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

```rust
use super::elements;

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

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

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

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

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

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

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

```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
    }
}

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

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

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

3: macro-client/src/abstract_factory_hand.rs

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

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

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

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

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

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

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

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

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

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

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

4: macro-client/src/visitor_hand.rs

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

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

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

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

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

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

```rust

struct NameVisitor {
    names: Vec<String>,
}

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

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

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

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

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

        let mut visitor = NameVisitor::new();

        button.apply(&mut visitor);

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

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

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

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

        let mut visitor = NameVisitor::new();

        window.apply(&mut visitor);
```

```
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

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

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

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

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

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

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

    #[test]
    fn parse() -> Result {
        let actual: AnnotatedType = parse_quote! {
            #[no_default]
            i32
        };
```

```
        let expected = AnnotatedType {
            attrs: parse_str("#[no_default]")?,
            inner_type: parse_str("i32")?,
        };

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

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

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

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

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

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

6: macro-lib/src/key_value.rs

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

/// Holds a single key value attribute, with the value being optional.
/// Streams in the following form will be parsed:
/// ```text
/// key = value
/// ```
///
/// The `value` is optional.
/// Thus, the following is also valid.
/// ```text
/// key
/// ```
#[derive(Debug)]
pub struct KeyValue {
```

```
    pub key: Ident,
    pub equal_token: Token![=],
    pub value: TokenTree,
}

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

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

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

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

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

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

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

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

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

```
        };

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

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

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

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

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

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

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

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

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

/// Holds an outer attribute filled with key-value options.
/// Streams in the following form will be parsed successfully:
/// ```text
/// #[key1 = value1, bool_key2, key3 = value]
/// ```
///
/// The value part of an option is optional.
/// Thus, `bool_key2` will have the value `default`.
#[derive(Eq, PartialEq, Debug, Default)]
pub struct OptionsAttribute {
    pub pound_token: Token![#],
    pub bracket_token: token::Bracket,
```

```rust
    pub options: Punctuated<KeyValue, Token![,]>,
}

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

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

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

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

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

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

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

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

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

/// Holds a simple type that is optionally annotated as `dyn`.
/// The following is an example of its input stream:
/// ```text
/// dyn SomeType
/// ```
///
/// The `dyn` keyword is optional.
#[derive(Eq, PartialEq, Debug)]
pub struct SimpleType {
    pub dyn_token: Option<Token![dyn]>,
    pub ident: Ident,
}

```

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

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

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

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

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

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

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

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

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

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

```
        let expected: TokenStream = parse_str("dyn Button")?;

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

9: macro-lib/src/token_stream_utils.rs

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

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

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

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

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

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

                // Check if identifier is in the replacement set
                if let Some(value) = replacements.get(ident_str) {
                    // Replace with replacement value
                    value.to_tokens(&mut new);

                    continue;
                }
```

```
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);
```

```

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

        Ok(())
    }

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

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

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

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

        Ok(())
    }
}
```

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

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

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

/// Make TraitSpecifier parsable from a token stream
impl Parse for TraitSpecifier {
    fn parse(input: ParseStream) -> Result<Self> {
        Ok(TraitSpecifier {
            abstract_trait: input.parse()?,
            arrow_token: input.parse()?,
            concrete: input.parse()?,
        })
    }
}
```

```rust

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

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

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

        interpolate(stream, &replacements)
    }
}

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

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

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

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

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

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

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

    #[test]
    fn interpolate() -> Result {
        let input = quote! {
            impl Factory<TRAIT> for Gnome {
                fn create(&self) -> CONCRETE {
                    CONCRETE{}
                }
```

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

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

        Ok(())
    }
}
```

A.1.2 macro-patterns

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

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

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

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

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

```
            let types = self.types.iter();
            let factory_name = &self.factory_trait;

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

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

        quote! {
            #input_trait
        }
    }
}

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

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

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

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

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

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

            Ok(())
        }

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

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

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

            let actual = &AbstractFactoryAttribute {
```

```
                factory_trait: parse_str("Factory")?,
                sep: Default::default(),
                types: input_types,
            }
            .expand(&mut t);

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

            Ok(())
        }
    }
}
```

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

```rust
mod abstract_factory;
mod visitor;

extern crate proc_macro;

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

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

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

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

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

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

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

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

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

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

```

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

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

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

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

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

            // Get helper function
            if options.has_helper {
```

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

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

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

            #(#helpers)*

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

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

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

        // Loop over each option given
        for option in options.iter() {
            // "no_default" turns no_default on
```

```
            if option.key == Ident::new("no_default", Span::call_site()) {
                no_default = true;
                continue;
            }

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

        Options {
            no_default,
            has_helper,
            helper_tmpl,
        }
    }
}

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

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

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

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

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

        assert_eq!(actual, expected);
    }

    #[test]
    fn parse_just_types() -> Result {
        let actual: VisitorFunction = parse_str("Button, dyn Text, Window")?;

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

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

```
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             #[tmpl = {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             #[tmpl = {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                 }
```

```
            }

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

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

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

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

        Ok(())
    }
}
```