



UNIVERSITEIT VAN PRETORIA  
UNIVERSITY OF PRETORIA  
YUNIBESITHI YA PRETORIA

Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

## **COS700 Research**

### **Design Pattern Metaprogramming Foundations in Rust**

**Abstract Factory and Visitor**

**Student number:** u19239395

**Supervisor:**

Dr. Linda Marshall

??? 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 metaprogramming style used by Rust is also explored in this report.

## **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 task that can be solved using metaprogramming.

Rust is a new programming language created by Mozilla and was announced the most loved language for the 5th consecutive year, according to the 2020 Developer Survey by StackOverflow<sup>1</sup>. 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 ??, 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. Both designs will be implemented in Rust in Section 2.

Section ?? will identify different types of metaprogramming and classify the types Rust uses. Some lesser-used programming concepts present in Rust will be explored in Section ??.

Finally, Section 2 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 foundations for the Abstract Factory implementation, the section will create a Rust macro for the Visitor implementation.

## 2 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.

### 2.1 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 *TokenStreams*, which were presented in Section ??, cannot be unit tested. *Syn* and *quote* thus operate against a wrapper found in *proc\_macro2*<sup>2</sup>, which the helper structures in this section will also use. This leads to the libraries shown in Figure 1.

---

<sup>1</sup><https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages>

<sup>2</sup>[https://docs.rs/proc-macro2/1.0.19/proc\\_macro2/index.html](https://docs.rs/proc-macro2/1.0.19/proc_macro2/index.html)

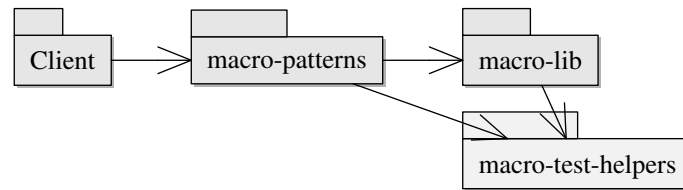


Figure 1: Layout of libraries

Client code will use the *macro-patterns* library. *Macro-patterns* will contain macro definitions, as was defined in Section ??, 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 next.

## 2.2 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.

### 2.2.1 Simple GUI

The design pattern implementations are built on the simple GUI library, shown in Listing 1, 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.

Listing 1: Simple GUI defined by client

```

1 pub trait Element {
2     fn new(name: String) -> Self
3     where
4         Self: Sized;
5     fn get_name(&self) -> &str;
6 }
7
8 pub trait Button: Element {
9     fn click(&self);
  
```

```

10     fn get_text(&self) -> &str;
11     fn set_text(&mut self, text: String);
12 }
13
14 pub trait Input: Element {
15     fn get_input(&self) -> String;
16     fn set_input(&mut self, input: String);
17 }
18
19 pub enum Child {
20     Button(Box<dyn Button>),
21     Input(Box<dyn Input>),
22 }
23
24 pub struct Window {
25     name: String,
26     children: Vec<Child>,
27 }
28
29 impl Window {
30     pub fn add_child(&mut self, child: Child) -> &mut Self {
31         self.children.push(child);
32
33         self
34     }
35     pub fn get_children(&self) -> &[Child] {
36         &self.children
37     }
38 }

```

The abstract *Button* and *Input* each have a concrete brand version. They are the *BrandButton* and *BrandInput* shown in Listing 2.

Listing 2: Brand GUI elements

```

1 use super::elements;
2
3 pub struct BrandButton {
4     name: String,
5     text: String,
6 }
7
8 impl elements::Element for BrandButton {
9     fn new(name: String) -> Self {
10         BrandButton {
11             name,
12             text: String::new(),
13         }
14     }
15     fn get_name(&self) -> &str {

```

```

16         &self.name
17     }
18 }
19
20 impl elements::Button for BrandButton {
21     fn click(&self) {
22         unimplemented!()
23     }
24     fn get_text(&self) -> &str {
25         &self.text
26     }
27     fn set_text(&mut self, text: String) {
28         self.text = text;
29     }
30 }
31
32 pub struct BrandInput {
33     name: String,
34     input: String,
35 }
36
37 impl elements::Element for BrandInput {
38     fn new(name: String) -> Self {
39         BrandInput {
40             name,
41             input: String::new(),
42         }
43     }
44     fn get_name(&self) -> &str {
45         &self.name
46     }
47 }
48
49 impl elements::Input for BrandInput {
50     fn get_input(&self) -> String {
51         self.input.to_owned()
52     }
53     fn set_input(&mut self, input: String) {
54         self.input = input
55     }
56 }

```

### 2.2.2 Manual Abstract Factory implementation

Listing 3 shows an implementation of AF for the GUI. This implementation maps directly to the UML presented for AF in Section ??.

Listing 3: Manual abstract factory

```

1 use crate::gui::{
2     brand_elements,
3     elements::{Button, Element, Input, Window},
4 };
5
6 pub trait Factory<T: Element + ?Sized> {
7     fn create(&self, name: String) -> Box<T>;
8 }
9
10 pub trait AbstractGuiFactory:
11     Display + Factory<dyn Button> + Factory<dyn Input> + Factory<Window>
12 {
13 }

```

Line 2 imports the concrete brand elements from Listing 2, with line 3 importing the abstract elements from Listing 1. A factory method is defined on lines 6 to 8. On line 10, 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))
    }
}

```

### 2.2.3 Manual Visitor implementation

A manual visitor implementation can be seen in Listing 4. Visitor consists of three parts:

- The abstract visitor, as defined on lines 3 to 14, maps to the UML for visitor, as defined in Section ??.

- Helper functions for traversing the object structure [GHJV94] on lines 16 to 37. 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* on line 27.
- Double dispatch reflections on lines 40 to 58. With these, the client does not need to remember/match each abstract visitor method with the element they are currently using. The client can just call *apply* on the element and have it redirect to the correct visitor method.

Listing 4: Manual visitor

```

1 use crate::gui::elements::{Button, Child, Input, Window};
2
3 // Abstract visitor for `Button`, `Input` and `Window`
4 pub trait Visitor {
5     fn visit_button(&mut self, button: &dyn Button) {
6         visit_button(self, button)
7     }
8     fn visit_input(&mut self, input: &dyn Input) {
9         visit_input(self, input)
10    }
11    fn visit_window(&mut self, window: &Window) {
12        visit_window(self, window)
13    }
14 }
15
16 // Helper functions for transversing a hierarchical data structure
17 pub fn visit_button<V>(_visitor: &mut V, _button: &dyn Button)
18 where
19     V: Visitor + ?Sized,
20 {
21 }
22 pub fn visit_input<V>(_visitor: &mut V, _input: &dyn Input)
23 where
24     V: Visitor + ?Sized,
25 {
26 }
27 pub fn visit_window<V>(visitor: &mut V, window: &Window)
28 where
29     V: Visitor + ?Sized,
30 {
31     window.get_children().iter().for_each(child {
32         match child {
33             Child::Button(button) => visitor.visit_button(button.as_ref()),
34             Child::Input(input) => visitor.visit_input(input.as_ref()),
35         };
36     });
37 }

```



```

38
39 // Extends each element with the reflective `apply` method
40 trait Visitable {
41     fn apply(&self, visitor: &mut dyn Visitor);
42 }
43
44 impl Visitable for dyn Button {
45     fn apply(&self, visitor: &mut dyn Visitor) {
46         visitor.visit_button(self);
47     }
48 }
49 impl Visitable for dyn Input {
50     fn apply(&self, visitor: &mut dyn Visitor) {
51         visitor.visit_input(self);
52     }
53 }
54 impl Visitable for Window {
55     fn apply(&self, visitor: &mut dyn Visitor) {
56         visitor.visit_window(self);
57     }
58 }

```

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 4 line 12 uses the helper on line 27, *VisitorName* 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 VisitorName {
    names: Vec<String>,
}

impl VisitorName {
    #[allow(dead_code)]
    pub fn new() -> Self {
        VisitorName { names: Vec::new() }
    }
}

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

```

The next client code shows 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 *VisitorName* is created and applied to the window. Lastly, a test shows the visitor collected the correct names.

```

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 = VisitorName::new();

window.apply(&mut visitor);

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

```

## 2.3 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.

### 2.3.1 AF 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 ?? – will be parsed to a model. This model will be able to expand

itself into its pattern implementation, as defined in Section 2.2. 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 ??:

```

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

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 ??) 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*<sup>3</sup> type to parse a list of elements separated by any punctuation marker. *Syn* also provides *Type*<sup>4</sup> 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.

*TraitSpecifier* is defined in Listing 5. It will use the syntax *trait => concrete* to map a trait type to its concrete definition.

Listing 5: Making a parsable trait specifier

```

1 use syn::parse::{Parse, ParseStream, Result};
2 use syn::{Token, Type};
3
4 /// Type that holds an abstract type and how it will map to a concrete type.
5 /// An acceptable stream will have the following form:
6 /// ```text
7 /// trait => concrete
8 /// ```
9 #[derive(Eq, PartialEq, Debug)]
10 pub struct TraitSpecifier {
11     pub abstract_trait: Type,
12     pub arrow_token: Token![=>],
13     pub concrete: Type,
14 }
15
16 /// Make TraitSpecifier parsable from a token stream
17 impl Parse for TraitSpecifier {
18     fn parse(input: ParseStream) -> Result<Self> {
19         Ok(TraitSpecifier {
20             abstract_trait: input.parse()?,
21             arrow_token: input.parse()?,
22             concrete: input.parse()?,
23         })
24     }
25 }
```

Lines 1 and 2 import the *syn* elements that will be used. The tests use line 9. The model is defined on lines 10 to 14 to hold the abstract trait, the arrow token, and the concrete. The *Token*<sup>5</sup> macro on line 12 is a helper from *syn* to expand Rust tokens and punctuations easily. Lines 17 to 25 implement the

<sup>3</sup><https://docs.rs/syn/1.0.48/syn/punctuated/struct.Punctuated.html>

<sup>4</sup><https://docs.rs/syn/1.0.48/syn/enum.Type.html>

<sup>5</sup><https://docs.rs/syn/1.0.48/syn/macro.Token.html>

*Parse*<sup>6</sup> 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 6. This will be the input passed to the *abstract\_factory* macro.

Listing 6: Meta-code model for abstract factory macro

```

1  /// Holds the tokens for the attributes passed to an AbstractFactory
   ↪ attribute macro
2  /// Expects input in the following format
3  /// ``text
4  /// some_abstract_factory_trait, type_1, type_2, ... , type_n
5  /// ``
6  #[derive(Eq, PartialEq, Debug)]
7  pub struct AbstractFactoryAttribute {
8      factory_trait: Type,
9      sep: Token![,],
10     types: Punctuated<Type, Token![,]>,
11 }
12
13 impl Parse for AbstractFactoryAttribute {
14     fn parse(input: ParseStream) -> Result<Self> {
15         Ok(AbstractFactoryAttribute {
16             factory_trait: input.parse()?,
17             sep: input.parse()?,
18             types: input.parse_terminated(Type::parse)?,
19         })
20     }
21 }

```

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.

The *expand* method for the *AbstractFactoryAttribute* model is defined as seen in Listing 7.

Listing 7: Expanding an abstract factory macro model

```

1  impl AbstractFactoryAttribute {
2      /// Add factory super traits to an `ItemTrait` to turn the `ItemTrait`
   ↪ into an Abstract Factory
3      pub fn expand(&self, input_trait: &mut ItemTrait) -> TokenStream {
4          let factory_traits: Punctuated<TypeParamBound, Token![+]> = {
5              let types = self.types.iter();
6              let factory_name = &self.factory_trait;
7
8              parse_quote! {
9                  #(#factory_name<#types>)**
10             }
11     }
12 }

```

<sup>6</sup><https://docs.rs/syn/1.0.48/syn/parse/trait.Parse.html>

```

11     };
12
13     // Add extra factory super traits
14     input_trait.supertraits.extend(factory_traits);
15
16     quote! {
17         #input_trait
18     }
19 }
20 }

```

The `expand` method takes in a trait definition syntax tree as `input_trait` on line 3. On lines 4 to 11, a factory method super trait is created for each type passed to the macro. Lines 5 and 6 create local variables to be passed to `quote`. Line 8 uses a `syn` helper function to turn a `quote` into a syntax tree. Since `types` defined on line 5 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. On line 9, 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 9 will create `MyFactory<Type1> + MyFactory<Type2>`.

Line 14 appends the factory super trait that was just constructed to the context input. The new context input is returned on line 17.

**Definitions:** The AF macro is shown in Listing 8 to be an attribute macro, as was defined in Section ???. Line 11 parses the input context – which is the `AbstractGuiFactory` definition in the meta-code – with line 12 parsing the macro inputs to `AbstractFactoryAttribute` as defined in Listing 6. Line 14 expands the inputs on the context as defined in Listing 7.

Listing 8: Abstract Factory macro definition

```

1  extern crate proc_macro;
2
3  use proc_macro::TokenStream;
4  use syn::punctuated::Punctuated;
5  use syn::{parse_macro_input, ItemTrait, Token};
6
7  use abstract_factory::AbstractFactoryAttribute;
8
9  #[proc_macro_attribute]
10 pub fn abstract_factory(tokens: TokenStream, trait_expr: TokenStream) ->
    TokenStream {
11     let mut input = parse_macro_input!(trait_expr as ItemTrait);
12     let attributes = parse_macro_input!(tokens as AbstractFactoryAttribute);
13
14     attributes.expand(&mut input).into()
15 }

```

The `interpolate_traits` macro – also being an attribute macro – is shown in Listing 9.

Listing 9: Interpolate traits macro definition

```

1 use macro_lib::token_stream_utils::Interpolate;
2 use macro_lib::TraitSpecifier;
3
4 #[proc_macro_attribute]
5 pub fn interpolate_traits(tokens: TokenStream, concrete_impl: TokenStream)
6     ↪ -> TokenStream {
7     let attributes =
8         parse_macro_input!(tokens with Punctuated:::<TraitSpecifier,
9         ↪ Token![,]>:::parse_terminated);
10
11     attributes.interpolate(concrete_impl.into()).into()
12 }

```

Line 6 parses the macro inputs to a comma-separated list of *TraitSpecifiers* defined in Listing 5. 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 10 defines such a helper for a *proc\_macro2* token stream.

Listing 10: Run-time string interpolator

```

1 use proc_macro2::{Group, TokenStream, TokenTree};
2 use quote::{ToTokens, TokenStreamExt};
3 use std::collections::HashMap;
4 use syn::punctuated::Punctuated;
5
6 /// Trait for tokens that can replace interpolation markers
7 pub trait Interpolate {
8     /// Take a token stream and replace interpolation markers with their
9     ↪ actual values in a new stream
10     fn interpolate(&self, stream: TokenStream) -> TokenStream;
11 }
12
13 /// Make a Punctuated list interpolatible if it holds interpolatible types
14 impl<T: Interpolate, P> Interpolate for Punctuated<T, P> {
15     fn interpolate(&self, stream: TokenStream) -> TokenStream {
16         self.iter()
17             .fold(TokenStream::new(), mut implementations, t {
18                 implementations.extend(t.interpolate(stream.clone()));
19                 implementations
20             })
21     }
22 }
23
24 /// Replace the interpolation markers in a token stream with a specific text
25 pub fn interpolate(

```

```

25     stream: TokenStream,
26     replacements: &HashMap<&str, &dyn ToTokens>,
27 ) -> TokenStream {
28     let mut new = TokenStream::new();
29
30     for token in stream.into_iter() {
31         match token {
32             TokenTree::Ident(ident) => {
33                 let ident_str: &str = &ident.to_string();
34
35                 if let Some(value) = replacements.get(ident_str) {
36                     value.to_tokens(&mut new);
37                     continue;
38                 }
39
40                 new.append(ident);
41             }
42             TokenTree::Literal(literal) => new.append(literal),
43             TokenTree::Group(group) => {
44                 let mut new_group =
45                     Group::new(group.delimiter(),
↪ interpolate(group.stream(), replacements));
46                 new_group.set_span(group.span());
47                 new.append(new_group);
48             }
49             TokenTree::Punct(punct) => {
50                 new.append(punct);
51             }
52         }
53     }
54
55     new
56 }

```

Line 7 defines an *Interpolate* trait for types that will be interpolatable at macro run-time. Line 8 implements the *Interpolate* trait for *syn*'s *Punctuated* type if the punctuated tokens implement the *Interpolate* trait – the *TraitSpecifier* token will be made interpolatable in the next listing.

The *interpolate* function on line 24 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 28 creates a *new* token stream that will be returned from the function on line 55. Each token in *stream* will be copied to *new* if the token does not need to be replaced.

Line 30 starts looping through the tokens, and line 31 matches on the token type. Four token types were presented in Section ???. The *Literal*, *Punct*, and *Group* tokens will be copied as-is. 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 on line 46 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 on line 35, then the replacement value is copied to the *new* stream on line 36. Otherwise, the identifier is copied on line 40.

Listing 11 shows interpolation being implemented for the *TraitSpecifier* – specified in Listing 5. Lines 5 and 6 set up the hash map to replace *TRAIT* with the abstract trait and *CONCRETE* with the concrete type. Line 8 calls *interpolate* as defined in Listing 10, line 24.

Listing 11: Implement *Interpolate* for *TraitSpecifier*

```

1  impl Interpolate for TraitSpecifier {
2      fn interpolate(&self, stream: TokenStream) -> TokenStream {
3          let mut replacements: HashMap<_, &dyn ToTokens> = HashMap::new();
4
5          replacements.insert("TRAIT", &self.abstract_trait);
6          replacements.insert("CONCRETE", &self.concrete);
7
8          interpolate(stream, &replacements)
9      }
10 }
```

Thus, line 9 in Listing 9 will use the context input as a template to interpolate each *TraitSpecifier* passed into the *interpolate\_traits* macro.

### 2.3.2 Visitor macro

The Visitor macro implementation will be a function-like macro – as was defined in section Section ???. 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 ???. This will result in the same code as Listing 4.

```

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 2.2.3.
2. *helper\_tmpl* to modify the helper template used – also defined in Section 2.2.3.

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*<sup>7</sup>. 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 12.

Listing 12: Parses a single key value option

```

1 use proc_macro2::TokenTree;
2 use syn::parse::{Parse, ParseStream, Result};
3 use syn::{parse_str, Ident, Token};
4
5 /// Holds a single key value attribute, with the value being optional
6 #[derive(Debug)]
7 pub struct KeyValue {

```

<sup>7</sup><https://docs.rs/syn/1.0.48/syn/struct.PathSegment.html>

```

8     pub key: Ident,
9     pub equal_token: Token![=],
10    pub value: TokenTree,
11  }
12
13  /// Make KeyValue parsable from a token stream
14  impl Parse for KeyValue {
15    fn parse(input: ParseStream) -> Result<Self> {
16      let key = input.parse()?;
17
18      // Stop if optional value is not given
19      if input.is_empty() input.peek(Token![,]) {
20        return Ok(KeyValue {
21          key,
22          equal_token: Default::default(),
23          value: parse_str("default")?,
24        });
25      }
26
27      // Get the equal sign
28      let equal = input.parse()?;
29
30      // Get the next token item from the parse stream
31      let value = input.parse()?;
32
33      Ok(KeyValue {
34        key,
35        equal_token: equal,
36        value,
37      })
38    }
39  }

```

*KeyValue* parses a single *key* = *value*. The *key* is an identifier with *value* holding a token – lines 8 to 10. The *value* part is optional for boolean options. Thus, line 19 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 20 to 24 returns the key from the parse function.

Line 28 parses the = (equal) sign with the *value* being parsed on line 30. Finally, the entire *KeyValue* is returned on lines 33 to 37.

The *OptionsAttribute* is simple, as seen in Listing 13. It parses the # token, a group of enclosing square brackets, and a comma punctuated list of *KeyValues*.

Listing 13: Parses an attribute with options

```

1 use crate::key_value::KeyValue;
2 use syn::parse::{Parse, ParseStream, Result};
3 use syn::punctuated::Punctuated;
4 use syn::{bracketed, token, Token};

```

```

5
6  /// Holds an outer attribute filled with key-value options
7  #[derive(Eq, PartialEq, Debug, Default)]
8  pub struct OptionsAttribute {
9      pub pound_token: Token![#],
10     pub bracket_token: token::Bracket,
11     pub options: Punctuated<KeyValue, Token![,]>,
12 }
13
14  /// Make OptionsAttribute parsable from a token stream
15  impl Parse for OptionsAttribute {
16     fn parse(input: ParseStream) -> Result<Self> {
17         let content;
18         Ok(OptionsAttribute {
19             pound_token: input.parse()?,
20             bracket_token: bracketed!(content in input),
21             options: content.parse_terminated(KeyValue::parse)?,
22         })
23     }
24 }

```

Listing 14 shows the *SimpleType* model. It consists of an optional *dyn* key-word followed by a type identifier. *ToTokens* is implemented on *SimpleType* to be able to use it in *quote* later.

Listing 14: Parses a simple type identifier with an optional *dyn*

```

1  use proc_macro2::TokenStream;
2  use quote::ToTokens;
3  use syn::parse::{Parse, ParseStream, Result};
4  use syn::{Ident, Token};
5
6  /// Holds a simple type that is optionally annotated as `dyn`
7  #[derive(Eq, PartialEq, Debug)]
8  pub struct SimpleType {
9      pub dyn_token: Option<Token![dyn]>,
10     pub ident: Ident,
11 }
12
13  /// Make SimpleType parsable from token stream
14  impl Parse for SimpleType {
15     fn parse(input: ParseStream) -> Result<Self> {
16         Ok(SimpleType {
17             dyn_token: input.parse()?,
18             ident: input.parse()?,
19         })
20     }
21 }
22
23  /// Turn SimpleType back into a token stream

```

```

24 impl ToTokens for SimpleType {
25     fn to_tokens(&self, tokens: &mut TokenStream) {
26         self.dyn_token.to_tokens(tokens);
27         self.ident.to_tokens(tokens);
28     }
29 }

```

*AnnotatedType* will combine a generic type with an optional *OptionsAttribute*, as seen in Listing 15.

Listing 15: Parses an annotated type

```

1  use proc_macro2::TokenStream;
2  use quote::ToTokens;
3  use syn::parse::{Parse, ParseStream, Result};
4  use syn::{Ident, Token};
5
6  /// Holds a simple type that is optionally annotated as `dyn`
7  #[derive(Eq, PartialEq, Debug)]
8  pub struct SimpleType {
9      pub dyn_token: Option<Token![dyn]>,
10     pub ident: Ident,
11 }
12
13 /// Make SimpleType parsable from token stream
14 impl Parse for SimpleType {
15     fn parse(input: ParseStream) -> Result<Self> {
16         Ok(SimpleType {
17             dyn_token: input.parse()?,
18             ident: input.parse()?,
19         })
20     }
21 }
22
23 /// Turn SimpleType back into a token stream
24 impl ToTokens for SimpleType {
25     fn to_tokens(&self, tokens: &mut TokenStream) {
26         self.dyn_token.to_tokens(tokens);
27         self.ident.to_tokens(tokens);

```

Lastly, *VisitorFunction* – the input to the Visitor macro – will be a comma punctuated list of *AnnotatedTypes*, as shown in Listing 16. The generic *T* in *AnnotatedType* is set to *SimpleType*.

Listing 16: Model to parse and expand Visitor macro inputs

```

1  use macro_lib::{extensions::ToLowercase, AnnotatedType, KeyValue,
2     ↪ SimpleType};
3  use proc_macro2::{Span, TokenStream, TokenTree};
4  use quote::{format_ident, quote};
5  use syn::parse::{Parse, ParseStream, Result};
6  use syn::punctuated::Punctuated;

```

```

6 use syn::{Ident, Token};
7
8 #[derive(Eq, PartialEq, Debug)]
9 pub struct VisitorFunction {
10     types: Punctuated<AnnotatedType<SimpleType>, Token![,]>,
11 }
12
13 impl Parse for VisitorFunction {
14     fn parse(input: ParseStream) -> Result<Self> {
15         Ok(VisitorFunction {
16             types: input.parse_terminated(AnnotatedType::parse)?,
17         })
18     }
19 }
20
21 impl VisitorFunction {
22     pub fn expand(&self) -> TokenStream {
23         let mut trait_functions: Vec<TokenStream> = Vec::new();
24         let mut helpers: Vec<TokenStream> = Vec::new();
25         let mut visitables: Vec<TokenStream> = Vec::new();
26
27         for t in self.types.iter() {
28             let elem_name = t.inner_type.ident.to_lowercase();
29             let elem_type = &t.inner_type;
30
31             let fn_name = format_ident!("visit_{}", elem_name);
32
33             let options = Options::new(&t.attrs.options);
34
35             let fn_def = if options.no_default {
36                 quote! {
37                     fn #fn_name(&mut self, #elem_name: &#elem_type);
38                 }
39             } else {
40                 quote! {
41                     fn #fn_name(&mut self, #elem_name: &#elem_type) {
42                         #fn_name(self, #elem_name)
43                     }
44                 }
45             };
46
47             if options.has_helper {
48                 if let Some(inner) = options.helper_tmpl {
49                     helpers.push(quote! {
50                         pub fn #fn_name<V>(visitor: &mut V, #elem_name:
51 ↪ &#elem_type)
52                             where
53                                 V: Visitor + ?Sized,
54                             {

```

```

54         #inner
55     }
56     });
57     } else {
58         let unused_elem_name = format_ident!("_{}", elem_name);
59         helpers.push(quote! {
60             pub fn #fn_name<V>(_visitor: &mut V,
↪ #unused_elem_name: &#elem_type)
61                 where
62                     V: Visitor + ?Sized,
63                     {
64                     }
65             });
66     }
67 };
68
69 trait_functions.push(fn_def);
70 visitables.push(quote! {
71     impl Visitable for #elem_type {
72         fn apply(&self, visitor: &mut dyn Visitor) {
73             visitor.#fn_name(self);
74         }
75     }
76 });
77 }
78
79 quote! {
80     pub trait Visitor {
81         #(#trait_functions)*
82     }
83
84     #(#helpers)*
85
86     trait Visitable {
87         fn apply(&self, visitor: &mut dyn Visitor);
88     }
89     #(#visitables)*
90 }
91 }
92 }
93
94 struct Options {
95     no_default: bool,
96     has_helper: bool,
97     helper_tmpl: Option<TokenStream>,
98 }
99
100 impl Options {

```

```

101     fn new(options: &Punctuated<KeyValue, Token![,]>) -> Self {
102         let mut no_default = false;
103         let mut has_helper = true;
104         let mut helper_tmpl = None;
105
106         for option in options.iter() {
107             if option.key == Ident::new("no_default", Span::call_site()) {
108                 no_default = true;
109                 continue;
110             }
111
112             if option.key == Ident::new("helper_tmpl", Span::call_site()) {
113                 match &option.value {
114                     TokenTree::Ident(ident) if ident == &Ident::new("false",
↪ Span::call_site()) => {
115                         has_helper = false;
116                     }
117                     TokenTree::Group(group) => {
118                         helper_tmpl = Some(group.stream());
119                     }
120                     _ => continue,
121                 }
122             }
123         }
124
125         Options {
126             no_default,
127             has_helper,
128             helper_tmpl,
129         }
130     }
131 }

```

*VisitorFunction* makes use of a private *Options* struct – lines 94 to 98 – to dissect the options passed to each type. On lines 102 to 104, *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 line 106. If the option has the *no\_default* key, then creating a default trait method for the type is turned off on lines 107 to 110. The option *helper\_tmpl = false* turns off creating a helper function on line 114, while the option *helper\_tmpl = {template}* activates a custom helper template on lines 117 to 119. Line 120 ignores anything else passed to the *helper\_tmpl* option.

The *expand* method for the *VisitorFunction* is defined on line 22. As defined in Section 2.2.3, the Visitor implementation will consist of three parts:



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 23 to 25 define variables to hold each of these three parts in a list. Line 27 starts an iterator over each type passed to the *Visit* macro. Line 28 gets the type name to construct a function name on line 31 – *format\_ident*<sup>8</sup> is a *quote* macro for creating an *Ident* token. Line 29 creates a local variable of the type for use in *quote*. The private *Options* is used on line 33 to dissect the attribute options.

Line 35 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 47 checks if the client has not turned it off. If the client supplied a custom helper template, then line 48 extracts it into *inner* and writes the helper function with *inner* as the function block. Otherwise, if no custom template was supplied, lines 58 to 65 create an empty helper function.

Lastly, the *Visitable* trait is implemented for the type on lines 70 to 76.

Once iterating over all the types is done, lines 79 to 90 write the macro output. The list for each of the three parts is expanded without a separator on lines 81, 84, and 89.

**Definition:** The function-like – as was defined in Section ?? – *visitor* macro is shown in Listing 17. It uses the *VisitorFunction* model defined in Listing 16 to parse the macro input on line 5. The model is expanded on line 7.

Listing 17: Visitor macro definition

```

1 use visitor::VisitorFunction;
2
3 #[proc_macro]
4 pub fn visitor(tokens: TokenStream) -> TokenStream {
5     let input = parse_macro_input!(tokens as VisitorFunction);
6
7     input.expand().into()
8 }
```

### 3 Conclusion

This report showed that Rust macros can be used to implement design patterns. The Abstract Factory pattern was first implemented using Rust attribute macros. The macro definition led to the foundation of parsing the macro input to a model using *syn*. This model is then expanded to the pattern's implementation using *quote*. The model was used to create a function-like Rust macro

<sup>8</sup>[https://docs.rs/quote/1.0.7/quote/macro.format\\_ident.html](https://docs.rs/quote/1.0.7/quote/macro.format_ident.html)

to implement the Visitor pattern. The same model can be implied to other 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”<sup>9</sup> 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 to 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<sup>10</sup> on each type to set the option. An inner attribute<sup>10</sup> can be used on the macro for global options.

## References

- [GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

---

<sup>9</sup><https://www.ncameron.org/blog/abstract-return-types-aka-impl-trait/>

<sup>10</sup><https://doc.rust-lang.org/reference/attributes.html>