to parse the input to a model. The model will be expanded to match a manual implementation using *quote*.

The following shows the client meta-code for the Visitor macro – meta-code being the macro input as defined in Section 3. This will result in the same code as Listing 18 and thus implements the UMLs in Section 2.3.

```
use macro_patterns::visitor;
use std::fmt;

use crate::gui::elements::{Button, Child, Input, Window};

visitor!(
    dyn Button,
    dyn Input,

    #[helper_tmpl = {
        window.get_children().iter().for_each(child {
            match child {
                Child::Button(button) => visitor.visit_button(button.as_ref()),
                Child::Input(input) => visitor.visit_input(input.as_ref()),
            };
        });

    }]
    Window,
);
```

A list of types is being passed to the *visitor* macro function. A type can also have two options inside the *#[options]* syntax:

1. *no_default* to turn off the default trait function implementation – as defined in Section 5.2.3.
2. *helper_tmpl* to modify the helper template used – also defined in Section 5.2.3.

The client code above shows how the *helper_tmpl* option is used on the *Window* type. *Syn* does not make provision for parsing complex options like this. Thus, this section will create new syntax elements to parse the input for the Visitor macro.

Models: Six parsable models need to be created:

1. ***KeyValue*** to parse a *key = value* stream. The *key* will be an option, with the *value* being the option value.
2. ***OptionsAttribute*** to hold a comma-separated list of *options* inside the *#[options]* syntax. Each option will be a *KeyValue*.
3. ***SimpleType*** to parse each type in the input list. The *Type* provided by *syn* holds a punctuated *PathSegments*[18]. The type will determine the function name, thus building a function name from each identifier in the *PathSegment* list is unnecessarily complex.
4. ***AnnotatedType***, which is a type annotated with an *OptionsAttribute* like *Window* in the client code above.
5. ***VisitorFunction*** to parse the input passed to the macro. The input is a list of *AnnotatedTypes*.

The ***KeyValue*** type is the most complex to parse. It is defined in Listing 20.

KeyValue parses a single *key = value*. The *key* is an identifier with *value* holding a token – lines 18 to 20. The *value* part is optional for boolean options. Thus, line 29 checks if a *value* part is present. A *value* will be present if the end of the stream has not been reached or if the next token is not a comma (,) – indicating the next key-value option. If no *value* is given, lines 30 to 34 returns the key from the parse function, with "default" as the value. Lines 38 to 42 parse the rest of the stream if a value is present and returns the *KeyValue*.

The ***OptionsAttribute*** is simple, as seen in Listing 21. It parses the *#[list-of-KeyValues]* stream, as defined in lines 15 to 19. The *bracketed*[19] helper from *syn* is used in line 29 to get the stream inside a square-bracket group.

Listing 22 shows the ***SimpleType*** model. It consists of an optional *dyn* keyword followed by a type identifier (*Ident*). *ToTokens* is implemented on *SimpleType* to be able to use it in *quote* later, lines 30 to 35.

AnnotatedType will combine a generic type with an optional *OptionsAttribute*, as seen in Listing 19.

Lastly, ***VisitorFunction*** – the input to the Visitor macro – will be a comma punctuated list of *AnnotatedTypes*, as shown in Listing 27. The generic *T* in *AnnotatedType* is set to *SimpleType* in line 30.

VisitorFunction makes use of a private *Options* struct – lines 120 to 124 – to dissect the options passed to each type. In lines 129 to 131, *Options* defaults to:

- Creating the default trait method that will call the helper function.
- Creating a helper function.
- The helper function being empty.

Each option passed to the type is iterated on, in line 134. If the option has the *no_default* key, then creating a default trait method for the type is turned off in lines 136 to 139. The option *helper_tmpl = false* turns off creating a helper function on 145, while the option *helper_tmpl = {template}* activates a custom helper template in line 149. Line 151 ignores anything else passed to the *helper_tmpl* option.

The *expand* method for the *VisitorFunction* is defined in line 44. As defined in Section 5.2.3, the Visitor implementation will consist of three parts:

[18] https://docs.rs/syn/1.0.48/syn/struct.PathSegment.html
[19] https://docs.rs/syn/1.0.48/syn/macro.bracketed.html

1. The *Visitor* trait. A *visit_<type_name>* trait method needs to be made for each type passed to the macro.

2. A helper function for each type.

3. A *Visitable* implementation for each type.

Lines 46 to 48 define variables to hold each of these three parts in a list. Line 51 starts an iterator over each type passed to the *Visit* macro. Line 52 gets the type name to construct a function name in line 54 – *format_ident*[20] is a *quote* macro for creating an *Ident* token. Line 53 creates a local variable of the type for use in *quote* – why *SimpleType* implemented *ToTokens*. The private *Options* is used in line 55 to dissect the attribute options.

Line 58 uses the *no_default* option to decide if the trait method should not have a default implementation. Otherwise, a default implementation that calls the helper function is created.

The macro will write the helper by default. Line 71 checks if the client has not turned it off. If the client supplied a custom helper template, then line 72 extracts it into *inner* and writes the helper function with *inner* as the function block. Otherwise, if no custom template was supplied, lines 83 to 88 create an empty helper function.

Lastly, the *Visitable* trait is implemented for the type in lines 94 to 100.

Once iterating over all the types is done, lines 104 to 115 write the macro output. The list for each of the three parts is expanded without a separator in lines 106, 109, and 114.

Definition: The function-like – as was defined in Section 3.3.1 – *visitor* macro is shown in Listing 26, lines 32 to 36. It uses the *VisitorFunction* model defined in Listing 27 to parse the macro input in line 33. The model is expanded in line 35.

6 Conclusion

This report showed that Rust's metaprogramming is capable of implementing design patterns that are identical to manual implementations. The Abstract Factory pattern was first implemented using Rust attribute macros. The macro definition led to a foundation that can be used to implement other design pattern macros. The foundation uses a model to parse the macro input into using *syn*. This model is then expanded to the pattern's implementation using *quote*. The foundation was applied to create a function-like Rust macro to implement the Visitor pattern. The same foundation can thus be applied to other design patterns.

Both macros have some limitations that future research can explore. The Abstract Factory implementation uses dynamic dispatch, which has a performance trade-off. "Abstract return types"[21] may be a solution to this problem. Visitor's implementation requires the client to supply the traversal code in the *helper_tmpl* option manually. However, Rust macros can read the file system. Thus, it might be possible for the macro the read all the files for a module and build a composition graph of all the module's types. The traversal code can then be automatically written by the macro – effectively reducing the visitor macro call to one line. Lastly, the Visitor has no global option to apply *no_default* or *helper_tmpl* to all its types but will require repeating the option on every type. The macro currently uses an outer attribute[22]on each type to set the option. An inner attribute[22] can be used on the macro for global options.

The *iterpolate_traits* macro creates code that almost forwards to a sub-method by mapping one type to another, close to the Decorator design pattern. Future research can investigate how

[20]https://docs.rs/quote/1.0.7/quote/macro.format_ident.html

[21]https://www.ncameron.org/blog/abstract-return-types-aka–impl-trait-/

a new macro will need to change from *interpolate_traits* to implement Decorator. The same goes for the Proxy, Adapter, and some parts of Mediator design patterns.

A study of automating this process will also be interesting. This will be an external metaprogram that identifies repeated patterns in a library, across libraries, or on whole repositories. It will then create a Rust macro for each repeated pattern and change the identified code sections to use the macro instead.

References

[AC12] Martin Abadi and Luca Cardelli. *A theory of objects*, chapter 2, page 18. Springer Science & Business Media, 2012.

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

[Ang17] Wisnu Anggoro. *Learning C++ Functional Programming*, chapter 6, pages 171–199. Packt Publishing Ltd, 2017.

[B+99] Alan Bawden et al. Quasiquotation in lisp. In *PEPM*, pages 4–12. Citeseer, 1999.

[BJ12] Aleksandar Bulajic and Slobodan Jovanovic. An approach to reducing complexity in abstract factory design pattern. *Journal of Emerging Trends in Computing and Information Sciences*, 3(10), 2012.

[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.

[CK03] David Carrington and S-K Kim. Teaching software design with open source software. In *33rd Annual Frontiers in Education, 2003. FIE 2003.*, volume 3, pages S1C–9. IEEE, 2003.

[DHV95] Karel Driesen, Urs Hölzle, and Jan Vitek. Message dispatch on pipelined processors. In *European Conference on Object-Oriented Programming*, pages 253–282. Springer, 1995.

[Fla06] David Flanagan. *JavaScript: The Definitive Guide*, chapter 11, pages 171–172. O'Reilly 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.

[22] https://doc.rust-lang.org/reference/attributes.html

[GY11] Davoud Keshvari Ghourbanpour and Mohhamd Hossien Yektaie. Towards pattern-based refactoring: Abstract factory. *International Journal of Advanced Research in Computer Science*, 2(3), 2011.

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

[Hin13] Konrad Hinsen. A glimpse of the future of scientific programming. *Computing in Science & Engineering*, 15(1):84–88, 2013.

[HSG18] Zhewei Hu, Yang Song, and Edward F Gehringer. Open-source software in class: students' common mistakes. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, pages 40–48, 2018.

[iee09] Ieee standard for information technology–systems design–software design descriptions. *IEEE STD 1016-2009*, pages 3–4, 2009.

[Ker05] Joshua Kerievsky. *Refactoring to patterns*. Pearson Deutschland GmbH, 2005.

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

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

[LS15] Yannis Lilis and Anthony Savidis. An integrated implementation framework for compile-time metaprogramming. *Software: Practice and Experience*, 45(6):727–763, 2015.

[LS19] Yannis Lilis and Anthony Savidis. A survey of metaprogramming languages. *ACM Computing Surveys (CSUR)*, 52(6):1–39, 2019.

[LZ95] Arthur H Lee and Joseph L Zachary. Reflections on metaprogramming. *IEEE Transactions on Software Engineering*, 21(11):883–893, 1995.

[Mal09] Davender S Malik. *Data structures using C++*, chapter 1, 2, 3, pages 4, 33, 79, 170. Cengage Learning, 2009.

[Mar06] Alex Martelli. *Python in a Nutshell (In a Nutshell (O'Reilly))*, chapter 13, pages 269–272. O'Reilly 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.

[MSD15] Stefan Marr, Chris Seaton, and Stéphane Ducasse. Zero-overhead metaprogramming: Reflection and metaobject protocols fast and without compromises. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 545–554, 2015.

[Nyk12] AA Nykonenko. Creational design patterns in computational linguistics: Factory method, prototype, singleton. *Cybernetics and Systems Analysis*, 48(1):138–145, 2012.

[PB02] Benjamin C Pierce and C Benjamin. *Types and programming languages*, chapter 18, page 226. MIT press, 2002.

[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.

[She01] Tim Sheard. Accomplishments and research challenges in meta-programming. In *International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44. Springer, 2001.

[SJB15] John W Satzinger, Robert B Jackson, and Stephen D Burd. *Systems analysis and design in a changing world*, chapter 1, 2, 13, pages 5, 44, 428. Cengage learning, 2015.

[Son98] Sabine Sonnentag. Expertise in professional software design: A process study. *Journal of applied psychology*, 83(5):703, 1998.

[Ste15] Rod Stephens. *Beginning software engineering*, chapter 13, page 284. John Wiley & Sons, 2015.

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

[Tah04] Walid Taha. A gentle introduction to multi-stage programming. In *Domain-Specific Program Generation*, pages 30–50. Springer, 2004.

[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.

[VPG+18] Brice Videau, Kevin Pouget, Luigi Genovese, Thierry Deutsch, Dimitri Komatitsch, Frédéric Desprez, and Jean-François Méhaut. Boast: A metaprogramming framework to produce portable and efficient computing kernels for hpc applications. *The International Journal of High Performance Computing Applications*, 32(1):28–44, 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 1, 10, 15, pages 47, 572–583, 892–893. Pearson Education Ltd, Edinburgh Gate, Harlow, England, 2015.

[Zhu05] Hong Zhu. *Software design methodology: From principles to architectural styles.* Elsevier, 2005.

A Appendix

Appendix only document can be found here: `https://github.com/chesedo/cos-700/blob/master/Report-Appendix.pdf`

Alternatively, the code is located here: `https://github.com/chesedo/cos-700/tree/master/Code/Rust` . The file path is at the top of each listing. The *.highlighted* listings contain embedded LaTeX command or are used to correct formatting issues. Each file will have a non-highlighted version.

A.1 Manual Implementations

15: macro-client/src/gui/brand_elements.rs

```rust
use super::elements;

pub struct BrandButton {
    name: String,
    text: String,
}

impl elements::Element for BrandButton {
    fn new(name: String) -> Self {
        BrandButton {
            name,
            text: String::new(),
        }
    }
    fn get_name(&self) -> &str {
        &self.name
    }
}

impl elements::Button for BrandButton {
    fn click(&self) {
        unimplemented!()
    }
    fn get_text(&self) -> &str {
        &self.text
    }
    fn set_text(&mut self, text: String) {
        self.text = text;
    }
}

pub struct BrandInput {
    name: String,
    input: String,
}

impl elements::Element for BrandInput {
    fn new(name: String) -> Self {
        BrandInput {
            name,
            input: String::new(),
        }
    }
    fn get_name(&self) -> &str {
        &self.name
    }
}

impl elements::Input for BrandInput {
    fn get_input(&self) -> String {
        self.input.to_owned()
    }
    fn set_input(&mut self, input: String) {
```

```
        self.input = input
    }
}
```

16: macro-client/src/gui/elements.rs

```
pub trait Element {
    fn new(name: String) -> Self
    where
        Self: Sized;
    fn get_name(&self) -> &str;
}

pub trait Button: Element {
    fn click(&self);
    fn get_text(&self) -> &str;
    fn set_text(&mut self, text: String);
}

pub trait Input: Element {
    fn get_input(&self) -> String;
    fn set_input(&mut self, input: String);
}

pub enum Child {
    Button(Box<dyn Button>),
    Input(Box<dyn Input>),
}

pub struct Window {
    name: String,
    children: Vec<Child>,
}

impl Window {
    pub fn add_child(&mut self, child: Child) -> &mut Self {
        self.children.push(child);

        self
    }
    pub fn get_children(&self) -> &[Child] {
        &self.children
    }
}

impl Element for Window {
    fn new(name: String) -> Self {
        Window {
            name,
            children: Vec::new(),
        }
    }
    fn get_name(&self) -> &str {
        &self.name
    }
}

impl From<Box<dyn Button>> for Child {
    fn from(button: Box<dyn Button>) -> Self {
        Child::Button(button)
    }
}

impl From<Box<dyn Input>> for Child {
```

```
    fn from(input: Box<dyn Input>) -> Self {
        Child::Input(input)
    }
}
```

17: macro-client/src/abstract_factory_hand.rs

```
#[allow(unused_imports)]
use macro_patterns::{abstract_factory, interpolate_traits};
use std::fmt::{Display, Formatter, Result};

use crate::gui::{
    brand_elements,
    elements::{Button, Element, Input, Window},
};

pub trait Factory<T: Element + ?Sized> {
    fn create(&self, name: String) -> Box<T>;
}

pub trait AbstractGuiFactory:
    Display + Factory<dyn Button> + Factory<dyn Input> + Factory<Window>
{
}

struct BrandFactory {}
impl AbstractGuiFactory for BrandFactory {}
impl Factory<dyn Button> for BrandFactory {
    fn create(&self, name: String) -> Box<dyn Button> {
        Box::new(brand_elements::BrandButton::new(name))
    }
}
impl Factory<dyn Input> for BrandFactory {
    fn create(&self, name: String) -> Box<dyn Input> {
        Box::new(brand_elements::BrandInput::new(name))
    }
}
impl Factory<Window> for BrandFactory {
    fn create(&self, name: String) -> Box<Window> {
        Box::new(Window::new(name))
    }
}

impl Display for BrandFactory {
    fn fmt(&self, f: &mut Formatter) -> Result {
        f.write_str("BrandFactory GUI creator")
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn button_factory() {
        let factory = BrandFactory {};
        let actual: Box<dyn Button> = factory.create(String::from("Button"));

        assert_eq!(actual.get_name(), "Button");
    }

    #[test]
    fn window_factory() {
        let factory = BrandFactory {};
```

```
        let actual: Box<Window> = factory.create(String::from("Window"));

        assert_eq!(actual.get_name(), "Window");
    }
}
```

18: macro-client/src/visitor_hand.rs

```rust
#[allow(unused_imports)]
use macro_patterns::visitor;
use std::fmt;

use crate::gui::elements::{Button, Child, Input, Window};

// Abstract visitor for `Button`, `Input` and `Window`
pub trait Visitor {
    fn visit_button(&mut self, button: &dyn Button) {
        visit_button(self, button)
    }
    fn visit_input(&mut self, input: &dyn Input) {
        visit_input(self, input)
    }
    fn visit_window(&mut self, window: &Window) {
        visit_window(self, window)
    }
}

// Helper functions for transversing a hierarchical data structure
pub fn visit_button<V>(_visitor: &mut V, _button: &dyn Button)
where
    V: Visitor + ?Sized,
{
}
pub fn visit_input<V>(_visitor: &mut V, _input: &dyn Input)
where
    V: Visitor + ?Sized,
{
}
pub fn visit_window<V>(visitor: &mut V, window: &Window)
where
    V: Visitor + ?Sized,
{
    window.get_children().iter().for_each(child {
        match child {
            Child::Button(button) => visitor.visit_button(button.as_ref()),
            Child::Input(input) => visitor.visit_input(input.as_ref()),
        };
    });
}

// Extends each element with the reflective `apply` method
trait Visitable {
    fn apply(&self, visitor: &mut dyn Visitor);
}

impl Visitable for dyn Button {
    fn apply(&self, visitor: &mut dyn Visitor) {
        visitor.visit_button(self);
    }
}
impl Visitable for dyn Input {
    fn apply(&self, visitor: &mut dyn Visitor) {
        visitor.visit_input(self);
    }
```

```rust
}
impl Visitable for Window {
    fn apply(&self, visitor: &mut dyn Visitor) {
        visitor.visit_window(self);
    }
}

struct NameVisitor {
    names: Vec<String>,
}

impl NameVisitor {
    #[allow(dead_code)]
    pub fn new() -> Self {
        NameVisitor { names: Vec::new() }
    }
}
impl Visitor for NameVisitor {
    fn visit_button(&mut self, button: &dyn Button) {
        self.names.push(button.get_name().to_string());
    }
    fn visit_input(&mut self, input: &dyn Input) {
        self.names
            .push(format!("{} ({})", input.get_name(), input.get_input()));
    }
}

impl fmt::Display for NameVisitor {
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "{}", self.names.join(", "))
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use crate::gui::brand_elements;
    use crate::gui::elements::{Child, Element};

    type Result = std::result::Result<(), Box<dyn std::error::Error>>;

    #[test]
    fn visit_button() {
        let button: &dyn Button = &brand_elements::BrandButton::new(String::from("Some Button"));

        let mut visitor = NameVisitor::new();

        button.apply(&mut visitor);

        assert_eq!(visitor.to_string(), "Some Button");
    }

    #[test]
    fn visit_window() -> Result {
        let mut window = Box::new(Window::new(String::from("Holding window")));
        let button: Box<dyn Button> = Box::new(brand_elements::BrandButton::new(String::from(
            "Some Button",
        )));
        let mut input: Box<dyn Input> =
            Box::new(brand_elements::BrandInput::new(String::from("Some Input")));

        input.set_input(String::from("John Doe"));

        window
```

```
            .add_child(Child::from(button))
            .add_child(Child::from(input));

        let mut visitor = NameVisitor::new();

        window.apply(&mut visitor);

        assert_eq!(visitor.to_string(), "Some Button, Some Input (John Doe)");

        Ok(())
    }
}
```

A.1.1 macro-lib

19: macro-lib/src/annotated_type.rs

```rust
use crate::options_attribute::OptionsAttribute;
use syn::parse::{Parse, ParseStream, Result};
use syn::{Token, Type};

/// Holds a type that is optionally annotated with key-value options.
/// An acceptable stream will have the following form:
/// ```text
/// #[option1 = value1, option2 = value2]
/// SomeType
/// ```
///
/// The outer attribute (hash part) is optional.
/// `SomeType` will be parsed to `T`.
#[derive(Eq, PartialEq, Debug)]
pub struct AnnotatedType<T = Type> {
    pub attrs: OptionsAttribute,
    pub inner_type: T,
}

/// Make AnnotatedType parsable from token stream
impl<T: Parse> Parse for AnnotatedType<T> {
    fn parse(input: ParseStream) -> Result<Self> {
        // Parse attribute options if the next token is a hash
        if input.peek(Token![#]) {
            return Ok(AnnotatedType {
                attrs: input.parse()?,
                inner_type: input.parse()?,
            });
        };

        // Parse without attribute options
        Ok(AnnotatedType {
            attrs: Default::default(),
            inner_type: input.parse()?,
        })
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use pretty_assertions::assert_eq;
    use syn::{parse_quote, parse_str, TypeTraitObject};

    type Result = std::result::Result<(), Box<dyn std::error::Error>>;

```

```rust
    #[test]
    fn parse() -> Result {
        let actual: AnnotatedType = parse_quote! {
            #[no_default]
            i32
        };
        let expected = AnnotatedType {
            attrs: parse_str("#[no_default]")?,
            inner_type: parse_str("i32")?,
        };

        assert_eq!(actual, expected);
        Ok(())
    }

    #[test]
    fn parse_simple_type() -> Result {
        let actual: AnnotatedType = parse_quote! {
            Button
        };
        let expected = AnnotatedType {
            attrs: Default::default(),
            inner_type: parse_str("Button")?,
        };

        assert_eq!(actual, expected);
        Ok(())
    }

    #[test]
    fn parse_trait_bounds() -> Result {
        let actual: AnnotatedType<TypeTraitObject> = parse_quote! {
            #[no_default]
            dyn Button
        };
        let expected = AnnotatedType::<TypeTraitObject> {
            attrs: parse_str("#[no_default]")?,
            inner_type: parse_str("dyn Button")?,
        };

        assert_eq!(actual, expected);
        Ok(())
    }

    #[test]
    #[should_panic(expected = "unexpected end of input")]
    fn missing_type() {
        parse_str::<AnnotatedType>("#[no_default]").unwrap();
    }
}
```

20: macro-lib/src/key_value.rs

```rust
use proc_macro2::TokenTree;
use syn::parse::{Parse, ParseStream, Result};
use syn::{parse_str, Ident, Token};

/// Holds a single key value attribute, with the value being optional.
/// Streams in the following form will be parsed:
/// ```text
/// key = value
/// ```
///
/// The `value` is optional.
```

```rust
/// Thus, the following is also valid.
/// ```text
/// key
/// ```
#[derive(Debug)]
pub struct KeyValue {
    pub key: Ident,
    pub equal_token: Token![=],
    pub value: TokenTree,
}

/// Make KeyValue parsable from a token stream
impl Parse for KeyValue {
    fn parse(input: ParseStream) -> Result<Self> {
        let key = input.parse()?;

        // Stop if optional value is not given
        if input.is_empty()  input.peek(Token![,]) {
            return Ok(KeyValue {
                key,
                equal_token: Default::default(),
                value: parse_str("default")?,
            });
        }

        // Parse with value
        Ok(KeyValue {
            key,
            equal_token: input.parse()?,
            value: input.parse()?,
        })
    }
}

// Just for testing
impl PartialEq for KeyValue {
    fn eq(&self, other: &Self) -> bool {
        self.key == other.key && format!("{}", self.value) == format!("{}", other.value)
    }
}
impl Eq for KeyValue {}

#[cfg(test)]
mod tests {
    use super::*;
    use pretty_assertions::assert_eq;
    use syn::parse_str;

    type Result = std::result::Result<(), Box<dyn std::error::Error>>;

    #[test]
    fn parse() -> Result {
        let actual: KeyValue = parse_str("some_key = \"value\"")?;
        let expected = KeyValue {
            key: parse_str("some_key")?,
            equal_token: Default::default(),
            value: parse_str("\"value\"")?,
        };

        assert_eq!(actual, expected);
        Ok(())
    }

    #[test]
```

```rust
    fn parse_missing_value() -> Result {
        let actual: KeyValue = parse_str("bool_key")?;
        let expected = KeyValue {
            key: parse_str("bool_key")?,
            equal_token: Default::default(),
            value: parse_str("default")?,
        };

        assert_eq!(actual, expected);
        Ok(())
    }

    #[test]
    fn parse_attribute_item_complex_stream() -> Result {
        let actual: KeyValue = parse_str("tmpl = {trait To {};}")?;
        let expected = KeyValue {
            key: parse_str("tmpl")?,
            equal_token: Default::default(),
            value: parse_str("{trait To {};}")?,
        };

        assert_eq!(actual, expected);
        Ok(())
    }

    // Test extra input after a value stream is ignored
    #[test]
    #[should_panic(expected = "expected token")]
    fn parse_attribute_item_complex_stream_extra() {
        parse_str::<KeyValue>("tmpl = {trait To {};}, key").unwrap();
    }

    #[test]
    #[should_panic(expected = "expected identifier")]
    fn missing_key() {
        parse_str::<KeyValue>("= true").unwrap();
    }

    #[test]
    #[should_panic(expected = "expected `=`")]
    fn missing_equal_sign() {
        parse_str::<KeyValue>("key  value").unwrap();
    }

    #[test]
    #[should_panic(expected = "expected token tree")]
    fn missing_value() {
        parse_str::<KeyValue>("key = ").unwrap();
    }
}
```

21: macro-lib/src/options_attribute.highlighted.rs

```rust
use crate::key_value::KeyValue;
use syn::parse::{Parse, ParseStream, Result};
use syn::punctuated::Punctuated;
use syn::{bracketed, token, Token};

/// Holds an outer attribute filled with key-value options.
/// Streams in the following form will be parsed successfully:
/// ```text
/// #[key1 = value1, bool_key2, key3 = value]
/// ```
///
```

```rust
/// The value part of an option is optional.
/// Thus, `bool_key2` will have the value `default`.
#[derive(Eq, PartialEq, Debug, Default)]
pub struct OptionsAttribute {
    pub pound_token: Token![#],
    pub bracket_token: token::Bracket,
    pub options: Punctuated<KeyValue, Token![,]>,
}

/// Make OptionsAttribute parsable from a token stream
impl Parse for OptionsAttribute {
    fn parse(input: ParseStream) -> Result<Self> {
        // Used to hold the stream inside square brackets
        let content;

        Ok(OptionsAttribute {
            pound_token: input.parse()?,
            bracket_token: bracketed!(content in input), // Use `syn` to extract the stream inside the square bracket group
            options: content.parse_terminated(KeyValue::parse)?,
        })
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use pretty_assertions::assert_eq;
    use syn::parse_str;

    type Result = std::result::Result<(), Box<dyn std::error::Error>>;

    #[test]
    fn parse() -> Result {
        let actual: OptionsAttribute =
            parse_str("#[tmpl = {trait To {};}, no_default, other = Top]")?;
        let mut expected = OptionsAttribute {
            pound_token: Default::default(),
            bracket_token: Default::default(),
            options: Punctuated::new(),
        };

        expected.options.push(parse_str("tmpl = {trait To {};}")?);
        expected.options.push(parse_str("no_default")?);
        expected.options.push(parse_str("other = Top")?);

        assert_eq!(actual, expected);
        Ok(())
    }
}
```

22: macro-lib/src/simple_type.highlighted.rs

```rust
use proc_macro2::TokenStream;
use quote::ToTokens;
use syn::parse::{Parse, ParseStream, Result};
use syn::{Ident, Token};

/// Holds a simple type that is optionally annotated as `dyn`.
/// The following is an example of its input stream:
/// ```text
/// dyn SomeType
/// ```
///
/// The `dyn` keyword is optional.
```

```rust
#[derive(Eq, PartialEq, Debug)]
pub struct SimpleType {
    pub dyn_token: Option<Token![dyn]>,
    pub ident: Ident,
}

/// Make SimpleType parsable from token stream
impl Parse for SimpleType {
    fn parse(input: ParseStream) -> Result<Self> {
        Ok(SimpleType {
            dyn_token: input.parse()?,
            ident: input.parse()?,
        })
    }
}

/// Turn SimpleType back into a token stream
impl ToTokens for SimpleType {
    fn to_tokens(&self, tokens: &mut TokenStream) {
        self.dyn_token.to_tokens(tokens);
        self.ident.to_tokens(tokens);
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use pretty_assertions::assert_eq;
    use quote::quote;
    use syn::parse_str;

    type Result = std::result::Result<(), Box<dyn std::error::Error>>;

    #[test]
    fn parse() -> Result {
        let actual: SimpleType = parse_str("dyn Button")?;
        let expected = SimpleType {
            dyn_token: Some(Default::default()),
            ident: parse_str("Button")?,
        };

        assert_eq!(actual, expected);
        Ok(())
    }

    #[test]
    fn parse_without_dyn() -> Result {
        let actual: SimpleType = parse_str("Button")?;
        let expected = SimpleType {
            dyn_token: None,
            ident: parse_str("Button")?,
        };

        assert_eq!(actual, expected);
        Ok(())
    }

    #[test]
    #[should_panic(expected = "expected identifier")]
    fn missing_type() {
        parse_str::<SimpleType>("dyn").unwrap();
    }

    #[test]
```

```
    fn to_tokens() -> Result {
        let input = SimpleType {
            dyn_token: Some(Default::default()),
            ident: parse_str("Button")?,
        };
        let actual = quote! { #input };
        let expected: TokenStream = parse_str("dyn Button")?;

        assert_eq!(format!("{}", actual), format!("{}", expected));
        Ok(())
    }
}
```

23: macro-lib/src/token_stream_utils.rs

```
use proc_macro2::{Group, TokenStream, TokenTree};
use quote::{ToTokens, TokenStreamExt};
use std::collections::HashMap;
use syn::punctuated::Punctuated;

/// Trait for tokens that can replace interpolation markers
pub trait Interpolate {
    /// Take a token stream and replace interpolation markers with their actual values into a new stream
    fn interpolate(&self, stream: TokenStream) -> TokenStream;
}

/// Make a Punctuated list interpolatible if it holds interpolatible types
impl<T: Interpolate, P> Interpolate for Punctuated<T, P> {
    fn interpolate(&self, stream: TokenStream) -> TokenStream {
        self.iter()
            .fold(TokenStream::new(), mut implementations, t {
                implementations.extend(t.interpolate(stream.clone()));
                implementations
            })
    }
}

/// Replace the interpolation markers in a token stream with a specific text
/// Thus, if `stream` is "let a: TRAIT;" and `replacements` has the key "TRAIT" with value "Button", then this will return
 ↪  a stream with "let a: Button;".
pub fn interpolate(
    stream: TokenStream,
    replacements: &HashMap<&str, &dyn ToTokens>,
) -> TokenStream {
    let mut new = TokenStream::new();

    // Loop over each token in the stream
    // `Literal`, `Punct`, and `Group` are kept as is
    for token in stream.into_iter() {
        match token {
            TokenTree::Literal(literal) => new.append(literal),
            TokenTree::Punct(punct) => new.append(punct),
            TokenTree::Group(group) => {
                // Recursively interpolate the stream in group
                let mut new_group =
                    Group::new(group.delimiter(), interpolate(group.stream(), replacements));
                new_group.set_span(group.span());

                new.append(new_group);
            }
            TokenTree::Ident(ident) => {
                let ident_str: &str = &ident.to_string();

                // Check if identifier is in the replacement set
```

```
                if let Some(value) = replacements.get(ident_str) {
                    // Replace with replacement value
                    value.to_tokens(&mut new);

                    continue;
                }

                // Identifier did not match, so copy as is
                new.append(ident);
            }
        }
    }

    new
}

#[cfg(test)]
mod tests {
    use super::*;
    use crate::trait_specifier::TraitSpecifier;
    use pretty_assertions::assert_eq;
    use quote::quote;
    use syn::{parse_str, Ident, Token, Type};

    type Result = std::result::Result<(), Box<dyn std::error::Error>>;

    #[test]
    fn complete_replacements() -> Result {
        let input = quote! {
            let VAR: TRAIT = if true {
                CONCRETE{}
            } else {
                Alternative{}
            }
        };

        let expected = quote! {
            let var: abstract_type = if true {
                concrete{}
            } else {
                Alternative{}
            }
        };

        let mut r: HashMap<&str, &dyn ToTokens> = HashMap::new();
        let v: Ident = parse_str("var")?;
        let a: Type = parse_str("abstract_type")?;
        let c: Type = parse_str("concrete")?;

        r.insert("VAR", &v);
        r.insert("TRAIT", &a);
        r.insert("CONCRETE", &c);

        assert_eq!(
            format!("{}", &interpolate(input, &r)),
            format!("{}", expected)
        );

        Ok(())
    }

    /// Partial replacements should preverse the uninterpolated identifiers
    #[test]
    fn partial_replacements() -> Result {
```

```
        let input: TokenStream = parse_str("let a: TRAIT = OTHER;")?;
        let expected: TokenStream = parse_str("let a: Display = OTHER;")?;

        let mut r: HashMap<&str, &dyn ToTokens> = HashMap::new();
        let t: Type = parse_str("Display")?;
        r.insert("TRAIT", &t);

        assert_eq!(
            format!("{}", interpolate(input, &r)),
            format!("{}", expected)
        );

        Ok(())
    }

    #[test]
    fn interpolate_on_punctuated() -> Result {
        let mut traits: Punctuated<TraitSpecifier, Token![,]> = Punctuated::new();

        traits.push(parse_str("IButton => BigButton")?);
        traits.push(parse_str("IWindow => MinimalWindow")?);

        let input = quote! {
            let _: TRAIT = CONCRETE{};
        };
        let expected = quote! {
            let _: IButton = BigButton{};
            let _: IWindow = MinimalWindow{};
        };

        assert_eq!(
            format!("{}", traits.interpolate(input)),
            format!("{}", expected)
        );

        Ok(())
    }
}
```

24: macro-lib/src/trait_specifier.highlighted.rs

```
use crate::token_stream_utils::{interpolate, Interpolate};
use proc_macro2::TokenStream;
use quote::ToTokens;
use std::collections::HashMap;
use syn::parse::{Parse, ParseStream, Result};
use syn::{Token, Type};

/// Type that holds an abstract type and how it will map to a concrete type.
/// An acceptable stream will have the following form:
/// ```text
/// trait => concrete
/// ```
#[derive(Eq, PartialEq, Debug)]
pub struct TraitSpecifier {
    pub abstract_trait: Type,
    pub arrow_token: Token![=>],
    pub concrete: Type,
}

/// Make TraitSpecifier parsable from a token stream
impl Parse for TraitSpecifier {
    fn parse(input: ParseStream) -> Result<Self> {
        Ok(TraitSpecifier {
```

```
            abstract_trait: input.parse()?,
            arrow_token: input.parse()?,
            concrete: input.parse()?,
        })
    }
}

/// Make TraitSpecifier interpolatible
impl Interpolate for TraitSpecifier {
    fn interpolate(&self, stream: TokenStream) -> TokenStream {
        let mut replacements: HashMap<_, &dyn ToTokens> = HashMap::new();

        // Replace each "TRAIT" with the absract trait
        replacements.insert("TRAIT", &self.abstract_trait);

        // Replace each "CONCRETE" with the concrete type
        replacements.insert("CONCRETE", &self.concrete);

        interpolate(stream, &replacements)
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use macro_test_helpers::reformat;
    use pretty_assertions::assert_eq;
    use quote::quote;
    use syn::parse_str;

    type Result = std::result::Result<(), Box<dyn std::error::Error>>;

    #[test]
    fn parse() -> Result {
        let actual: TraitSpecifier = parse_str("abstract_trait => concrete")?;
        let expected = TraitSpecifier {
            abstract_trait: parse_str("abstract_trait")?,
            arrow_token: Default::default(),
            concrete: parse_str("concrete")?,
        };

        assert_eq!(actual, expected);
        Ok(())
    }

    #[test]
    #[should_panic(expected = "expected one of")]
    fn missing_trait() {
        parse_str::<TraitSpecifier>("=> concrete").unwrap();
    }

    #[test]
    #[should_panic(expected = "expected `=>`")]
    fn missing_arrow_joiner() {
        parse_str::<TraitSpecifier>("IButton -> RoundButton").unwrap();
    }

    #[test]
    #[should_panic(expected = "unexpected end of input")]
    fn missing_concrete() {
        parse_str::<TraitSpecifier>("abstract_trait => ").unwrap();
    }

    #[test]
```

```
    fn interpolate() -> Result {
        let input = quote! {
            impl Factory<TRAIT> for Gnome {
                fn create(&self) -> CONCRETE {
                    CONCRETE{}
                }
            }
        };
        let expected = quote! {
            impl Factory<abstract_trait> for Gnome {
                fn create(&self) -> concrete {
                    concrete{}
                }
            }
        };
        let specifier = TraitSpecifier {
            abstract_trait: parse_str("abstract_trait")?,
            arrow_token: Default::default(),
            concrete: parse_str("concrete")?,
        };

        assert_eq!(reformat(&specifier.interpolate(input)), reformat(&expected));

        Ok(())
    }
}
```

A.1.2 macro-patterns

25: macro-patterns/src/abstract_factory.highlighted.rs

```
use proc_macro2::TokenStream;
use quote::quote;
use syn::parse::{Parse, ParseStream, Result};
use syn::punctuated::Punctuated;
use syn::{parse_quote, ItemTrait, Token, Type, TypeParamBound};

/// Holds the tokens for the attributes passed to an AbstractFactory attribute macro
/// Expects input in the following format
/// ```text
/// some_factory_method_trait, type_1, type_2, ... , type_n
/// ```
///
/// `some_factory_method_trait` needs to be created by the client.
/// It should have one generic type.
/// Every `type_1` ... `type_n` will be filled into this generic type.
#[derive(Eq, PartialEq, Debug)]
pub struct AbstractFactoryAttribute {
    factory_trait: Type,
    sep: Token![,],
    types: Punctuated<Type, Token![,]>,
}

/// Make AbstractFactoryAttribute parsable
impl Parse for AbstractFactoryAttribute {
    fn parse(input: ParseStream) -> Result<Self> {
        Ok(AbstractFactoryAttribute {
            factory_trait: input.parse()?,
            sep: input.parse()?,
            types: input.parse_terminated(Type::parse)?,
        })
    }
}
```

```rust

impl AbstractFactoryAttribute {
    /// Add factory super traits to an `ItemTrait` to turn the `ItemTrait` into an Abstract Factory
    pub fn expand(&self, input_trait: &mut ItemTrait) -> TokenStream {
        // Build all the super traits
        let factory_traits: Punctuated<TypeParamBound, Token![+]> = {
            let types = self.types.iter();
            let factory_name = &self.factory_trait;

            parse_quote! {
                #(#factory_name<#types>)+*
            }
        };

        // Append extra factory super traits
        input_trait.supertraits.extend(factory_traits);

        quote! {
            #input_trait
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use macro_test_helpers::reformat;
    use syn::parse_str;

    type Result = std::result::Result<(), Box<dyn std::error::Error>>;

    mod abstract_factory {
        use super::*;
        use pretty_assertions::assert_eq;

        #[test]
        fn parse() -> Result {
            let actual: AbstractFactoryAttribute = parse_str("Factory, u32, i64")?;
            let mut expected_types = Punctuated::new();

            expected_types.push(parse_str("u32")?);
            expected_types.push(parse_str("i64")?);

            assert_eq!(
                actual,
                AbstractFactoryAttribute {
                    factory_trait: parse_str("Factory")?,
                    sep: Default::default(),
                    types: expected_types,
                }
            );

            Ok(())
        }

        #[test]
        #[should_panic(expected = "expected `,`")]
        fn missing_types() {
            parse_str::<AbstractFactoryAttribute>("Factory").unwrap();
        }

        #[test]
        fn expand() -> Result {
            let mut t = parse_str::<ItemTrait>("pub trait Abstraction<T>: Display + Extend<T> {}")?;
```

```
            let mut input_types = Punctuated::new();

            input_types.push(parse_str("u32")?);
            input_types.push(parse_str("i64")?);

            let actual = &AbstractFactoryAttribute {
                factory_trait: parse_str("Factory")?,
                sep: Default::default(),
                types: input_types,
            }
            .expand(&mut t);

            assert_eq!(
                reformat(&actual),
                "pub trait Abstraction<T>: Display + Extend<T> + Factory<u32> + Factory<i64> {}\n"
            );

            Ok(())
        }
    }
}
```

26: macro-patterns/src/lib.highlighted.rs

```
mod abstract_factory;
mod visitor;

extern crate proc_macro;

use proc_macro::TokenStream;
use syn::punctuated::Punctuated;
use syn::{parse_macro_input, ItemTrait, Token};

use abstract_factory::AbstractFactoryAttribute;
use macro_lib::token_stream_utils::Interpolate;
use macro_lib::TraitSpecifier;
use visitor::VisitorFunction;

#[proc_macro_attribute]
pub fn abstract_factory(tokens: TokenStream, trait_expr: TokenStream) -> TokenStream {
    let mut input = parse_macro_input!(trait_expr as ItemTrait);
    let attributes = parse_macro_input!(tokens as AbstractFactoryAttribute);

    attributes.expand(&mut input).into()
}

#[proc_macro_attribute]
pub fn interpolate_traits(tokens: TokenStream, concrete_impl: TokenStream) -> TokenStream {
    let attributes =
        parse_macro_input!(tokens with Punctuated::<TraitSpecifier, Token![,]>::parse_terminated);

    attributes.interpolate(concrete_impl.into()).into()
}

#[proc_macro]
pub fn visitor(tokens: TokenStream) -> TokenStream {
    let input = parse_macro_input!(tokens as VisitorFunction);

    input.expand().into()
}
```

27: macro-patterns/src/visitor.highlighted.rs

```rust
use macro_lib::{extensions::ToLowercase, AnnotatedType, KeyValue, SimpleType};
use proc_macro2::{Span, TokenStream, TokenTree};
use quote::{format_ident, quote};
use syn::parse::{Parse, ParseStream, Result};
use syn::punctuated::Punctuated;
use syn::{Ident, Token};

/// Model for holding the input passed to the visitor macro
/// It expects a stream in the following format:
/// ```text
/// ConcreteType,
///
/// dyn DynamicType,
///
/// #[no_defuault]
/// NoDefault,
///
/// #[helper_tmpl = {visitor.visit_button(window.button);}]
/// CustomTemplate,
/// ```
///
/// Thus, it takes a list of types that will be visited.
/// A type can be concrete or dynamic.
///
/// Options can also be passed to type:
/// - `no_default` to turn-off the defualt implementation for the trait method.
/// - 'helper_tmpl` to be filled into the helper template for traversing a types internal structure.
#[derive(Eq, PartialEq, Debug)]
pub struct VisitorFunction {
    types: Punctuated<AnnotatedType<SimpleType>, Token![,]>,
}

/// Make VisitorFunction parsable
impl Parse for VisitorFunction {
    fn parse(input: ParseStream) -> Result<Self> {
        Ok(VisitorFunction {
            types: input.parse_terminated(AnnotatedType::parse)?,
        })
    }
}

impl VisitorFunction {
    /// Expand the visitor model into its implementation
    pub fn expand(&self) -> TokenStream {
        // Store each of the three parts
        let mut trait_functions: Vec<TokenStream> = Vec::new();
        let mut helpers: Vec<TokenStream> = Vec::new();
        let mut visitables: Vec<TokenStream> = Vec::new();

        // Loop over each type given
        for t in self.types.iter() {
            let elem_name = t.inner_type.ident.to_lowercase();
            let elem_type = &t.inner_type;
            let fn_name = format_ident!("visit_{}", elem_name);
            let options = Options::new(&t.attrs.options);

            // Get trait function
            if options.no_default {
                trait_functions.push(quote! {
                    fn #fn_name(&mut self, #elem_name: &#elem_type);
                })
            } else {
```

```
                trait_functions.push(quote! {
                    fn #fn_name(&mut self, #elem_name: &#elem_type) {
                        #fn_name(self, #elem_name)
                    }
                })
            };

            // Get helper function
            if options.has_helper {
                if let Some(inner) = options.helper_tmpl {
                    helpers.push(quote! {
                        pub fn #fn_name<V>(visitor: &mut V, #elem_name: &#elem_type)
                        where
                            V: Visitor + ?Sized,
                        {
                            #inner
                        }
                    });
                } else {
                    let unused_elem_name = format_ident!("_{}", elem_name);
                    helpers.push(quote! {
                        pub fn #fn_name<V>(_visitor: &mut V, #unused_elem_name: &#elem_type)
                        where
                            V: Visitor + ?Sized,
                        {
                        }
                    });
                }
            };

            // Make visitable
            visitables.push(quote! {
                impl Visitable for #elem_type {
                    fn apply(&self, visitor: &mut dyn Visitor) {
                        visitor.#fn_name(self);
                    }
                }
            });
        }

        // Built complete visitor implementation
        quote! {
            pub trait Visitor {
                #(#trait_functions)*
            }

            #(#helpers)*

            trait Visitable {
                fn apply(&self, visitor: &mut dyn Visitor);
            }
            #(#visitables)*
        }
    }
}

/// Private struct for dissecting each option passed to a visitor type
struct Options {
    no_default: bool,
    has_helper: bool,
    helper_tmpl: Option<TokenStream>,
}

impl Options {
```

```
    fn new(options: &Punctuated<KeyValue, Token![,]>) -> Self {
        // Defaults
        let mut no_default = false;
        let mut has_helper = true;
        let mut helper_tmpl = None;

        // Loop over each option given
        for option in options.iter() {
            // "no_default" turns no_default on
            if option.key == Ident::new("no_default", Span::call_site()) {
                no_default = true;
                continue;
            }

            if option.key == Ident::new("helper_tmpl", Span::call_site()) {
                match &option.value {
                    TokenTree::Ident(ident) if ident == &Ident::new("false", Span::call_site()) => {
                        // "helper_tmpl = false" turns helper template off
                        has_helper = false;
                    }
                    TokenTree::Group(group) => {
                        // Custom helper template was given
                        helper_tmpl = Some(group.stream());
                    }
                    _ => continue,
                }
            }
        }

        Options {
            no_default,
            has_helper,
            helper_tmpl,
        }
    }
}

#[cfg(test)]
mod tests {
    use super::*;
    use macro_test_helpers::reformat;
    use pretty_assertions::assert_eq;
    use syn::{parse_quote, parse_str};

    type Result = std::result::Result<(), Box<dyn std::error::Error>>;

    #[test]
    fn parse() {
        let actual: VisitorFunction = parse_quote! {
            #[no_default]
            dyn Button
        };

        let mut expected = VisitorFunction {
            types: Punctuated::new(),
        };

        expected.types.push(parse_quote! {#[no_default] dyn Button});

        assert_eq!(actual, expected);
    }

    #[test]
    fn parse_just_types() -> Result {
```

```
        let actual: VisitorFunction = parse_str("Button, dyn Text, Window")?;

        let mut expected = VisitorFunction {
            types: Punctuated::new(),
        };

        expected.types.push(parse_str("Button")?);
        expected.types.push(parse_str("dyn Text")?);
        expected.types.push(parse_str("Window")?);

        assert_eq!(actual, expected);

        Ok(())
    }

    #[test]
    fn parse_mixed() -> Result {
        let actual: VisitorFunction = parse_quote! {
            Button,

            #[tmpl = {trait T {};}]
            Text,

            dyn Window
        };

        let mut expected = VisitorFunction {
            types: Punctuated::new(),
        };

        expected.types.push(parse_str("Button")?);
        expected.types.push(parse_quote! {
            #[tmpl = {trait T {};}]
            Text
        });
        expected.types.push(parse_str("dyn Window")?);

        assert_eq!(actual, expected);

        Ok(())
    }

    #[test]
    fn expand() -> Result {
        let mut input = VisitorFunction {
            types: Punctuated::new(),
        };

        input.types.push(parse_quote! {
            #[helper_tmpl = false]
            Button
        });
        input.types.push(parse_quote! {
            #[no_default]
            dyn Text
        });
        input.types.push(parse_quote! {
           #[helper_tmpl = {
               visitor.visit_button(window.button);
           }]
           Window
        });

        let actual = input.expand();
```

```
        let expected = quote! {
            pub trait Visitor{
                fn visit_button(&mut self, button: &Button) {
                    visit_button(self, button)
                }
                fn visit_text(&mut self, text: &dyn Text);
                fn visit_window(&mut self, window: &Window) {
                    visit_window(self, window)
                }
            }

            pub fn visit_text<V>(_visitor: &mut V, _text: &dyn Text)
            where
                V: Visitor + ?Sized,
            {
            }

            pub fn visit_window<V>(visitor: &mut V, window: &Window)
            where
                V: Visitor + ?Sized,
            {
               visitor.visit_button(window.button);
            }

            trait Visitable {
                fn apply(&self, visitor: &mut dyn Visitor);
            }
            impl Visitable for Button {
                fn apply(&self, visitor: &mut dyn Visitor) {
                    visitor.visit_button(self);
                }
            }
            impl Visitable for dyn Text {
                fn apply(&self, visitor: &mut dyn Visitor) {
                    visitor.visit_text(self);
                }
            }
            impl Visitable for Window {
                fn apply(&self, visitor: &mut dyn Visitor) {
                    visitor.visit_window(self);
                }
            }
        };

        assert_eq!(
            reformat(&actual).lines().collect::<Vec<_>>(),
            reformat(&expected).lines().collect::<Vec<_>>()
        );

        Ok(())
    }
}
```