MP2: Design Patterns

1 Introduction

Since you are now familiar with the Jsoup project (at least you know how to build it ◡̈), let's practice program analysis on the Jsoup project!

Imagine you are a software engineer working on a large project. There will be various programming tasks you need to perform perform, including understanding existing code, modifying existing code, and of course producing new code. As a graduate student in CS, you probably are already very good at such tasks as you have learned such skills from various courses. Meanwhile, other courses mainly teach you how to automate various tasks in our daily life (such as financial services, transportation, healthcare, and more) via programming, and you still have to manually perform those programming tasks. In contrast, with program analysis, you can further automate programming tasks themselves!

2 Instructions

JavaParser (https://javaparser.org/) is a widely used analysis engine for Java programs. It can transform any Java source code snippet into its abstract syntax tree (AST) form. Then, we can build programs to traverse the ASTs to perform various code parsing, manipulation, and generation tasks at the AST level. Note that JavaParser is largely built based on the Visitor design pattern (https://en.wikipedia.org/wiki/Visitor_pattern). Here are some useful links for getting started with JavaParser:

- https://javaparser.org: JavaParser homepage with various useful resources, including blogs on how to use JavaParser in various ways.
- https://javadoc.io/doc/com.github.javaparser/javaparser-core/latest/index.html: JavaDoc information for all JavaParser APIs.
- https://gitter.im/javaparser/home: the gitter channel for JavaParser, where you can ask questions and search previously solved questions.
- https://tomassetti.me/wp-content/uploads/2017/12/JavaParser-JUG-Milano.pdf: a tutorial about JavaParser.

To simulate real-world software development, in this MP, you are expected to get familiar with the popular JavaParser analysis engine by yourself, and perform the following three tasks. To simplify the implementation and auto-grading, you will revise an existing template Maven-based project. You can download the template project using the command below:

```
git clone https://github.com/CS427-UIUC/cs427-mp2
```

Note that the template project already includes the dependency of a popular JavaParser version from the Maven Central Repo. Furthermore, all the source code for Jsoup (which will be used as the input data) has also already been included as test resources, which will be automatically copied into `target/test-classes` directory when running `mvn test`. Therefore, you do not need to install or change anything for the project, except adding your implementation to the locations with **"TODO"** comments (details will be shown in the later task description).

Your implementation should pass all JUnit tests included in the `edu.illinois.cs.analysis` package of the template project. You can run tests via `mvn test` command or through the IDE. If you execute the tests through the command line, you can look at the generated Surefire reports (`target/surefire-reports`) for the details about your test failure(s), if any. Note that at the beginning, all the tests will fail, as the code is incomplete and cannot produce the expected results.

2.1 Task 1: Parsing existing code

Assume you are working on the Jsoup project, and one day your manager asks you to compute some statistics (such as the number of classes or methods) of the project. You have multiple options even without knowing the concept of program analysis:

- Just do it manually. However, this will take you forever in case of huge projects.
- Write some simple scripts based on string processing. It also works, but will take you a tremendous amount of time to consider all corner cases.
- Try to find some existing tools. Depending on the specific statistics, you may not find an ideal tool that can compute all statistics.

In this task, you are expected to compute the number of methods satisfying ALL of the following conditions within a Java file based on the JavaParser program analysis tool:

- The method has actual body declaration (i.e., for method declaration `n`: `n.getBody().isPresent()` should be `true`).
- The method has at least one input parameter.
- The method is `public`.
- The method is **not** `static`.
- The method has a return type that is **not** `void`.

More specifically, you should complete the TODO parts in the `edu.illinois.cs.analysis.CodeParser` class so that it can automatically compute the precise number of methods. By default, it only computes the total number of methods within a file. When you are done with this task, make sure your implementation passes all JUnit tests included in the `edu.illinois.cs.analysis.CodeParserTest` class of the template project.

When you are done, please take a moment to think about the benefits of using such a program analysis engine compared with all the other three ways for completing the same task. It could save you even more time for a more complicated or specialized code-parsing task!

2.2 Task 2: Modifying existing code

Another day, your manager asks you to find all instances of `null` checks in any given Java file, and then switch the operands for all found `null` checks (what a weird request! ◡̈). Note that you shall consider ANY binary expression that:

- includes `null` as an operand (either left or right side), and
- includes `==` or `!=` as the operator.

You are expected to complete the TODO portions in the `edu.illinois.cs.analysis.CodeModifier` class so that it can automatically make the change. When you are done with this task, make sure your implementation passes all JUnit tests included in the `edu.illinois.cs.analysis.CodeModifierTest` class of the template project.

2.3 Task 3: Generating new code

Yet another day, your manager comes and asks you to build a program that can automatically generate the following method and add it to any given class:

```
@Override
public String toString() {
        String str = super.toString();
        int len = str.length();
        if (len > 40)
                return "OMITTED";
        else
                return str;
}
```

The reason for the change is that the original `toString` method (e.g., inherited from class `Object`) may return very long string and cram the hard drive with large chunk of such log data. The new `toString` method simply returns "`OMITTED`" for any class with string representation longer than 40 characters. In this way, the space issue can be mitigated. You can assume that the given Java file does not include `toString` declarations before your change (so you can directly add it without any check). You also do not need to worry about the precision issue caused by the string omission.

You are expected to complete the TODO portions in the `edu.illinois.cs.analysis.CodeGenerator` class so that it can automatically add the desired code for the given file. To help with your implementation, we have included partial implementation. Furthermore, we have also included the AST tree information for the body of added method; you may find it helpful to follow this tree structure to construct the corresponding code structure. You can also create the AST tree yourself by running the `main` method from class `edu.illinois.cs.analysis.CodeGenerator`. The execution will dump a file named `ast.dot` in the root directory of this template project. You can simply copy the file content and display it using any GraphViz software (e.g., you can also use the online version: https://dreampuf.github.io/GraphvizOnline).

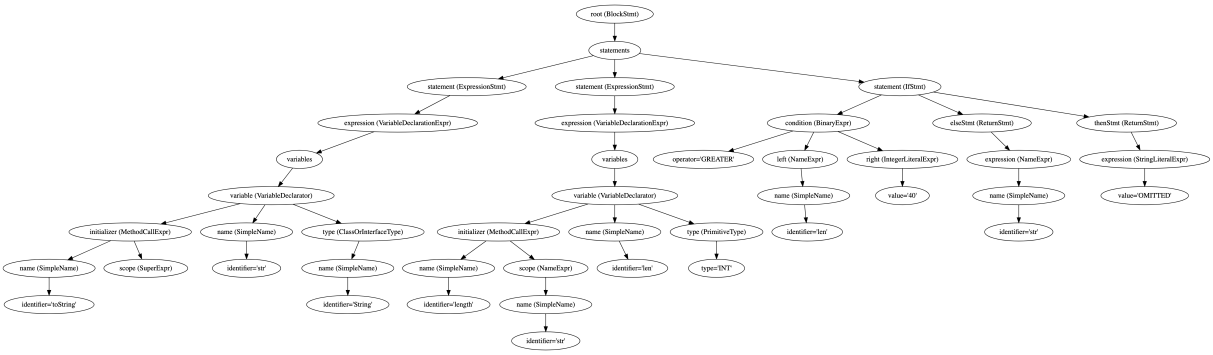

Figure 1: AST of the code to be generated

3 Deliverables

You are expected to upload a zip file including only the `CodeParser.java`, `CodeModifier.java`, and `CodeGenerator.java` files you completed (no directories please). The name of the zip file should be your NetID.

Warning: you may lose all your points if you violate the following rules:

- Please make sure that your zip file is named with your NetID and only includes the three specified files: **DO NOT include any directories in the zip file**, and **DO NOT change anything (including names) of the three files except the TODO parts**.
- Please **DO NOT use any absolute paths** in your solution since your absolute paths will not match our grading machines. Also, please **only use "/"** as the file separator in your relative paths (if needed for this MP) since Java will usually make the correct conversions if you use the Unix file separator "/" in relative paths (for both Windows and Unix-style systems).

- Please **DO NOT fork any assignment repo** from our GitHub organization or share your solution online.

4 Grading rubric

The autograder will run tests on your submitted code and your final score for this MP will be based on the success rate of test execution (including the 5 provided tests plus 5 hidden tests). The overall score is 5pt:

- In general, your score will be proportional to the total pass rate, e.g., if you pass 8 out of the 10 tests, you will receive 4pt.
- However, we may manually check the actual code for some sample solutions (those with more hidden tests failing will get a higher chance to be sampled), and your score can be proportional to your test pass rate or (very likely) lower.