



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

APPENDIX

Student number: u19239395

Supervisor:

Dr. Linda Marshall

6 November 2020

A Appendix

A.1 Manual Implementations

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

```
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 }
```

```

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 }
39
40 impl Element for Window {
41     fn new(name: String) -> Self {
42         Window {
43             name,
44             children: Vec::new(),
45         }
46     }
47     fn get_name(&self) -> &str {
48         &self.name
49     }
50 }
51
52 impl From<Box<dyn Button>> for Child {
53     fn from(button: Box<dyn Button>) -> Self {
54         Child::Button(button)
55     }
56 }
57
58 impl From<Box<dyn Input>> for Child {
59     fn from(input: Box<dyn Input>) -> Self {
60         Child::Input(input)
61     }
62 }

```

```

1  #[allow(unused_imports)]
2  use macro_patterns::{abstract_factory, interpolate_traits};
3  use std::fmt::{Display, Formatter, Result};
4
5  use crate::gui::{
6      brand_elements,
7      elements::{Button, Element, Input, Window},
8  };
9
10 pub trait Factory<T: Element + ?Sized> {
11     fn create(&self, name: String) -> Box<T>;
12 }
13
14 pub trait AbstractGuiFactory:
15     Display + Factory<dyn Button> + Factory<dyn Input> + Factory<Window>
16 {
17 }
18
19 struct BrandFactory {}
20 impl AbstractGuiFactory for BrandFactory {}
21 impl Factory<dyn Button> for BrandFactory {
22     fn create(&self, name: String) -> Box<dyn Button> {
23         Box::new(brand_elements::BrandButton::new(name))
24     }
25 }
26 impl Factory<dyn Input> for BrandFactory {
27     fn create(&self, name: String) -> Box<dyn Input> {
28         Box::new(brand_elements::BrandInput::new(name))
29     }
30 }
31 impl Factory<Window> for BrandFactory {
32     fn create(&self, name: String) -> Box<Window> {
33         Box::new(Window::new(name))
34     }
35 }
36
37 impl Display for BrandFactory {
38     fn fmt(&self, f: &mut Formatter) -> Result {
39         f.write_str("BrandFactory GUI creator")
40     }
41 }
42
43 #[cfg(test)]
44 mod tests {
45     use super::*;
46
47     #[test]
48     fn button_factory() {
49         let factory = BrandFactory {};
50         let actual: Box<dyn Button> = factory.create(String::from("Button"));
51
52         assert_eq!(actual.get_name(), "Button");
53     }
54
55     #[test]
56     fn window_factory() {
57         let factory = BrandFactory {};
58         let actual: Box<Window> = factory.create(String::from("Window"));
59
60         assert_eq!(actual.get_name(), "Window");
61     }
62 }

```

```

1  #[allow(unused_imports)]
2  use macro_patterns::visitor;
3  use std::fmt;
4
5  use crate::gui::elements::{Button, Child, Input, Window};
6
7  // Abstract visitor for `Button`, `Input` and `Window`
8  pub trait Visitor {
9      fn visit_button(&mut self, button: &dyn Button) {
10         visit_button(self, button)
11     }
12     fn visit_input(&mut self, input: &dyn Input) {
13         visit_input(self, input)
14     }
15     fn visit_window(&mut self, window: &Window) {
16         visit_window(self, window)
17     }
18 }
19
20 // Helper functions for transversing a hierarchical data structure
21 pub fn visit_button<V>(_visitor: &mut V, _button: &dyn Button)
22 where
23     V: Visitor + ?Sized,
24 {
25 }
26 pub fn visit_input<V>(_visitor: &mut V, _input: &dyn Input)
27 where
28     V: Visitor + ?Sized,
29 {
30 }
31 pub fn visit_window<V>(visitor: &mut V, window: &Window)
32 where
33     V: Visitor + ?Sized,
34 {
35     window.get_children().iter().for_each(child {
36         match child {
37             Child::Button(button) => visitor.visit_button(button.as_ref()),
38             Child::Input(input) => visitor.visit_input(input.as_ref()),
39         };
40     });
41 }
42
43 // Extends each element with the reflective `apply` method
44 trait Visitable {
45     fn apply(&self, visitor: &mut dyn Visitor);
46 }
47
48 impl Visitable for dyn Button {
49     fn apply(&self, visitor: &mut dyn Visitor) {
50         visitor.visit_button(self);
51     }
52 }
53 impl Visitable for dyn Input {
54     fn apply(&self, visitor: &mut dyn Visitor) {
55         visitor.visit_input(self);
56     }
57 }
58 impl Visitable for Window {
59     fn apply(&self, visitor: &mut dyn Visitor) {
60         visitor.visit_window(self);
61     }
62 }

```

```

63
64 struct NameVisitor {
65     names: Vec<String>,
66 }
67
68 impl NameVisitor {
69     #[allow(dead_code)]
70     pub fn new() -> Self {
71         NameVisitor { names: Vec::new() }
72     }
73 }
74
75 impl Visitor for NameVisitor {
76     fn visit_button(&mut self, button: &dyn Button) {
77         self.names.push(button.get_name().to_string());
78     }
79     fn visit_input(&mut self, input: &dyn Input) {
80         self.names
81             .push(format!("{}", input.get_name(), input.get_input()));
82     }
83 }
84
85 impl fmt::Display for NameVisitor {
86     fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
87         write!(f, "{}", self.names.join(", "))
88     }
89 }
90
91 #[cfg(test)]
92 mod tests {
93     use super::*;
94     use crate::gui::brand_elements;
95     use crate::gui::elements::{Child, Element};
96
97     type Result = std::result::Result<(), Box<dyn std::error::Error>>;
98
99     #[test]
100     fn visit_button() {
101         let button: &dyn Button = &brand_elements::BrandButton::new(String::from("Some Button"));
102
103         let mut visitor = NameVisitor::new();
104
105         button.apply(&mut visitor);
106
107         assert_eq!(visitor.to_string(), "Some Button");
108     }
109
110     #[test]
111     fn visit_window() -> Result {
112         let mut window = Box::new(Window::new(String::from("Holding window")));
113         let button: Box<dyn Button> = Box::new(brand_elements::BrandButton::new(String::from(
114             "Some Button",
115         )));
116         let mut input: Box<dyn Input> =
117             Box::new(brand_elements::BrandInput::new(String::from("Some Input")));
118
119         input.set_input(String::from("John Doe"));
120
121         window
122             .add_child(Child::from(button))
123             .add_child(Child::from(input));
124
125         let mut visitor = NameVisitor::new();
126
127         window.apply(&mut visitor);

```

```

127
128     assert_eq!(visitor.to_string(), "Some Button, Some Input (John Doe)");
129
130     Ok(())
131 }
132 }

```

A.1.1 macro-lib

```

5: macro-lib/src/annotated_type.rs

1 use crate::options_attribute::OptionsAttribute;
2 use syn::parse::{Parse, ParseStream, Result};
3 use syn::{Token, Type};
4
5 /// Holds a type that is optionally annotated with key-value options.
6 /// An acceptable stream will have the following form:
7 /// ```text
8 /// #[option1 = value1, option2 = value2]
9 /// SomeType
10 /// ```
11 ///
12 /// The outer attribute (hash part) is optional.
13 /// `SomeType` will be parsed to `T`.
14 #[derive(Eq, PartialEq, Debug)]
15 pub struct AnnotatedType<T = Type> {
16     pub attrs: OptionsAttribute,
17     pub inner_type: T,
18 }
19
20 /// Make AnnotatedType parsable from token stream
21 impl<T: Parse> Parse for AnnotatedType<T> {
22     fn parse(input: ParseStream) -> Result<Self> {
23         // Parse attribute options if the next token is a hash
24         if input.peek(Token![#]) {
25             return Ok(AnnotatedType {
26                 attrs: input.parse()?,
27                 inner_type: input.parse()?,
28             });
29         }
30
31         // Parse without attribute options
32         Ok(AnnotatedType {
33             attrs: Default::default(),
34             inner_type: input.parse()?,
35         })
36     }
37 }
38
39 #[cfg(test)]
40 mod tests {
41     use super::*;
42     use pretty_assertions::assert_eq;
43     use syn::{parse_quote, parse_str, TypeTraitObject};
44
45     type Result = std::result::Result<(), Box<dyn std::error::Error>>;
46
47     #[test]
48     fn parse() -> Result {
49         let actual: AnnotatedType = parse_quote! {
50             #[no_default]
51             i32
52         };

```

```

53     let expected = AnnotatedType {
54         attrs: parse_str("#[no_default]")?,
55         inner_type: parse_str("i32")?,
56     };
57
58     assert_eq!(actual, expected);
59     Ok(())
60 }
61
62 #[test]
63 fn parse_simple_type() -> Result {
64     let actual: AnnotatedType = parse_quote! {
65         Button
66     };
67     let expected = AnnotatedType {
68         attrs: Default::default(),
69         inner_type: parse_str("Button")?,
70     };
71
72     assert_eq!(actual, expected);
73     Ok(())
74 }
75
76 #[test]
77 fn parse_trait_bounds() -> Result {
78     let actual: AnnotatedType<TypeTraitObject> = parse_quote! {
79         #[no_default]
80         dyn Button
81     };
82     let expected = AnnotatedType:::<TypeTraitObject> {
83         attrs: parse_str("#[no_default]")?,
84         inner_type: parse_str("dyn Button")?,
85     };
86
87     assert_eq!(actual, expected);
88     Ok(())
89 }
90
91 #[test]
92 #[should_panic(expected = "unexpected end of input")]
93 fn missing_type() {
94     parse_str:::<AnnotatedType>("#[no_default]").unwrap();
95 }
96 }

```

6: macro-lib/src/key_value.rs

```

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  /// Streams in the following form will be parsed:
7  /// ```text
8  /// key = value
9  /// ```
10 ///
11 /// The `value` is optional.
12 /// Thus, the following is also valid.
13 /// ```text
14 /// key
15 /// ```
16 #[derive(Debug)]
17 pub struct KeyValue {

```



```

18     pub key: Ident,
19     pub equal_token: Token![=],
20     pub value: TokenTree,
21 }
22
23 /// Make KeyValue parsable from a token stream
24 impl Parse for KeyValue {
25     fn parse(input: ParseStream) -> Result<Self> {
26         let key = input.parse()?;
27
28         // Stop if optional value is not given
29         if input.is_empty() input.peek(Token![,]) {
30             return Ok(KeyValue {
31                 key,
32                 equal_token: Default::default(),
33                 value: parse_str("default")?,
34             });
35         }
36
37         // Parse with value
38         Ok(KeyValue {
39             key,
40             equal_token: input.parse()?,
41             value: input.parse()?,
42         })
43     }
44 }
45
46 // Just for testing
47 impl PartialEq for KeyValue {
48     fn eq(&self, other: &Self) -> bool {
49         self.key == other.key && format!("{}", self.value) == format!("{}", other.value)
50     }
51 }
52 impl Eq for KeyValue {}
53
54 #[cfg(test)]
55 mod tests {
56     use super::*;
57     use pretty_assertions::assert_eq;
58     use syn::parse_str;
59
60     type Result = std::result::Result<(), Box<dyn std::error::Error>>;
61
62     #[test]
63     fn parse() -> Result {
64         let actual: KeyValue = parse_str("some_key = \"value\"");
65         let expected = KeyValue {
66             key: parse_str("some_key")?,
67             equal_token: Default::default(),
68             value: parse_str("\"value\"")?,
69         };
70
71         assert_eq!(actual, expected);
72         Ok(())
73     }
74
75     #[test]
76     fn parse_missing_value() -> Result {
77         let actual: KeyValue = parse_str("bool_key");
78         let expected = KeyValue {
79             key: parse_str("bool_key")?,
80             equal_token: Default::default(),
81             value: parse_str("default")?,

```

```

82     };
83
84     assert_eq!(actual, expected);
85     Ok(())
86 }
87
88 #[test]
89 fn parse_attribute_item_complex_stream() -> Result {
90     let actual: KeyValue = parse_str("tmpl = {trait To {};}");
91     let expected = KeyValue {
92         key: parse_str("tmpl")?,
93         equal_token: Default::default(),
94         value: parse_str("{trait To {};}")?,
95     };
96
97     assert_eq!(actual, expected);
98     Ok(())
99 }
100
101 // Test extra input after a value stream is ignored
102 #[test]
103 #[should_panic(expected = "expected token")]
104 fn parse_attribute_item_complex_stream_extra() {
105     parse_str::<KeyValue>("tmpl = {trait To {};} key").unwrap();
106 }
107
108 #[test]
109 #[should_panic(expected = "expected identifier")]
110 fn missing_key() {
111     parse_str::<KeyValue>("= true").unwrap();
112 }
113
114 #[test]
115 #[should_panic(expected = "expected `=`")]
116 fn missing_equal_sign() {
117     parse_str::<KeyValue>("key value").unwrap();
118 }
119
120 #[test]
121 #[should_panic(expected = "expected token tree")]
122 fn missing_value() {
123     parse_str::<KeyValue>("key = ").unwrap();
124 }
125 }

```

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

```

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 /// Streams in the following form will be parsed successfully:
8 /// ```text
9 /// #[key1 = value1, bool_key2, key3 = value]
10 /// ```
11 ///
12 /// The value part of an option is optional.
13 /// Thus, `bool_key2` will have the value `default`.
14 #[derive(Eq, PartialEq, Debug, Default)]
15 pub struct OptionsAttribute {
16     pub pound_token: Token![#],
17     pub bracket_token: token::Bracket,

```

```

18 pub options: Punctuated<KeyValue, Token![,]>,
19 }
20
21 /// Make OptionsAttribute parsable from a token stream
22 impl Parse for OptionsAttribute {
23     fn parse(input: ParseStream) -> Result<Self> {
24         // Used to hold the stream inside square brackets
25         let content;
26
27         Ok(OptionsAttribute {
28             pound_token: input.parse()?,
29             bracket_token: bracketed!(content in input), // Use `syn` to extract the stream inside the square bracket group
30             options: content.parse_terminated(KeyValue::parse)?,
31         })
32     }
33 }
34
35 #[cfg(test)]
36 mod tests {
37     use super::*;
38     use pretty_assertions::assert_eq;
39     use syn::parse_str;
40
41     type Result = std::result::Result<(), Box<dyn std::error::Error>>;
42
43     #[test]
44     fn parse() -> Result {
45         let actual: OptionsAttribute =
46             parse_str("#[tmpl = {trait To {};}], no_default, other = Top]")?;
47         let mut expected = OptionsAttribute {
48             pound_token: Default::default(),
49             bracket_token: Default::default(),
50             options: Punctuated::new(),
51         };
52
53         expected.options.push(parse_str("tmpl = {trait To {};}")?);
54         expected.options.push(parse_str("no_default")?);
55         expected.options.push(parse_str("other = Top")?);
56
57         assert_eq!(actual, expected);
58         Ok(())
59     }
60 }

```

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

```

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 /// The following is an example of its input stream:
8 /// ```text
9 /// dyn SomeType
10 /// ```
11 ///
12 /// The `dyn` keyword is optional.
13 #[derive(Eq, PartialEq, Debug)]
14 pub struct SimpleType {
15     pub dyn_token: Option<Token![dyn]>,
16     pub ident: Ident,
17 }
18

```

```

19 /// Make SimpleType parsable from token stream
20 impl Parse for SimpleType {
21     fn parse(input: ParseStream) -> Result<Self> {
22         Ok(SimpleType {
23             dyn_token: input.parse()?,
24             ident: input.parse()?,
25         })
26     }
27 }
28
29 /// Turn SimpleType back into a token stream
30 impl ToTokens for SimpleType {
31     fn to_tokens(&self, tokens: &mut TokenStream) {
32         self.dyn_token.to_tokens(tokens);
33         self.ident.to_tokens(tokens);
34     }
35 }
36
37 #[cfg(test)]
38 mod tests {
39     use super::*;
40     use pretty_assertions::assert_eq;
41     use quote::quote;
42     use syn::parse_str;
43
44     type Result = std::result::Result<(), Box<dyn std::error::Error>>;
45
46     #[test]
47     fn parse() -> Result {
48         let actual: SimpleType = parse_str("dyn Button")?;
49         let expected = SimpleType {
50             dyn_token: Some(Default::default()),
51             ident: parse_str("Button")?,
52         };
53
54         assert_eq!(actual, expected);
55         Ok(())
56     }
57
58     #[test]
59     fn parse_without_dyn() -> Result {
60         let actual: SimpleType = parse_str("Button")?;
61         let expected = SimpleType {
62             dyn_token: None,
63             ident: parse_str("Button")?,
64         };
65
66         assert_eq!(actual, expected);
67         Ok(())
68     }
69
70     #[test]
71     #[should_panic(expected = "expected identifier")]
72     fn missing_type() {
73         parse_str::<SimpleType>("dyn").unwrap();
74     }
75
76     #[test]
77     fn to_tokens() -> Result {
78         let input = SimpleType {
79             dyn_token: Some(Default::default()),
80             ident: parse_str("Button")?,
81         };
82         let actual = quote! { #input };

```

```

83     let expected: TokenStream = parse_str("dyn Button"?);
84
85     assert_eq!(format!("{}", actual), format!("{}", expected));
86     Ok(())
87 }
88 }

```

9: macro-lib/src/token_stream_utils.rs

```

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 actual values into a new stream
9      fn interpolate(&self, stream: TokenStream) -> TokenStream;
10 }
11
12 /// Make a Punctuated list interpolatable if it holds interpolatable types
13 impl<T: Interpolate, P> Interpolate for Punctuated<T, P> {
14     fn interpolate(&self, stream: TokenStream) -> TokenStream {
15         self.iter()
16             .fold(TokenStream::new(), mut implementations, t {
17                 implementations.extend(t.interpolate(stream.clone()));
18                 implementations
19             })
20     }
21 }
22
23 /// Replace the interpolation markers in a token stream with a specific text
24 /// Thus, if `stream` is "let a: TRAIT;" and `replacements` has the key "TRAIT" with value "Button", then this will return
25 ↳ a stream with "let a: Button;".
26 pub fn interpolate(
27     stream: TokenStream,
28     replacements: &HashMap<&str, &dyn ToTokens>,
29 ) -> TokenStream {
30     let mut new = TokenStream::new();
31
32     // Loop over each token in the stream
33     // `Literal`, `Punct`, and `Group` are kept as is
34     for token in stream.into_iter() {
35         match token {
36             TokenTree::Literal(literal) => new.append(literal),
37             TokenTree::Punct(punct) => new.append(punct),
38             TokenTree::Group(group) => {
39                 // Recursively interpolate the stream in group
40                 let mut new_group =
41                     Group::new(group.delimiter(), interpolate(group.stream(), replacements));
42                 new_group.set_span(group.span());
43
44                 new.append(new_group);
45             }
46             TokenTree::Ident(ident) => {
47                 let ident_str: &str = &ident.to_string();
48
49                 // Check if identifier is in the replacement set
50                 if let Some(value) = replacements.get(ident_str) {
51                     // Replace with replacement value
52                     value.to_tokens(&mut new);
53
54                     continue;
55                 }
56             }
57         }
58     }
59 }

```

```

55
56         // Identifier did not match, so copy as is
57         new.append(ident);
58     }
59 }
60 }
61
62 new
63 }
64
65 #[cfg(test)]
66 mod tests {
67     use super::*;
68     use crate::trait_specifier::TraitSpecifier;
69     use pretty_assertions::assert_eq;
70     use quote::quote;
71     use syn::{parse_str, Ident, Token, Type};
72
73     type Result = std::result::Result<(), Box<dyn std::error::Error>>;
74
75     #[test]
76     fn complete_replacements() -> Result {
77         let input = quote! {
78             let VAR: TRAIT = if true {
79                 CONCRETE{}
80             } else {
81                 Alternative{}
82             }
83         };
84
85         let expected = quote! {
86             let var: abstract_type = if true {
87                 concrete{}
88             } else {
89                 Alternative{}
90             }
91         };
92
93         let mut r: HashMap<&str, &dyn ToTokens> = HashMap::new();
94         let v: Ident = parse_str("var")?;
95         let a: Type = parse_str("abstract_type")?;
96         let c: Type = parse_str("concrete")?;
97
98         r.insert("VAR", &v);
99         r.insert("TRAIT", &a);
100        r.insert("CONCRETE", &c);
101
102        assert_eq!(
103            format!("{}", &interpolate(input, &r)),
104            format!("{}", expected)
105        );
106
107        Ok(())
108    }
109
110    /// Partial replacements should preverse the uninterpolated identifiers
111    #[test]
112    fn partial_replacements() -> Result {
113        let input: TokenStream = parse_str("let a: TRAIT = OTHER;")?;
114        let expected: TokenStream = parse_str("let a: Display = OTHER;")?;
115
116        let mut r: HashMap<&str, &dyn ToTokens> = HashMap::new();
117        let t: Type = parse_str("Display")?;
118        r.insert("TRAIT", &t);

```

```

119
120     assert_eq!(
121         format!("{}", interpolate(input, &r)),
122         format!("{}", expected)
123     );
124
125     Ok(())
126 }
127
128 #[test]
129 fn interpolate_on_punctuated() -> Result {
130     let mut traits: Punctuated<TraitSpecifier, Token![,]> = Punctuated::new();
131
132     traits.push(parse_str("IButton => BigButton")?);
133     traits.push(parse_str("IWindow => MinimalWindow")?);
134
135     let input = quote! {
136         let _: TRAIT = CONCRETE{};
137     };
138     let expected = quote! {
139         let _: IButton = BigButton{};
140         let _: IWindow = MinimalWindow{};
141     };
142
143     assert_eq!(
144         format!("{}", traits.interpolate(input)),
145         format!("{}", expected)
146     );
147
148     Ok(())
149 }
150 }

```

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

```

1 use crate::token_stream_utils::{interpolate, Interpolate};
2 use proc_macro2::TokenStream;
3 use quote::ToTokens;
4 use std::collections::HashMap;
5 use syn::parse::{Parse, ParseStream, Result};
6 use syn::{Token, Type};
7
8 /// Type that holds an abstract type and how it will map to a concrete type.
9 /// An acceptable stream will have the following form:
10 /// ```text
11 /// trait => concrete
12 /// ```
13 #[derive(Eq, PartialEq, Debug)]
14 pub struct TraitSpecifier {
15     pub abstract_trait: Type,
16     pub arrow_token: Token![=>],
17     pub concrete: Type,
18 }
19
20 /// Make TraitSpecifier parsable from a token stream
21 impl Parse for TraitSpecifier {
22     fn parse(input: ParseStream) -> Result<Self> {
23         Ok(TraitSpecifier {
24             abstract_trait: input.parse()?,
25             arrow_token: input.parse()?,
26             concrete: input.parse()?,
27         })
28     }
29 }

```

```

30
31 /// Make TraitSpecifier interpolatible
32 impl Interpolate for TraitSpecifier {
33     fn interpolate(&self, stream: TokenStream) -> TokenStream {
34         let mut replacements: HashMap<_, &dyn ToTokens> = HashMap::new();
35
36         // Replace each "TRAIT" with the abstract trait
37         replacements.insert("TRAIT", &self.abstract_trait);
38
39         // Replace each "CONCRETE" with the concrete type
40         replacements.insert("CONCRETE", &self.concrete);
41
42         interpolate(stream, &replacements)
43     }
44 }
45
46 #[cfg(test)]
47 mod tests {
48     use super::*;
49     use macro_test_helpers::reformat;
50     use pretty_assertions::assert_eq;
51     use quote::quote;
52     use syn::parse_str;
53
54     type Result = std::result::Result<(), Box<dyn std::error::Error>>;
55
56     #[test]
57     fn parse() -> Result {
58         let actual: TraitSpecifier = parse_str("abstract_trait => concrete")?;
59         let expected = TraitSpecifier {
60             abstract_trait: parse_str("abstract_trait")?,
61             arrow_token: Default::default(),
62             concrete: parse_str("concrete")?,
63         };
64
65         assert_eq!(actual, expected);
66         Ok(())
67     }
68
69     #[test]
70     #[should_panic(expected = "expected one of")]
71     fn missing_trait() {
72         parse_str::<TraitSpecifier>("=> concrete").unwrap();
73     }
74
75     #[test]
76     #[should_panic(expected = "expected `=>`")]
77     fn missing_arrow_joiner() {
78         parse_str::<TraitSpecifier>("IButton -> RoundButton").unwrap();
79     }
80
81     #[test]
82     #[should_panic(expected = "unexpected end of input")]
83     fn missing_concrete() {
84         parse_str::<TraitSpecifier>("abstract_trait => ").unwrap();
85     }
86
87     #[test]
88     fn interpolate() -> Result {
89         let input = quote! {
90             impl Factory<TRAIT> for Gnome {
91                 fn create(&self) -> CONCRETE {
92                     CONCRETE{}
93                 }

```



```

94     }
95 };
96 let expected = quote! {
97     impl Factory<abstract_trait> for Gnome {
98         fn create(&self) -> concrete {
99             concrete{}
100         }
101     }
102 };
103 let specifier = TraitSpecifier {
104     abstract_trait: parse_str("abstract_trait")?,
105     arrow_token: Default::default(),
106     concrete: parse_str("concrete")?,
107 };
108
109 assert_eq!(reformat(&specifier.interpolate(input)), reformat(&expected));
110
111 Ok(())
112 }
113 }

```

A.1.2 macro-patterns

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

```

1 use proc_macro2::TokenStream;
2 use quote::quote;
3 use syn::parse::{Parse, ParseStream, Result};
4 use syn::punctuated::Punctuated;
5 use syn::{parse_quote, ItemTrait, Token, Type, TypeParamBound};
6
7 /// Holds the tokens for the attributes passed to an AbstractFactory attribute macro
8 /// Expects input in the following format
9 /// ``text
10 /// some_factory_method_trait, type_1, type_2, ... , type_n
11 /// ``
12 ///
13 /// `some_factory_method_trait` needs to be created by the client.
14 /// It should have one generic type.
15 /// Every `type_1` ... `type_n` will be filled into this generic type.
16 #[derive(Eq, PartialEq, Debug)]
17 pub struct AbstractFactoryAttribute {
18     factory_trait: Type,
19     sep: Token![,],
20     types: Punctuated<Type, Token![,]>,
21 }
22
23 /// Make AbstractFactoryAttribute parsable
24 impl Parse for AbstractFactoryAttribute {
25     fn parse(input: ParseStream) -> Result<Self> {
26         Ok(AbstractFactoryAttribute {
27             factory_trait: input.parse()?,
28             sep: input.parse()?,
29             types: input.parse_terminated(Type::parse)?,
30         })
31     }
32 }
33
34 impl AbstractFactoryAttribute {
35     /// Add factory super traits to an `ItemTrait` to turn the `ItemTrait` into an Abstract Factory
36     pub fn expand(&self, input_trait: &mut ItemTrait) -> TokenStream {
37         // Build all the super traits
38         let factory_traits: Punctuated<TypeParamBound, Token![+]> = {

```

```

39     let types = self.types.iter();
40     let factory_name = &self.factory_trait;
41
42     parse_quote! {
43         #(#factory_name<#types>)+*
44     }
45 };
46
47 // Append extra factory super traits
48 input_trait.supertraits.extend(factory_traits);
49
50 quote! {
51     #input_trait
52 }
53 }
54 }
55
56 #[cfg(test)]
57 mod tests {
58     use super::*;
59     use macro_test_helpers::reformat;
60     use syn::parse_str;
61
62     type Result = std::result::Result<(), Box<dyn std::error::Error>>;
63
64     mod abstract_factory {
65         use super::*;
66         use pretty_assertions::assert_eq;
67
68         #[test]
69         fn parse() -> Result {
70             let actual: AbstractFactoryAttribute = parse_str("Factory, u32, i64")?;
71             let mut expected_types = Punctuated::new();
72
73             expected_types.push(parse_str("u32")?);
74             expected_types.push(parse_str("i64")?);
75
76             assert_eq!(
77                 actual,
78                 AbstractFactoryAttribute {
79                     factory_trait: parse_str("Factory")?,
80                     sep: Default::default(),
81                     types: expected_types,
82                 }
83             );
84
85             Ok(())
86         }
87
88         #[test]
89         #[should_panic(expected = "expected `,`")]
90         fn missing_types() {
91             parse_str::<AbstractFactoryAttribute>("Factory").unwrap();
92         }
93
94         #[test]
95         fn expand() -> Result {
96             let mut t = parse_str::<ItemTrait>("pub trait Abstraction<T>: Display + Extend<T> {}")?;
97             let mut input_types = Punctuated::new();
98
99             input_types.push(parse_str("u32")?);
100             input_types.push(parse_str("i64")?);
101
102             let actual = &AbstractFactoryAttribute {

```

```

103         factory_trait: parse_str("Factory")?,
104         sep: Default::default(),
105         types: input_types,
106     }
107     .expand(&mut t);
108
109     assert_eq!(
110         reformat(&actual),
111         "pub trait Abstraction<T>: Display + Extend<T> + Factory<u32> + Factory<i64> {} \n"
112     );
113
114     Ok(())
115 }
116 }
117 }

```

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

```

1 mod abstract_factory;
2 mod visitor;
3
4 extern crate proc_macro;
5
6 use proc_macro::TokenStream;
7 use syn::punctuated::Punctuated;
8 use syn::{parse_macro_input, ItemTrait, Token};
9
10 use abstract_factory::AbstractFactoryAttribute;
11 use macro_lib::token_stream_utils::Interpolate;
12 use macro_lib::TraitSpecifier;
13 use visitor::VisitorFunction;
14
15 #[proc_macro_attribute]
16 pub fn abstract_factory(tokens: TokenStream, trait_expr: TokenStream) -> TokenStream {
17     let mut input = parse_macro_input!(trait_expr as ItemTrait);
18     let attributes = parse_macro_input!(tokens as AbstractFactoryAttribute);
19
20     attributes.expand(&mut input).into()
21 }
22
23 #[proc_macro_attribute]
24 pub fn interpolate_traits(tokens: TokenStream, concrete_impl: TokenStream) -> TokenStream {
25     let attributes =
26         parse_macro_input!(tokens with Punctuated::<TraitSpecifier, Token![,]>::parse_terminated);
27
28     attributes.interpolate(concrete_impl.into()).into()
29 }
30
31 #[proc_macro]
32 pub fn visitor(tokens: TokenStream) -> TokenStream {
33     let input = parse_macro_input!(tokens as VisitorFunction);
34
35     input.expand().into()
36 }

```

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

```

1 use macro_lib::{extensions::ToLowercase, AnnotatedType, KeyValue, SimpleType};
2 use proc_macro2::{Span, TokenStream, TokenTree};
3 use quote::{format_ident, quote};
4 use syn::parse::{Parse, ParseStream, Result};
5 use syn::punctuated::Punctuated;
6 use syn::{Ident, Token};
7

```

```

8  /// Model for holding the input passed to the visitor macro
9  /// It expects a stream in the following format:
10 /// ``text
11 /// ConcreteType,
12 ///
13 /// dyn DynamicType,
14 ///
15 /// #[no_default]
16 /// NoDefault,
17 ///
18 /// #[helper_tmpl = {visitor.visit_button(window.button);}]
19 /// CustomTemplate,
20 /// ``
21 ///
22 /// Thus, it takes a list of types that will be visited.
23 /// A type can be concrete or dynamic.
24 ///
25 /// Options can also be passed to type:
26 /// - `no_default` to turn-off the default implementation for the trait method.
27 /// - `helper_tmpl` to be filled into the helper template for traversing a types internal structure.
28 #[derive(Eq, PartialEq, Debug)]
29 pub struct VisitorFunction {
30     types: Punctuated<AnnotatedType<SimpleType>, Token![,]>,
31 }
32
33 /// Make VisitorFunction parsable
34 impl Parse for VisitorFunction {
35     fn parse(input: ParseStream) -> Result<Self> {
36         Ok(VisitorFunction {
37             types: input.parse_terminated(AnnotatedType::parse)?,
38         })
39     }
40 }
41
42 impl VisitorFunction {
43     /// Expand the visitor model into its implementation
44     pub fn expand(&self) -> TokenStream {
45         // Store each of the three parts
46         let mut trait_functions: Vec<TokenStream> = Vec::new();
47         let mut helpers: Vec<TokenStream> = Vec::new();
48         let mut visitables: Vec<TokenStream> = Vec::new();
49
50         // Loop over each type given
51         for t in self.types.iter() {
52             let elem_name = t.inner_type.ident.to_lowercase();
53             let elem_type = &t.inner_type;
54             let fn_name = format_ident!("visit_{}", elem_name);
55             let options = Options::new(&t.attrs.options);
56
57             // Get trait function
58             if options.no_default {
59                 trait_functions.push(quote! {
60                     fn #fn_name(&mut self, #elem_name: &#elem_type);
61                 })
62             } else {
63                 trait_functions.push(quote! {
64                     fn #fn_name(&mut self, #elem_name: &#elem_type) {
65                         #fn_name(self, #elem_name)
66                     }
67                 })
68             };
69
70             // Get helper function
71             if options.has_helper {

```

```

72         if let Some(inner) = options.helper_tmpl {
73             helpers.push(quote! {
74                 pub fn #fn_name<V>(visitor: &mut V, #elem_name: &#elem_type)
75                     where
76                         V: Visitor + ?Sized,
77                     {
78                         #inner
79                     }
80             });
81         } else {
82             let unused_elem_name = format_ident!("_{}", elem_name);
83             helpers.push(quote! {
84                 pub fn #fn_name<V>(_visitor: &mut V, #unused_elem_name: &#elem_type)
85                     where
86                         V: Visitor + ?Sized,
87                     {
88                     }
89             });
90         }
91     };
92
93     // Make visitable
94     visitables.push(quote! {
95         impl Visitable for #elem_type {
96             fn apply(&self, visitor: &mut dyn Visitor) {
97                 visitor.#fn_name(self);
98             }
99         }
100     });
101 }
102
103 // Built complete visitor implementation
104 quote! {
105     pub trait Visitor {
106         #(#trait_functions)*
107     }
108
109     #(#helpers)*
110
111     trait Visitable {
112         fn apply(&self, visitor: &mut dyn Visitor);
113     }
114     #(#visitables)*
115 }
116 }
117
118
119 /// Private struct for dissecting each option passed to a visitor type
120 struct Options {
121     no_default: bool,
122     has_helper: bool,
123     helper_tmpl: Option<TokenStream>,
124 }
125
126 impl Options {
127     fn new(options: &Punctuated<KeyValue, Token![,]>) -> Self {
128         // Defaults
129         let mut no_default = false;
130         let mut has_helper = true;
131         let mut helper_tmpl = None;
132
133         // Loop over each option given
134         for option in options.iter() {
135             // "no_default" turns no_default on

```

```

136     if option.key == Ident::new("no_default", Span::call_site()) {
137         no_default = true;
138         continue;
139     }
140
141     if option.key == Ident::new("helper_tmpl", Span::call_site()) {
142         match &option.value {
143             TokenTree::Ident(ident) if ident == &Ident::new("false", Span::call_site()) => {
144                 // "helper_tmpl = false" turns helper template off
145                 has_helper = false;
146             }
147             TokenTree::Group(group) => {
148                 // Custom helper template was given
149                 helper_tmpl = Some(group.stream());
150             }
151             _ => continue,
152         }
153     }
154 }
155
156 Options {
157     no_default,
158     has_helper,
159     helper_tmpl,
160 }
161 }
162 }
163
164 #[cfg(test)]
165 mod tests {
166     use super::*;
167     use macro_test_helpers::reformat;
168     use pretty_assertions::assert_eq;
169     use syn::{parse_quote, parse_str};
170
171     type Result = std::result::Result<(), Box<dyn std::error::Error>>;
172
173     #[test]
174     fn parse() {
175         let actual: VisitorFunction = parse_quote! {
176             #[no_default]
177             dyn Button
178         };
179
180         let mut expected = VisitorFunction {
181             types: Punctuated::new(),
182         };
183
184         expected.types.push(parse_quote! { #[no_default] dyn Button });
185
186         assert_eq!(actual, expected);
187     }
188
189     #[test]
190     fn parse_just_types() -> Result {
191         let actual: VisitorFunction = parse_str("Button, dyn Text, Window"?);
192
193         let mut expected = VisitorFunction {
194             types: Punctuated::new(),
195         };
196
197         expected.types.push(parse_str("Button"?);
198         expected.types.push(parse_str("dyn Text"?);
199         expected.types.push(parse_str("Window"?);

```

```

200
201     assert_eq!(actual, expected);
202
203     Ok(())
204 }
205
206 #[test]
207 fn parse_mixed() -> Result {
208     let actual: VisitorFunction = parse_quote! {
209         Button,
210
211         #[templ = {trait T {};}]]
212         Text,
213
214         dyn Window
215     };
216
217     let mut expected = VisitorFunction {
218         types: Punctuated::new(),
219     };
220
221     expected.types.push(parse_str("Button")?);
222     expected.types.push(parse_quote! {
223         #[templ = {trait T {};}]]
224         Text
225     });
226     expected.types.push(parse_str("dyn Window")?);
227
228     assert_eq!(actual, expected);
229
230     Ok(())
231 }
232
233 #[test]
234 fn expand() -> Result {
235     let mut input = VisitorFunction {
236         types: Punctuated::new(),
237     };
238
239     input.types.push(parse_quote! {
240         #[helper_tmpl = false]
241         Button
242     });
243     input.types.push(parse_quote! {
244         #[no_default]
245         dyn Text
246     });
247     input.types.push(parse_quote! {
248         #[helper_tmpl = {
249             visitor.visit_button(window.button);
250         }]
251         Window
252     });
253
254     let actual = input.expand();
255     let expected = quote! {
256         pub trait Visitor{
257             fn visit_button(&mut self, button: &Button) {
258                 visit_button(self, button)
259             }
260             fn visit_text(&mut self, text: &dyn Text);
261             fn visit_window(&mut self, window: &Window) {
262                 visit_window(self, window)
263             }

```

```

264     }
265
266     pub fn visit_text<V>(_visitor: &mut V, _text: &dyn Text)
267     where
268         V: Visitor + ?Sized,
269     {
270     }
271
272     pub fn visit_window<V>(visitor: &mut V, window: &Window)
273     where
274         V: Visitor + ?Sized,
275     {
276         visitor.visit_button(window.button);
277     }
278
279     trait Visitable {
280         fn apply(&self, visitor: &mut dyn Visitor);
281     }
282     impl Visitable for Button {
283         fn apply(&self, visitor: &mut dyn Visitor) {
284             visitor.visit_button(self);
285         }
286     }
287     impl Visitable for dyn Text {
288         fn apply(&self, visitor: &mut dyn Visitor) {
289             visitor.visit_text(self);
290         }
291     }
292     impl Visitable for Window {
293         fn apply(&self, visitor: &mut dyn Visitor) {
294             visitor.visit_window(self);
295         }
296     }
297 };
298
299 assert_eq!(
300     reformat(&actual).lines().collect::<Vec<_>>(),
301     reformat(&expected).lines().collect::<Vec<_>>()
302 );
303
304 Ok(())
305 }
306 }

```