JML Template Generation

2017

Kushal Raghav Poojari
University of Central Florida

JML TEMPLATE GENERATION

by

KUSHAL RAGHAV POOJARI
B.S. Jawaharlal Nehru Technological University, 2015

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2017

ABSTRACT

The Java Modeling Language (JML) is a behavioral interface specific language designed to specify Java modules (which are Java classes and interfaces). Specifications are used to describe the intended functionality without considering the way it is implemented. In JML, if a user wants to write specifications for a Java file, he or she must undertake several steps. To help automate the process of creating annotations for method specifications, a tool Jmlspec was created. Jmlspec generated a file that refines the source file and has empty placeholders in which one can write specifications. Although Jmlspec worked with older versions of Java, it does not work with the current version of Java (Java 8). This thesis describes the implementation of a new version of the Jmlspec tool that is compatible with newest versions of Java. This tool will be more maintainable than the older version of Jmlspec and easier to extend.

I dedicate this thesis to my parents P. Satyanarayana and E. Renuka Devi, who have loved me unconditionally and whose good examples have taught me to work hard for the things that I aspire to achieve.

ACKNOWLEDGMENTS

Firstly, I would like to extend most profound appreciation to my advisor, Dr. Gary T. Leavens, for his encouragement and excellent guidance. Without his support, this thesis would not have been possible. I would also like to thank the members of my thesis committee, Professors Damian Dechev and Damla Turgut for their advice and guidance during the entire process. Special thanks to my brother, Kumar Raghav Poojari, who has been a constant source of support and encouragement during the challenges of graduate school and life.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

ANTLR	ANother Tool for Language Recognition
AST	Abstract Syntax Tree
AWT	Abstract Windowing Toolkit
BISL	Behavioral Interface Specific Language
DBC	Design by Contract
EBNF	Extended Backus-Naur Form
IDE	Integrated Development Environment
JDK	Java Development Kit
JFC	Java Foundation Classes
JML	Java Modeling Language
UI	User Interface

CHAPTER ONE: INTRODUCTION

The Java Modeling Language (JML) is a behavioral interface specific language (BISL) designed to specify Java modules. JML is based on the Design by Contract (DBC) approach and model-based specification approach of Larch family of interface specification languages [1]. The JML Template Generation tool generates specification skeletons for the Java files and is compatible with the newest versions of Java. This chapter gives a brief description of JML background, need for the JML Template Generation Tool, the old Jmlspec tool, and the problem with it and the approach for new design.

Background on JML

Specifications are used to describe the intended functionality of the software. JML specifications are written for Java modules like Java classes and interfaces. Specifications are a contract between a class and the clients of the class. In this contract, the client guarantees some conditions to be satisfied before calling the method of the class, and in return the class guarantees some conditions that will be satisfied after the call. So, a contract is written by specifying a methods pre-conditions and its post-conditions. Using JML specifications to describe the intended behavior of methods and classes adds the advantage of finding bugs easily.

JML specifications are written by adding special annotation comments to the Java code. These special annotation comments start with an at-sign `(@)`. Single line annotations are of the form `//@` and multi-line annotations are of the form `/*@...@*/`. These special annotations are placed within Java comments so they will be ignored by a Java Compiler whereas any specifications written with these special annotations are used by the JML compiler. The JML

Compiler outputs code that raises a runtime exception if the code's behavior does not match the specifications written. The `@` must be right next to the `//` in `//@`, otherwise the comment (e.g., one beginning with `// @`) will be ignored by the JML compiler. Similarly, one should use `/*@` as `/* @` will also be ignored by JML. An example of JML specification with single and multi-line annotations is shown in Figure 1.

Figure 1 shows pre-conditions and post-conditions written for a method, `sqrt` that takes a number and returns the square of the number.

```
//@ requires x >= 0.0;
/*@ ensures JMLDouble
  @         .approximatelyEqualTo
  @          (x, \result * \result, eps);
  @*/
public static double sqrt(double x) {
  /*...*/
}
```

Figure 1 Pre-and post-conditions for sqrt method (adapted from [2])

A method's pre-conditions specify the conditions on the program state that must be satisfied to call the method. JML uses the keyword `requires` to introduce preconditions. Figure 1 says that the precondition for `sqrt()` is that `x` should be a non-negative value. A method's post-conditions specify the method's responsibilities; that is, after the method returns, the post-conditions must be true. JML uses the keyword `ensures` to introduce post-conditions. Post-conditions can be distinguished into normal post-conditions and exceptional post-conditions. Normal post-conditions specify the conditions that must be true when the method returns without throwing an exception. For example, normal post-conditions for `sqrt()` is specified as follows.

```
/*@ ensures JMLDouble
  @         .approximatelyEqualTo
  @          (x, \result * \result, eps);
  @*/
```

Figure 2 Normal post-conditions for square root method (adapted from [2])

Exceptional post-conditions use a `signals` or a `signals_only` clause, which must be true when a method terminates with an exception. Java allows a class or method to throw a runtime exception, but JML allows a method to throw a runtime exception only if it is specified in method's header (in method's throw clause) or if specified in method contract's `signals_only` clause. The default `signals_only` clause is to allow the exceptions that are specified in method's `throws` clause to be thrown. In Figure 1 there is no `throws` clause in method's header which prohibits the method from throwing an exception. An example of a method with an exceptional post-condition is shown in Figure 3.

```
1  class SettableClock extends TickTockClock {
2
3      // ...
4
5      /*@ public normal_behavior
6        @   requires   0 <= hour && hour <= 23 &&
7        @              0 <= minute && minute <= 59;
8        @   assignable _time_state;
9        @   ensures    getHour() == hour &&
10       @              getMinute() == minute && getSecond() == 0;
11       @ also
12       @  public exceptional_behavior
13       @   requires     !(0 <= hour && hour <= 23 &&
14       @                  0 <= minute && minute <= 59);
15       @   assignable   \nothing;
16       @   signals      (IllegalArgumentException e) true;
17       @   signals_only IllegalArgumentException;
18       @*/
19     public void setTime(int hour, int minute) {
20       if (!(0 <= hour & hour <= 23 & 0 <= minute & minute <= 59)) {
21         throw new IllegalArgumentException();
22       }
23       this.hour = hour;
24       this.minute = minute;
25       this.second = 0;
26     }
27 }
```

Figure 3 Exceptional post-conditions example (adapted from [3])

JML does not expect one to specify behavior completely. A specification case is where a user decides to describe the specifications either shortly assuming defaults or precisely including exceptional behaviors. JML has two styles of method specification cases: lightweight and heavyweight. Lightweight specification cases are used to give partial specifications, in which the user only describes what he is interested in and heavyweight specifications are used to give a complete specification for some pre-condition [4]. In heavyweight specification cases, JML expects that the user is aware of the defaults involved and omits the part of specification where defaults are appropriate. Lightweight specifications can be used for documentation purposes. Heavyweight specifications can be used with runtime assertion checking and static checking (including verification).

The Need for JML Template Generation

Automation is everywhere. When you are at a stoplight, the lights change from red to green automatically without a person acting as a light operator. Automation of mundane programming tasks can make programming more pleasant and productive. So it is with the automation of specification generation for JML. The User should go through steps like selecting the file, opening it in an IDE, adding annotations for methods where he or she wants to write specifications to, saving the file while editing the extension to be .jml. These mundane tasks can be automated with the use of the JML Template Generation tool.

JMLSPEC

Jmlspec is a tool designed to automate the process of adding the annotations for methods to write specifications. Jmlspec and the graphical user interface (GUI) version Jmlspec-gui have

two major modes: generation of specification skeleton and comparing specification files [5]. Jmlspec without `-diff` option selects the generation mode, whereas Jmlspec with a `diff` option sets the comparison mode.

In generation mode, for each Java source file given on the command-line, Jmlspec generates a file that refines the given source file and has an empty placeholder for one to write in the specifications [5]. In the old version of Jmlspec, a file with the generated specification skeleton had the name same as the Java source file but with a `.refines-spec` extension. For example, if the Java source file is `HelloWorld.java` then the file with the generated specification skeleton using jmlspec will be named `HelloWorld.refines-spec`. The user had to edit the `.refines-spec` file (`HelloWorld.refines-spec`) file to add the desired JML specifications.

In comparison mode, Jmlspec found all the matching files in the user's CLASSPATH and compared the declarations in those files to each file that was given on the command line. After the comparison was done, it printed the differences found.

Problem with Jmlspec

The original version of the Jmlspec was built on MultiJava compiler. MultiJava is an extension to the Java programming language that adds symmetric multiple dispatch and open classes which allow programmers to add methods to existing classes without editing those classes, or even having their source code [6]. The latest version of MultiJava released is version 1.3.2 in August 2006. This compiler does not support the Java 5 source language. So, the old version of Jmlspec works with the older versions of Java but not with the newest versions (like

Java8). Another problem with Jmlspec is that it generates wrong suffix for files, `.refines-spec`, the `refines-spec` extension is no longer supported by JML.

The problem is to create a new JML Template Generation tool that automates the process of generating skeleton specifications for Java files and that works with the newest version of Java.

Approach for New Design

JML Template Generation design is a four-phase process. First, a Java source file is passed as input to the tool. A Lexer takes this input and converts it into tokens. In the second phase, the Parser asks for the tokens which are generated by the Lexer. Tokens generated by the Lexer are passed into the Parser as input. The Parser then generates an Abstract Syntax Tree (AST) and passes this AST to the Tree Parser which checks the syntax of the source code and validates it. Each leaf node in the AST is a token passed from the Lexer phase. In the next phase, the AST along with tokens are passed into a Transformer which modifies the AST by adding annotation blocks (nodes or tokens) to the AST. Tokens modified in the Transformer phase are passed into a Printer which formats the program's indentation and prints the resulting output. This output is saved as a .jml file.

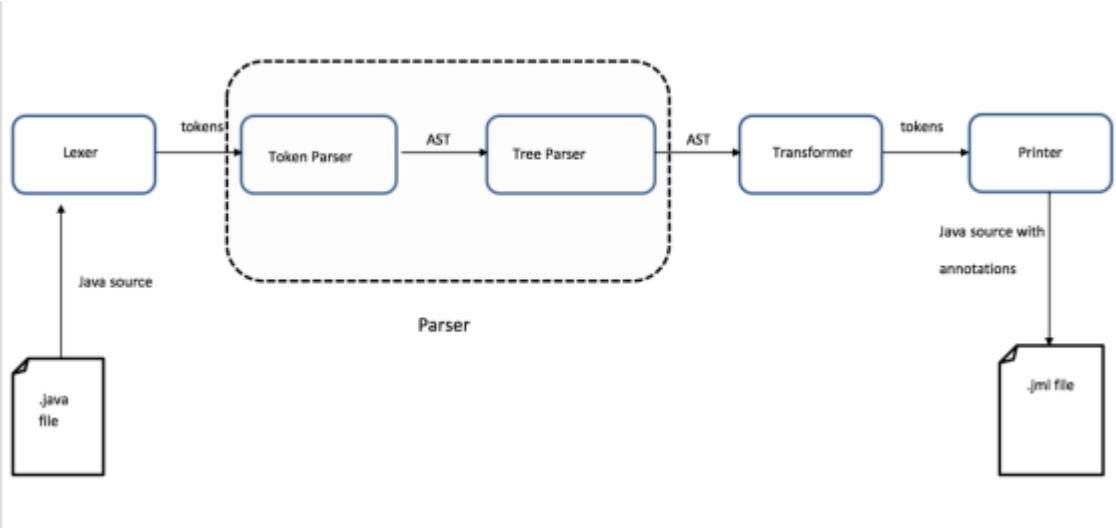

Figure 4 JML Template Generation Design

Overview of the Thesis

Chapter Two explains more details about the need for JML Template Generation. Chapter Three discusses the implementation details of the JML Template Generation tool proposed in this thesis. The Organization of the implementation's code is discussed in Chapter Four. A discussion about future work, alternate designs proposals and contributions are presented in Chapter Five.

CHAPTER TWO: THE PROBLEM

The old Jmlspec tool generates specification skeletons for files, but it does not support the newest versions of Java, as it was built on the old MultiJava compiler. The MultiJava compiler does not support the Java 5 source language, so Jmlspec also does not support any of the newest versions of Java. The new JML Template Generation tool supports the newest versions of Java as the Lexer, and the Parser are generated using a grammar file for Java's newest version. In addition, the new JML Template Generation tool will be able to support any upcoming version of Java by replacing the Lexer and the Parser with an updated version of grammar file. Tree Parser can also be replaced with an updated version of the Parser file generated with an updated grammar file for the Tree Parser (`TreeParser.g`).

The JML Template Generation tool will generate annotations as tokens in the transformer phase and sends the entire source code as a stream of tokens to the Printer. The Printer then converts these tokens to strings, adds indentation to it and prints the output. Care must be taken to place the annotation tokens at the correct position so that they appear exactly above method and variable declaration and with the correct amount of indentation. This can be done by traversing through each node and finding the method nodes and variable nodes, then adding these annotation tokens as a child node to the method node or variable node just before the existing tokens of the method or variable nodes.

Annotations must contain `@also` in the child class method, if same method exists in parent class and child class. This can be done by keeping track of all the classes in parent and child classes and their methods. If a same method exists in parent and child class, then a prefix

`also` is added to the block annotation generated. If no common method exists in parent and child class, then the prefix defaults to an empty string.

CHAPTER THREE: THE SOLUTION APPROACH

Introduction

This chapter discusses the key implementation details of JML Template Generation. ANother Tool for Language Recognition (ANTLR) is used to generate a Lexer, a Parser, and a TreeParser and to transform the AST generated after the parsing phase.

ANTLR is a tool that generates the parser that uses LL(*) parsing. In order to generate these, ANTLR takes a context-free grammar expressed using EBNF as input that specifies a language.

Why ANTLR?

The JML Template Generation tool is designed using ANTLR, thus making it a standalone tool. ANTLR is self-contained with several methods to modify the AST with ease. The main alternative to building a standalone tool would be to build the tool using the OpenJML infrastructure. However, designing a tool with OpenJML makes it depend on other parts of OpenJML. If the tool has to be updated in future, these dependencies would make it harder to update the tool. On the other hand, if OpenJML must be updated in any case, making the tool depend on OpenJML may be a way to have the tool stay in sync with updates to Java (and OpenJML). On the other hand, it is easier to maintain a software with fewer dependencies on it when compared to a software with more dependencies. Maintaining and updating this tool designed with ANTLR will be easier, as updating the tool only requires one to replace the Lexer,

the Parser and the TreeParser files with an updated version of these files generated with an updated grammar file.

Architecture

Key components of JML Template Generation are a Lexer, a Parser, a TreeParser which are generated by ANTLR and a Transformer which adds annotations as tokens and a Printer which adjusts positioning and spacing of the JML file.

ANTLR Generated Java Lexer

The ANTLR generated Java Lexer is built on top of the lexical rules present in the grammar file which is passed to ANTL. A Lexer breaks an input stream of characters sent to it into tokens.

In the grammar file used to generate a Lexer, lexical rules are defined. These rules defined in the lexical grammar are implicitly matched to the input stream of characters. The following rule defines a rule called `COMMENT`, if the characters are matched to this rule then they are ignored.

```
COMMENT
    :   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
    ;
```

Figure 5 Lexer rule for a Comment

This rule, defined in the grammar file, will generate a part of the Lexer. The generated code would appear as a method called `mCOMMENT().` The generated method is shown in Figure 8. The above lexical rule treats text in between `/*` and `*/` as a comment. `{greedy=false;}` implies the author wants the sub-rules that loop, which appear as `(..)*` and `(..)+`, to not exit

until they see a lookahead consistent with what follows the loop [7]. The notation `$channel=HIDDEN` tells ANTLR to have the lexer generate a token, but to hide the token from the parser by adding it to a hidden channel.

In a lexer, typically white spaces, comments and any other content which does not have a semantic value are removed by the Lexer. In lexical rules, a `skip` command notifies the Lexer to throw out the current text and get the next token. A lexical rule where the `skip` command is used is shown in Figure 7.

```
WS  :  [ \t\r\n\u000C]+ -> skip
    ;
```

Figure 6 Example of a lexical rule where a skip command is used

The `skip` command prevents the generation of a token, unlike `$channel=HIDDEN`, which generates the token but hides it from the parser. In some grammar files, `$channel=HIDDEN` and `skip` are enabled using a condition. The user must change the condition values to either generate the token or hide the token from the Parser.

```
if (!mPreserveBlockComments) {
    skip();
} else {
    $channel = HIDDEN;
}
```

Figure 7 Example that uses a condition to prevent generation of the token or hide the token from parser.

```
// $ANTLR start "COMMENT"
public final void mCOMMENT() throws RecognitionException {
    try {
        int _type = COMMENT;
        int _channel = DEFAULT_TOKEN_CHANNEL;
        // Java.g:1176:5: ( '/*' ( options {greedy=false; } : . )* '*/' )
        // Java.g:1176:9: '/*' ( options {greedy=false; } : . )* '*/'
        {
            match("/*");
            // Java.g:1176:14: ( options {greedy=false; } : . )*
            loop25: while (true) {
                int alt25 = 2;
                int LA25_0 = input.LA(1);
                if ((LA25_0 == '*')) {
                    int LA25_1 = input.LA(2);
                    if ((LA25_1 == '/')) {
                        alt25 = 2;
                    } else if (((LA25_1 >= '\u0000' && LA25_1 <= '.') || (LA25_1 >= '0' && LA25_1 <= '\uFFFF'))) {
                        alt25 = 1;
                    }

                } else if (((LA25_0 >= '\u0000' && LA25_0 <= ')') || (LA25_0 >= '+' && LA25_0 <= '\uFFFF'))) {
                    alt25 = 1;
                }

                switch (alt25) {
                case 1:
                // Java.g:1176:42: .
                {
                    matchAny();
                }
                    break;

                default:
                    break loop25;
                }
            }
            match("*/");

            if (!preserveWhitespacesAndComments) {
                skip();
            } else {
                _channel = HIDDEN;
            }
        }
        state.type = _type;
        state.channel = _channel;
    } finally {
        // do for sure before leaving
    }
}
// $ANTLR end "COMMENT"
```

Figure 8 `mComment()` method generated for the Lexer

ANTLR Generated Java Parser

The job of the ANTLR generated parser is to determine what sequences of tokens are valid. The Parser receives the stream of tokens and organizes these tokens into a sequence as defined in the grammar. If the language is used as defined in the grammar, then the Parser will recognize the series of tokens and group them together to make ASTs. An error is issued if the series of tokens issued by the parser does not match the grammar.

The Parser converts these valid sequences of tokens into an AST. Symbol tables are generated by the Parser that contains information about the tokens. Symbol tables are used for type checking and generation of object code.

The Parser consists of a set of rules defined in the grammar file passed to ANTLR. A rule may be a basic rule or the one which may contain alternatives. A rule is defined by one or more alternatives terminated with a semicolon. A Basic rule is defined by one. For example, a basic rule is shown in Figure 9. A rule with more alternatives is shown in Figure 10.

```
fieldDeclaration
    :   typeType variableDeclarators ';'
    ;
```

Figure 9 Basic Parser rule

```
interfaceMemberDeclaration
    :   constDeclaration
    |   interfaceMethodDeclaration
    |   genericInterfaceMethodDeclaration
    |   interfaceDeclaration
    |   annotationTypeDeclaration
    |   classDeclaration
    |   enumDeclaration
    ;
```

Figure 10 Parser rule with alternatives

Alternatives can be a basic rule, a list of rules or empty. A rule with an empty alternative is shown in Figure 11.

```
interfaceBodyDeclaration
    :   modifier* interfaceMemberDeclaration
    |   ';'
    ;
```

Figure 11 Parser rule with an empty alternative.

A TreeParser is generated by ANTLR from a grammar for the TreeParser (`TreeParser.g`). This TreeParser uses the AST generated by the TokenParser to traverse it and check for syntax errors. The TreeParser expects the AST generated from the TokenParser (which is generated using `Java.g`) as an input.

AST and tokens are passed into `JavaTransformer` to modify the AST and add annotation blocks to the code before variable and method declarations. `JavaTransformer` will be discussed in details in later sections.

Design of the Tool

For convenience, the design of the tool is categorized into the design of the User Interface (UI) and the design of the core part of the tool. Each is covered in a separate subsection below.

UI of the tool

The UI of the tool is designed using Java Foundation Classes (JFC). JFC is a Graphical User Interface Toolkit which is the successor of Abstract Windowing Toolkit (AWT). AWT and Swing are used to develop the UI of the tool. Swing is a Graphical User Interface Widget Toolkit for Java [10].

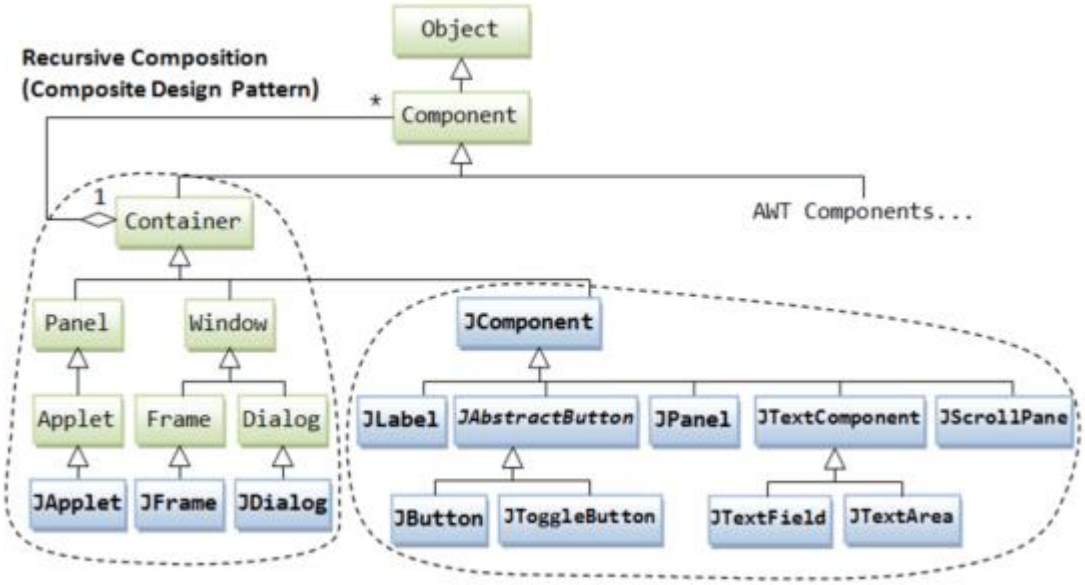

Figure 12 Class Hierarchy of Swing GUI classes (adapted from [9])

Figure 12 shows the class hierarchy of Swing GUI classes. There are two groups of classes Containers and Components. A container holds components and other classes. `JFrame`, `JDialog` and `JApplet` are three top-level containers in Swing.

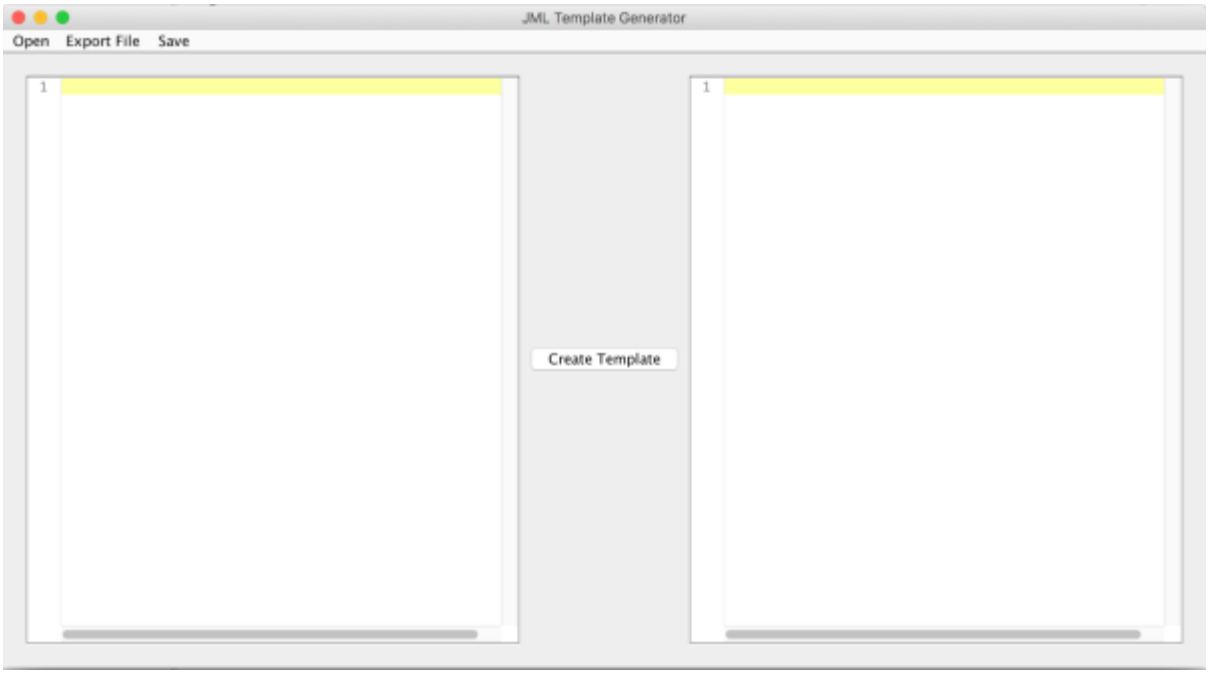

Figure 13 JML Template Generation UI - using JFC

Figure 13 shows the UI of the tool. This UI contains a Frame (title set to JML Template Generator), `MenuBar`, text component and a button. `MenuBar` provides options to open existing Java files located in the file system and to save the JML Template generated to the file system with a `.jml` extension (done using `JFileChooser` which is a swing component). Input is provided by selecting a desired Java file from the file system. Once the input is provided, then the JML Template is created by pressing the "Create Template" button (done using `JButton`, a swing component). Figure 14 shows the JML Template generated. The User can either get the annotations for the entire Java source code provided or get annotations for only a selected portion of the code. Annotations for the entire source code are generated by default. If a specific portion of the Java source code is selected and then Create template button is clicked, then the annotations are generated for only the selected portion of the Java source code.

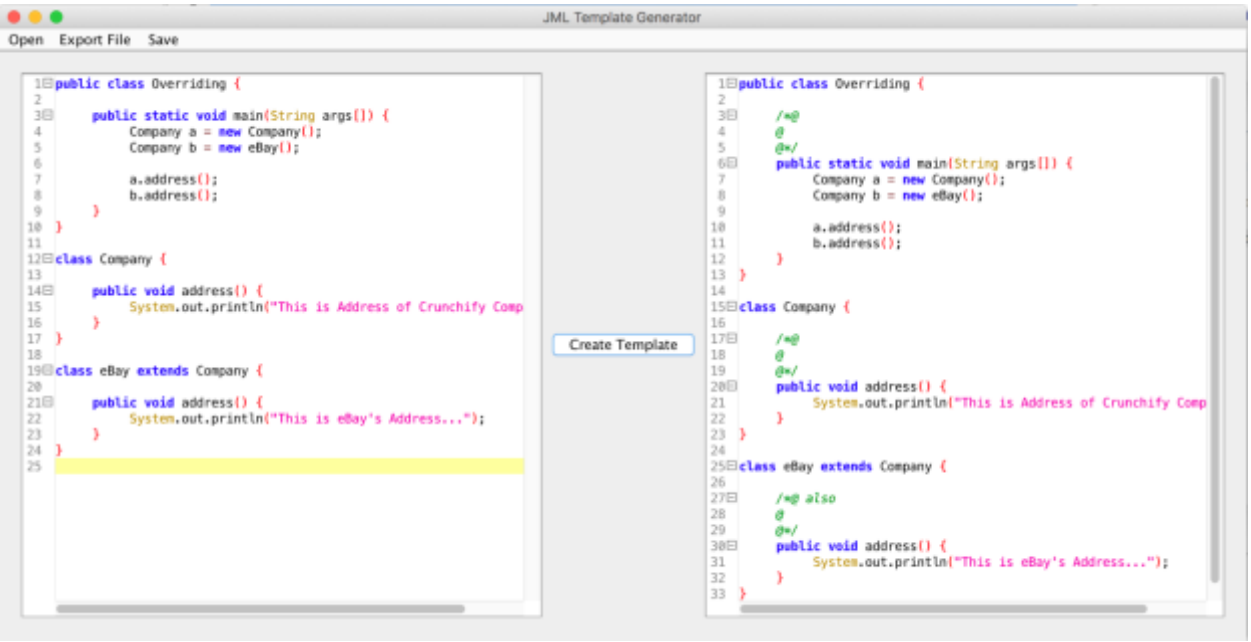

Figure 14 Final JML Template generated

Figure 15 shows the JML Template generated for the selected portion of the source code.

After the template is generated it can be exported and saved as a `.jml` file in the file system by clicking on the Save button on the `MenuBar`. Clicking on the Save Button creates a `.jml` file with the same name as the Java source file provided as input and is saved in the same directory as the Java source file. If a different name or different location has to be provided for the `.jml` file, it can be done by clicking on the Export File button in the `MenuBar`. Figure 16 shows the template being saved as a `.jml` file when Save button is clicked and Figure 17 shows the template being saved as a `.jml` file when Export File button is clicked.

After selecting a specific location for the file to be saved, a new file is created at the location and the contents of the text component where annotations are generated are taken and written into the newly created file and then saved with the file name as given by the user.

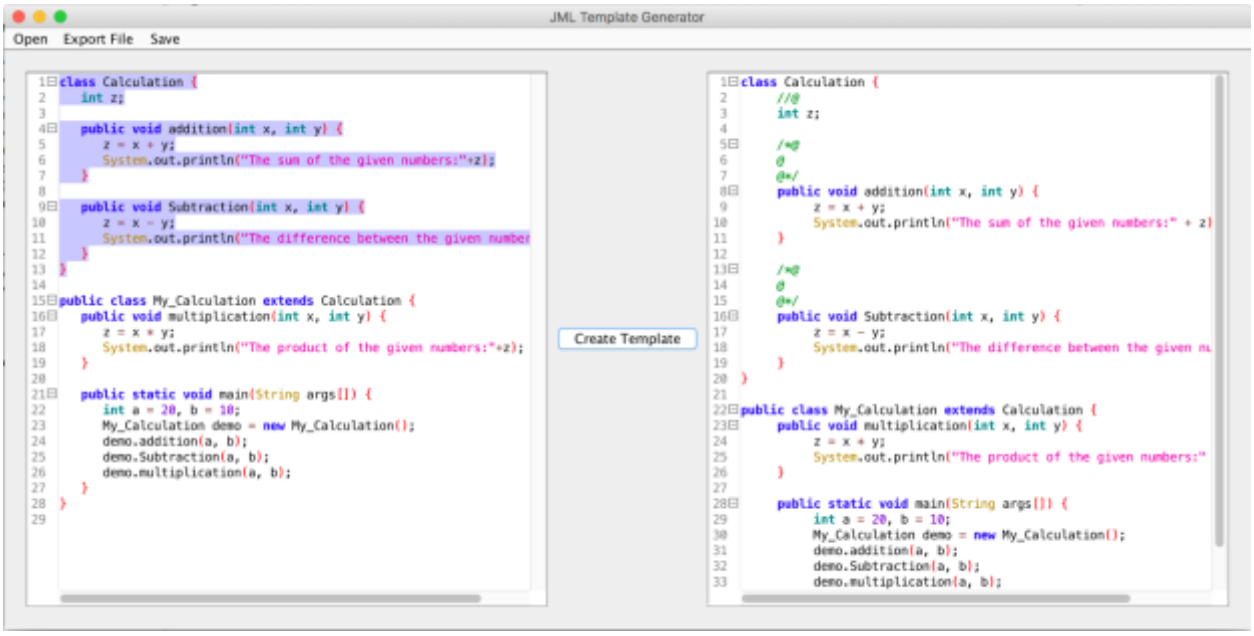

Figure 15 JML Template generated for selected portion of Java source code.

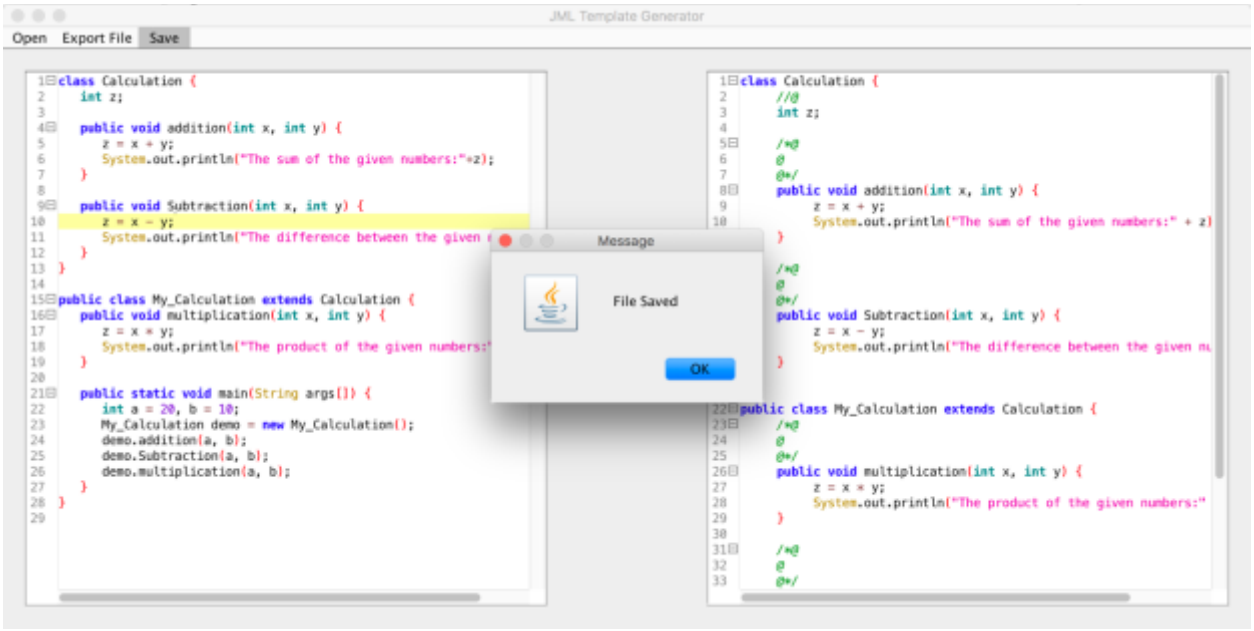

Figure 16 JML Template generated and saved using Save button

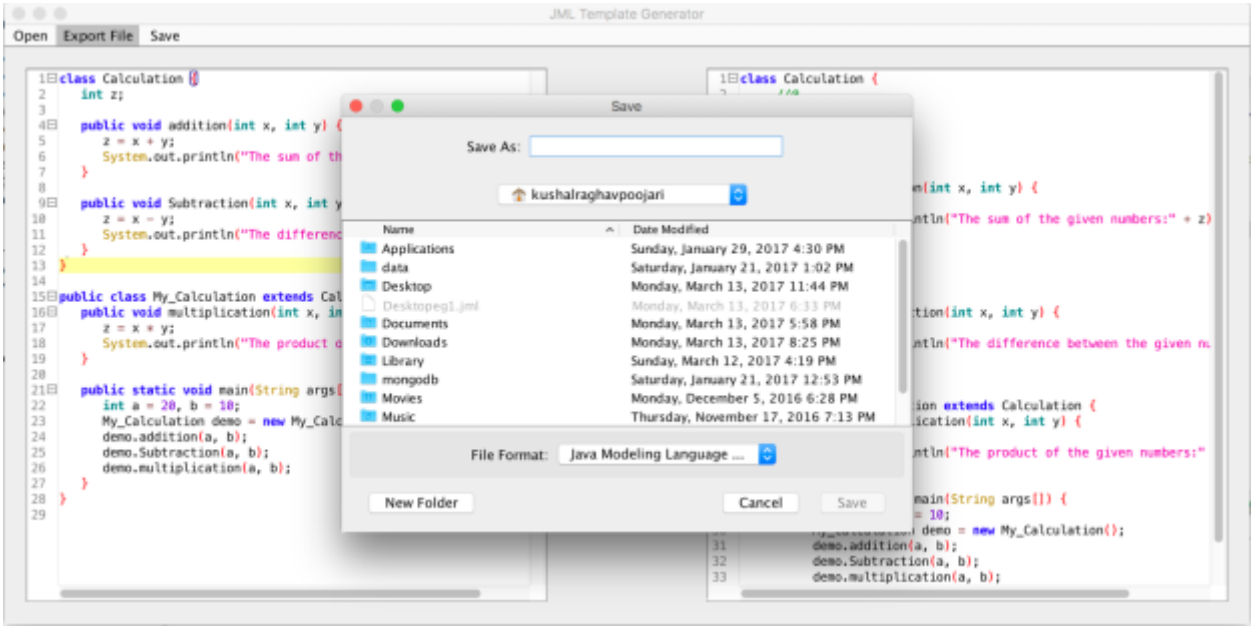

Figure 17 JML Template generated and saved using Export File button

Core

The Core of the tool's architecture consists of the classes and methods used to add annotations to the Java source code. This process runs in the background as soon as the user provides the input and clicks on the Create Template button.

Once the Create Template button is clicked, the input from the text component is provided to the `convert()` in `Maincore.java`. If a specific portion of the Java source code is selected and the Create Template button is clicked, then the entire input from the text component along with the starting line number from which the code is selected and the ending line number are sent to the `convert()` in `Maincore.java.` If the code is not selected, then the entire Java source code and `-1` and `-1`, which indicates the entire Java source code to be converted, are sent as the starting line and ending line to `convert()` in `Maincore.java.`

The Java source code is passed into `convert()` as a string. This string is passed into `ANTLRStringStream` which takes the string and copies it into a local array and the values from the array are taken and converted into a `CharStream`, a source of characters for an ANTLR Lexer. CharStream input is passed to the `JavaLexer`, further this Lexer is consumed by `TokenRewriteStream` as a `TokenSource` (a source to provide the sequence of tokens). `TokenRewriteStream` gives the flexibility to modify tokens. `TokenRewriteStream` provides a stream of tokens, accessing tokens from a `TokenSource`, this stream of tokens is consumed by the `JavaParser`.

```
CommonTree tree = (CommonTree) parser.javaSource().getTree();
```

This line of code generates the AST. For example, let us take a simple Java source code shown in Figure 18. The AST for this Java source code is shown in Figure 19.

```
public class HelloWorld {
     public int x;
     public void print() {
          System.out.println("Hello world");
     }
}
```

Figure 18 Simple Java source code example

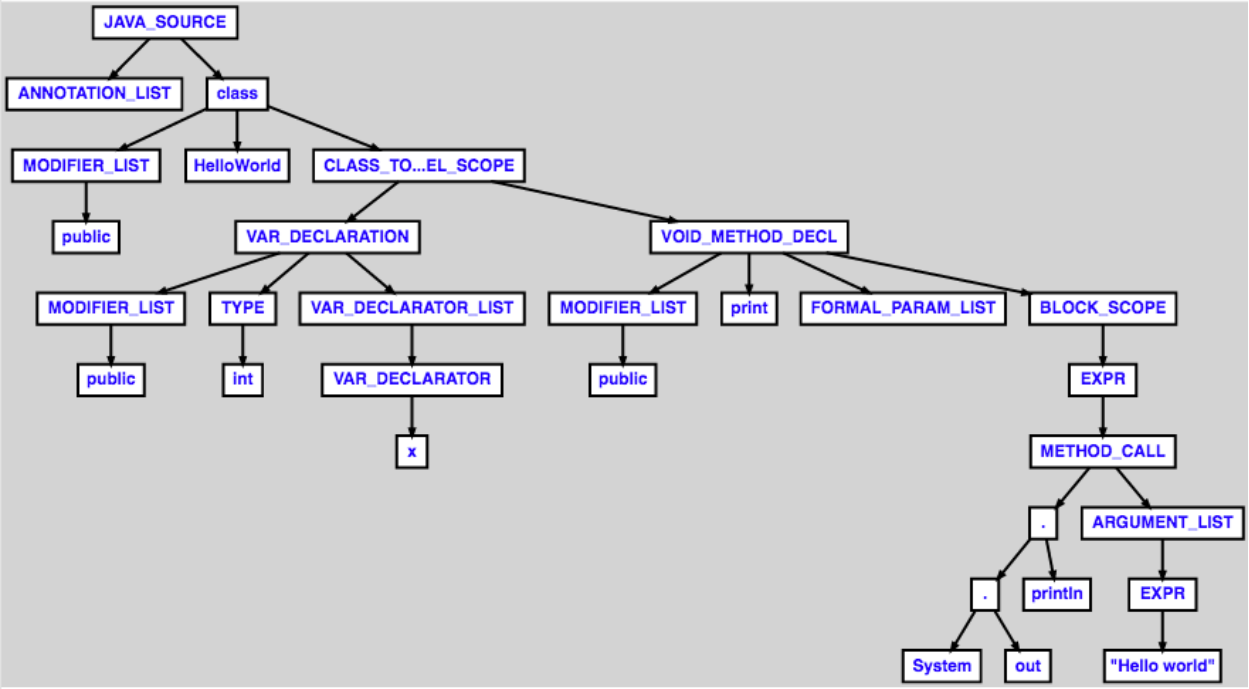

Figure 19 AST generated for Java source code in Figure 18.

Once the AST is generated it has to be passed to the `TreeParser` to validate it. `TreeParser` (`JavaTreeParser`) consumes a stream of tokens as input. The generated AST is passed into `CommonTreeNodeStream` and then converted into `TokenStream` (stream of tokens) using `setTokenStream()` of `CommonTreeNodeStream`., an error is thrown if there are any syntax errors. If there are no syntax errors then the tree (AST), tokens along with the starting line number and ending line number of the source code are passed to the `JavaTransformer` where the AST is modified by inserting annotations as tokens. Once the AST is modified the stream of tokens are passed to the JavaPrinter. These tokens are then converted to a string; indentation is provided and then printed to the UI.

Code

The Heart of the JML Template Generation tool is the `JavaTransformer` which modifies the AST by adding the annotation tokens before variable and method declarations. The AST is modified by passing the tree, tokens, starting line number and ending line number of the selected Java source code to `JavaTransformer` and then invoking the `modifyTree()`.

The method `isClassNode()` returns a boolean. If the current node is a class node, then it returns true.

The method `isMethodNode()` returns a boolean. If the current node is a method node, then it returns true.

The method `isInSelectedCode()` return a boolean. It checks whether the current node is among the selected lines of code.

The method `genBlockComment()` and `genLineComment()` are used to generate single line and multi-line annotations as tokens.

On invoking `modifyTree()`, that method initializes a `HashMap` where the key is expected to be the name of the class and the value is the set of methods in that particular class. To get the list of methods for each class, the tree and the HashMap are passed to `getAllClassMethods()`. This method traverses through the entire tree and checks for class nodes using `isClassNode()`. If the node is a class node then it's child nodes are traversed to find the list of methods, it checks for a method node using `isMethodNode()`. If it is a method node then it is added to the set which contains the list of methods for the specific class, if the node is a class node then `getAllClassMethods()` is called. `getAllClassMethods()` is called recursively until all the methods for each class is added into the set. Once all the methods for each class is added into

the HashMap `buildComment()` is called. This method traverses through the entire tree and checks if the current node is a method node or a constructor node using `isMethodNode()` and `isConstructorNode()`. If the node is a method or constructor, then it checks if the current node is among the selected lines of code using `isInSelectedCode()` or else it checks if the current node is a variable node using `isVariableNode()` and if it is true it checks whether the current node is among the selected lines of code. If the current node is found out to be a method node or a constructor node or a variable node, then the first token of that node is grabbed using `getFirstToken()` and annotations are generated using `genBlockComment()` and `genLineComment()` based on the node and these annotations are inserted as token before the first token. If the current node is found out to be a method or a constructor node and is among the selected lines of code, then the node is checked to see if the same method exists in the parent class of the node using `isOverriddenMethodNode()`. If the parent class method exists then a prefix **`also`** is added to the block comment token. These tokens can be modified easily as these tokens are an instance of `TokenRewriteStream`. The AST for Figure 18 after inserting annotations are shown in Figure 20. The modified stream of tokens is then passed to the Printer. In the Printer, these tokens are converted into strings and indentation is added to the string and printed on the UI.

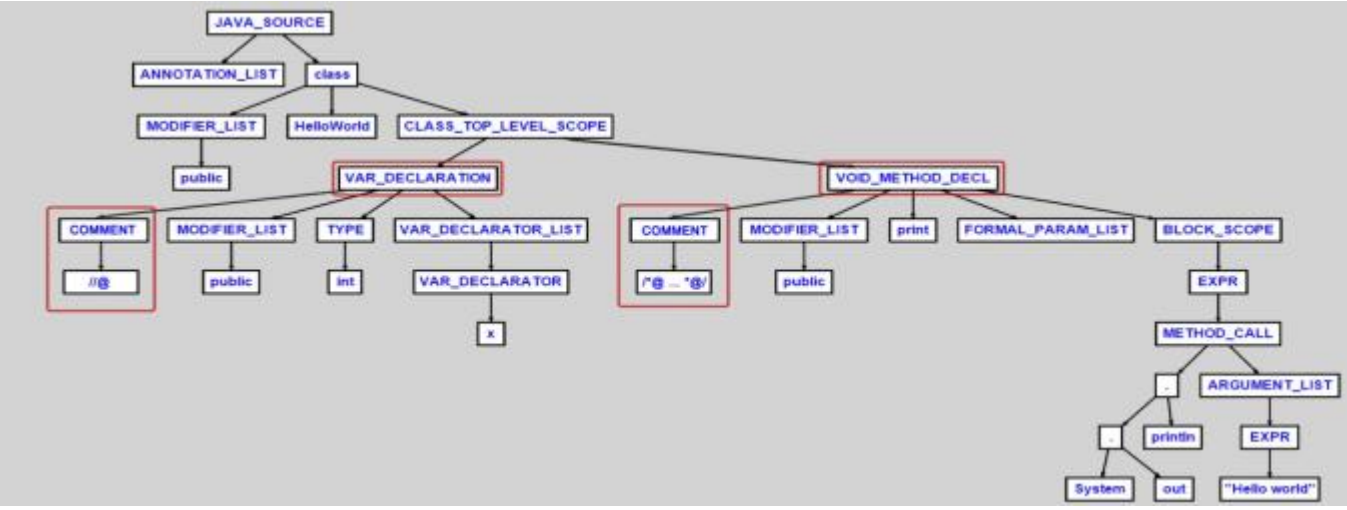

Figure 20 AST of Figure 18 after inserting annotations

Tests

JML Template Generation is tested by writing JUnit tests. Tests are written to check whether annotations are generated before variable and method declaration. A JUnit test for the Template generated in Figure 14 is shown in Figure 21.

```
1  package jml.template.test;
2
3+ import java.io.BufferedReader;
13
14 public class OverridingTest {
15
16     @Test
17     public void testFile() throws Exception {
18         String file1 = FileUtils.readFileToString(new File("test/Overriding.java"));
19         String finalsource = MainCore.convert(file1, -1, -1);
20         BufferedReader reader = new BufferedReader(new StringReader(finalsource));
21         String line;
22         ArrayList<String> lines = new ArrayList<String>();
23         while((line=reader.readLine())!=null){
24             lines.add(line.trim());
25         }
26         Assert.assertTrue(lines.get(2).startsWith("/*@"));
27         Assert.assertTrue(lines.get(3).startsWith("@"));
28         Assert.assertTrue(lines.get(4).startsWith("@*/"));
29         Assert.assertTrue(lines.get(16).startsWith("/*@"));
30         Assert.assertTrue(lines.get(17).startsWith("@"));
31         Assert.assertTrue(lines.get(18).startsWith("@*/"));
32         Assert.assertTrue(lines.get(26).startsWith("/*@ also"));
33         Assert.assertTrue(lines.get(27).startsWith("@"));
34         Assert.assertTrue(lines.get(28).startsWith("@*/"));
35
36     }
37
38 }
```

Figure 21 A JUnit test for the Template generated in Figure 14

Result for the JUnit test is shown in Figure 21 is shown in Figure 22.

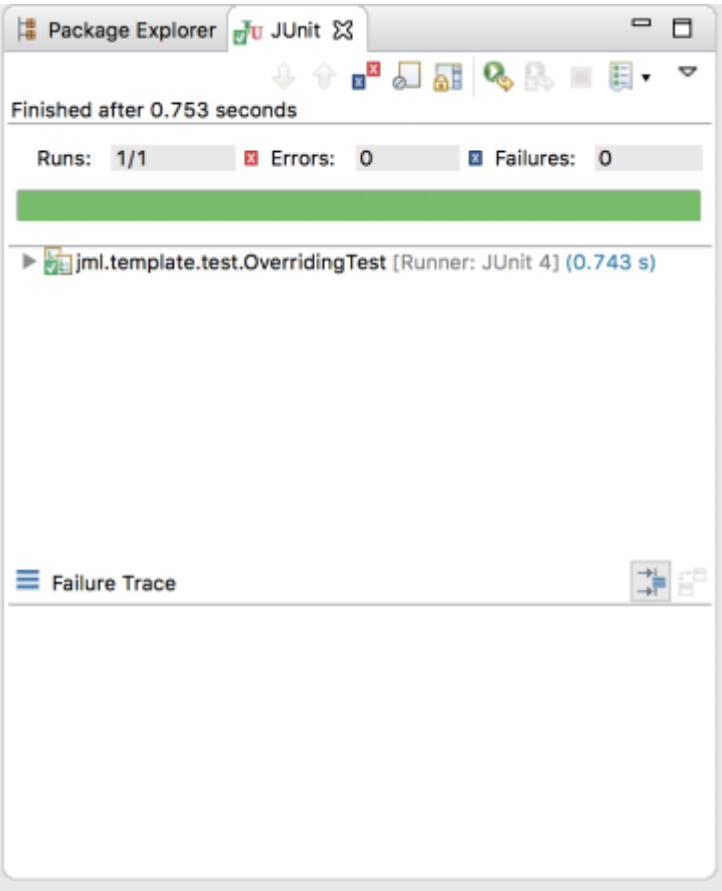

Figure 22 JUnit test results for Figure 21

CHAPTER FOUR: DISCUSSION

Organization of code

The JML Template Generation project has source files organized into `src`, `lib` and `test` folders. `src` contains three packages: `jml.template.core` where all the core code like the Lexer, the Parser, the Transformer and the Printer is placed, `jml.template.ui` contains the code for UI, `jml.template.test` contains test cases for the Java programs place in the `test` folder of the project. Lib folder contains a set of external jar files used for the project and `test` folder contains few examples of Java programs that will be used for the Junit tests.

Other issues

The JML Template Generation tool can be made to work with next versions of Java by generating Lexer and Parser with the updated grammar file (`Java.g`). By generating a Lexer and a Parser from the updated grammar file (`Java.g`) all the rules for the Lexer and the Parser are updated. The TreeParser can also be updated by using an updated grammar file for the TreeParser (`TreeParser.g`) and then generating TreeParser with the updated grammar file.

Future Work

The JML Template Generation tool creates a template for Java files. This tool accepts Java files as input and generates annotations for all the method declarations and variable declarations. This work can be extended by adding the feature of generating missing annotations for JML files. A JML file can be given as an input; the output is a JML file with missing annotations generated for methods and variables declared. Another feature that can be added in future would be,

generating specification skeleton files for an entire directory that contains Java files. The directory where the Java files are present are to be passed as input, and the JML Template Generation tool should generate specification skeleton files for the Java files present in the directory and saved with the file name same as the Java file name but with a `.jml` extension.

CHAPTER FIVE: CONCLUSION

The JML Template Generation tool automates the mundane task of going through several steps to add annotations to variables and methods. Annotations can be even generated for the selected part of Java source code. As it is a standalone tool, future works can be easier to implement and the cost of implementing the future work, cost of maintaining the JML Template Generation tool would be relatively less when compared to the other tools.

APPENDIX: SOURCE CODE

```java
package jml.template.core;

import org.antlr.runtime.ANTLRStringStream;
import org.antlr.runtime.TokenRewriteStream;
import org.antlr.runtime.tree.CommonTree;
import org.antlr.runtime.tree.CommonTreeNodeStream;

public class MainCore {

    public static String convert(String source, int selectedStartLine, int selectedEndLine) throws Exception {
        ANTLRStringStream input = new ANTLRStringStream(source);
        // Init Lexer for generating Tokens from source code
        JavaLexer lexer = new JavaLexer(input);
        TokenRewriteStream tokens = new TokenRewriteStream(lexer);

        // Init Parser and build AST tree
        JavaParser parser = new JavaParser(tokens);
        CommonTree tree = (CommonTree) parser.javaSource().getTree();

        CommonTreeNodeStream nodes = new CommonTreeNodeStream(tree);
        nodes.setTokenStream(tokens);

        // Init GrammarParser to validate grammar source code
        JavaTreeParser grammarParser = new JavaTreeParser(nodes);
        grammarParser.javaSource();
        // Init Transformer to modify AST Tree: add comment nodes to function nodes and variable nodes.
        JavaTransformer transformer = new JavaTransformer(tokens, tree, selectedStartLine, selectedEndLine);
        transformer.modifyTree();

        // Init Printer to print
        JavaPrinter printer = new JavaPrinter(tokens);
        return printer.getSource();

    }
}
```

Figure 23 Source code MainCore.java

```java
package jml.template.core;

import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

import org.antlr.runtime.CommonToken;
import org.antlr.runtime.Token;
import org.antlr.runtime.TokenRewriteStream;
import org.antlr.runtime.tree.CommonTree;
import org.apache.commons.lang3.StringUtils;

public class JavaTransformer {
    private TokenRewriteStream tokens;
    private CommonTree tree;
    private int selectedStartLine;
    private int selectedEndLine;

    public JavaTransformer(TokenRewriteStream tokens, CommonTree tree, int selectedStartLine, int selectedEndLine) {
        this.tokens = tokens;
        this.tree = tree;
        this.selectedStartLine = selectedStartLine;
        this.selectedEndLine = selectedEndLine;
    }

    private Token getFirstToken(CommonTree node) {
        if (node.getChildren() == null) {
            if (node.getToken().getTokenIndex() != -1) {
                return node.getToken();
            } else {
                return null;
            }
        } else {
            List<CommonTree> listChildNodes = (List<CommonTree>) node.getChildren();
            for (CommonTree childNode : listChildNodes) {
                Token token = getFirstToken(childNode);
                if (token != null) {
                    return token;
                }
            }
        }

        return null;
    }
```

Figure 24 Source code JavaTransformer.java Part a

```
48@   private boolean isClassNode(CommonTree node) {
49        return node.getText().equals("class");
50    }
51
52@   private boolean isMethodNode(CommonTree node) {
53        return node.getText().equals("VOID_METHOD_DECL") || node.getText().equals("FUNCTION_METHOD_DECL");
54    }
55
56@   private boolean isVariableNode(CommonTree node) {
57        return node.getParent().getText().equals("CLASS_TOP_LEVEL_SCOPE") && (node.getText().equals("VAR_DECLARATION"));
58    }
59
60@   private boolean isConstructorNode(CommonTree node) {
61        return node.getParent().getText().equals("CLASS_TOP_LEVEL_SCOPE") && node.getParent().getParent().getText().equals("class")
62                && (node.getText().equals("CONSTRUCTOR_DECL"));
63    }
64
65
66@   private boolean isInSelectedCode(CommonTree node) {
67        if (selectedStartLine >= 0 && selectedEndLine >= 0) {
68            return node.getLine() >= selectedStartLine && node.getLine() <= selectedEndLine;
69        }
70        return true;
71    }
72
73@   private CommonToken genBlockComment(String prefix, Token firstToken) {
74        final String comment = " /*@" + prefix + "\n" + "@ \n" + "@*/\n";
75@       CommonToken token = new CommonToken(JavaLexer.BLOCK_COMMENT) {
76@           @Override
77            public String toString() {
78                return comment;
79            }
80        };
81
82        return token;
83    }
84
```

Figure 25 Source code JavaTransformer.java Part b

```
85    private CommonToken genLineComment(Token firstToken) {
86        StringBuffer strTab = new StringBuffer();
87        for (int j = 0; j < firstToken.getCharPositionInLine(); j++) {
88            strTab.append(" ");
89        }
90        final String comment = strTab + "//@\n";
91        CommonToken tokenxx = new CommonToken(JavaLexer.LINE_COMMENT) {
92            @Override
93            public String toString() {
94                return comment;
95            }
96        };
97
98        return tokenxx;
99    }
100
101   private void buildComment(CommonTree t, int indent, Map<String, Set<String>> mapAllClassMethods) {
102       if (t != null) {
103           for (int i = 0; i < t.getChildCount(); i++) {
104               CommonTree node = (CommonTree) t.getChild(i);
105               if (isMethodNode(node) || isConstructorNode(node)) {
106                   Token firstToken = getFirstToken(node);
107                   if (isInSelectedCode(node)) {
108                       String prefix = "";
109                       if (isOverriddenMethodNode(node, mapAllClassMethods)) {
110                           prefix = " also";
111                       }
112                       CommonToken tokenComment = genBlockComment(prefix, firstToken);
113                       tokens.insertBefore(firstToken.getTokenIndex()-1, tokenComment);
114                   }
115               } else if (isVariableNode(node)) {
116                   Token firstToken = getFirstToken(node);
117                   if (isInSelectedCode(node)) {
118                       CommonToken tokenComment = genLineComment(firstToken);
119                       tokens.insertBefore(firstToken.getTokenIndex()-1, tokenComment);
120                   }
121               }
122
123               buildComment(node, indent + 1, mapAllClassMethods);
124           }
125       }
126   }
127
```

Figure 26 Source code JavaTransformer.java Part c

```
127
128  private boolean isOverriddenMethodNode(CommonTree node, Map<String, Set<String>> mapAllClassMethods) {
129      String extendClass = "";
130      CommonTree tempNode = (CommonTree) node.getParent();
131      while (!isClassNode(tempNode)) {
132          tempNode = (CommonTree) tempNode.getParent();
133      }
134
135      if (tempNode.getChild(2) != null && tempNode.getChild(2).getText().equals("EXTENDS_CLAUSE")) {
136          tempNode = (CommonTree) tempNode.getChild(2).getChild(0);
137          while (tempNode.getChildren() != null) {
138              tempNode = (CommonTree) tempNode.getChild(0);
139          }
140
141          extendClass = tempNode.getText();
142      }
143
144      if (StringUtils.isNoneEmpty(extendClass)) {
145          if (mapAllClassMethods.get(extendClass) != null && mapAllClassMethods.get(extendClass).contains(node.getChild(1).getText()))
146              return true;
147          }
148      }
149
150      return false;
151  }
152
```

Figure 27 Source code JavaTransformer.java Part d

```
154    private void getAllClassMethods(CommonTree t, Map<String, Set<String>> mapAllClassMethods) {
155        if (t != null) {
156            for (int i = 0; i < t.getChildCount(); i++) {
157                CommonTree node = (CommonTree) t.getChild(i);
158                if (isClassNode(node)) {
159                    String className = node.getChild(1).getText();
160                    CommonTree classTopLevelScope = (CommonTree) node.getChild(node.getChildCount() - 1);
161                    Set<String> methodNames = new HashSet<String>();
162                    for (int j = 0; j < classTopLevelScope.getChildCount(); j++) {
163                        CommonTree methodNode = (CommonTree) classTopLevelScope.getChild(j);
164                        if (isMethodNode(methodNode)) {
165                            methodNames.add(methodNode.getChild(1).getText());
166                        }
167                    }
168                    mapAllClassMethods.put(className, methodNames);
169                } else {
170                    getAllClassMethods(node, mapAllClassMethods);
171                }
172            }
173        }
174    }
175
176    public void modifyTree() {
177        Map<String, Set<String>> mapAllClassMethods = new HashMap<String, Set<String>>();
178        getAllClassMethods(tree, mapAllClassMethods);
179        buildComment(tree, 0, mapAllClassMethods);
180    }
181 }
```

Figure 28 Source code JavaTransformer.java Part e

```java
package jml.template.core;

import java.util.Map;

import org.antlr.runtime.TokenRewriteStream;
import org.eclipse.jdt.core.JavaCore;
import org.eclipse.jdt.core.ToolFactory;
import org.eclipse.jdt.core.formatter.CodeFormatter;
import org.eclipse.jdt.core.formatter.DefaultCodeFormatterConstants;
import org.eclipse.jface.text.Document;
import org.eclipse.jface.text.IDocument;
import org.eclipse.text.edits.TextEdit;

public class JavaPrinter {
    private TokenRewriteStream tokens;
    private CodeFormatter codeFormatter;

    public JavaPrinter(TokenRewriteStream tokens) {
        this.tokens = tokens;

        Map options = DefaultCodeFormatterConstants.getEclipseDefaultSettings();
        options.put(JavaCore.COMPILER_COMPLIANCE, JavaCore.VERSION_1_8);
        options.put(JavaCore.COMPILER_CODEGEN_TARGET_PLATFORM, JavaCore.VERSION_1_8);
        options.put(JavaCore.COMPILER_SOURCE, JavaCore.VERSION_1_8);
        options.put(DefaultCodeFormatterConstants.FORMATTER_ALIGNMENT_FOR_ENUM_CONSTANTS, DefaultCodeFormatterConstants
                .createAlignmentValue(true, DefaultCodeFormatterConstants.WRAP_ONE_PER_LINE,
                        DefaultCodeFormatterConstants.INDENT_ON_COLUMN));
        codeFormatter = ToolFactory.createCodeFormatter(options);
    }

    public String getSource() throws Exception {
        String source = tokens.toString();
        TextEdit edit = codeFormatter.format(CodeFormatter.K_COMPILATION_UNIT, source, // source to format
                0, source.length(), 0, System.getProperty("line.separator"));
        IDocument document = new Document(source);
        edit.apply(document);
        return document.get();
    }
}
```

Figure 29 Source code JavaPrinter.java

```java
package jml.template.ui;


import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.io.File;
import java.io.IOException;
import java.nio.file.Path;
import java.nio.file.Paths;

import javax.swing.JButton;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JMenuBar;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.border.EmptyBorder;

import org.apache.commons.io.FileUtils;
import org.apache.commons.io.FilenameUtils;
import org.apache.commons.lang3.StringUtils;
import org.fife.ui.rsyntaxtextarea.RSyntaxTextArea;
import org.fife.ui.rsyntaxtextarea.SyntaxConstants;
import org.fife.ui.rtextarea.RTextScrollPane;

import jml.template.core.MainCore;
import net.miginfocom.swing.MigLayout;

public class MainDisplay extends JFrame {

    private JPanel contentPane;
    private RSyntaxTextArea txtAreaSource;
    private RSyntaxTextArea txtAreaTarget;
    private JButton btnConvert;
    private String path;

    /**
     * Launch the application.
     */
    public static void main(String[] args) {
        EventQueue.invokeLater(new Runnable() {
            public void run() {
                try {
                    MainDisplay frame = new MainDisplay();
                    frame.setVisible(true);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });
    }

```

Figure 30 Source code MainDisplay.java Part a

```
54    private int getLineNumber(String text, int position) {
55        int count = 0;
56        for (int i = 0; i <= position; i++) {
57            if (text.charAt(i) == '\n') {
58                count++;
59            }
60        }
61        return count;
62    }
63
64    /**
65     * Create the frame.
66     */
67    public MainDisplay() {
68        setTitle("JML Template Generator");
69        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
70        setBounds(100, 100, 1200, 600);
71        contentPane = new JPanel();
72        contentPane.setBorder(new EmptyBorder(5, 5, 5, 5));
73        contentPane.setLayout(new MigLayout("", "[grow][][grow]", "[grow]"));
74        setContentPane(contentPane);
75
76        txtAreaSource = new RSyntaxTextArea(20, 60);
77        txtAreaSource.setSyntaxEditingStyle(SyntaxConstants.SYNTAX_STYLE_JAVA);
78        txtAreaSource.setCodeFoldingEnabled(true);
79        contentPane.add(new RTextScrollPane(txtAreaSource), "cell 0 0, grow");
80
81        btnConvert = new JButton("Create Template");
82        btnConvert.addActionListener(new ActionListener() {
83            @Override
84            public void actionPerformed(ActionEvent e) {
85                try {
86                    if (StringUtils.isNotEmpty(txtAreaSource.getText())) {
87                        int selectedStartLine = -1;
88                        int selectedEndLine = -1;
89                        if (StringUtils.isNotEmpty(txtAreaSource.getSelectedText())) {
90                            selectedStartLine = getLineNumber(txtAreaSource.getText(),
91                                    txtAreaSource.getSelectionStart());
92                            selectedEndLine = getLineNumber(txtAreaSource.getText(), txtAreaSource.getSelectionEnd());
93                        }
94
95                        String source = MainCore.convert(txtAreaSource.getText(), selectedStartLine, selectedEndLine);
96                        txtAreaTarget.setText(source);
97                    }
98                } catch (Exception ex) {
99                    JOptionPane.showMessageDialog(MainDisplay.this, "Cannot parse source code", "Error",
100                            JOptionPane.ERROR_MESSAGE);
101                }
102            }
103        });
104
105        contentPane.add(btnConvert, "cell 1 0");
106
107        txtAreaTarget = new RSyntaxTextArea(20, 60);
108        txtAreaTarget.setSyntaxEditingStyle(SyntaxConstants.SYNTAX_STYLE_JAVA);
109        txtAreaTarget.setCodeFoldingEnabled(true);
110        contentPane.add(new RTextScrollPane(txtAreaTarget), "cell 2 0, grow");
111
112        setLocationRelativeTo(null);
113
```

Figure 31 Source code MainDisplay.java Part b

```
114         JButtonMenu menuOpen = new JButtonMenu("Open", 50);
115         menuOpen.addActionListener(new ActionListener() {
116             @Override
117             public void actionPerformed(ActionEvent e) {
118                 JOpenFileDialog openDialog = new JOpenFileDialog();
119                 int returnVal = openDialog.show(MainDisplay.this);
120                 if (returnVal == JFileChooser.APPROVE_OPTION) {
121                     String source = openDialog.readFile();
122                     path = openDialog.getPath();
123                     txtAreaSource.setText(source);
124                 }
125             }
126         });
127 
128         JButtonMenu menuExportFile = new JButtonMenu("Export File", 80);
129         menuExportFile.addActionListener(new ActionListener() {
130             @Override
131             public void actionPerformed(ActionEvent e) {
132                 if (StringUtils.isNotEmpty(txtAreaTarget.getText())) {
133                     JSaveFileDialog saveDialog = new JSaveFileDialog();
134                     int returnVal = saveDialog.show(MainDisplay.this);
135                     if (returnVal == JFileChooser.APPROVE_OPTION) {
136                         saveDialog.saveFile(txtAreaTarget.getText());
137                     }
138                 } else {
139                     JOptionPane.showMessageDialog(MainDisplay.this, "Source Target is empty");
140                 }
141             }
142         });
143 
144         JButtonMenu save = new JButtonMenu("Save", 50);
145         save.addActionListener(new ActionListener() {
146             @Override
147             public void actionPerformed(ActionEvent e) {
148                 if (StringUtils.isNotEmpty(txtAreaTarget.getText())) {
149                     Path pathTemp1 = Paths.get(path);
150                     String pathTemp = pathTemp1.getParent().toString();
151                     String filename = FilenameUtils.getBaseName(path);
152                     File file = new File(pathTemp + "/" + filename + ".jml");
153                     try {
154                         if(!file.exists()) {
155                             FileUtils.writeStringToFile(file,txtAreaTarget.getText() );
156                             JOptionPane.showMessageDialog(MainDisplay.this, "File Saved");
157                         } else {
158                             int result = JOptionPane.showConfirmDialog(MainDisplay.this, "File Already Exists,"
159                                     + " Do you want to override it?"
160                                     ,"File Already Exists", JOptionPane.YES_NO_CANCEL_OPTION);
161                             switch(result) {
162                             case JOptionPane.YES_OPTION:
163                                 FileUtils.writeStringToFile(file,txtAreaTarget.getText() );
164                                 JOptionPane.showMessageDialog(MainDisplay.this, "File Saved");
165                             case JOptionPane.NO_OPTION:
166                                 return;
167                             case JOptionPane.CLOSED_OPTION:
168                                 return;
169                             case JOptionPane.CANCEL_OPTION:
170                                 return;
171                             }
172                         }
```

Figure 32 Source code MainDisplay.java Part c

```
                    } catch (IOException e1) {
                        JOptionPane.showMessageDialog(MainDisplay.this, "Problem Saving file");
                    }
                } else {
                    JOptionPane.showMessageDialog(MainDisplay.this, "Source Target is empty");
                }
            }
        });

        JMenuBar menuBar = new JMenuBar();
        menuBar.add(menuOpen);
        menuBar.add(menuExportFile);
        menuBar.add(save);

        setJMenuBar(menuBar);
    }
}
```

Figure 33 Source code MainDisplay.java part d

```
package jml.template.ui;


import java.io.File;

public class JOpenFileDialog extends JFileChooser {
    private JFrame parent;
    private String path;

    public JOpenFileDialog() {
        setFileFilter(new FileFilter() {
            @Override
            public String getDescription() {
                return "Java Files (*.java)";
            }

            @Override
            public boolean accept(File file) {
                return file.isDirectory() || file.getAbsolutePath().endsWith(".java");
            }
        });
    }

    public int show(JFrame parent) {
        this.parent = parent;
        return showOpenDialog(parent);
    }

    public String readFile() {
        File file = getSelectedFile();
        try {
            path = file.getAbsolutePath();
            return IOUtils.toString(new FileReader(path));
        } catch (IOException ex) {
            JOptionPane.showMessageDialog(parent, "problem accessing file " + file.getAbsolutePath());
            return "";
        }
    }

    public String getPath() {
        try {
            return path;
        }catch (Exception ex) {
            JOptionPane.showMessageDialog(parent, "problem accessing file path ");
            return "";
        }
    }
}
```

Figure 34 Source code JOpenFileDialog.java

```java
package jml.template.ui;


import java.io.File;

import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.filechooser.FileFilter;

import org.apache.commons.io.FileUtils;

public class JSaveFileDialog extends JFileChooser {
    private JFrame parent;
    private static final String FILE_EXTENSION = ".jml";

    public JSaveFileDialog() {
        setFileFilter(new FileFilter() {
            @Override
            public String getDescription() {
                return "Java Modeling Language (*" + FILE_EXTENSION + ")";
            }

            @Override
            public boolean accept(File file) {
                return file.isDirectory() || file.getAbsolutePath().endsWith(FILE_EXTENSION);
            }
        });
    }

    public int show(JFrame parent) {
        this.parent = parent;
        return showSaveDialog(parent);
    }

    public void saveFile(String text) {
        File file = getSelectedFile();
        try {
            String filename = file.toString();
            if (!filename.endsWith(FILE_EXTENSION)) {
                filename += FILE_EXTENSION;
            }
            FileUtils.writeStringToFile(new File(filename), text);
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(parent, "problem save file " + file.getAbsolutePath());
        }
    }
}

```

Figure 35 Source code JSaveFileDialog.java

```java
package jml.template.ui;

import java.awt.Color;
import java.awt.Dimension;

import javax.swing.ButtonModel;
import javax.swing.JButton;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;

public class JButtonMenu extends JButton {
    private static final int DEFAULT_HEIGHT = 20;

    public JButtonMenu(String title, int width) {
        setText(title);

        setOpaque(false); // remove button filling and border
        setContentAreaFilled(false);
        setBorder(null);

        setRolloverEnabled(true); // Allows the button to detect when mouse is over it
        getModel().addChangeListener(new ChangeListener() {
            @Override
            public void stateChanged(ChangeEvent e) {
                ButtonModel model = (ButtonModel) e.getSource();

                if (model.isRollover()) {
                    JButtonMenu.this.setBackground(Color.LIGHT_GRAY); // Changes the color of the button
                    JButtonMenu.this.setOpaque(true);
                }

                else {
                    JButtonMenu.this.setBackground(null);
                    JButtonMenu.this.setOpaque(false);
                }
            }
        });

        Dimension dBt = new Dimension(width, DEFAULT_HEIGHT); // Sets the size of the button in the JMenuBar
            setMinimumSize(dBt);
            setPreferredSize(dBt);
            setMaximumSize(dBt);
        }
    }
```

Figure 36 Source code JButtonMenu.java

REFERENCES

[1] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, Department of Computer Science, February 2000. See www.cs.iastate.edu/~leavens/J ML.html.

[2] Gary T. Leavens, Yoonsik Cheon. Design by Contract with JML.

[3] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, Patrice Chalin, and Daniel M. Zimmerman. JML Reference Manual (DRAFT), May, 2013.

[4] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. In *Formal Methods for Components and Objects (FMCO) 2005, Revised Lectures*, pages 342-363. Volume 4111 of Lecture Notes in Computer Science, Springer Verlag, 2006.

[5] JMLSPEC tool : http://www.eecs.ucf.edu/~leavens/JML-release/docs/man/jmlspec.html

[6] MultiJava Project : http://multijava.sourceforge.net/

[7] ANother Tool for Language Recognition Reference Manual: http://www.antlr3.org/share/1084743321127/ANTLR_Reference_Manual.pdf

[8] Using ANTLR: https://theantlrguy.atlassian.net/wiki/display/ANTLR3/Using+ANTLR

[9] Java Foundation Classes: https://docs.oracle.com/javase/tutorial/uiswing/

[10] Swing (Java): https://en.wikipedia.org/wiki/Swing

[11] Eclipse Help: Formatting Java code.

[12] Graph visualization software: https://mdaines.github.io/viz.js/