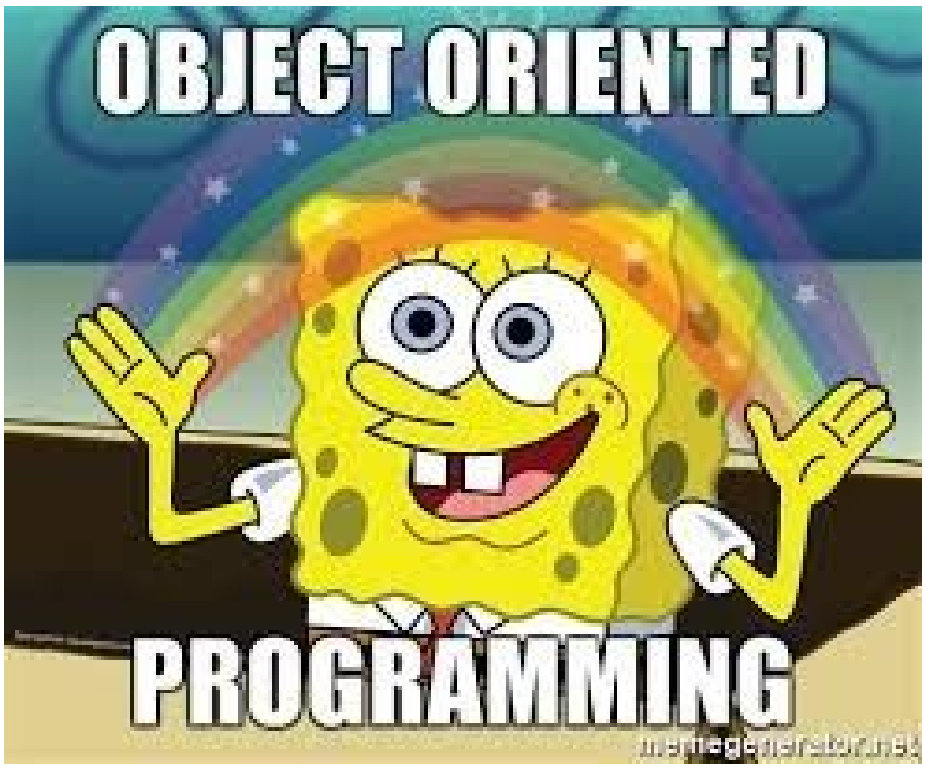
Homework 3 - DePaCoGe IntelliJ Plugin with Name Clashing Analyzer



Emily Lin

CS474

Spring 2020

Table of Contents

Project Overview

In this project, we will be creating the design pattern code generator IntelliJ plugin using Java with the name clashing analyzer function. The SDK for this project is Java8. In order to run the plugin, users have to have the same version of SDK. The design patterns that will be generated are Abstract Factory, Builder, Factory, Facade, Chain of Responsibility, Mediator, Template, and Visitor. This DePaCoGe plugin is designed as a dockable window pane. The tab to open the window is located on the right side of the IDE. When users open the window, they will see a list of available design patterns. After the user chooses the design pattern they want, the pop-up window will prompt the user to enter the interface, class, methods and field names. If the names that the user enters don't have name clash error, the generated code will be inside the user's project's src folder so that the user can easily locate the generated design pattern. Otherwise, the error dialog window will show the user which name they entered is clashing with other classes, methods or objects in the same scope. All generated files are in .java format.

Project Design

Project Model

In this project, I use Gradle with IntelliJ plugin function to build the project and use IntelliJ as my IDE. The reason why I chose Gradle is that Gradle avoids unnecessary work by only running the tasks that need to run because their inputs or outputs have

changed. Also, when we want to import the libraries or use the frameworks, we can add the libraries and frameworks to the dependencies in Gradle. Gradle uses dependency management, which is a technique for declaring, resolving, and using dependencies required by the project in an automated fashion. The parser I use to generate the design pattern is Javapoet. The reason behind it is Javapoet is easy to follow and use. Also, Javapoet provides the flexibility for me to take user input from the pop-up windows and generate the design pattern in runtime.

Design Pattern Usage

For implementing the hw3, I import the builder part of my code from hw2. The reason is that the builder part is the part that generates the code. The builder part will be interacting with the new plugin code that I implemented.

Implementation of Plugin

The approach I used to implement this project is I have a class which implements the ToolWindowFactory. This ToolWindowFactory is an interface which contains methods needed to create a tool window which is the dockable window pane. The tool window serves as a container for the UI that users will be interacting with. After that, I have a class that extends the SimpleToolWindowPanel abstract class. In this class, I create a string label that asks users to choose a design pattern and a list of design patterns that users can click on. In this class, I also have a loop that will generate the dialog depending on what design pattern users chose. I also implemented eight dialog classes for each design pattern. The dialog will be presented after users

click on the design patterns' list. Each dialog contains the string labels and text fields. The string labels will ask users to put the name of classes, interfaces, methods and fields' name. The text fields will hold the users' input. Each dialog contains different questions depending on what design pattern users want. Each generated design pattern shows the main functionality and it's easy for the user to follow and extend it if needed.

Implementation of Name Clashing Analyzer

The approach I took for implementing the name clashing analyzer for my plugin is I have a class which acts as a file storage for the user's project. This class will iterate through the project that the user currently opened. If the file is in .java, I put that file as PsiFile in my file list. Since my generated codes are in Java, I only check the name clash for the files that are in the .java format. After iterating through the user's project and when the user selects the design pattern they want and enters the names that needed to generate the design pattern, I have a name clash checker method in each design pattern's dialog class. The reason why I took this approach is because each design pattern has different things that need to be checked. Some design patterns only need to check if there's a name clash for class. Others need to also check methods or objects. If there's no name clash after the user enters the name of the classes, methods or objects, the code of the design pattern will be generated. Otherwise, ther error dialog will pop up and tell the user which name they entered has a name clash and they need to change their input. The design pattern won't be generated if there's a name clash.

Results

Abstract Factory

For the Abstract Factory design pattern, I generate one abstract factory interface which has one method, which returns a product instance, and one factory concrete class that implements the abstract factory interface. In this factory concrete class, I have a method which overrides the method in the abstract factory interface and will return a new instance of the product concrete class. Then I have a product interface with a method in which the return type is String. The purpose of the method is to get the name of the product concrete class. Finally, I have a product concrete class which implements the product interface. In this product concrete class, I have one method overrides the method in the product interface. This method will return the name of the product concrete class.

Builder

For the Builder design pattern, I generate one builder interface which has two methods in it. One of the methods serves the purpose of building the product with return void type. The second method is to get the result which returns a complex object instance. I generate one builder concrete class which implements the builder interface. In the builder concrete class, I have one complex object class object and two methods which override from the builder interface. In the build product method, I create a new product concrete class object. I have one complex object class in which the user can extend the

functionality in their desired way. I have a product interface with a method that has the String return type. The purpose of this method is to return the product concrete class name. I have another subproduct interface which extends the product interface. Finally, I have a product concrete class which implements the subproduct interface. In this concrete class, I override the method in the product interface which will return the product concrete class name.

Factory

For the Factory design pattern, I have one creator abstract class with a private product class instance and an abstract method with the product return type. I have a creator concrete class that extends the creator abstract class. In the concrete class, I override the method which returns a new product concrete class object. I have a product interface with a method which has the String return type. This method will get the name of the product concrete class. Finally, I have a product concrete class which has a method overrides from the product interface which returns the name of this product concrete class.

Facade

For the Facade design pattern, I have a facade abstract class with an abstract void method. The purpose of this method is to do the user's operation of this facade design pattern. I have a facade concrete class which extends the facade abstract class. In this facade concrete class, I have a private instance of the class which will do the operation.

Also, I have a constructor for the facade concrete class and a method override from the facade abstract class. In this method, I use the class instance to call the operation method. Finally, I have a class which has a method for the user to do the operation they want.

Chain of Responsibility

For the Chain of Responsibility design pattern, I have a handler abstract class with a private instance of the handler abstract class, a handler constructor with no parameter, another handler constructor with parameter, and a void method which is to handle the request and a boolean method which is to check if the class can handle the request or not. I have two handler concrete classes in which the first concrete class will pass down the request to the second concrete class.

Mediator

For the Mediator design pattern, I have a mediator abstract class with an abstract method with the purpose of mediating the colleague. I have a mediator concrete class which extends the mediator abstract class. In this concrete class, I have instances for both colleague concrete classes, a method for setting the colleague, and a method overrides from the mediator abstract class for mediating the colleagues. I have a colleague abstract class with a mediator abstract class instance and a constructor. Finally, I have two colleague concrete classes which will set the state.

Template

For the Template design pattern, I have an abstract class which contains a protected void method that the subclass will implement, and a method which invokes the protected void method. Finally, I have a subclass extending from the abstract class and a method override from the abstract class.

Visitor

For the Visitor design pattern, I have an element abstract class which contains an abstract void method which will be dealing with visiting the element class. I have an element concrete class that extends the element abstract class with an override method from the element abstract class and a void method for doing the operation for this design pattern. I have a visitor abstract class with an abstract void method for visiting the element. Finally, I have a visitor concrete class that extends from the visitor abstract class, and a method in this concrete class. The method is overridden from the visitor abstract class, which will be dealing with calling the operation method in the element concrete class.

Test Cases

For the test cases, I used the Mockito framework to test since most of my methods are void methods. Also, mocking the target classes can prevent creating a lot of string objects in the test cases since every class I have all has arguments. My test cases focus on

if the right file path is passing around the classes. I also tested if the string object which stores the user input is matching with the corresponding string object in my generator classes. For testing the name clash analyzer, I tested if the name clash checker is working properly by putting some random classes' names in the list, and passing the list to the name clash checker.

Conclusion

The limitation of my DePaCoGe IntelliJ plugin is that I did not let the user choose how many concrete classes they want that will either implement the interface or extend the abstract class. Also, the other limitation of my DePaCoGe is that I didn't let the user define the class that will do the operation of the design pattern. My DePaCoGe provides the general idea of what each design pattern is supposed to do. The user can easily extend the functionality or change the signature of the methods based on their need in the future. For the test cases, I tested if the right file path is passing to the classes that needed. Also, for each design pattern, I tested if the user input is stored to the right variable that will be used to generate the code. Adding the name clash analyzer will make sure that there's no name clash error appearing in the same scope in the user's currently opened project. This will save some time for the user trying to find where the error is after the design pattern is generated.

Index

Gradle : https://docs.gradle.org/current/userguide/userguide.html

IntelliJ plugin : https://www.jetbrains.org/intellij/sdk/docs/tutorials/build_system/prerequisites.html

JavaPoet : https://github.com/square/javapoet

Mockito : https://site.mockito.org