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

In this section we will discuss the layout of our macro library, the hand-written factory implementations we want the macros to write, the AF macro to write the AF hand-written implementation, and the Visitor macro to write the hand-written Visitor implementation.

2.1 Library layout

Parts of the framework we are creating can be used by other macros. Thus, we want to separate the macro implementations from the structures they will be using. This leads to the libraries shown in Figure 1.

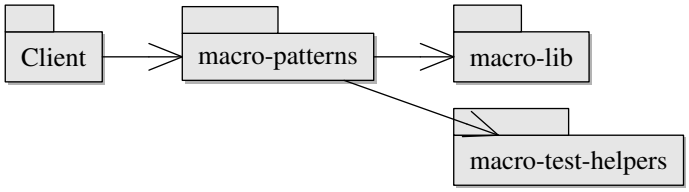

Figure 1: Layout of libraries

Client code will use the *macro-patterns* library. *Macro-patterns* will contain macro definitions as defined in Section **??** for Abstract Factory and Visitor. *Macro-lib* will provide syntax trees 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 patterns implementations will be written by hand. Even though we want to create macros to replace this manual process, we will be writing them here manually once 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.

Listing 1: Simple GUI defined by client

```
1 pub trait Element {
2     fn new(name: String) -> Self
3     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
    }
}
```

Listing 1 defines:

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

The abstract *Button* and *Input* each have a concrete brand version – i.e. *BrandButton* and *BrandInput* that will not be shown.

2.2.2 Hand-written AF

Listing 2 shows a hand-written AF implementation for the GUI.

Listing 2: Hand-written abstract factory

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

Line 2 imports the elements from Listing 1, with line 3 importing the concrete brand elements. 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(BrandElements::BrandButton::new(name))
    }
}
impl Factory<dyn Input> for BrandFactory {
    fn create(&self, name: String) -> Box<dyn Input> {
        Box::new(BrandElements::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

A hand-written visitor can be seen in Listing 3.

Listing 3: Hand-written visitor

```rust
use crate::gui::elements::{Button, Child, Element, 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()),
        };
    });
}
```

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

Visitor consists of three parts:

- The abstract visitor as defined on lines 3 to 14.
- Helper functions for transversing 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 transversal 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.

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. However, because the default implementation in Listing 3 line 12 uses the helper on line 27, *VisitorName* does not need to implement anything for *Window* to be able to transvers into a *Window*'s children.

```
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()));
    }
}
```

2.3 AF macro

The implementation of the AF macro will be used as a foundation to implement the Visitor macro. Thus, we propose reading meta-code – the input to a metaprogram as defined in Section **??** – to a model. This model will be able to expand itself into the pattern implementation as defined in Section 2.2.

The clien meta-code for AF is as follow:

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

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 {}
```

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

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

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

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

2.3.1 Models

Both *abstract_factory* and *interpolate_traits* take in a comma-seperated list of inputs to work against. The *syn* library provides the *Punctuated*[1] type to parse a list of elements seperated by any punctuation marker. *Syn* also provides *Type*[2] for parsing Rust types as used in *abstract_factory*. We will have to create our own element to parse the items passed to *interpolate_trait*.

TraitSpecifier We will call this element *TraitSpecifier* as seen in Listing 4. It will be used to map a trait type to its concrete definition using the syntax *trait => concrete*.

Listing 4: Making a parsable trait specifier

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

```

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

```rust
/// 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()?,
        })
    }
}
```

Lines 1 and 2 import the *syn* elements we will need – imports will not be shown for the remainder of the examples. The model is defined on lines 10 to 14 to hold the abstract trait, the arrow token, and the concrete. The *Token*[3] macro on line 12 is a helper from *syn* to easily expand Rust tokens and punctuations. Lines 17 to 25 implement the *Parse*[4] trait from *syn* for parsing a token stream to this model. Here parsing is simple, we only *parse* each stream token and propagate the errors. *Syn* will take care of converting the errors into compiler errors.

AbstractFactoryAttribute The AF attribute macro will be using the model defined in Listing 5.

Listing 5: Meta-code model for abstract factory macro

```rust
/// Holds the tokens for the attributes passed to an AbstractFactory
 ↪  attribute macro
/// Expects input in the following format
/// ```text
/// some_abstract_factory_trait, type_1, type_2, ... , type_n
```

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

```rust
/// ```
#[derive(Eq, PartialEq, Debug)]
pub struct AbstractFactoryAttribute {
    factory_trait: Type,
    sep: Token![,],
    types: Punctuated<Type, Token![,]>,
}

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)?,
        })
    }
}
```

The model takes the factory method trait as the first input followed by a list of types the abstract factory will create.

2.4 Visitor macro

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.