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

Keywords:

1 Introduction

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, hand-written implementation 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 output.

2.1 Library layout

Parts of the framework being created can be used by other macros/libraries. Thus, the macro implementations will be separated from the structures they will use. Another reason for this choice is because the *TokenStream*s, which were presented in Section **??**, cannot be unit tested. *Syn* and *quote* thus operate against a wrapper found in *proc_macro2*[1] which the helper structures in this section will also use. This leads to the libraries shown in Figure 1.

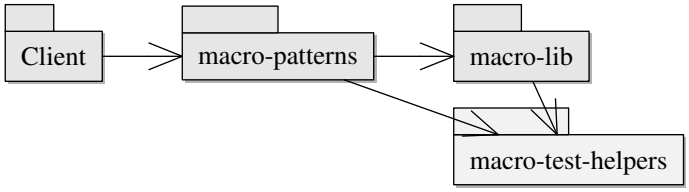

Figure 1: Layout of libraries

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, but the reader should note that automated tests are used to ensure the macro outputs are identical to the hand-written implementations covered next.

2.2 Hand-written 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:

[1] https://docs.rs/proc-macro2/1.0.19/proc_macro2/index.html

- 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

```rust
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
    }
}
```

The abstract *Button* and *Input* each have a concrete brand version – i.e. *BrandButton* and *BrandInput* shown in Listing 2.

Listing 2: Brand GUI elements

```
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
    }
}
```

2.2.2 Hand-written 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: Hand-written abstract factory

```
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>
{
}
```

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 Hand-written Visitor implementation

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

- The abstract visitor as defined on lines 3 to 14 which 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 that has 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: Hand-written visitor

```rust
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
```

```
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 be able to travers 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 is 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 hand-written implementations. The outputs of each macro should be exactly the same as the manually implementations written by a programmer. Three macros will be created in total: 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 hand-written 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))
    }
}
```

```rust
#[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 the factory method for each element. 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-seperated list of inputs. The *syn* library provides the *Punctuated*[2] type to parse a list of elements seperated by any punctuation marker. *Syn* also provides *Type*[3] for parsing Rust types that will used by the *abstract_factory* macro. The elements passed to *interpolate_traits* need to be custom made. Two models needs 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

```rust
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 {
```

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

```
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. Line 9 is used by the tests. The model is defined on lines 10 to 14 to hold the abstract trait, the arrow token, and the concrete. The *Token*[4] macro on line 12 is a helper from *syn* to easily expand Rust tokens and punctuations. Lines 17 to 25 implement the *Parse*[5] 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> {
```

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

```
        Ok(AbstractFactoryAttribute {
            factory_trait: input.parse()?,
            sep: input.parse()?,
            types: input.parse_terminated(Type::parse)?,
        })
    }
}
```

The model takes the factory method trait as the first input followed by a 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

```
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
    ↪ {
        let factory_traits: Punctuated<TypeParamBound, Token![+]> =
        ↪ {
            let types = self.types.iter();
            let factory_name = &self.factory_trait;

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

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

        quote! {
            #input_trait
        }
    }
}
```

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 element in the list. 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 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) -> TokenStream {
6     let attributes =
7         parse_macro_input!(tokens with Punctuated::<TraitSpecifier,
   ↪  Token![,]>::parse_terminated);
8
```

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

Line 6 parses the macro inputs to a list of *TraitSpecifier*s 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 at compile – i.e. the compile-time of the macro. But, these templates need to be expanded when the macro is run – i.e. the compile-time of the client code. Thus a string interpolation like *quote* is needed that can run at run-time. Listing 10 defines such a helper for a *proc_macro2* token stream.

Listing 10: Run-time string interpolator

```
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 in 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
pub fn interpolate(
    stream: TokenStream,
    replacements: &HashMap<&str, &dyn ToTokens>,
) -> TokenStream {
    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* tokens 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 Listing **??**. 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* in its stream and create a new group from the result – the span that is 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. Else the identifier is copied on line 40.

Listing 11 shows interpolation being implemented for *TraitSpecifier* – specified in Listing 5. Lines 5 and 6 set up the hash map to be replaced *TRAIT* with the abstract trait and *CONCRETE* to be replaced 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 hand-written implementation using *quote*.

The following shows the client meta-code for the Visitor macro – meta-code being the input to the macro 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 show how the *helper_tmpl* option is used on the *Window* type. Nothing in *syn* makes it easy to parse a set of 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 *value* being the option value.
2. *OptionsAttribute* to hold a comma seperated 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 to much information to use it easily here.
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::TokenStream;
2 use quote::TokenStreamExt;
3 use syn::parse::{Parse, ParseStream, Result};
4 use syn::{Ident, Token};
```

```rust

/// Holds a single key value attribute, with the value being optional
#[derive(Debug)]
pub struct KeyValue {
    pub key: Ident,
    pub equal_token: Token![=],
    pub value: TokenStream,
}

/// 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: Default::default(),
            });
        }

        // Get the equal sign
        let equal = input.parse()?;

        // Get the next token item from the parse stream
        let value = input.step(cursor {
            let mut s = TokenStream::new();

            if let Some((inner, rest)) = cursor.token_tree() {
                s.append(inner);
                return Ok((s, rest));
            }

            Err(cursor.error("value not supplied"))
        })?;

        Ok(KeyValue {
            key,
            equal_token: equal,
```

```
46              value,
47          })
48      }
49  }
```

KeyValue parses a single *key = value*. The *key* is an identifier with *value* holding a token stream – lines 9 to 11. The *value* part is optional for boolean options. Thus, line 20 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 21 to 25 returns the key from the parse function.

Line 29 parses the = (equal) sign with the *value* being parsed on lines 32 to 41. The *value* needs to be parsed as a token stream, but only a single token needs to be parsed from the current stream. The *step()* method[6] with the *token_tree()* method[7] allows the extraction of a single token leaving the *rest* of the stream intact. If no *value* was given, line 40 creates a compile error at the current stream position using the *error()* method[8]. Finally, the entire *KeyValue* is returned on lines 43 to 47.

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 *KeyValue*s.

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),
```

[6] https://docs.rs/syn/1.0.48/syn/parse/struct.ParseBuffer.html#method.step
[7] https://docs.rs/syn/1.0.48/syn/buffer/struct.Cursor.html#method.token_tree
[8] https://docs.rs/syn/1.0.48/syn/parse/struct.StepCursor.html#method.error

```
            options: content.parse_terminated(KeyValue::parse)?,
        })
    }
}
```

Listing 14 shows the *SimpleType* model. It consists of an optional *dyn* keyword 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*

```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`
#[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);
    }
}
```

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

Listing 15: Parses an annotated type

```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
#[derive(Eq, PartialEq, Debug)]
pub struct RichType<T = Type> {
    pub attrs: OptionsAttribute,
    pub ident: T,
}

/// Make RichType parsable from token stream
impl<T: Parse> Parse for RichType<T> {
    fn parse(input: ParseStream) -> Result<Self> {
        if input.peek(Token![#]) {
            return Ok(RichType {
                attrs: input.parse()?,
                ident: input.parse()?,
            });
        };

        Ok(RichType {
            attrs: Default::default(),
            ident: input.parse()?,
        })
    }
}
```

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

```rust
use macro_lib::{extensions::ToLowercase, KeyValue, RichType,
↪  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};

#[derive(Eq, PartialEq, Debug)]
```

```rust
pub struct VisitorFunction {
    types: Punctuated<RichType<SimpleType>, Token![,]>,
}

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

impl VisitorFunction {
    pub fn expand(&self) -> TokenStream {
        let mut trait_functions: Vec<TokenStream> = Vec::new();
        let mut helpers: Vec<TokenStream> = Vec::new();
        let mut visitables: Vec<TokenStream> = Vec::new();

        for t in self.types.iter() {
            let elem_name = t.ident.ident.to_lowercase();
            let elem_type = &t.ident;

            let fn_name = format_ident!("visit_{}", elem_name);

            let options = Options::new(&t.attrs.options);

            let fn_def = if options.no_default {
                quote! {
                    fn #fn_name(&mut self, #elem_name: &#elem_type);
                }
            } else {
                quote! {
                    fn #fn_name(&mut self, #elem_name: &#elem_type)
 ↪  {
                        #fn_name(self, #elem_name)
                    }
                }
            };

            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,
                        {
                        }
                    });
                }
            };

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

        quote! {
            pub trait Visitor {
                #(#trait_functions)*
            }

            #(#helpers)*

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

```
            #(#visitables)*
        }
    }
}

struct Options {
    no_default: bool,
    has_helper: bool,
    helper_tmpl: Option<TokenStream>,
}

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

        for option in options.iter() {
            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.clone().into_iter().next().unwrap() {
                    TokenTree::Ident(ident) if ident ==
↪   Ident::new("false", Span::call_site()) => {
                        has_helper = false;
                    }
                    TokenTree::Group(group) => {
                        helper_tmpl = Some(group.stream());
                    }
                    _ => continue,
                }
            }
        }

        Options {
            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 line 107 to 110. The option *helper_tmpl* = *false* turns off creating a helper function on 114, while the option *helper_tmpl* = {*template*} activates a custom helper template on lines 117 to 119. Line 120 ignores anything else passed to *helper_tmpl*.

Definition

3 Conclusion

References

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