UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

MURILO WOLFART

Analysis and validation of test case redundancy in testing suites of Java libraries

Work presented in partial fulfillment of the requirements for the degree of Bachelor in Computer Science

Advisor: Prof. Dra. Érika Fernandes Cota

Porto Alegre
May 2021

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos André Bulhões
Vice-Reitora: Prof^{a}. Patricia Pranke
Pró-Reitora de Graduação: Prof^{a}. Cíntia Inês Boll
Diretora do Instituto de Informática: Prof^{a}. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Ciência de Computação: Prof. Rodrigo Machado
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

ABSTRACT

When studying the test suites from some Java libraries, it is possible to observe a certain redundancy in the execution paths that some of the test cases take, whose reason is not known. In order to understand this issue, we establish two hypotheses: one that this redundancy happens due to the suite being an automatically generated suite, and one that it comes from test cases that test inputs of different natures, such as empty strings or null strings. To validate these hypotheses, we perform a detailed study by analysing the behavior of the test cases and the associated application code. We consider a code coverage criterion, computed by generating the test path of each test case in the suite and comparing these with the prime path coverage of the tested methods. For this procedure we first adapted a coverage analysis tool that considers the prime path coverage criterion, and afterwards we collected additional data about the application code and the test suite in order to understand the rationale behind the tests. Finally, a manual analysis is made in the source code of the methods being tested and their respective test cases. Results show that the projects' suites are closer to a developer-implemented suite, and not a generated one. It also shows that, when studying the tests with redundant test paths, most consist of cases that test different types of input - our second hypothesis. There are some cases that do not have a clear purpose and test very similar inputs, with some being even duplicated, but this occurs in a considerably insignificant scale.

Keywords: Test suites. Java Libraries. Prime Path Coverage. Test case redundancy.

Análise e validação de redundância em casos de teste em suítes de bibliotecas Java

RESUMO

Ao estudar suítes de testes de algumas bibliotecas Java, é possível perceber uma certa redundância no caminho que alguns dos casos de teste tomam, cuja razão é desconhecida. A fim de entender este problema, estabelecemos duas hipóteses: uma de que esta redundância acontece devido à suíte ser automaticamente gerada utilizando uma ferramenta de geração automática de testes, e uma de que ela se origina em casos de teste que testam parâmetros de entrada de naturezas diferentes, como palavra vazia e palavra nula. Para validar estas hipóteses, fazemos um estudo detalhado, analisando o comportamento dos casos de teste associado ao código da aplicação. Levamos em consideração um critério de cobertura de código, computado gerando o caminho de execução de cada caso de teste presente na suíte e comparando estes com a cobertura de caminhos primos do método. Para este procedimento primeiro adaptamos uma ferramenta de análise de cobertura que considera o critério de caminhos primos, e posteriormente coletamos dados adicionais acerca do código da aplicação e da suíte para entender os objetivos dos testes. Após isto, uma análise manual é feita no código dos métodos sendo testado e seus respectivos casos de teste. Resultados mostram que as suítes dos projetos são mais próximas de uma suíte implementada por desenvolvedores, e não de uma suíte gerada automaticamente. Também é mostrado que, ao analisar os testes com caminhos de execução redundantes, a maioria consiste de casos que testam diferentes tipos de parâmetros de entrada - nossa segunda hipótese. Existem alguns testes que não possuem propósito e alguns que testam valores de entrada semelhantes, sendo alguns inclusive duplicados, mas isto ocorre em uma escala consideravelmente insignificante.

Palavras-chave: Bibliotecas Java. Suítes de teste. Cobertura de caminhos primos. Redundância de testes.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

SUT	Software under test
MUT	Method under test
CFG	Control flow graph
TP	Test path
PP	Prime path
PPC	Prime path coverage
TR	Test requirement
CC	Cyclomatic complexity

CONTENTS

1 INTRODUCTION

One of the most important parts of software development is software testing, which is responsible for detecting faults in the system. Part of this process includes the elaboration of a test suite - a set of functions that execute the program's logic with specific properties and verify its results. There are multiple ways to implement a suite - it can be built based on testing requirements, obtained through a detailed analysis of the logic; generated through an automated test generation tool; or, in most cases, manually written by the software's development team, based on the nature of the program's functionalities. Moreover, a test suite contains different properties such as code coverage, number of test cases, number of redundant tests and, the most important, effectiveness. In this work, we define test suite effectiveness as a synonym for test power, defined by (ZHANG et al., 2019) as the power of a test suite in detecting faults.

Estimating the suite effectiveness becomes important in the process of fault detection, and the most ideal measure for it is the number of faults detected. However, counting the number of faults detected can be difficult and experiments that do it usually consider a small set of real faults, preventing rigorous statistical analysis (GLIGORIC et al., 2013). The second best measure is mutation testing, which consists of checking the ability of the test suite to detect small changes to the source code. This is yet another process that is considerably difficult to apply due to its computational cost (GLIGORIC et al., 2013). Finally, the third best measure we have is code coverage, which detects the amount of source code that the suite covers. Code coverage is usually the preferred way of estimating a suite's reach, and is also the focus of this work. There are multiple levels of code coverage, such as node, edge, edge-pair and prime paths. We will focus mainly on the latter one because it tests longer paths, providing better loop testing and ensuring in a more reliable way that the system does not fail.

This work is part of a broader project produced by Keslley Lima and Érika F. Cota, which aims to help developers in creating and maintaining an effective test suite using a machine learning technique, with models derived from real test suites in Java libraries. We focus initially on facilitating the analysis of the execution paths that the test suite covers, in relation to all possible execution paths of the algorithm. Part of this process included mapping the execution flow that the test suites from Java projects take, and during this phase a notable redundancy was identified, i.e. there was a subset of test cases that had the program execute the same set of instructions, in the same order. This raised the following question: why do these tests with duplicate flows exist and what is their source? Furthermore, does this redundancy affect the previously mentioned generation of models, needed to apply the machine learning technique and create effective test suites?

To answer this question we derive two hypotheses: one that the suite was automatically generated, and thus containing redundant tests; and also one that the test cases with equal execution flow cover different types of parameters (for instance, empty string and null string). In order to validate these hypothesis, we compare the properties of the test suites in the aforementioned projects with the properties of two other suites: a suite manually created based only on the methods' signatures and descriptions (when available), and a suite generated using the automatic test generation tool EvoSuite.

To compare the test suites we analyzed their coverage for different criteria and also consider some characteristics of the associated application code. Besides the basic criteria of code and branch coverage, we used the prime path criterion as well. For this step we need a tool that generates test requirements for the methods under test at a prime path level, and since we do not currently have a tool for this we must create or look for one. We go for the second option, as the operation of computing prime paths is not quite simple, and there are a couple of solutions on-line which we can reuse. Some adaptations are needed, which is also part of this work, so that the tool supports a great variation of Java source code and can be run for large libraries instead of small pieces of code. Once the tool is ready we use it to generate the ideal prime path coverage of the method under test and compare it with the test execution paths in order to compute the coverage of the test suite.

Therefore, the contributions made by this work are as follows: a tool that generates prime path coverage criteria for a Java source code, adapted and made to work with a greater code base, available at its BitBucket repository (WOLFART; LEE; PLATT, 2020); and an analysis on the redundancy present in suites from Java libraries, providing a conclusion on where they come from and a partial conclusion on their impact.

This work is organized as follows: Chapter 2 contains basic definitions and concepts which help the reader understand the work described here, as well as information regarding related work. Chapter 3 summarizes the main tools found in order to compute the PPC criteria of the projects being studied. Chapter 4 goes through the modifications made to one of the tools that were found to be the best for generating PPC criteria. Lastly, Chapter 5 describes the research done under the redundancy in the test sets. Chapter 6 discusses the conclusions and possible future work.

2 TECHNICAL BACKGROUND AND RELATED WORK

There are some concepts that must be understood before getting into the details of this work. We may recall that a test suite (or a test set) is a set of test cases that evaluate the software's functions (AMMANN; OFFUTT, 2016).

When talking about software testing, it must be mentioned that we have four levels of testing: unit testing, integration testing, system testing and acceptance testing. The main goal of this work is the first one, unit testing, which is usually the main focus of a project's test suite. For this work we assume that unit testing means to test each method separately, evaluating its output given a specific input.

Besides test levels, we should also speak briefly about testing types. Some traditional terms are black box testing and white box testing. Black box testing consists of deriving test cases from the software's specifications, requirements and external descriptions in general. On the other hand, white box testing consists of deriving tests from the software implementation. Later, the term grey box testing appeared, combining elements from both types and, soon, making this distinction quite obsolete. Ammann and Offutt (2016) argue that the best way of deriving tests is to use mathematical abstractions and, for unit testing, graphs are the most common abstraction used. Some types of graphs include byte code graph and control flow graph. We will be using mainly the latter one since it deals with the original, high-level source code.

2.1 Control flow graph

A control flow graph (CFG) is a graphical representation of the computation of the method under test (MUT), where each node represents a set of instructions that are executed in the same flow, and each edge represents a possible flow the execution can take. The entry node represents the start of the execution flow, while the exit nodes represent the end of the flow (blocks where the program returns).

We may mathematically define a CFG as a graph $G = (V, E)$, where $N_0 \in V$ represents the starting node and $V_\mathrm{f} \subset V$ represents the set of i exit nodes ${N_\mathrm{f}}^0, ..., {N_\mathrm{f}}^\mathrm{i}$.

After defining the CFG, we may now also define some notions of paths in the graph.

2.2 Test path

Normally we have one CFG for each MUT, and when executing a test case for that method, the program will produce a path p in the CFG, starting at node N_0 and ending at an exit node ${N_{\mathrm{f}}}^{\mathrm{x}}$, which represents the execution flow of the application. The path p is called a Test Path (TP). It is important to note that the TP may be of length zero, in case the set of nodes V from the CFG is empty.

2.3 Prime path

Before stating the definition of a prime path, we must define a simple path. A simple path is a path p_{s} in the CFG where no nodes appear more than once, with exception that the first and the last nodes may be identical. This means that no simple paths can contain internal loops.

A prime path (PP) is a path p_{p} in the CFG that satisfies two rules: 1. It is a simple path and 2. It is not a sub-path of any other simple path in the graph.

2.4 Test Requirement

A Test Requirement (TR) is a specific flow in the program execution that a test case must cover. In the case of a CFG, it is a path p_{r} that must be taken. As Ammann and Offutt (2016) exemplify, from an `if-else` structure two TRs would be derived, one for the *false* branch and another for the *true* branch.

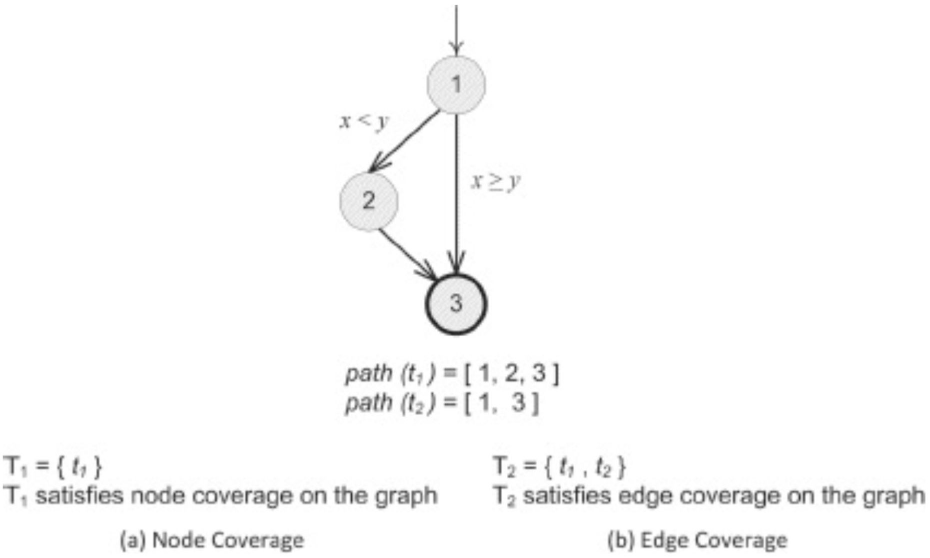
2.5 Test coverage criteria

A test coverage criteria is a set of TRs that a test suite must satisfy. There are many types of coverage criteria, but we will talk briefly about the main ones.

Node coverage (sometimes shown as "statement coverage"), as stated by Ammann and Offutt (2016), requires that the test set visits each node in the CFG of the MUT at least once. In other words, it requires all instructions in the source code to have been called at least once. This is the simplest type of graph-based coverage.

Edge coverage (sometimes shown as "branch coverage"), requires that the test set

Figure 2.1 – Difference between node coverage and edge coverage.



Source: Ammann and Offutt (2016)

visits each edge in the CFG at least once. It should be noted that a test set that satisfies node coverage may not satisfy edge coverage. Figure 2.1 shows the difference between node and edge coverage. Edge-pair coverage works similarly to edge coverage, but requires all paths of length up to 2 (that is, all paths that visit 3 nodes or less) to be covered by the test set.

Finally, we have prime path coverage (PPC), which requires the test set to cover all prime paths in the CFG at least once. This coverage is particularly interesting because, as mentioned in Chapter 1, it provides a better assurance that the system will not fail, mainly because it tests longer paths and loops.

2.6 Input value equivalence partition

In this work we will also talk about boundary partition values, or inputs of equivalent partitions. Two different input values come from different equivalence partitions if they are, for the program's logic, semantically different. For instance, in a method that returns `true` for numbers greater than 20 and `false` for other numbers, values `30` and `1` are not partition-equivalent, while values `21` and `22` are partition-equivalent. More information can be found in the book *The Art of Software Testing* (2011).

Having introduced the basic concepts we can now begin describing the work. Before getting into the details, we will briefly present related approaches that discuss the effectiveness of a test suite.

2.7 Related work

Namin and Andrews (2009) analyze the correlation among test suite effectiveness, size and coverage. More specifically, the idea of that work is to analyze whether achieving high coverage will lead to more effectiveness when preventing software faults, or if it will only make the suite larger. Results show that using both size and coverage can yield a more accurate prediction of effectiveness than when using size alone or coverage alone. The work also shows that a nonlinear correlation exists among the three variables.

Inozemtseva and Holmes (2014) study the correlation between code coverage and test suite effectiveness. It analyzes some common types of coverage, such as edge and node coverage, and also some more specific types of coverage, such as dynamic basic block coverage and predicate complete coverage. However, basic coverage types are more deeply analyzed. The study shows that these coverage types are not suitable for measuring the effectiveness of fault prevention in a test suite, but they work for identifying over-tested and under-tested parts of the software under test (SUT) - i.e. parts of the software that the suite covers with multiple test cases and parts that it covers with few test cases. The study does not cover prime path coverage, probably due to the computational cost of dealing with it at the time.

Gopinath, Jensen and Groce (2014) check how suitable are statement and branch coverage when measuring a suite's effectiveness. A test suite's effectiveness is often measured by its ability to kill mutants - modifications inserted into the program's behavior. The study shows that, between these two types of coverage, statement works best when predicting mutants. This study also does not take prime path coverage into consideration.

None of the previously listed works deal with prime path coverage, and that is the main difference between them and this work. Moreover, none of them study the occurrence of redundancy in test suite's.

3 ANALYSIS OF CFG GENERATION TOOLS

As it was previously mentioned, we want to understand the possible redundancy that was identified in the test suite of some Java projects. We need to investigate whether this redundancy comes from a generated test suite or not and, if it does not, we must also study the purpose of the test cases. For that, we must obtain the coverage properties of the test suite and its reach, and that requires three steps: one to calculate the PPC of the methods being tested; one to map the TPs of the test suite; and one to compute the coverage percent itself, using the two previously obtained data.

To perform the first step, we need to read a source code and compute its CFG in a non-visual way, so that we can manipulate the information (generating a PNG file with the visual representation of the graph is not enough). After the CFG has been generated, we can apply the PPC generation algorithm to obtain the requirements. In short, we first generate the CFG from a source code, and later use it to output the PPCs.

To perform the second step of the property generation, which requires us to map the TPs of the test suite, we use a tool called Execution Flow (NIEMEC, 2020). It is a JUnit add-on that may be run for classes and projects, which goes through a JUnit test and maps the TPs of each test case. More details can be found at the project's repository.

For the third step, we simply calculate the percentage using any simple mathematics software.

This chapter focuses on the first step, and consists of an on-line survey on different CFG and PPC generation tools. Table 3.1 shows the tools that were found. We will study each of them, describe their characteristics, what they do and which limitations they contain. We will then choose which one works best for us.

Table 3.1 – Researched CFG and PPC generation tools.

Name	*Author*	*Year*
Dr. Garbage Tools	Dr. Garbage Community	2014
TRGeneration	Stan Lee, Evan Platt	2014
Soot	Open-Source Community	2020
Atlas	EnSoft Corp	2020
Control Flow Graph	A. Pena, N. Brondin-Bernard, J. Bardon	2014
Binnavi	Zynamics	2010

Source: author

3.1 Dr. Garbage Tools

Dr. Garbage Tools is "a suite of Eclipse Plugins released under Apache Open Source license". It consists of a plugin with three main tools: the byte code visualizer; the source code visualizer; and the CFG factory. Each one of these tools generate a flow graph for the selected method(s) of the application. The first one generates a byte code[1] graph, along with the byte code instructions in a sequential format. The second generates a graph for the method's source code, line by line, similar to what we need for this work. Finally, the third generates a flow graph where similar instructions (such as a sequential block of assignments) are grouped in one entire node. All of these features are interesting to us since we need to get the prime path requirements for the MUT. However, there are some problems which we will list below.

3.1.1 Compatibility

The first problem we have with Dr. Garbage Tools is that it is only supported by Eclipse versions lower than Mars[2], which is as old as 2015. The tools themselves were last updated in 2014. These issues are problematic because most of the libraries we are analyzing are not supported by older versions of Eclipse.

3.1.2 Dense code base

Since the project is not supported by newer versions of Eclipse, we analyze the possibility of extracting the parts of the tool's logic that we need, i.e. the logic necessary to take a source code and generate its flow graph in a textual format. However, upon inspecting the source of the project, it shows that it is very complex, consisting of many folders and files, making this extraction difficult. Most of this complexity is probably due to the nature of the tool - it is an Eclipse plug-in, which adds buttons to context menus with different actions.

[1] See https://techterms.com/definition/bytecode.

[2] See https://www.eclipse.org/mars/.

3.1.3 Lack of non-visual output

Lastly, another problem we have with these tools is that they do not provide the CFG output in a text format, which is what we need in order to generate PPCs. In order to get this version of the graph, we would have to modify parts of the tool's logic, which is complicated due to the dense code base as mentioned before.

3.2 TR Generation

This is a tool developed in Java by Seung Hun Lee and Evan Platt which reads from a source code and provides its CFG, as well as a desired coverage criteria, which is specified by the user. It is quite simple and objective, consisting of two main modules: the graph generator, which parses the source code while performing some clean up and adaption, and produces a graph that represents it; and the test requirement generator, which generates a coverage criteria based on the graph built in the previous module. The criteria may be at node, edge, edge-pair or prime path levels. Although the tool is very direct and suitable to us at first glance, mainly because it not only generates the CFG but also computes the PPC of the MUT, it contains some issues.

3.2.1 Logic limitation

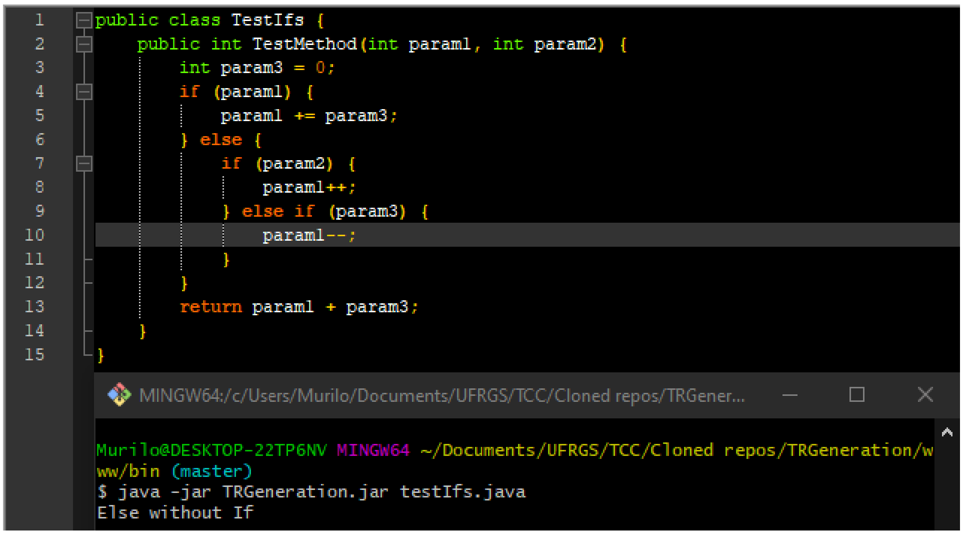
There are a couple of structures that the application does not support, such as try-catch blocks, ternary operators, for-each loops, annotations, multi-line comments, loop break instructions and do-while loops. This deficiency causes a huge limitation since most of the libraries we want to analyze contain at least a small part of these structures. However, most of these can be fixed by working on the tool's logic.

3.2.2 Multiple parsing issues

Some instructions are supported by the tool but may occasionally fail to work, such as an if-else when it is contained into some specific blocks. See Figure 3.1.

Another problem is that the logic of the tool will look for specific characters in order to take an action, such as semicolons, double slashes and brackets, however it does

Figure 3.1 – Source code with if-else statement fails with "Else without if".



```
public class TestIfs {
    public int TestMethod(int param1, int param2) {
        int param3 = 0;
        if (param1) {
            param1 += param3;
        } else {
            if (param2) {
                param1++;
            } else if (param3) {
                param1--;
            }
        }
        return param1 + param3;
    }
}
```

MINGW64:/c/Users/Murilo/Documents/UFRGS/TCC/Cloned repos/TRGener...

```
Murilo@DESKTOP-22TP6NV MINGW64 ~/Documents/UFRGS/TCC/Cloned repos/TRGeneration/w
ww/bin (master)
$ java -jar TRGeneration.jar testIfs.java
Else without If
```

Source: author

not check if they are indeed part of the source code logic or if they are just part of a variable (such as a string), and the tool will break in such cases. See Figure 3.2.

3.2.3 Loss of source code line information

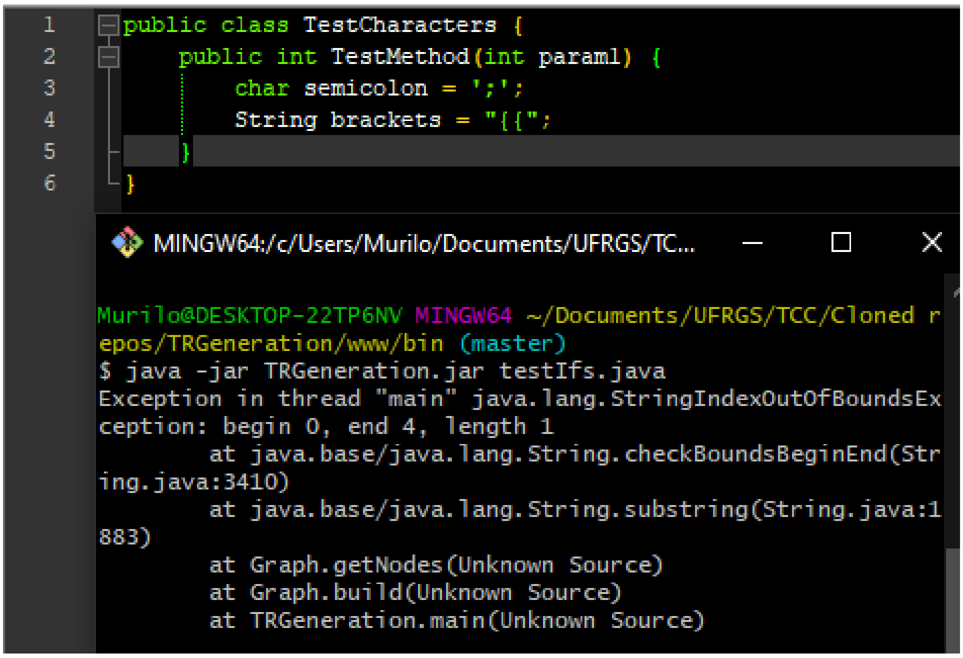
Another problem is that the test requirements that the tool generates are based directly on the graph generated in the first module of the application, and thus will contain the index of the nodes, and not the source code lines. This is a problem because we cannot directly compare the graph with the original source code.

Basically what the tool does is, it generates the graph where each node represents a block of similar instructions (such as three sequential assignments, for instance), and the test requirement tool will provide the PPC describing each path as a sequence of nodes in that graph. For example, Figure 3.4 represents a graph that was generated for the code shown in Figure 3.3. The node 0 represents everything that comes before the `if` instruction in one entire node.

3.2.4 No support of entire classes

Lastly, the application will parse a source code as one entire method, i.e. it does not fully support full classes with multiple methods. In order to execute it this way the

Figure 3.2 – String containing special characters causes buggy behavior.

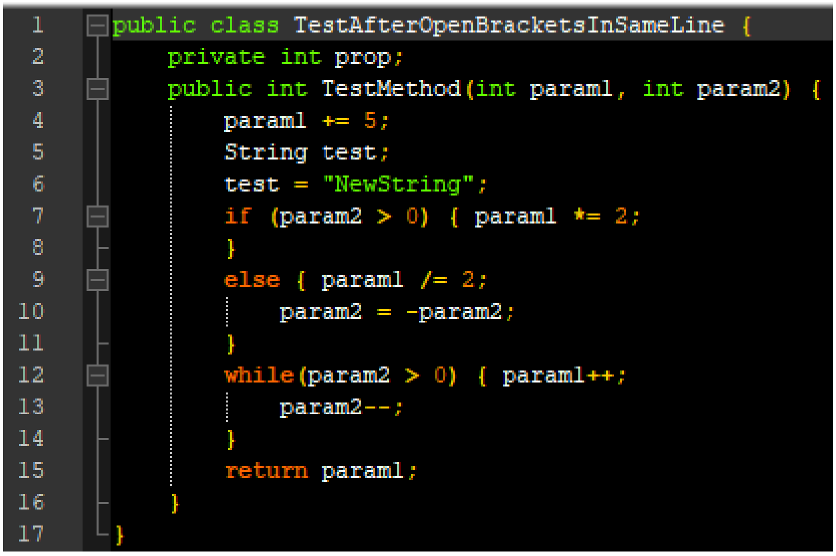


```
public class TestCharacters {
    public int TestMethod(int param1) {
        char semicolon = ';';
        String brackets = "{{";
    }
}
```

```
Murilo@DESKTOP-22TP6NV MINGW64 ~/Documents/UFRGS/TCC/Cloned r
epos/TRGeneration/www/bin (master)
$ java -jar TRGeneration.jar testIfs.java
Exception in thread "main" java.lang.StringIndexOutOfBoundsEx
ception: begin 0, end 4, length 1
        at java.base/java.lang.String.checkBoundsBeginEnd(Str
ing.java:3410)
        at java.base/java.lang.String.substring(String.java:1
883)
        at Graph.getNodes(Unknown Source)
        at Graph.build(Unknown Source)
        at TRGeneration.main(Unknown Source)
```

Source: author

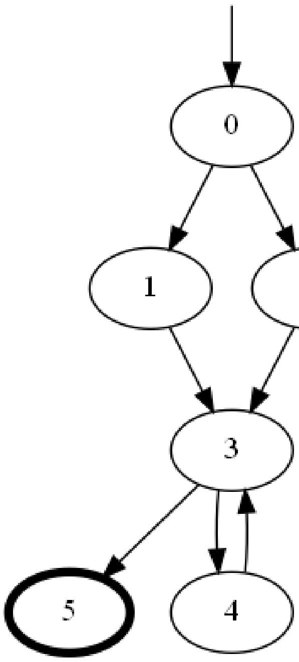
Figure 3.3 – Source code used to generate a CFG.



```
public class TestAfterOpenBracketsInSameLine {
    private int prop;
    public int TestMethod(int param1, int param2) {
        param1 += 5;
        String test;
        test = "NewString";
        if (param2 > 0) { param1 *= 2;
        }
        else { param1 /= 2;
            param2 = -param2;
        }
        while(param2 > 0) { param1++;
            param2--;
        }
        return param1;
    }
}
```

Source: author

Figure 3.4 – CFG generated for the source code in Figure 3.3.



Source: author

user would have to split all methods into different files and run the tool once for each file.

3.3 Soot

Soot is a Java optimization framework, containing many features and among them, one for generating CFGs for a source code. This one is interesting because it is more updated, still having changes pushed in the last few months. It works in a similar way to Dr. Garbage Tools, since it may be used as an Eclipse plug-in, however it may also be used as a terminal application.

The difficulty here is that, since the application is very complex, it is also hard to understand it and obtain the desired data. The documentation is not clear on how to produce a CFG from a source code, and there are not many guides in the internet, so this was scratched from the list in favor of the other two tools described before. It should still be noted that it is a powerful tool for many other uses.

3.4 Other analyzed tools

There are a few other tools that were found but were not analyzed in depth due to more suitable tools having been found earlier or their complexity being too high. Nevertheless, they will be briefly mentioned here.

Atlas[3] is a powerful tool for source code analysis, also available for C, which can be used to generate CFGs. However, it only exports the CFGs as images, and also requires a license in order to be used.

There is also a tool titled *Control Flow Graph*[4], which generates the CFG of a source code. However, it contains few documentation and was found to be hard to use.

Lastly, Binnavi[5] is a binary analysis IDE copyrighted by Google. It allows for generation and manipulation of CFGs from disassembled code. It was skipped since the project seems a little bit ahead of what we need and is not under active development anymore.

3.5 Concluding remarks and selected tool

After the research on CFG and PPC generation tools is over, the tool we pick to work with is the Structural Test Requirement Generator by Platt and Lee, described in Section 3.2. It contains many limitations, but they are easier to fix than the other projects because the code is considerably more readable, and it contains just what we need, and nothing else.

In the next chapter we will describe the modifications made to the tool and the main difficulties encountered in the process.

[3] See http://www.ensoftcorp.com/atlas/.

[4] See https://github.com/masters-info-nantes/control-flow-graph.

[5] See https://github.com/google/binnavi.

4 ADAPTATION OF TEST REQUIREMENT GENERATION TOOL

Throughout this chapter we will dig into each of the main modifications that were necessary for the chosen tool, TRGeneration, to work. Some more specific, small and complex modifications will be just briefly mentioned for simplicity reasons. At the end of the chapter we present a summary of what has been changed.

4.1 Changes to the project structure

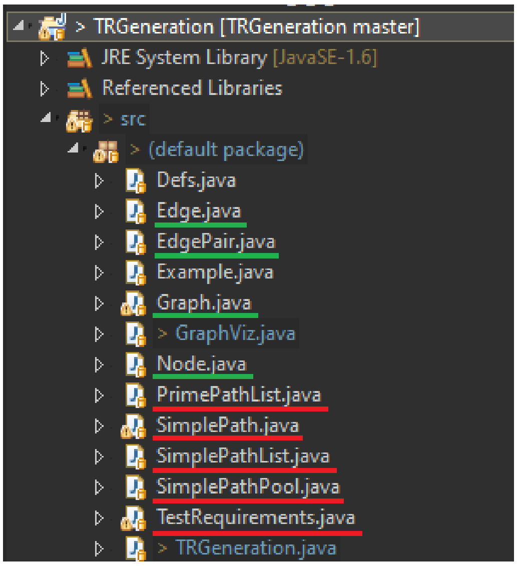
The first main change to the project was in its structure. To recapitulate, the project was initially composed of two main modules: the graph generator and the test requirement generator. A screenshot of the project's original structure can be viewed in Figure 4.1 where green underlined classes represent the logic responsible for the graph generation, and red underlined classes represent the logic responsible for the test requirement generation. Other non-marked classes are not relevant for now.

Upon inspection of the project, it is possible to see that the building of the graph, shown in Figure 4.2, consists of six steps: a cleanup of the source code, which removes and adapts parts of the source code being analyzed; an insertion of blank dummy nodes, used as mid-way points in junctions of nodes; an identification of the nodes based on the clean source code; a method to number these nodes; a method to combine nodes that represent a sequential set of instructions; and a method to fix the node numbers after the combination has been made.

These methods can be mainly split into two groups: one that performs a cleanup (pre-processing) of the project, consisted of the two first steps of the build process, and one that builds the graph nodes and edges, consisted of the four latter steps. Taking this into consideration, the first modification we perform is to split the pre-processing of the source code of the MUT into its own class, called *CodeCleaner*, and we move all methods related to the source code manipulation into it. This makes the adaptation of the tool easier since all modifications we must do regarding the pre-processing are contained in only one file, and it also makes it easier to test the application. The initial version of the *CodeCleaner* class is shown in Figure 4.3.

Another modification we make relates to the lack of support for entire classes - the tool will not run correctly if provided a Java file with multiple methods or classes. In order to solve this, we must create a piece of logic that will parse an entire file and identify the

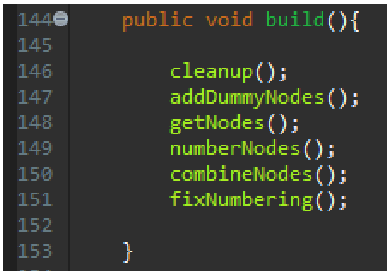
Figure 4.1 – Current project structure.



```
> TRGeneration [TRGeneration master]
    JRE System Library [JavaSE-1.6]
    Referenced Libraries
    > src
        > (default package)
            Defs.java
            Edge.java
            EdgePair.java
            Example.java
            Graph.java
            > GraphViz.java
            Node.java
            PrimePathList.java
            SimplePath.java
            SimplePathList.java
            SimplePathPool.java
            TestRequirements.java
            > TRGeneration.java
```

Source: author

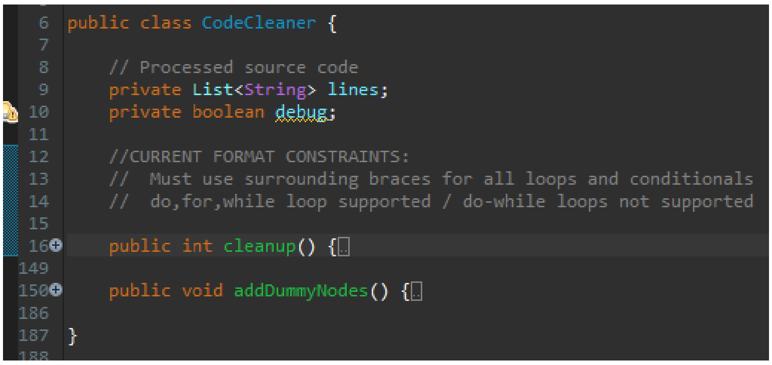
Figure 4.2 – Method responsible for graph building.



```java
    public void build(){

        cleanup();
        addDummyNodes();
        getNodes();
        numberNodes();
        combineNodes();
        fixNumbering();

    }
```

Source: author

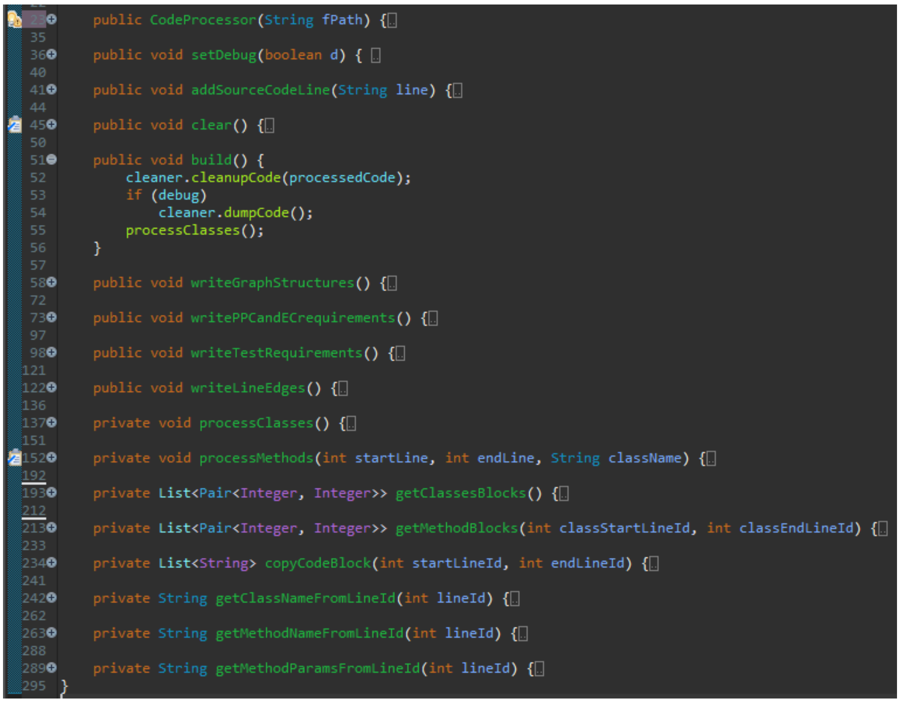
Figure 4.3 – Initial version of the class responsible for cleaning the source code.



```java
public class CodeCleaner {

    // Processed source code
    private List<String> lines;
    private boolean debug;

    //CURRENT FORMAT CONSTRAINTS:
    //  Must use surrounding braces for all loops and conditionals
    //  do,for,while loop supported / do-while loops not supported

    public int cleanup() {□

    public void addDummyNodes() {□

}
```

Source: author

Figure 4.4 – *CodeProcessor* is the class responsible for general class and method parsing. The cleanup method that is called will execute both `cleanup` and `addDummyNodes` methods present in Figure 4.3



```
    public CodeProcessor(String fPath) {□

    public void setDebug(boolean d) { □

    public void addSourceCodeLine(String line) {□

    public void clear() {□

    public void build() {
        cleaner.cleanupCode(processedCode);
        if (debug)
            cleaner.dumpCode();
        processClasses();
    }

    public void writeGraphStructures() {□

    public void writePPCandECrequirements() {□

    public void writeTestRequirements() {□

    public void writeLineEdges() {□

    private void processClasses() {□

    private void processMethods(int startLine, int endLine, String className) {□

    private List<Pair<Integer, Integer>> getClassesBlocks() {□

    private List<Pair<Integer, Integer>> getMethodBlocks(int classStartLineId, int classEndLineId) {□

    private List<String> copyCodeBlock(int startLineId, int endLineId) {□

    private String getClassNameFromLineId(int lineId) {□

    private String getMethodNameFromLineId(int lineId) {□

    private String getMethodParamsFromLineId(int lineId) {□
}
```

Source: author

blocks of code that represent a class and a method. We also must store the names of each one, in order to label them when saving the files with the outputs. This piece of logic will be placed in a class we will call *CodeProcessor* and can be viewed in Figure 4.4. Besides method identification, it also contains some methods used to instantiate and manipulate the test requirement generator.

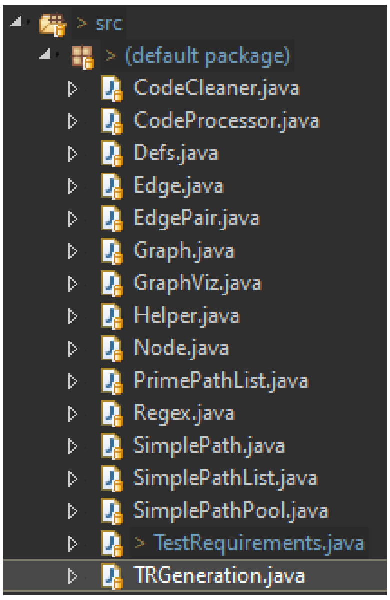
Finally, we create a helper class to contain simple methods that are reused throughout the application, and later a regex class to contain all regular expressions that are used in the pre-processing of the MUT's code to identify patterns and make decisions. We will talk more about these classes in Section 4.2.4.

The final project file structure can be viewed in Figure 4.5.

4.2 Changes to the cleanup process

The cleanup process originally consists of one entire method that performs many changes to the source code (called `cleanup`) and a second one that adds dummy nodes to specific parts of it (called `addDummyNodes`). Since the first method is the one that

Figure 4.5 – Updated project structure.



```
> src
  > (default package)
    CodeCleaner.java
    CodeProcessor.java
    Defs.java
    Edge.java
    EdgePair.java
    Graph.java
    GraphViz.java
    Helper.java
    Node.java
    PrimePathList.java
    Regex.java
    SimplePath.java
    SimplePathList.java
    SimplePathPool.java
    > TestRequirements.java
    TRGeneration.java
```

Source: author

contains the most of the cleanup logic, we will focus mainly on it.

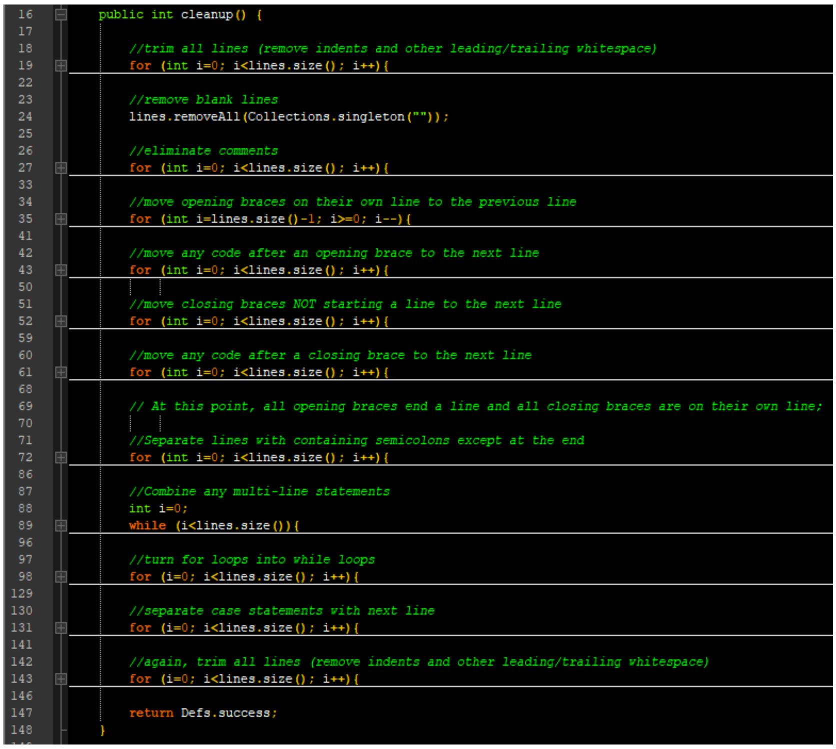
If we take a look into the `cleanup` method, shown in Figure 4.6, we can see it consists of many instruction blocks that perform independent adaptations to the code (such as removal of comments, re-positioning of brackets, etc). This is evidenced by the comments in the code. So taking this into consideration, it is easy to see we can split the method into different sub-methods, each responsible for a different modification in the MUT's source code. We can also add some debugging logs for future problem handling. See Figure 4.7.

After splitting the clean up process into different methods, we can now bring support to some things that were not supported by the tool initially (such as `try-catch` blocks, one liner `if` blocks and loops, `for-each` loops, etc) and fix some of the structures that were having problems (annotations, specific constructions of `if-then-else` statements, etc). Let us take a look into each one carefully.

4.2.1 For-each support

Before modifying the application to support the `for-each` structure, we must first take a look into how it handles the `for` structure, since they are very similar. Understanding this requires looking at two of the cleanup steps: the first one inserts a line break

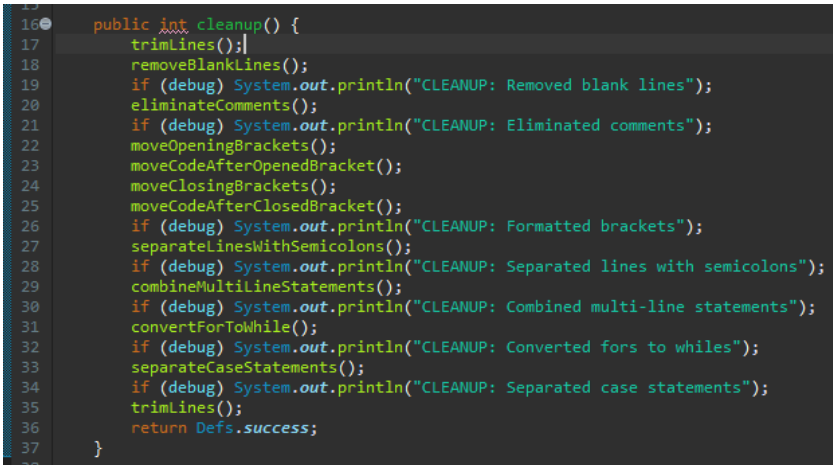
Figure 4.6 – Current state of `cleanup` method.



```
public int cleanup() {

    //trim all lines (remove indents and other leading/trailing whitespace)
    for (int i=0; i<lines.size(); i++){

    //remove blank lines
    lines.removeAll(Collections.singleton(""));

    //eliminate comments
    for (int i=0; i<lines.size(); i++){

    //move opening braces on their own line to the previous line
    for (int i=lines.size()-1; i>=0; i--){

    //move any code after an opening brace to the next line
    for (int i=0; i<lines.size(); i++){

    //move closing braces NOT starting a line to the next line
    for (int i=0; i<lines.size(); i++){

    //move any code after a closing brace to the next line
    for (int i=0; i<lines.size(); i++){

    // At this point, all opening braces end a line and all closing braces are on their own line;

    //Separate lines with containing semicolons except at the end
    for (int i=0; i<lines.size(); i++){

    //Combine any multi-line statements
    int i=0;
    while (i<lines.size()){

    //turn for loops into while loops
    for (i=0; i<lines.size(); i++){

    //separate case statements with next line
    for (i=0; i<lines.size(); i++){

    //again, trim all lines (remove indents and other leading/trailing whitespace)
    for (i=0; i<lines.size(); i++){

    return Defs.success;
}
```

Source: author

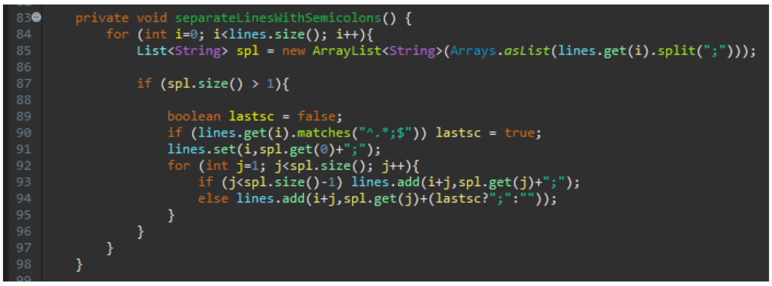
Figure 4.7 – Refactored version of `cleanup` method.



```
public int cleanup() {
    trimLines();
    removeBlankLines();
    if (debug) System.out.println("CLEANUP: Removed blank lines");
    eliminateComments();
    if (debug) System.out.println("CLEANUP: Eliminated comments");
    moveOpeningBrackets();
    moveCodeAfterOpenedBracket();
    moveClosingBrackets();
    moveCodeAfterClosedBracket();
    if (debug) System.out.println("CLEANUP: Formatted brackets");
    separateLinesWithSemicolons();
    if (debug) System.out.println("CLEANUP: Separated lines with semicolons");
    combineMultiLineStatements();
    if (debug) System.out.println("CLEANUP: Combined multi-line statements");
    convertForToWhile();
    if (debug) System.out.println("CLEANUP: Converted fors to whiles");
    separateCaseStatements();
    if (debug) System.out.println("CLEANUP: Separated case statements");
    trimLines();
    return Defs.success;
}
```

Source: author

Figure 4.8 – Method that splits any lines with semicolons, which breaks down the for declaration into three lines (initialization, condition and step).



```
    private void separateLinesWithSemicolons() {
        for (int i=0; i<lines.size(); i++){
            List<String> spl = new ArrayList<String>(Arrays.asList(lines.get(i).split(";")));

            if (spl.size() > 1){

                boolean lastsc = false;
                if (lines.get(i).matches("^.*;$")) lastsc = true;
                lines.set(i,spl.get(0)+";");
                for (int j=1; j<spl.size(); j++){
                    if (j<spl.size()-1) lines.add(i+j,spl.get(j)+";");
                    else lines.add(i+j,spl.get(j)+(lastsc?";":""));
                }
            }
        }
    }
```

Source: author

after any semicolon, and the second one extracts the `for` loop properties (initialization, step and exit condition) in order to convert it to a `while` loop. They are shown in Figures 4.8 and 4.9 respectively.

Taking this into consideration, we can conclude that, if we convert our `for-each` loop into a `for` loop, it can be later converted into a `while` loop and correctly processed by the application, so that is all we have to do - create a new method in the cleanup process that executes before the ones that process the `for` loop.

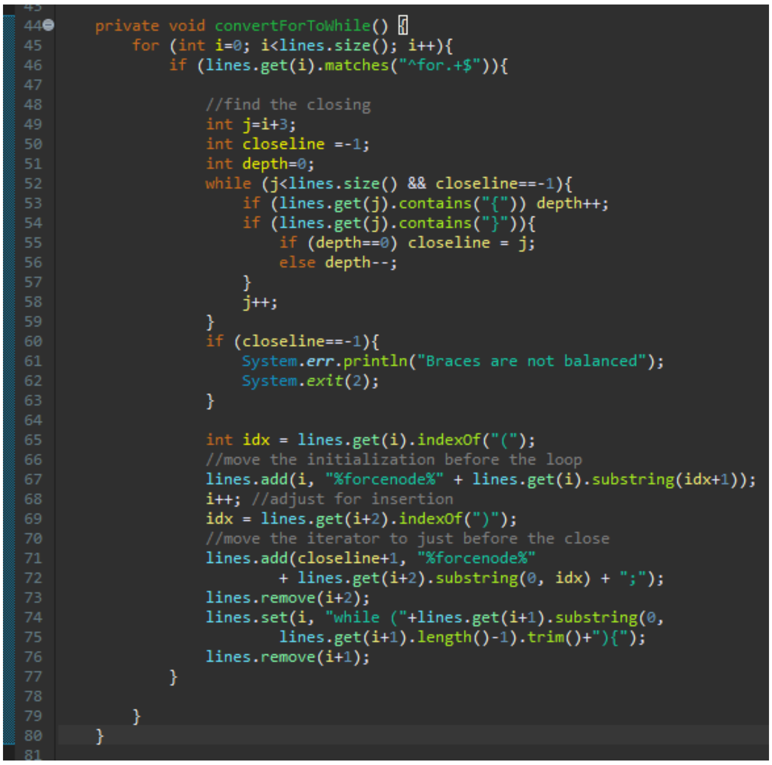
Since the difference between a `for` loop and a `for-each` loop is basically just the initialization line, we can focus on just that: we memorize the type of the item variable (let us call it T), its name and the object being iterated. We then create an iterator of type T[1] and initialize it over the object being iterated. This is the initialization step of the resulting `for` loop. The end condition will test for the `hasNext()` property of the iterator, and the step will be left blank. A visual representation can be seen in Figure 4.10.

Besides that, we must place an instruction after the `for` declaration line, which will declare the item variable and assign it to the iterator current value, moving it to the next item in the object being iterated - which is why we can leave the step blank. This can be done using the instruction `next()` of the iterator structure. The resulting logic can be seen in Figure 4.11.[2]

[1] For primitive types, such as `int` and `char`, it may be necessary to convert them to non-primitive types. However, since semantic errors do not really affect the test requirement generation, we can ignore this.

[2] This is one of the places we use a helper method to look for reserved characters while parsing the code. This is more well-detailed in Section 4.2.4.

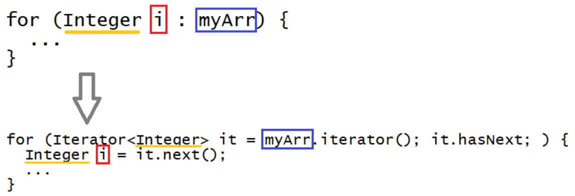
Figure 4.9 – Method that converts any `for` instruction into `while`.



```java
    private void convertForToWhile() {
        for (int i=0; i<lines.size(); i++){
            if (lines.get(i).matches("^for.+$")){

                //find the closing
                int j=i+3;
                int closeline =-1;
                int depth=0;
                while (j<lines.size() && closeline==-1){
                    if (lines.get(j).contains("{")) depth++;
                    if (lines.get(j).contains("}")){
                        if (depth==0) closeline = j;
                        else depth--;
                    }
                    j++;
                }
                if (closeline==-1){
                    System.err.println("Braces are not balanced");
                    System.exit(2);
                }

                int idx = lines.get(i).indexOf("(");
                //move the initialization before the loop
                lines.add(i, "%forcenode%" + lines.get(i).substring(idx+1));
                i++; //adjust for insertion
                idx = lines.get(i+2).indexOf(")");
                //move the iterator to just before the close
                lines.add(closeline+1, "%forcenode%"
                        + lines.get(i+2).substring(0, idx) + ";");
                lines.remove(i+2);
                lines.set(i, "while ("+lines.get(i+1).substring(0,
                        lines.get(i+1).length()-1).trim()+"){");
                lines.remove(i+1);
            }

        }
    }
```

Source: author

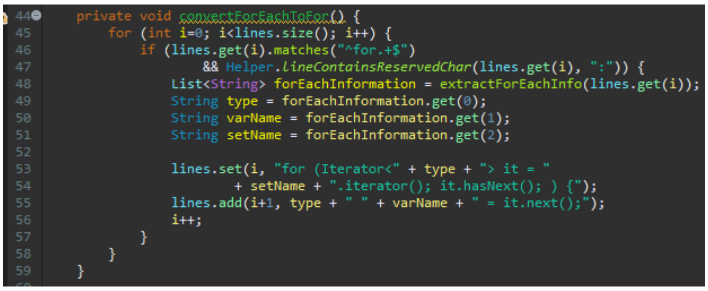
Figure 4.10 – Procedure to convert `for-each` loops into `for` loops.



```java
for (Integer i : myArr) {
    ...
}

for (Iterator<Integer> it = myArr.iterator(); it.hasNext; ) {
  Integer i = it.next();
  ...
}
```

Source: author

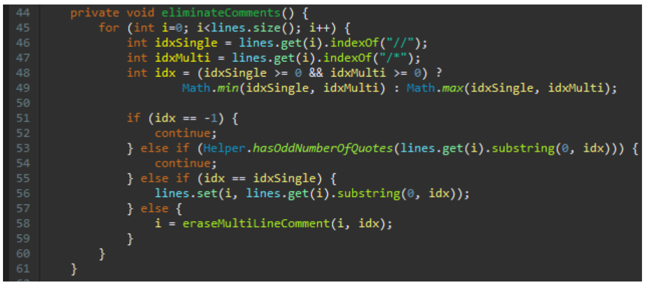
Figure 4.11 – Logic written to perform the `for-each` conversion.



```java
    private void convertForEachToFor() {
        for (int i=0; i<lines.size(); i++) {
            if (lines.get(i).matches("^for.+$")
                    && Helper.lineContainsReservedChar(lines.get(i), ":")) {
                List<String> forEachInformation = extractForEachInfo(lines.get(i));
                String type = forEachInformation.get(0);
                String varName = forEachInformation.get(1);
                String setName = forEachInformation.get(2);

                lines.set(i, "for (Iterator<" + type + "> it = "
                        + setName + ".iterator(); it.hasNext(); ) {");
                lines.add(i+1, type + " " + varName + " = it.next();");
                i++;
            }
        }
    }
```

Source: author

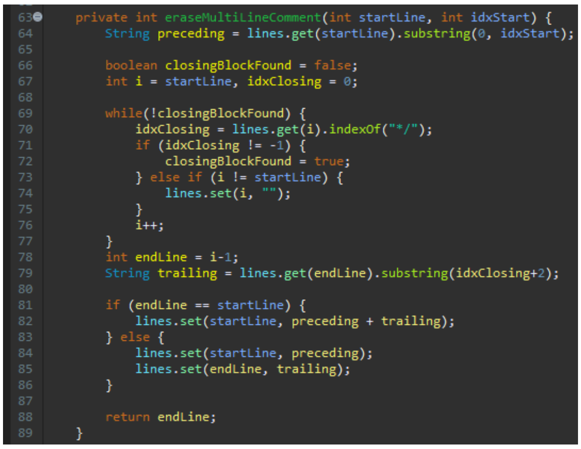
Figure 4.12 – Removal of comments in the source code: main method.



```java
    private void eliminateComments() {
        for (int i=0; i<lines.size(); i++) {
            int idxSingle = lines.get(i).indexOf("//");
            int idxMulti = lines.get(i).indexOf("/*");
            int idx = (idxSingle >= 0 && idxMulti >= 0) ?
                    Math.min(idxSingle, idxMulti) : Math.max(idxSingle, idxMulti);

            if (idx == -1) {
                continue;
            } else if (Helper.hasOddNumberOfQuotes(lines.get(i).substring(0, idx))) {
                continue;
            } else if (idx == idxSingle) {
                lines.set(i, lines.get(i).substring(0, idx));
            } else {
                i = eraseMultiLineComment(i, idx);
            }
        }
    }
```

Source: author

Figure 4.13 – Removal of comments in the source code: processing of multi-line comments.



```java
    private int eraseMultiLineComment(int startLine, int idxStart) {
        String preceding = lines.get(startLine).substring(0, idxStart);

        boolean closingBlockFound = false;
        int i = startLine, idxClosing = 0;

        while(!closingBlockFound) {
            idxClosing = lines.get(i).indexOf("*/");
            if (idxClosing != -1) {
                closingBlockFound = true;
            } else if (i != startLine) {
                lines.set(i, "");
            }
            i++;
        }
        int endLine = i-1;
        String trailing = lines.get(endLine).substring(idxClosing+2);

        if (endLine == startLine) {
            lines.set(startLine, preceding + trailing);
        } else {
            lines.set(startLine, preceding);
            lines.set(endLine, trailing);
        }

        return endLine;
    }
```

Source: author

4.2.2 Comment processing improvement

Since the libraries' source codes contain lots of comments of both single-line and multi-line types, you just have to run the tool in one of the files to see that multi-line comments are not supported.

The current method for eliminating comments will only check for a double slash in the line and remove everything after it. So we have to add an extra check that looks for a multi-line comment start token, and if it is found we set a Boolean variable that will tell the logic it is looking for a multi-line comment closing token. It is important to note that the comment may end in the same line it has started, so we need to take that into consideration.

While the token has not been found, we store the data we need to remove, and once it is, we remove everything. The final logic is presented in Figures 4.12 and 4.13.

There is another problem that arises from the way the application gets the index of the symbols (for both single and multi-line comment), which was already described in Section 3.2.2. We will present a generic solution for it in Section 4.2.4.

4.2.3 Tracking of source code original lines

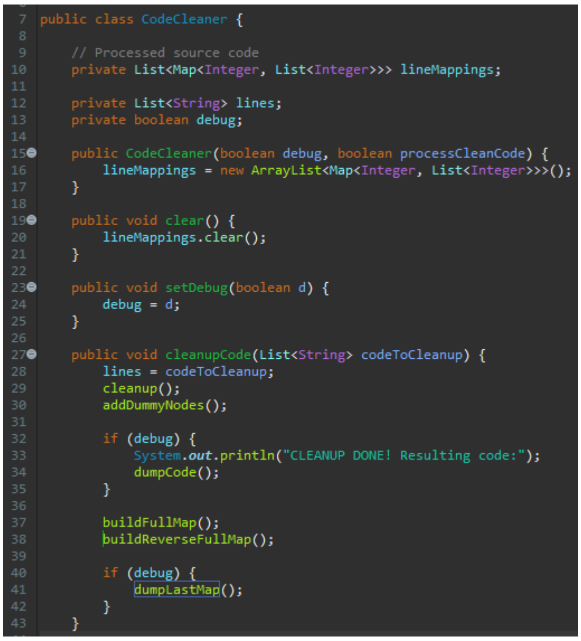
One of the major changes in the cleanup process is related to the output of the tool. The PPC generated by TRGeneration is done at node level, as described in Section 3.2.3. Besides this, the *Node* class already contains a field for storing the first line index that makes it, and another that stores the source code present in it. However, the line relates to the clean source code, and not the original, unmodified one. This implies in two changes we must perform: one to map the original source code line indexes to the ones in the clean source code, and another to make the nodes store which lines they represent (instead of only the first line). The first change is made in the cleanup process.

What we do is, whenever a method that transforms the source code is executed, we store a map from the index of the lines before the transformation to the indexes after the transformation. This implies in one map for each modification we have the source code go through. Since we have many transformations (for each loops into for loops, for loops into while loop, removal of comments, etc) we modify the class to initialize a list of maps at its construction, and we append a new map to it every time we transform the code. When all transformations are done, we call a helper method to build the final map, which contains the original line indexes mapped to the clean line indexes. See Figure 4.14.

This change is not very simple since some transformations move many things around. For instance, to convert a `for` loop into a `while` loop the line containing the declaration of the instruction is split into three other lines. Moreover, there are some small tweaks that must be made to ensure that it works correctly. An example of the logic to store the mapping can be seen in Figure 4.15.

The other change that composes this feature is done in the NODE class, which is basically storing all lines that make the instruction, and looking for the original source code line through the map and replacing the stored line reference with that value.

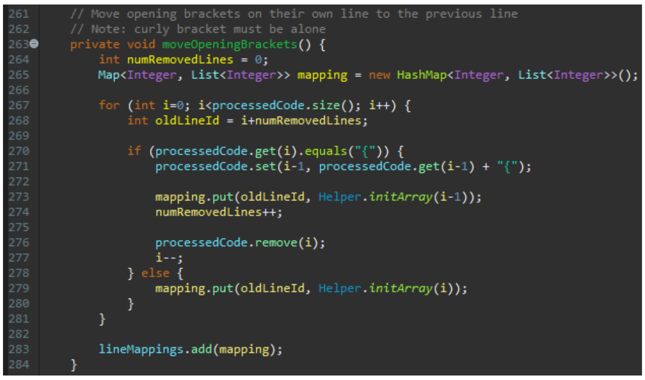
Figure 4.14 – Declaration and initialization of map structure containing the mapping from original line indexes to target line indexes.



```java
public class CodeCleaner {

    // Processed source code
    private List<Map<Integer, List<Integer>>> lineMappings;

    private List<String> lines;
    private boolean debug;

    public CodeCleaner(boolean debug, boolean processCleanCode) {
        lineMappings = new ArrayList<Map<Integer, List<Integer>>>();
    }

    public void clear() {
        lineMappings.clear();
    }

    public void setDebug(boolean d) {
        debug = d;
    }

    public void cleanupCode(List<String> codeToCleanup) {
        lines = codeToCleanup;
        cleanup();
        addDummyNodes();

        if (debug) {
            System.out.println("CLEANUP DONE! Resulting code:");
            dumpCode();
        }

        buildFullMap();
        buildReverseFullMap();

        if (debug) {
            dumpLastMap();
        }
    }
```

Source: author

Figure 4.15 – Example of mapping manipulation in the method that moves sole opening brackets to the line before them, along with its instruction.



```java
    // Move opening brackets on their own line to the previous line
    // Note: curly bracket must be alone
    private void moveOpeningBrackets() {
        int numRemovedLines = 0;
        Map<Integer, List<Integer>> mapping = new HashMap<Integer, List<Integer>>();

        for (int i=0; i<processedCode.size(); i++) {
            int oldLineId = i+numRemovedLines;

            if (processedCode.get(i).equals("{")) {
                processedCode.set(i-1, processedCode.get(i-1) + "{");

                mapping.put(oldLineId, Helper.initArray(i-1));
                numRemovedLines++;

                processedCode.remove(i);
                i--;
            } else {
                mapping.put(oldLineId, Helper.initArray(i));
            }
        }

        lineMappings.add(mapping);
    }
```

.

Source: author

4.2.4 Other fixed issues

There are some other small changes that were made. Support for codes with annotation is provided (they should be treated as comments and removed as they do not characterize an executable instruction). Another set of small changes are performed in order to support `switch-case` statements where multiple conditions produce the same execution. Also, when dealing with `for` loops, the step is cloned to the line before any continue instructions, in order to provide support for it.

There is a group of methods that format the source code based on the brackets - the pattern has any open bracket stay in the line of the instruction that initializes its block, and the closing bracket stay in the line below the last instruction of that block. Since we will need to perform this patterning more than once (due to other transformations[3] we make in the project), we refactor this group of methods into a single method, which will be called any time a reformat is needed.

Support for single instruction `if`/`for`/`while` blocks is also provided, since they do not require brackets in this case. The line is rewritten with newly added brackets in the correct positions, and afterwards the logic re-formats the code by calling the method we created earlier, that re-positions the brackets.

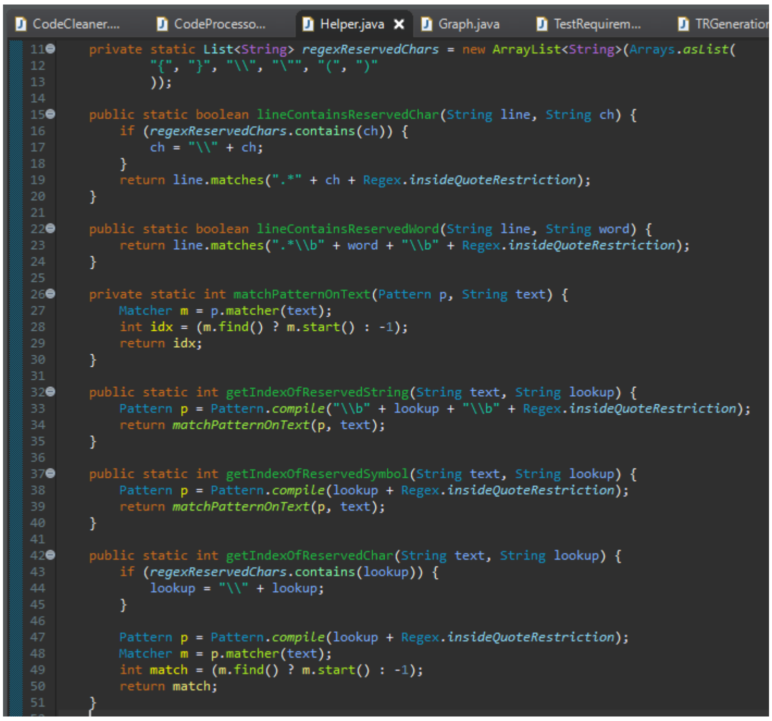
There is also a couple of modifications that are made to the queries the code does to find specific characters and take a decision - such as slashes for comments, brackets and reserved instructions. The tool will initially fail if these tokens are present inside a string, as shown in Figure 3.2, because it does not test for quotes or other syntax elements. To prevent this, we use some detailed regex queries that will match the tokens only if they are valid (for instance, URLs in a string will not be interpreted as a comment due to the slashes after the protocol). See Figure 4.16.

Finally, we call some of the cleanup steps more than once due to specific modifications we made throughout the logic (for instance, after adding brackets to one-liner ifs or whiles, we must reformat the brackets again). This ends up causing some code repetition, but it does the job we want it to do.

The final logic of the cleanup method is shown in Figure 4.17.

[3] one-liner `if` and loop blocks

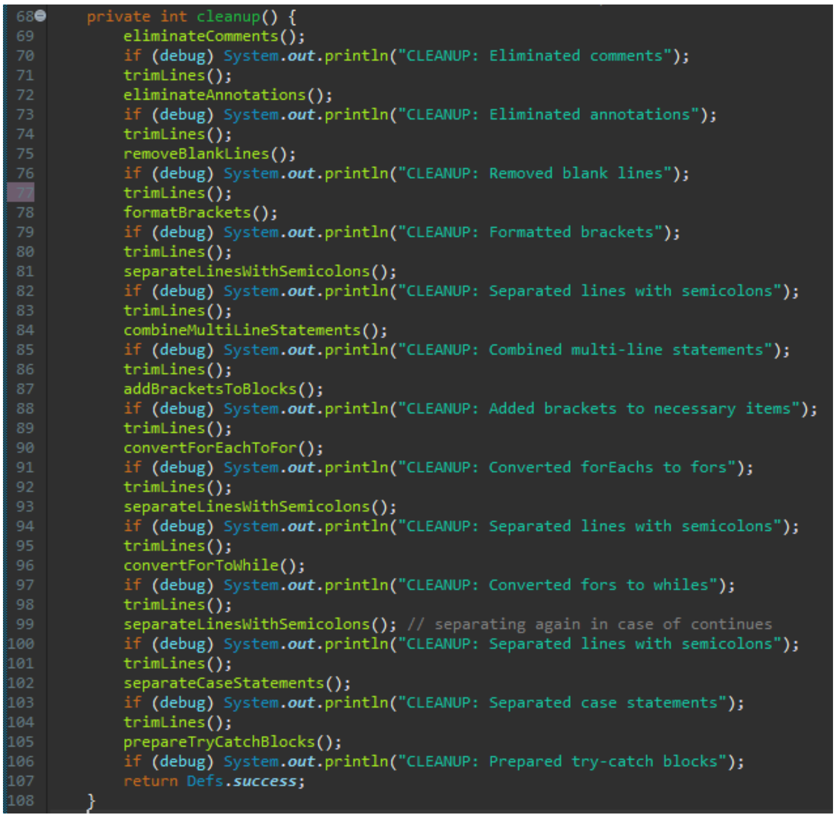
Figure 4.16 – Use of helper methods to test for reserved words and tokens.



```java
    private static List<String> regexReservedChars = new ArrayList<String>(Arrays.asList(
            "{", "}", "\\", "\"", "(", ")"
            ));

    public static boolean lineContainsReservedChar(String line, String ch) {
        if (regexReservedChars.contains(ch)) {
            ch = "\\" + ch;
        }
        return line.matches(".*" + ch + Regex.insideQuoteRestriction);
    }

    public static boolean lineContainsReservedWord(String line, String word) {
        return line.matches(".*\\b" + word + "\\b" + Regex.insideQuoteRestriction);
    }

    private static int matchPatternOnText(Pattern p, String text) {
        Matcher m = p.matcher(text);
        int idx = (m.find() ? m.start() : -1);
        return idx;
    }

    public static int getIndexOfReservedString(String text, String lookup) {
        Pattern p = Pattern.compile("\\b" + lookup + "\\b" + Regex.insideQuoteRestriction);
        return matchPatternOnText(p, text);
    }

    public static int getIndexOfReservedSymbol(String text, String lookup) {
        Pattern p = Pattern.compile(lookup + Regex.insideQuoteRestriction);
        return matchPatternOnText(p, text);
    }

    public static int getIndexOfReservedChar(String text, String lookup) {
        if (regexReservedChars.contains(lookup)) {
            lookup = "\\" + lookup;
        }

        Pattern p = Pattern.compile(lookup + Regex.insideQuoteRestriction);
        Matcher m = p.matcher(text);
        int match = (m.find() ? m.start() : -1);
        return match;
    }
```

Source: author

Figure 4.17 – Final state of the cleanup method.



```java
    private int cleanup() {
        eliminateComments();
        if (debug) System.out.println("CLEANUP: Eliminated comments");
        trimLines();
        eliminateAnnotations();
        if (debug) System.out.println("CLEANUP: Eliminated annotations");
        trimLines();
        removeBlankLines();
        if (debug) System.out.println("CLEANUP: Removed blank lines");
        trimLines();
        formatBrackets();
        if (debug) System.out.println("CLEANUP: Formatted brackets");
        trimLines();
        separateLinesWithSemicolons();
        if (debug) System.out.println("CLEANUP: Separated lines with semicolons");
        trimLines();
        combineMultiLineStatements();
        if (debug) System.out.println("CLEANUP: Combined multi-line statements");
        trimLines();
        addBracketsToBlocks();
        if (debug) System.out.println("CLEANUP: Added brackets to necessary items");
        trimLines();
        convertForEachToFor();
        if (debug) System.out.println("CLEANUP: Converted forEachs to fors");
        trimLines();
        separateLinesWithSemicolons();
        if (debug) System.out.println("CLEANUP: Separated lines with semicolons");
        trimLines();
        convertForToWhile();
        if (debug) System.out.println("CLEANUP: Converted fors to whiles");
        trimLines();
        separateLinesWithSemicolons(); // separating again in case of continues
        if (debug) System.out.println("CLEANUP: Separated lines with semicolons");
        trimLines();
        separateCaseStatements();
        if (debug) System.out.println("CLEANUP: Separated case statements");
        trimLines();
        prepareTryCatchBlocks();
        if (debug) System.out.println("CLEANUP: Prepared try-catch blocks");
        return Defs.success;
    }
```

Source: author

4.3 Changes to the graph generation process

The graph generation is one of the most complex parts of the logic since it deals with edge manipulation based on the instructions of the clean source code. Here we will avoid digging too much into the logic since it can become confusing, but we will focus more on the theory of the solutions.

The main changes are in the method that adds the edges themselves, and not in the ones that simplify the graph.

4.3.1 Try-catch support

Among the changes made to the graph generation process, a notable one is the added support for `try-catch` and `finally` blocks. Before processing the code, it is necessary to add some tokens in the clean up process that prevents the logic from grouping `try` blocks with `finally` blocks or outer blocks. This is one of the cases where three sequential instructions may not be in sequential line indexes (an assumption that the original tool makes).

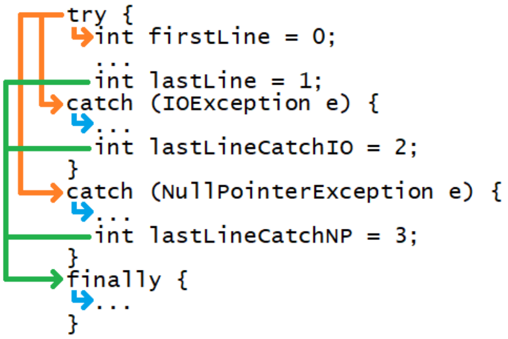
The `try-catch` case is a bit difficult to deal with since the program can jump from the try block to the `catch` block at any moment if a instruction fail, and it is hard to determine every possible path. So our target here will be to support the `try` and `finally` blocks, which can be treated almost as default sequential blocks, and support the `catch` blocks as best as we can.

The proposed solution is to add an edge from the `try` starting line to any `catch` starting lines, and then adding an edge from each of the `catch` blocks endings to the finally block (or to the first line outside of the system if there is no finally block). This will generate at least one PP for each `catch` block, making the code pass through them. Of course this might not be the best way to handle the `try-catch` system, but it is the definition we use for it. See Figure 4.18.

4.3.2 Switch-case support improvement

The way the tool deals with `switch` structures is slightly different from what we would expect. It adds edges from the `switch` declaration line to each of the `case` lines,

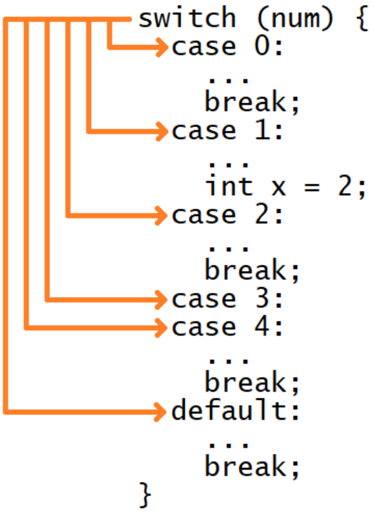
Figure 4.18 – Solution provided for the `try-catch` problem.



```
try {
  int firstLine = 0;
  ...
  int lastLine = 1;
catch (IOException e) {
  ...
  int lastLineCatchIO = 2;
}
catch (NullPointerException e) {
  ...
  int lastLineCatchNP = 3;
}
finally {
  ...
}
```

Source: author

Figure 4.19 – The way the application currently designs the graph of a `switch` structure.



```
switch (num) {
  case 0:
    ...
    break;
  case 1:
    ...
    int x = 2;
  case 2:
    ...
    break;
  case 3:
  case 4:
    ...
    break;
  default:
    ...
    break;
}
```

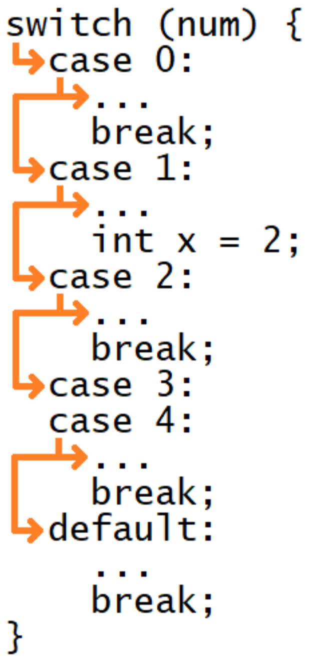
Source: author

while we would expect it to add edges throughout the clause lines in a more sequential form, almost as if it were a block of `if-else` statements. A visual representation can be viewed in Figure 4.19. Besides this, it does not really support fall through, causing some mistaken outputs when it occurs.

In order to fix these issues we will rewrite the logic that deals with `switch` structures, since it is not a complicated one. We must take care to keep track of each conditional block and add the edges from the previous accordingly.

The new algorithm for adding `switch-case` edges can be seen in Figure 4.20.

Figure 4.20 – New proposed way of dealing with switch blocks.



```
switch (num) {
 case 0:
   ...
   break;
 case 1:
   ...
   int x = 2;
 case 2:
   ...
   break;
 case 3:
 case 4:
   ...
   break;
 default:
   ...
   break;
}
```

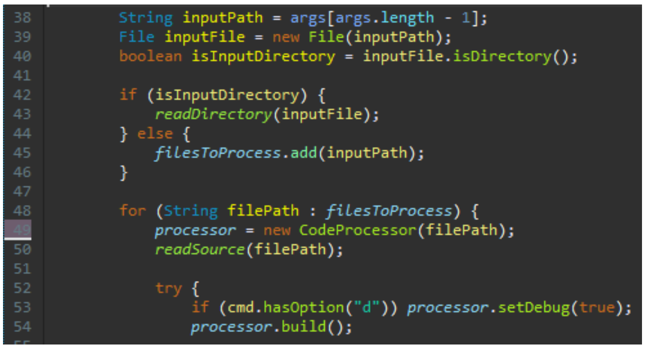
Source: author

4.3.3 Other fixed issues

Some other small changes are added. Support for `continue` and `break` statements inside loops is provided, since it is not present in the original logic. This requires us to look for the latest declared loop and can be a bit complex since the statement may be deep inside the loop structure.

There is also two new methods worth pointing out: one that is implemented to generate the line version of the graph, after it has been generated at node level; and another that iterates through the nodes of the graph and updates the source code line indexes that they reference, summing it to a value that is informed. This latter method is used because, as mentioned in 4.1, the tool does not support multiple methods initially, so we generate one graph for each method. The problem is that the graph will always start at line zero, because it is analysing the method separately, so we need to update the indexes after the computation has been finished.

Figure 4.21 – Partial version of the input processing.



```java
String inputPath = args[args.length - 1];
File inputFile = new File(inputPath);
boolean isInputDirectory = inputFile.isDirectory();

if (isInputDirectory) {
    readDirectory(inputFile);
} else {
    filesToProcess.add(inputPath);
}

for (String filePath : filesToProcess) {
    processor = new CodeProcessor(filePath);
    readSource(filePath);

    try {
        if (cmd.hasOption("d")) processor.setDebug(true);
        processor.build();
```

Source: author

4.4 Changes to the application input and output

Besides logic changes to the application, there is also a set of changes that were made in the way the application receives and outputs information, which is done in the class called *TRGeneration*. Currently, the tool will receive a file to be processed and output two things: a file containing a visual representation of the CFG, and some text in the terminal containing the test requirements. This is fine for short source codes and methods, but when dealing with a big set of files (in the libraries we want to analyze, for instance), this will cause a lot of confusion since everything will be sent to the terminal together.

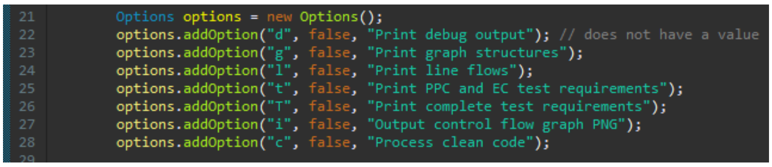
The tool currently accepts three inputs: the source code being analyzed, a flag `-o` that specified the file name for the outputted PNG file, and a debug flag `-d`. Since we are going to modify the tool to accept many classes and methods across multiple folders and sub-folders, the output name flag does not matter anymore, so we can remove it.

In chapter 4.1 we already explained how we made the application support multiple classes and methods in one Java file. However, we also need to make it iterate through the files of a directory, since libraries usually contain many files across many sub-directories. The logic we implement to solve this can be viewed in Figure 4.21.

Also, in order to prevent all output from going to the console together in a giant block of data, we make the application store each output in a file solely created for the method it is analysing. For large projects, the output basically follows the project's tree. This makes the analysis much easier, and only errors and debug information (if specified) are printed to the console.

Since the application deals with multiple types of TRs, we add some input flags

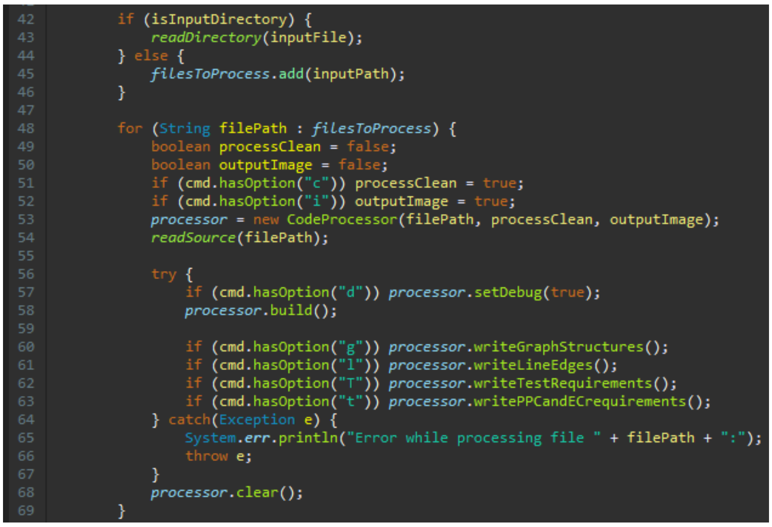
Figure 4.22 – Newly added inputs.



```java
        Options options = new Options();
        options.addOption("d", false, "Print debug output"); // does not have a value
        options.addOption("g", false, "Print graph structures");
        options.addOption("l", false, "Print line flows");
        options.addOption("t", false, "Print PPC and EC test requirements");
        options.addOption("T", false, "Print complete test requirements");
        options.addOption("i", false, "Output control flow graph PNG");
        options.addOption("c", false, "Process clean code");
```

Source: author

Figure 4.23 – Final version of the input processing.



```java
        if (isInputDirectory) {
            readDirectory(inputFile);
        } else {
            filesToProcess.add(inputPath);
        }

        for (String filePath : filesToProcess) {
            boolean processClean = false;
            boolean outputImage = false;
            if (cmd.hasOption("c")) processClean = true;
            if (cmd.hasOption("i")) outputImage = true;
            processor = new CodeProcessor(filePath, processClean, outputImage);
            readSource(filePath);

            try {
                if (cmd.hasOption("d")) processor.setDebug(true);
                processor.build();

                if (cmd.hasOption("g")) processor.writeGraphStructures();
                if (cmd.hasOption("l")) processor.writeLineEdges();
                if (cmd.hasOption("T")) processor.writeTestRequirements();
                if (cmd.hasOption("t")) processor.writePPCandECrequirements();
            } catch(Exception e) {
                System.err.println("Error while processing file " + filePath + ":");
                throw e;
            }
            processor.clear();
        }
```

Source: author

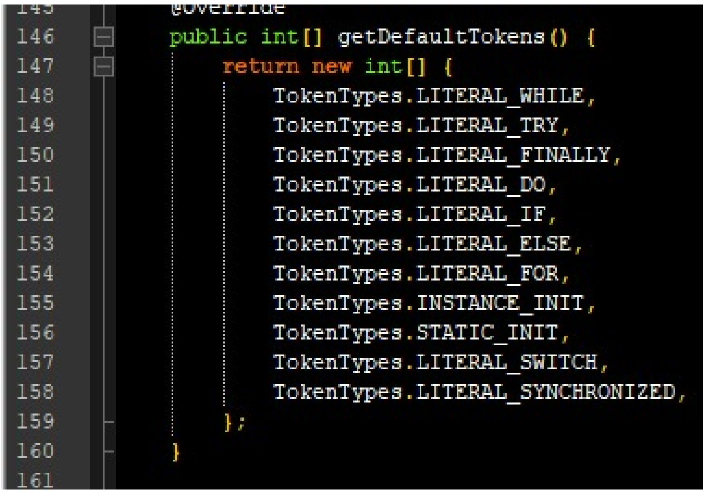
for the user to decide which of them they want to obtain. The flag `-T`, when specified, will print a file for each processed method containing the original application output, with node, edge and prime-path requirements. The flag `-t` makes the application output only edge and prime-path requirements, each in different files. The flag `-l` prints the edge coverage with a more well-formatted output.

For debugging purposes and analysis of the graphs, a `-g` flag is added that, when set, will make the application output one file for each processed method containing the detailed structure of the graph, with information on its nodes and edges and the source code they represent.

Another optional flag was added, `-c`, that when set makes the tool generate the test requirements based on the clean source code (i.e. it will not try to look into the original source code line indexes when outputting the requirements).

The newly added inputs and the final state of the file processing can be seen in Figures 4.22 and 4.23.

Figure 4.24 – Array initialization is one of the things that the tool does not yet support.



```
    @Override
    public int[] getDefaultTokens() {
        return new int[] {
            TokenTypes.LITERAL_WHILE,
            TokenTypes.LITERAL_TRY,
            TokenTypes.LITERAL_FINALLY,
            TokenTypes.LITERAL_DO,
            TokenTypes.LITERAL_IF,
            TokenTypes.LITERAL_ELSE,
            TokenTypes.LITERAL_FOR,
            TokenTypes.INSTANCE_INIT,
            TokenTypes.STATIC_INIT,
            TokenTypes.LITERAL_SWITCH,
            TokenTypes.LITERAL_SYNCHRONIZED,
        };
    }
```

Source: author

4.5 Final state of the tool

After all the changes and fixes we have implemented, although the tool is now able to process most Java projects out there, it still contains some more specific limitations, which should not be a big deal in a general case. We will list them here in order for a future developer to work on it should they want to.

The main limitation is that ternary operators are not processed as `if-else` instructions, but as single-line instructions. They do not cause errors and do not prevent the application from finishing, but the test requirements listed may not be completely correct since it will not derive two flows for the instructions - i.e. it will not treat the instruction as an `if-else` statement.

Complex objects and object initialization is also not fully supported. Some library classes have objects being invoked through return instructions. This sometimes causes the application to get lost, providing wrong information on the output, or even failing with an error.

Array and matrix initialization, as shown in 4.24, is also not fully supported due to the bracket syntax. The clean up script does not understand it pretty well and it may lead to failures.

`try-catch` blocks are supported, but one may argue if the way the application deals with them is correct or not, since the flow is more abstract in their cases.

Table 4.1 – Adaptations made to the tool

Description	*Effort*
Refactor of clean up procedure	3
Code processing improvement	7
`for-each` support	6
`try-catch` support	5
Improvement to comments support	6
Tracking of source code line ids	10
Support for code with annotations	1
`switch-case` structure improvement	6
Support for continue and break statements	5
One-liner `if` and loops support	6
Character query system improvement	8
Generation of line version of CFG	9
Support for multiple files and directories	6
Input & output modifications	5

Source: author

Table 4.2 – Current limitations of the tool

Description	*Effort*
Ternary operator support	10
Lambda expression support	10
Array initialization support	9
Matrix & object initialization support	10

Source: author

Lastly, Java 8 allows for the use of lambda expressions[4], and these are also not supported, with great chance of causing the tool to fail.

These listed limitations have a small impact in the reliability of our results, since roughly most of the projects we are addressing make few to no use of the instructions that cause them. Furthermore, the methods we are selecting for analysis in Chapter 5 do not use any of the the aforementioned instructions.

Table 4.1 and Table 4.2 display, respectively, a summary of all modifications that were made and limitations that were left unchanged, correlated with the effort to work on them, where 0 represents no effort at all, and 10 represents very high effort.

[4] See https://www.w3schools.com/java/java_lambda.asp.

5 ANALYSIS OF TEST SUITES

We should recall the problems listed in Chapter 1: there is an apparent redundancy in the test suites of the analyzed Java projects, and we want to know why they happen, if they are related to an automatically generated test suite, and what is the impact of this redundancy. Now that we have the PPC generation tool, we can proceed with our methodology and analyse the test suites.

In order to solve the aforementioned questions, we should select a set of methods from the projects to work with. We want to consider methods mainly with different natures, sizes and complexities, so that we can have a consistent analysis. We also want to give a special attention to methods with loops, since they are more interesting to analyze due to the complex paths derived from the loops.

Previous research on the TPs from the projects show that the apparent redundancy shows up mainly in methods of *lang3* and *math4* libraries (two libraries from *org.apache.commons*). The former contains many functions related to language and string manipulation, while the latter deals with mathematical problems. Thus, this pair works well for the research as both libraries come from issues of different natures (language versus math). Moreover, when taking a look into the methods where a problem of redundant TPs was identified, it is possible to see many different complexities and structures.

The methods observed and their cyclomatic complexities (CC) are shown in Tables 5.1 and 5.2.

From these tables, we select fifteen methods to study, since we want to focus more on a depth approach rather than a breadth approach. Again, we look for small groups of 2-3 methods, each with different characteristics or natures. We also want to look for methods with loops, since they tend to be more complex and interesting to study.

The selected methods for this work can be seen in Table 5.3. Now, let us take a

Table 5.1 – Methods with identified TP redundancy - *math4*

Method	*CC*
math4.dfp.Dfp.divide(Dfp)	33
math4.dfp.Dfp.divide(int)	9
math4.dfp.Dfp.intLog10()	4
math4.dfp.Dfp.isZero()	1
math4.dfp.DfpMath.pow(Dfp,Dfp)	28
math4.genetics.ListPopulation.getFittestChromosome()	3
math4.geometry.euclidean.threed.SphericalCoordinatesTest.testHessian()	8
math4.stat.regression.RegressionResults.getCovarianceOfParameters(int,int)	4

Source: Silva, Keslley

Table 5.2 – Methods with identified TP redundancy - *lang3*

Method	*CC*
lang3.Conversion.binaryBeMsb0ToHexDigit(boolean[],int)	17
lang3.Conversion.binaryToHexDigit(boolean[],int)	17
lang3.StringUtils.getLevenshteinDistance(CharSequence,CharSequence,int)	17
lang3.time.DurationFormatUtils.formatPeriod(long,long,String,boolean,TimeZone)	20
lang3.Conversion.longToIntArray(long,int,int[],int,int)	4
lang3.StringUtils.compare(String,String,boolean)	6
lang3.StringUtils.compareIgnoreCase(String,String,boolean)	6
lang3.StringUtils.containsOnly(CharSequence,char...)	4
lang3.StringUtils.indexOfAnyBut(CharSequence,CharSequence)	6
lang3.StringUtils.isAllUpperCase(CharSequence)	4
lang3.StringUtils.isAsciiPrintable(CharSequence)	4
lang3.StringUtils.lastIndexOfIgnoreCase(CharSequence,CharSequence,int)	7
lang3.StringUtils.startsWithAny(CharSequence,CharSequence...)	4
lang3.StringUtils.stripStart(String,String)	6
lang3.LocaleUtils.toLocale(String)	11
lang3.StringUtils.containsAny(CharSequence,char...)	8
lang3.StringUtils.indexOfDifference(CharSequence...)	11
lang3.text.WordUtils.wrap(String,int,String,boolean,String)	14
lang3.time.DurationFormatUtils.formatDurationWords(long,boolean,boolean)	10
lang3.StringUtils.uncapitalize(String)	4
lang3.AnnotationUtils.isValidAnnotationMemberType(Class<?>)	3
lang3.StringUtils.center(String,int,char)	3
lang3.Conversion.byteArrayToUuid(byte[],int)	2
lang3.Conversion.hexDigitToInt(char)	2
lang3.Conversion.intToHexDigit(int)	2
lang3.StringUtils.stripAccents(String)	2

Source: Silva, Keslley

Table 5.3 – Methods selected for the analysis

Method	*CC*
lang3.StringUtils.getLevenshteinDistance(CharSequence,CharSequence,int)	17
lang3.time.DurationFormatUtils.formatPeriod(long,long,String,boolean,TimeZone)	20
lang3.Conversion.longToIntArray(long,int,int[],int,int)	4
lang3.StringUtils.isAllUpperCase(CharSequence)	4
lang3.StringUtils.startsWithAny(CharSequence,CharSequence...)	4
lang3.StringUtils.containsAny(CharSequence,char...)	8
lang3.StringUtils.indexOfDifference(CharSequence...)	11
lang3.text.WordUtils.wrap(String,int,String,boolean,String)	14
lang3.StringUtils.uncapitalize(String)	4
math4.dfp.Dfp.divide(Dfp)	33
math4.dfp.Dfp.divide(int)	9
math4.dfp.Dfp.intLog10()	4
math4.dfp.Dfp.isZero()	1
math4.dfp.DfpMath.pow(Dfp,Dfp)	28
math4.genetics.ListPopulation.getFittestChromosome()	3

Source: Silva, Keslley and author

look at the process of obtaining the required data for the analysis.

5.1 Obtaining data

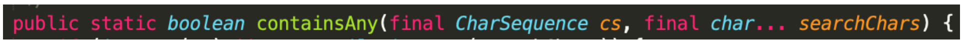
To recap, we need to compare the project's test suite properties with two other test suites: one that is created based only in the method's signatures and descriptions, and another that is generated through an automatic test generation tool. Doing so will let us know whether the project's test suite was generated automatically or if it is manually developed by a person. We also need to understand if this is related to the redundant TPs and, if not, why they occur.

More specifically, we will perform the following steps for each of the selected methods: a. manually create a test suite; b. generate test suites using a selected test generation tool; c. compute the TPs of each suite; d. compute the TRs of each MUT. We will now describe in detail each step and the difficulties that showed up in each one of them.

5.1.1 Blind creation of new test suites

In order to create a new suite, we must analyze the signatures of the MUTs and, eventually, their description (in case the name is not self-explanatory). We want to avoid

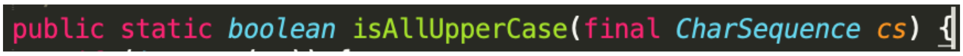
Figure 5.1 – Signature of method `containsAny`.



```java
public static boolean containsAny(final CharSequence cs, final char... searchChars) {
```

Source: author

Figure 5.2 – Signature of method `isAllUpperCase`.



```java
public static boolean isAllUpperCase(final CharSequence cs) {
```

Source: author

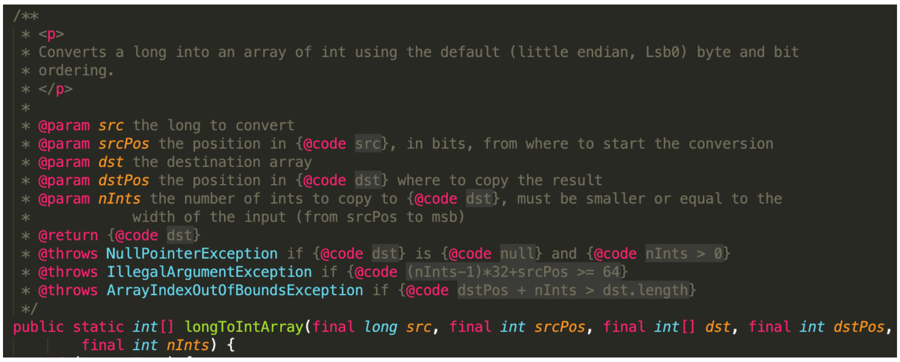
looking into the method's logic and the project's suites, so as not to build biased test cases, although this will be necessary for some of the MUTs as we will see in a moment.

For a couple of methods, developing test cases is easy, because they are very clear on what they do, which inputs they receive an which outputs they return. Such is the case of `containsAny` and `isAllUpperCase`, whose signatures are displayed in Figures 5.1 and 5.2. It is clear to see that the first will check if the first parameter, a string, contains any of the subsequent characters, and the second will check if the string passed contains only uppercase characters.

For another set of methods, understanding them in order to write test cases is a bit harder. For instance, `longToIntArray` and `formatPeriod` seem easy to understand at first glance, but the amount of parameters that are passed may cause a bit of confusion, making us check at least their descriptions (displayed in Figures 5.3 and 5.4). In the case of `formatPeriod`, it is also needed to analyze the test cases to see how to use the format parameter.

Moreover, all methods from the *Dfp* class are difficult to understand by looking at

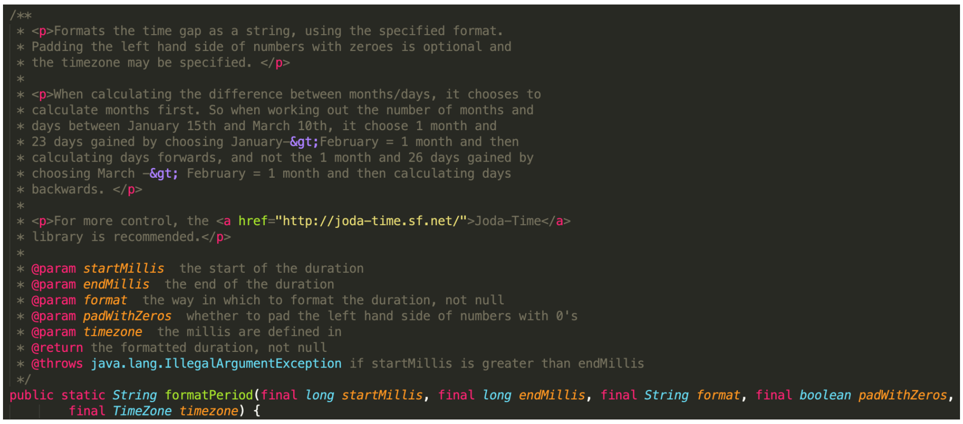
Figure 5.3 – Signature and description of method `longToIntArray`.



```java
/**
 * <p>
 * Converts a long into an array of int using the default (little endian, Lsb0) byte and bit
 * ordering.
 * </p>
 *
 * @param src the long to convert
 * @param srcPos the position in {@code src}, in bits, from where to start the conversion
 * @param dst the destination array
 * @param dstPos the position in {@code dst} where to copy the result
 * @param nInts the number of ints to copy to {@code dst}, must be smaller or equal to the
 *            width of the input (from srcPos to msb)
 * @return {@code dst}
 * @throws NullPointerException if {@code dst} is {@code null} and {@code nInts > 0}
 * @throws IllegalArgumentException if {@code (nInts-1)*32+srcPos >= 64}
 * @throws ArrayIndexOutOfBoundsException if {@code dstPos + nInts > dst.length}
 */
public static int[] longToIntArray(final long src, final int srcPos, final int[] dst, final int dstPos,
        final int nInts) {
```

Source: author

Figure 5.4 – Signature and description of method `formatPeriod`.



```
/**
 * <p>Formats the time gap as a string, using the specified format.
 * Padding the left hand side of numbers with zeroes is optional and
 * the timezone may be specified. </p>
 *
 * <p>When calculating the difference between months/days, it chooses to
 * calculate months first. So when working out the number of months and
 * days between January 15th and March 10th, it choose 1 month and
 * 23 days gained by choosing January-&gt;February = 1 month and then
 * calculating days forwards, and not the 1 month and 26 days gained by
 * choosing March -&gt; February = 1 month and then calculating days
 * backwards. </p>
 *
 * <p>For more control, the <a href="http://joda-time.sf.net/">Joda-Time</a>
 * library is recommended.</p>
 *
 * @param startMillis  the start of the duration
 * @param endMillis  the end of the duration
 * @param format  the way in which to format the duration, not null
 * @param padWithZeros  whether to pad the left hand side of numbers with 0's
 * @param timezone  the millis are defined in
 * @return the formatted duration, not null
 * @throws java.lang.IllegalArgumentException if startMillis is greater than endMillis
 */
public static String formatPeriod(final long startMillis, final long endMillis, final String format, final boolean padWithZeros,
        final TimeZone timezone) {
```

Source: author

the signatures alone. It is a complex class with attributes, representing a decimal floating point[1]. This makes us have to study at least the project's test suite before creating a new one from scratch, since it requires specific uses of the constructor and test values. See Figure 5.5. The same applies to *ListPopulation*.

After the MUT has been significantly understood, we create a number of test cases, trying to cover all the situations we believe the method should cover. We make an effort to always include guard clauses in our process (for example, if the parameter is a string we should always test for null and empty strings). This is the case for most of the methods (although some have exceptions[2]).

After the manually developed suite has been created, we can look into the project's test suite for a comparison. For methods with many parameters and/or more specific uses (such as `longToIntArray` and `formatPeriod`), we may take a look in the project's suite before finishing ours, but we try as best as we can not to study the test cases and their individual purposes with the method.

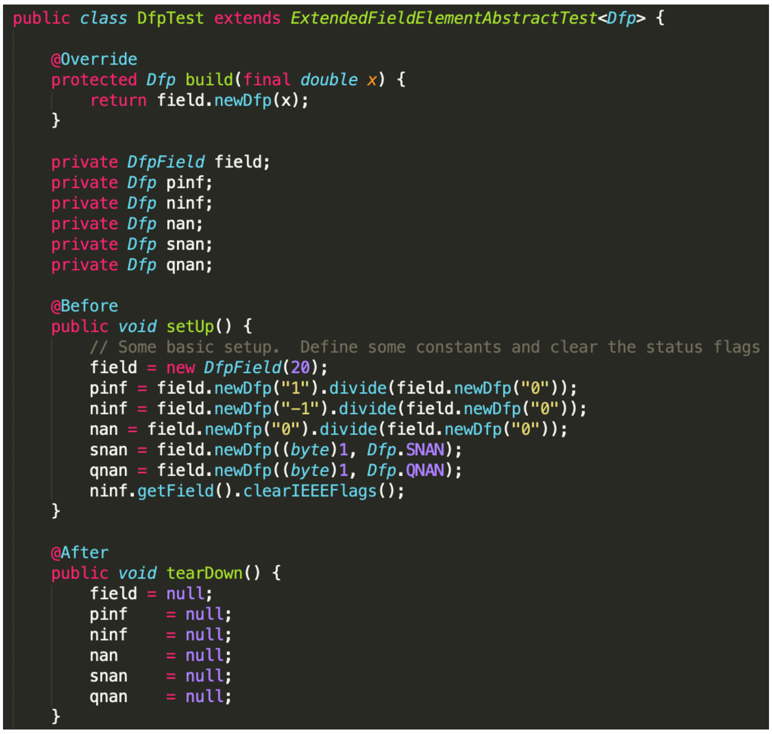
Some examples of suites developed based on the methods signatures can be seen in Figures 5.6, 5.7 and 5.8[3].

[1] See https://en.wikipedia.org/wiki/Decimal_floating_point.

[2] `longToIntArray` expects the passed array to be non-null.

[3] Since the TP generation tool only computes the TP of the test case, we do not need to necessarily use an assertion, like the project's suite does.

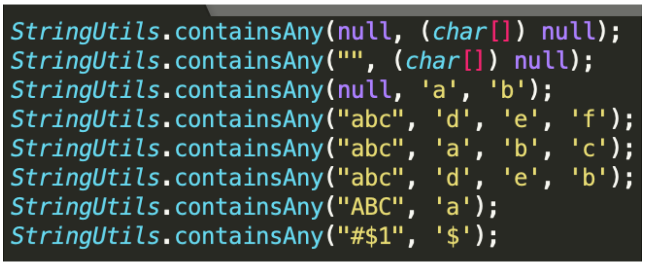
Figure 5.5 – The tests for *Dfp* class require some setup before being run.



```
public class DfpTest extends ExtendedFieldElementAbstractTest<Dfp> {

    @Override
    protected Dfp build(final double x) {
        return field.newDfp(x);
    }

    private DfpField field;
    private Dfp pinf;
    private Dfp ninf;
    private Dfp nan;
    private Dfp snan;
    private Dfp qnan;

    @Before
    public void setUp() {
        // Some basic setup.  Define some constants and clear the status flags
        field = new DfpField(20);
        pinf = field.newDfp("1").divide(field.newDfp("0"));
        ninf = field.newDfp("-1").divide(field.newDfp("0"));
        nan = field.newDfp("0").divide(field.newDfp("0"));
        snan = field.newDfp((byte)1, Dfp.SNAN);
        qnan = field.newDfp((byte)1, Dfp.QNAN);
        ninf.getField().clearIEEEFlags();
    }

    @After
    public void tearDown() {
        field = null;
        pinf    = null;
        ninf    = null;
        nan     = null;
        snan    = null;
        qnan    = null;
    }
```

Source: author

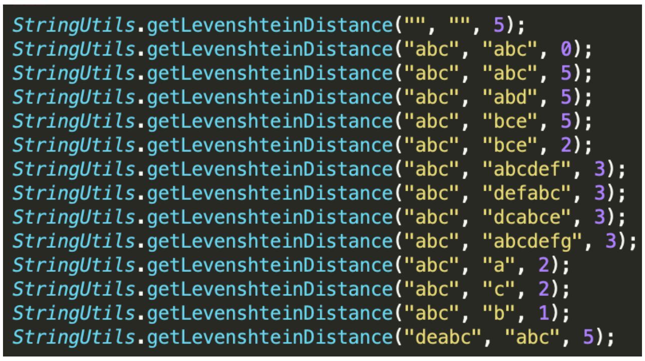
Figure 5.6 – Suite created for method `containsAny`.



```
StringUtils.containsAny(null, (char[]) null);
StringUtils.containsAny("", (char[]) null);
StringUtils.containsAny(null, 'a', 'b');
StringUtils.containsAny("abc", 'd', 'e', 'f');
StringUtils.containsAny("abc", 'a', 'b', 'c');
StringUtils.containsAny("abc", 'd', 'e', 'b');
StringUtils.containsAny("ABC", 'a');
StringUtils.containsAny("#$1", '$');
```

Source: author

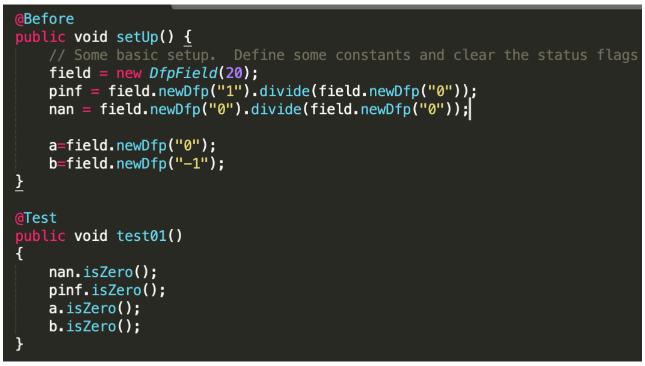
Figure 5.7 – Suite created for method `getLevenshteinDistance`.



```
StringUtils.getLevenshteinDistance("", "", 5);
StringUtils.getLevenshteinDistance("abc", "abc", 0);
StringUtils.getLevenshteinDistance("abc", "abc", 5);
StringUtils.getLevenshteinDistance("abc", "abd", 5);
StringUtils.getLevenshteinDistance("abc", "bce", 5);
StringUtils.getLevenshteinDistance("abc", "bce", 2);
StringUtils.getLevenshteinDistance("abc", "abcdef", 3);
StringUtils.getLevenshteinDistance("abc", "defabc", 3);
StringUtils.getLevenshteinDistance("abc", "dcabce", 3);
StringUtils.getLevenshteinDistance("abc", "abcdefg", 3);
StringUtils.getLevenshteinDistance("abc", "a", 2);
StringUtils.getLevenshteinDistance("abc", "c", 2);
StringUtils.getLevenshteinDistance("abc", "b", 1);
StringUtils.getLevenshteinDistance("deabc", "abc", 5);
```

Source: author

Figure 5.8 – Suite created for method `isZero`.



```
@Before
public void setUp() {
    // Some basic setup.  Define some constants and clear the status flags
    field = new DfpField(20);
    pinf = field.newDfp("1").divide(field.newDfp("0"));
    nan = field.newDfp("0").divide(field.newDfp("0"));

    a=field.newDfp("0");
    b=field.newDfp("-1");
}

@Test
public void test01()
{
    nan.isZero();
    pinf.isZero();
    a.isZero();
    b.isZero();
}
```

Source: author

5.1.2 Generation of test suites using EvoSuite

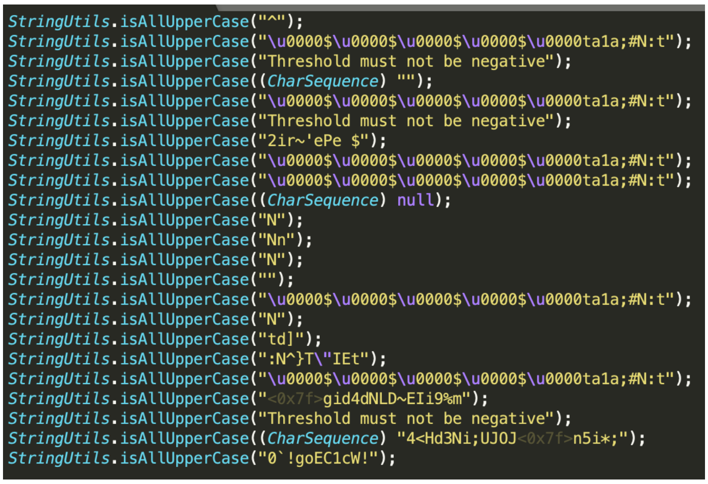
In parallel with the creation of the test suite, we will use EvoSuite to generate another set of test suites. EvoSuite is an open-source test generation tool that is widely used by the Java community. It is also easy to use, containing a command line integration which we can use to generate the tests we want. A guide to the tool can be found in their GitHub (2020).

The generation of the test suites may take a while since they run at class level, and some classes are quite large (*StringUtils*, for instance) and will have the tool run for about 1 hour before finishing.

The output of the tool is quite messy, as it does not create one test method for each MUT, but rather creates many methods, each executing a group of MUTs. In order to later generate the TPs of the suite we must find and extract the parts of the suite that test the desired MUT, and afterwards merge them in one entire method that we will use to generate the TPs. One example can be seen in Figure 5.9. This is probably the part of the generation that takes the most time since the generated suite is sometimes very big and the calls to test the methods may use multiple variables, so we cannot just copy them separately. This can be seen in Figure 5.10.

For more complex classes (mainly the ones belonging to *math4*), the generated suite is very poor, probably due to the lack of understanding of the classes by the tool. For instance, the suite generated for *ListPopulation* contains only 25 methods and 343 lines, and the only calls to `getFittestChromosome` test an exception, so the MUT's purpose is never really tested. Figure 5.11 displays part of the suite generated for *ListPopulation*. Besides this, the suite generated for *Dfp* class contains only one call to one

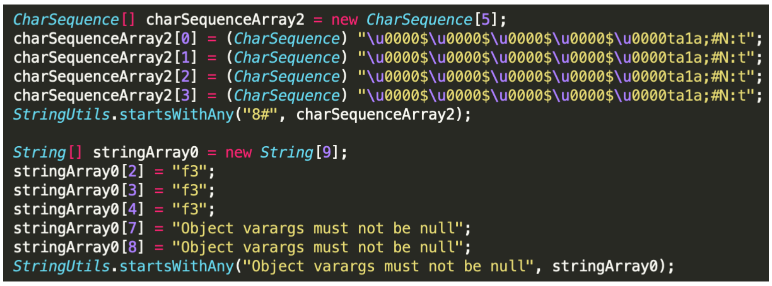
Figure 5.9 – Suite generated for method `isAllUpperCase`.



```
StringUtils.isAllUpperCase("^");
StringUtils.isAllUpperCase("\u0000$\u0000$\u0000$\u0000$\u0000ta1a;#N:t");
StringUtils.isAllUpperCase("Threshold must not be negative");
StringUtils.isAllUpperCase((CharSequence) "");
StringUtils.isAllUpperCase("\u0000$\u0000$\u0000$\u0000$\u0000ta1a;#N:t");
StringUtils.isAllUpperCase("Threshold must not be negative");
StringUtils.isAllUpperCase("2ir~'ePe $");
StringUtils.isAllUpperCase("\u0000$\u0000$\u0000$\u0000$\u0000ta1a;#N:t");
StringUtils.isAllUpperCase("\u0000$\u0000$\u0000$\u0000$\u0000ta1a;#N:t");
StringUtils.isAllUpperCase((CharSequence) null);
StringUtils.isAllUpperCase("N");
StringUtils.isAllUpperCase("Nn");
StringUtils.isAllUpperCase("N");
StringUtils.isAllUpperCase("");
StringUtils.isAllUpperCase("\u0000$\u0000$\u0000$\u0000$\u0000ta1a;#N:t");
StringUtils.isAllUpperCase("N");
StringUtils.isAllUpperCase("td]");
StringUtils.isAllUpperCase(":N^}T\"IEt");
StringUtils.isAllUpperCase("\u0000$\u0000$\u0000$\u0000$\u0000ta1a;#N:t");
StringUtils.isAllUpperCase("<0x7f>gid4dNLD~EIi9%m");
StringUtils.isAllUpperCase("Threshold must not be negative");
StringUtils.isAllUpperCase((CharSequence) "4<Hd3Ni;UJOJ<0x7f>n5i*;");
StringUtils.isAllUpperCase("0`!goEC1cW!");
```

Source: author

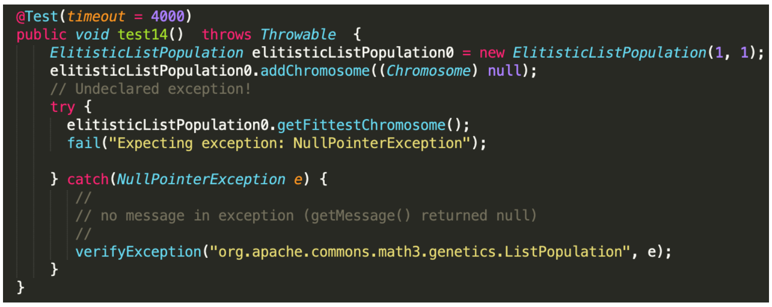
Figure 5.10 – Test cases generated for method `startsWithAny`. Before executing the test cases, a variable is created and manipulated in order to use it as parameter.



```
CharSequence[] charSequenceArray2 = new CharSequence[5];
charSequenceArray2[0] = (CharSequence) "\u0000$\u0000$\u0000$\u0000$\u0000ta1a;#N:t";
charSequenceArray2[1] = (CharSequence) "\u0000$\u0000$\u0000$\u0000$\u0000ta1a;#N:t";
charSequenceArray2[2] = (CharSequence) "\u0000$\u0000$\u0000$\u0000$\u0000ta1a;#N:t";
charSequenceArray2[3] = (CharSequence) "\u0000$\u0000$\u0000$\u0000$\u0000ta1a;#N:t";
StringUtils.startsWithAny("8#", charSequenceArray2);

String[] stringArray0 = new String[9];
stringArray0[2] = "f3";
stringArray0[3] = "f3";
stringArray0[4] = "f3";
stringArray0[7] = "Object varargs must not be null";
stringArray0[8] = "Object varargs must not be null";
StringUtils.startsWithAny("Object varargs must not be null", stringArray0);
```

Source: author

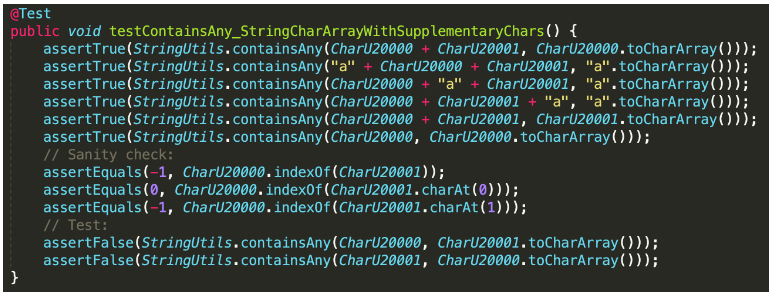
Figure 5.11 – Test case generated for the method `getFittestChromosome`



```java
@Test(timeout = 4000)
public void test14()  throws Throwable  {
    ElitisticListPopulation elitisticListPopulation0 = new ElitisticListPopulation(1, 1);
    elitisticListPopulation0.addChromosome((Chromosome) null);
    // Undeclared exception!
    try {
      elitisticListPopulation0.getFittestChromosome();
      fail("Expecting exception: NullPointerException");

    } catch(NullPointerException e) {
       //
       // no message in exception (getMessage() returned null)
       //
       verifyException("org.apache.commons.math3.genetics.ListPopulation", e);
    }
}
```

Source: author

Figure 5.12 – Part of project’s test suite for method `containsAny`, which tests for high surrogate characters.



```java
@Test
public void testContainsAny_StringCharArrayWithSupplementaryChars() {
    assertTrue(StringUtils.containsAny(CharU20000 + CharU20001, CharU20000.toCharArray()));
    assertTrue(StringUtils.containsAny("a" + CharU20000 + CharU20001, "a".toCharArray()));
    assertTrue(StringUtils.containsAny(CharU20000 + "a" + CharU20001, "a".toCharArray()));
    assertTrue(StringUtils.containsAny(CharU20000 + CharU20001 + "a", "a".toCharArray()));
    assertTrue(StringUtils.containsAny(CharU20000 + CharU20001, CharU20001.toCharArray()));
    assertTrue(StringUtils.containsAny(CharU20000, CharU20000.toCharArray()));
    // Sanity check:
    assertEquals(-1, CharU20000.indexOf(CharU20001));
    assertEquals(0, CharU20000.indexOf(CharU20001.charAt(0)));
    assertEquals(-1, CharU20000.indexOf(CharU20001.charAt(1)));
    // Test:
    assertFalse(StringUtils.containsAny(CharU20000, CharU20001.toCharArray()));
    assertFalse(StringUtils.containsAny(CharU20001, CharU20000.toCharArray()));
}
```

Source: author

of the class’ methods, the rest testing only the class initialization.

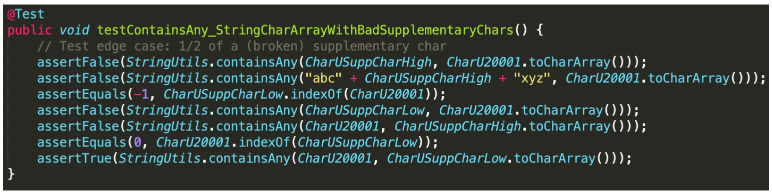
5.1.3 Generation of TPs

After all suites have been gathered, we should generate their TPs. For this, we use a tool called *Execution Flow*, as mentioned in Chapter 3.

In this process it was observed that in the project’s suite, for some methods, their test cases were divided into more than one test method. For instance, `containsAny` has test cases split over three methods (shown in Figures 5.12, 5.13 and 5.14) - one for high surrogate characters, one for bad high surrogate characters, and one for normal characters.

Furthermore, `wrap` has multiple overloads due to optional parameters, having the simpler overloads call the more complex ones, as shown in Figure 5.15. This makes any call to the simpler overloads go through all the method’s logic. However, our tool to generate the TPs will only generate the TP for the overload being called, so if we are to obtain the TPs of all the test cases for the `wrap` method, we need to change the suite so

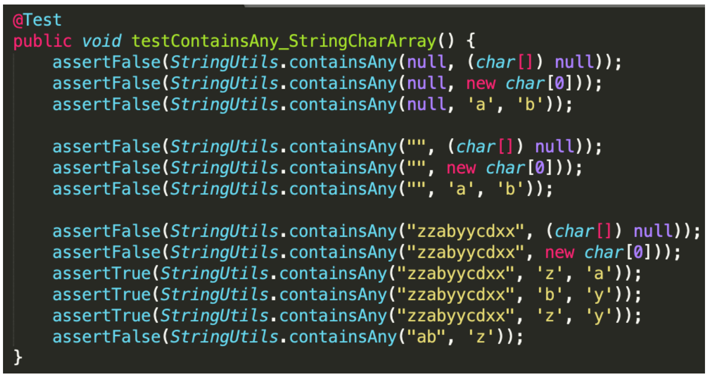
Figure 5.13 – Part of project’s test suite for method `containsAny`, which tests for bad high surrogate characters.



```java
@Test
public void testContainsAny_StringCharArrayWithBadSupplementaryChars() {
    // Test edge case: 1/2 of a (broken) supplementary char
    assertFalse(StringUtils.containsAny(CharUSuppCharHigh, CharU20001.toCharArray()));
    assertFalse(StringUtils.containsAny("abc" + CharUSuppCharHigh + "xyz", CharU20001.toCharArray()));
    assertEquals(-1, CharUSuppCharLow.indexOf(CharU20001));
    assertFalse(StringUtils.containsAny(CharUSuppCharLow, CharU20001.toCharArray()));
    assertFalse(StringUtils.containsAny(CharU20001, CharUSuppCharHigh.toCharArray()));
    assertEquals(0, CharU20001.indexOf(CharUSuppCharLow));
    assertTrue(StringUtils.containsAny(CharU20001, CharUSuppCharLow.toCharArray()));
}
```

Source: author

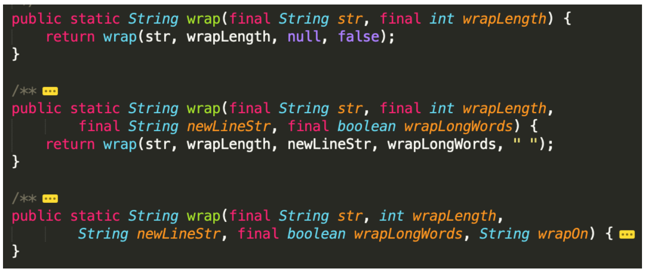
Figure 5.14 – Part of project’s test suite for method `containsAny`, which tests for normal characters.



```java
@Test
public void testContainsAny_StringCharArray() {
    assertFalse(StringUtils.containsAny(null, (char[]) null));
    assertFalse(StringUtils.containsAny(null, new char[0]));
    assertFalse(StringUtils.containsAny(null, 'a', 'b'));

    assertFalse(StringUtils.containsAny("", (char[]) null));
    assertFalse(StringUtils.containsAny("", new char[0]));
    assertFalse(StringUtils.containsAny("", 'a', 'b'));

    assertFalse(StringUtils.containsAny("zzabyycdxx", (char[]) null));
    assertFalse(StringUtils.containsAny("zzabyycdxx", new char[0]));
    assertTrue(StringUtils.containsAny("zzabyycdxx", 'z', 'a'));
    assertTrue(StringUtils.containsAny("zzabyycdxx", 'b', 'y'));
    assertTrue(StringUtils.containsAny("zzabyycdxx", 'z', 'y'));
    assertFalse(StringUtils.containsAny("ab", 'z'));
}
```

Source: author

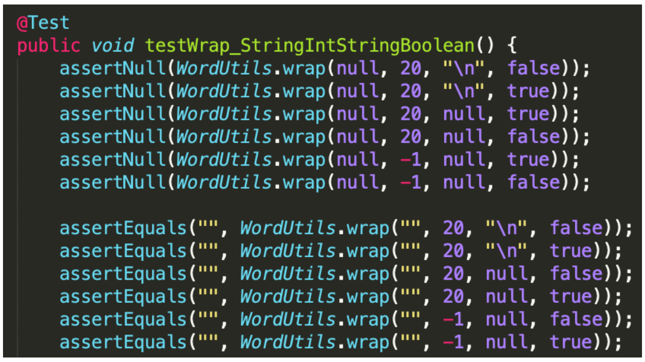
Figure 5.15 – The `wrap` method contains three overloads due to optional parameters.



```
public static String wrap(final String str, final int wrapLength) {
    return wrap(str, wrapLength, null, false);
}

/** ...
public static String wrap(final String str, final int wrapLength,
        final String newLineStr, final boolean wrapLongWords) {
    return wrap(str, wrapLength, newLineStr, wrapLongWords, " ");
}

/** ...
public static String wrap(final String str, int wrapLength,
        String newLineStr, final boolean wrapLongWords, String wrapOn) { ...
}
```

Source: author

Figure 5.16 – Project's suite for method `wrap` using one of the simpler overloads.



```
@Test
public void testWrap_StringIntStringBoolean() {
    assertNull(WordUtils.wrap(null, 20, "\n", false));
    assertNull(WordUtils.wrap(null, 20, "\n", true));
    assertNull(WordUtils.wrap(null, 20, null, true));
    assertNull(WordUtils.wrap(null, 20, null, false));
    assertNull(WordUtils.wrap(null, -1, null, true));
    assertNull(WordUtils.wrap(null, -1, null, false));

    assertEquals("", WordUtils.wrap("", 20, "\n", false));
    assertEquals("", WordUtils.wrap("", 20, "\n", true));
    assertEquals("", WordUtils.wrap("", 20, null, false));
    assertEquals("", WordUtils.wrap("", 20, null, true));
    assertEquals("", WordUtils.wrap("", -1, null, false));
    assertEquals("", WordUtils.wrap("", -1, null, true));
```

Source: author

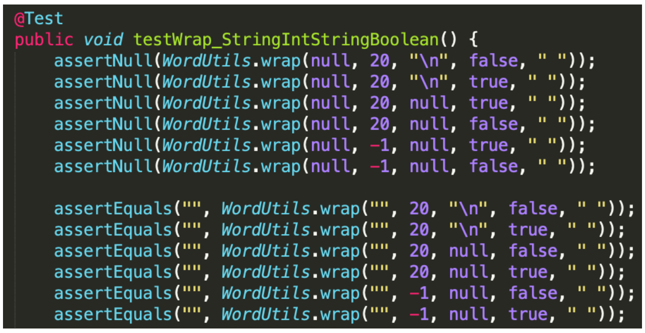
that it will always call the overload to be tested. See Figures 5.16 and 5.17. The same thing happens to `formatPeriod`.

After we fix these issues in the project's suite, we are good to go. We may create a temporary test class in which we will run the TP generation tool, and place the test cases there each time we change the MUT.

5.1.4 Generation of PPCs

Finally, for each of the selected methods we must use the tool adapted in Chapter 4 to generate the prime path TRs. This allows us to check which TPs cover which PPs, and completes the set of data we need for our analysis.

Figure 5.17 – Project's suite for method `wrap` after changing the suite to call the overload with five parameters.



```
@Test
public void testWrap_StringIntStringBoolean() {
    assertNull(WordUtils.wrap(null, 20, "\n", false, " "));
    assertNull(WordUtils.wrap(null, 20, "\n", true, " "));
    assertNull(WordUtils.wrap(null, 20, null, true, " "));
    assertNull(WordUtils.wrap(null, 20, null, false, " "));
    assertNull(WordUtils.wrap(null, -1, null, true, " "));
    assertNull(WordUtils.wrap(null, -1, null, false, " "));

    assertEquals("", WordUtils.wrap("", 20, "\n", false, " "));
    assertEquals("", WordUtils.wrap("", 20, "\n", true, " "));
    assertEquals("", WordUtils.wrap("", 20, null, false, " "));
    assertEquals("", WordUtils.wrap("", 20, null, true, " "));
    assertEquals("", WordUtils.wrap("", -1, null, false, " "));
    assertEquals("", WordUtils.wrap("", -1, null, true, " "));
```

Source: author

5.2 Data comparison

Table 5.4 displays the number of TRs to satisfy PPC on the selected methods, and Tables 5.5, 5.6 and 5.7 display coverage properties for the project's suites, the created suites and the automatically generated suites respectively. These coverage properties consist of:

(a) number of test cases (number of times the MUT is called in the suite).

(b) percentage of the PPC that was not covered, i.e. uncovered prime paths (UPPs).

(c) number of redundant TPs (RTPs) - not necessarily redundant test cases because the redundant TP might come from multiple test cases that test different classes of input values, which we will explain in a moment.

Property (a) is obtained by counting the number of TPs found for each suite (since the number of TPs is the number of test cases that were executed), while properties (b) and (c) were obtained using two simple Python scripts, also developed in this work, that operate over the PPC and TP data.

With these data, it is possible to draw some conclusions. First, we can see that the properties of the project's suite are closer to the properties of the manually created suite than to the generated one. The generated suite contains a lot of redundant TPs, and considerably more test cases than the other ones. It is also observed that the generated suite does not cover the PPC criteria efficiently. Actually, if we manually take a look at the test cases generated by EvoSuite, we can see that the inputs are quite random (see Figure 5.9), very different from the inputs in the project's suite. Moreover, as mentioned

Table 5.4 – Number of PPCs for each analyzed method

Method	*# PPCs*
getLevenshteinDistance(CharSequence,CharSequence,int)	95
formatPeriod(long,long,String,boolean,TimeZone)	108
longToIntArray(long,int,int[],int,int)	9
isAllUpperCase(CharSequence)	9
startsWithAny(CharSequence,CharSequence...)	7
containsAny(CharSequence,char...)	35
indexOfDifference(CharSequence...)	87
wrap(String,int,String,boolean,String)	457
uncapitalize(String)	9
divide(Dfp)	7999
divide(int)	26
intLog10()	4
isZero()	2
pow(Dfp,Dfp)	47
getFittestChromosome()	9

Source: author

Table 5.5 – Properties of project's suite for selected methods

Method	*N^o Tests*	*% UPPCs*	*N^o RTPs*
getLevenshteinDistance(CharSequence,CharSequence,int)	38	62,1	14
formatPeriod(long,long,String,boolean,TimeZone)	35	95,4	25
longToIntArray(long,int,int[],int,int)	14	22,2	11
isAllUpperCase(CharSequence)	10	11,1	5
startsWithAny(CharSequence,CharSequence...)	12	14,3	6
containsAny(CharSequence,char...)	25	51,4	12
indexOfDifference(CharSequence...)	17	65,5	4
wrap(String,int,String,boolean,String)	46	81,8	20
uncapitalize(String)	10	0	4
divide(Dfp)	29	99,3	13
divide(int)	16	42,3	8
intLog10()	16	0	12
isZero()	11	0	9
pow(Dfp,Dfp)	90	36,2	60
getFittestChromosome()	1	27,3	0

Source: author

Table 5.6 – Properties of created suite for selected methods

Method	*N^o Tests*	*% UPPCs*	*N^o RTPs*
getLevenshteinDistance(CharSequence,CharSequence,int)	14	75,8	1
formatPeriod(long,long,String,boolean,TimeZone)	17	95,4	6
longToIntArray(long,int,int[],int,int)	8	22,2	4
isAllUpperCase(CharSequence)	6	11,1	2
startsWithAny(CharSequence,CharSequence...)	11	14,3	7
containsAny(CharSequence,char...)	8	68,6	2
indexOfDifference(CharSequence...)	8	69	1
wrap(String,int,String,boolean,String)	18	87,1	5
uncapitalize(String)	6	11,1	2
divide(Dfp)	34	99,3	17
divide(int)	10	46,2	3
intLog10()	10	50	8
isZero()	4	0	2
pow(Dfp,Dfp)	8	89,4	3
getFittestChromosome()	3	18,2	1

Source: author

Table 5.7 – Properties of automatically generated suite for selected methods

Method	*N^o Tests*	*% UPPCs*	*N^o RTPs*
getLevenshteinDistance(CharSequence,CharSequence,int)	38	62,1	21
formatPeriod(long,long,String,boolean,TimeZone)	9	88,9	2
longToIntArray(long,int,int[],int,int)	3	77,7	1
isAllUpperCase(CharSequence)	23	22,2	19
startsWithAny(CharSequence,CharSequence...)	47	14,3	37
containsAny(CharSequence,char...)	21	71,4	14
indexOfDifference(CharSequence...)	30	69	18
wrap(String,int,String,boolean,String)	13	87,7	3
uncapitalize(String)	71	11,1	60
divide(Dfp)	0	100	0
divide(int)	0	100	0
intLog10()	0	100	0
isZero()	0	100	0
pow(Dfp,Dfp)	0	100	0
getFittestChromosome()	0	100	0

Source: author

Table 5.8 – Properties of project's suite for selected methods: updated

Method	*N^o^ Tests*	*% UPPs*	*N^o^ RTPs*	*RTPG*
getLevenshteinDistance(CharSequence,...,int)	38	62,1	14	10
formatPeriod(long,...,TimeZone)	35	95,4	25	9
longToIntArray(long,...,int)	14	22,2	11	3
isAllUpperCase(CharSequence)	10	11,1	5	4
startsWithAny(CharSequence,CharSequence...)	12	14,3	6	4
containsAny(CharSequence,char...)	25	51,4	12	5
indexOfDifference(CharSequence...)	17	65,5	4	3
wrap(String,int,String,boolean,String)	46	81,8	20	9
uncapitalize(String)	10	0	4	4
divide(Dfp)	29	99,3	13	6
divide(int)	16	42,3	8	4
intLog10()	16	0	12	4
isZero()	11	0	9	2
pow(Dfp,Dfp)	90	36,2	60	24
getFittestChromosome()	1	27,3	0	0

Source: author

before and as seen in Table 5.7, for some of the methods the generated suite contains no tests whatsoever.

Another thing we can observe is that the project's suite contains slightly more tests than the manually created suite, and also slight more redundant TPs. However, the project's suite covers slightly more PPs than the manually created suite. In order to study whether this increased TP redundancy improves the efficiency of the suite or not, we must manually analyze the test cases to see if the inputs provided come from different, non-equivalent partitions. The definition of equivalent value partition is provided in Section 2.6.

5.3 Manual comparison

To make this analysis more objective, we will add one extra column to Table 5.5 and Table 5.6 to track the number of "redundant TP groups" (RTPG), i.e. the number of sets that contain more than one test case producing the same TP. For each group, we study what causes the TP redundancy. See Table 5.8 and Table 5.9.

For the project's suite, in at least 4 methods (`containsAny`, `getLevenshteinDistance`, `startsWithAny` and `formatPeriod`) a considerable part of the redundant TPs come from testing guard clauses, such as empty array, null string, empty string, interval of length zero, etc. These compose non-equivalent classes of inputs and may not be considered redundant.

Table 5.9 – Properties of created suite for selected methods: updated

Method	*N^o Tests*	*% UPPs*	*N^o RTPs*	*RTPG*
getLevenshteinDistance(CharSequence,...,int)	14	75,8	1	1
formatPeriod(long,...,TimeZone)	17	95,4	6	4
longToIntArray(long,...,int)	8	22,2	4	3
isAllUpperCase(CharSequence)	6	11,1	2	2
startsWithAny(CharSequence,CharSequence...)	11	14,3	7	2
containsAny(CharSequence,char...)	8	68,6	2	1
indexOfDifference(CharSequence...)	8	69	1	1
wrap(String,int,String,boolean,String)	18	87,1	5	2
uncapitalize(String)	6	11,1	2	2
divide(Dfp)	34	99,3	17	9
divide(int)	10	46,2	3	4
intLog10()	10	50	8	1
isZero()	4	0	2	1
pow(Dfp,Dfp)	8	89,4	3	3
getFittestChromosome()	3	18,2	1	1

Source: author

`getLevenshteinDistance` and `indexOfDifference` contain some redundant TPs also due to parameter swap - there is one test case to call the method with parameters `a` and `b`, in that order, and another that calls it with parameters `b` and `a`. When this is done once, it may not be considered redundant since it tests a functionality (is the order of the parameters relevant to the method?), however when it is tested more than once it may indeed define a redundant test case.

`getLevenshteinDistance` and `startsWithAny` also contain sets of test cases that test different classes of equivalence in strings, i.e. uppercase x lowercase letters, alphabetic characters x numeric characters, and others. This also defines non-redundant test cases.

Some methods such as `formatPeriod`, `longToIntArray` and `wrap` contain redundant TPs also due to changing of parameters that do not affect the execution. In method `wrap` this happens because it contains one Boolean parameter that, when set to `true`, will allow the application to break long words (such as URLs). However, the suite contains many pairs of test cases testing both values of this parameter, while no long words are present, which makes the Boolean have no impact whatsoever. For method `formatPeriod`, the previously mentioned issue happens because the *format* parameter is actually used by another method that it returns, and not by the `formatPeriod` itself. All these cases define redundant test cases, and could be removed or improved by studying them carefully.

For most of *Dfp*-related test cases, many class-related special values (such as positive infinite, negative infinite, NaN, etc) are used, which makes it hard to determine whether they are indeed redundant test cases or not, but we interpret them as not redundant (with exception of some test cases from `intLog10` and `pow`. The former contains some sets of test cases with multiple numeric values producing the same TP, and the latter contains some identical, duplicated, test cases).

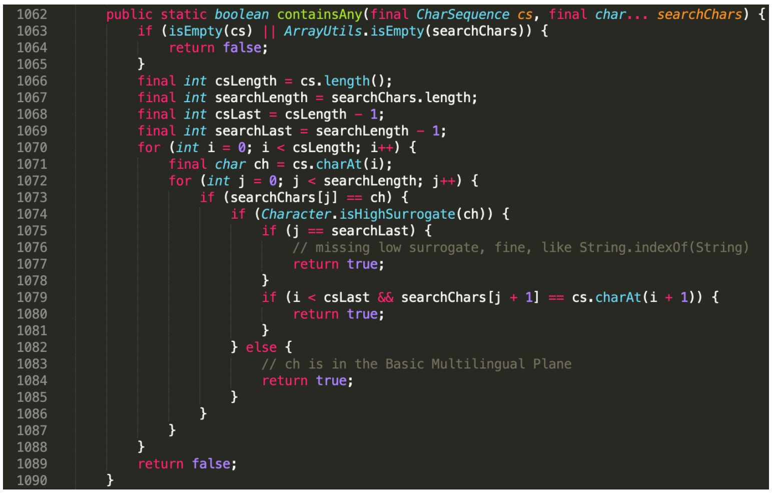
For the created suite, roughly most of the redundancy comes from guard clauses, and occasionally values of same types but different classes of equivalence, so very few are really redundant. `getFittestChromosome` has one redundant test that comes from two tests with very similar setups.

Overall, in the project's suite, the redundant TPs tend to compose different kinds of inputs, and indeed improving the suite's effectiveness. However, performing this estimate as to whether the tests test different kinds of inputs or not is hard and, sometimes, even though the combination of parameters that are passed to the function compose an equal equivalence class, there might be a reason as to why those tests are being executed. For some of the methods, such as `wrap`, there are multiple comments sustaining the purpose of the test cases. However there are cases that the comments do not specify a reason to the redundancy (also in `wrap`, all the methods that test both Boolean values contain no clarification whatsoever).

We may try to study the source of the test cases, i.e. check the context of the project by the time that they were developed. These projects are all maintained in GitHub, so we can check commit messages, timestamps and pull requests for information. However, the only information we get by analyzing the projects' repositories is that most tests were created 11-18 years ago. Besides that, no other information is given nor as pull request descriptions, nor as commit messages.

Another observation we have to make is that the percentage of uncovered PPs is considerably high for some methods, which might induce one to believe that the suite's are not effective. However, a detailed analysis in these cases show that there are many prime paths that are invalid, i.e. represent execution paths that will never happen. For example, Figure 5.19, that shows the PPC criteria of method `containsAny`, defines a prime path that makes line 1070 jump to line 1089 without entering the first line of the loop (selected in the figure). However, if we check the source code of the method in Figure 5.18, it is easy to see that this will never happen due to the guard clause in line 1062, which makes the logic return if the length of `cs` is zero (condition necessary in order for the loop to be skipped in its first step). This issue happens to many of the studied methods, which is the

Figure 5.18 – Source code of method `containsAny`.



```java
public static boolean containsAny(final CharSequence cs, final char... searchChars) {
    if (isEmpty(cs) || ArrayUtils.isEmpty(searchChars)) {
        return false;
    }
    final int csLength = cs.length();
    final int searchLength = searchChars.length;
    final int csLast = csLength - 1;
    final int searchLast = searchLength - 1;
    for (int i = 0; i < csLength; i++) {
        final char ch = cs.charAt(i);
        for (int j = 0; j < searchLength; j++) {
            if (searchChars[j] == ch) {
                if (Character.isHighSurrogate(ch)) {
                    if (j == searchLast) {
                        // missing low surrogate, fine, like String.indexOf(String)
                        return true;
                    }
                    if (i < csLast && searchChars[j + 1] == cs.charAt(i + 1)) {
                        return true;
                    }
                } else {
                    // ch is in the Basic Multilingual Plane
                    return true;
                }
            }
        }
    }
    return false;
}
```

Source: author

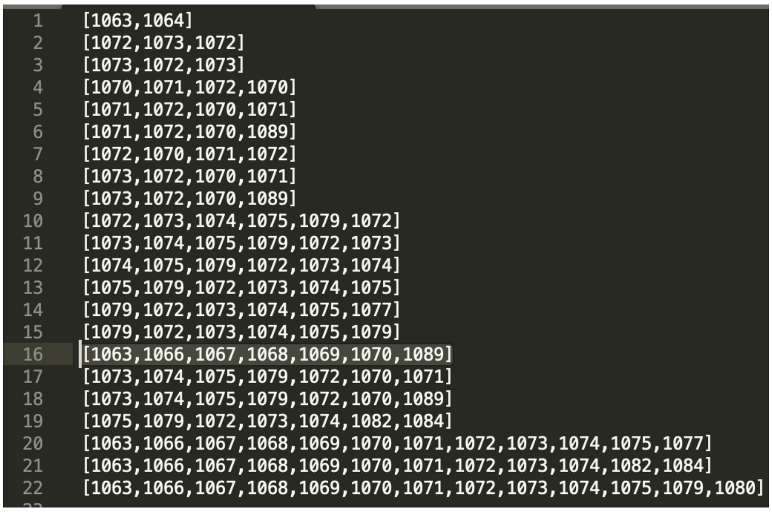
main reason for the high amount of UPPs.

Another item that draws attention is the extremely high amount of UPPs in the method `formatPeriod`. Upon inspection of the method's source code and the TPs of both suites, we can see that there is a big piece of source code that neither suite reaches due to the conditions established in the logic. This is another issue whose source is unknown.

Overall, in spite of these observations, we can see that a slight increase in the TP redundancy in the project's suite does indeed increase the suite's effectiveness in fault prevention and test coverage, mainly because most of TP redundancies come from guard clause testing. There are only two cases where a higher amount of real redundancy is identified (textttwrap and `pow` methods), and even in these, the number is still not very significant.

With these observations we end our analysis of the project's suites, the conclusions being as follows: the redundancy that was previously identified comes not from an automatically generated suite, but from a suite that was manually developed. Furthermore, it comes mostly from tests cases that test different types of inputs - i.e. parameters of different equivalence partitions.

Figure 5.19 – PPC criteria for method `containsAny`.



```
[1063,1064]
[1072,1073,1072]
[1073,1072,1073]
[1070,1071,1072,1070]
[1071,1072,1070,1071]
[1071,1072,1070,1089]
[1072,1070,1071,1072]
[1073,1072,1070,1071]
[1073,1072,1070,1089]
[1072,1073,1074,1075,1079,1072]
[1073,1074,1075,1079,1072,1073]
[1074,1075,1079,1072,1073,1074]
[1075,1079,1072,1073,1074,1075]
[1079,1072,1073,1074,1075,1077]
[1079,1072,1073,1074,1075,1079]
[1063,1066,1067,1068,1069,1070,1089]
[1073,1074,1075,1079,1072,1070,1071]
[1073,1074,1075,1079,1072,1070,1089]
[1075,1079,1072,1073,1074,1082,1084]
[1063,1066,1067,1068,1069,1070,1071,1072,1073,1074,1075,1077]
[1063,1066,1067,1068,1069,1070,1071,1072,1073,1074,1082,1084]
[1063,1066,1067,1068,1069,1070,1071,1072,1073,1074,1075,1079,1080]
```

Source: author

6 CONCLUSION

This work has analyzed and discussed a possible redundancy present in the test suites of Java projects, originated from test cases with duplicate execution flows. Hereby, the question we aim to answer is as follows: why do these tests with duplicate flows exist and what is their source?

We have defined two hypothesis: one that the suite was automatically generated, and thus containing redundant tests; and also one that the test cases with equal execution flow cover different types of parameters (for instance, empty string and null string). We validated these hypothesis by comparing the properties of three test suites: one created using a black-box technique, one generated using an automatic test generation tool, and the project suite's itself. Among the properties that were analyzed are the prime-path coverage of the suite and the number of redundant tests in it. To obtain the prime-path coverage of the test suite, we have researched for a CFG and PPC generation tool, and later adapted it in order to expand its support of Java code.

After the tool was adapted, we took 15 Java methods and, for each method, we compared the three test suites, looking for their prime-path coverage and the number of redundant tests. This allowed us to conclude that the project's test suite is closer to a developer-made test suite, rather than a generated one, as is shown in Tables 5.5, 5.6 and 5.7. This refutes our first hypothesis.

A deeper, manual analysis was made to see which test cases cause the redundancy and what exactly are they doing. In this step, it was found that most of the redundancy comes from test cases that test parameters of different equivalence partitions, these being mainly edge cases. There is, however, a portion of test cases that test values of same equivalence partitions, and even some tests that are duplicated. However, no apparent reason is given to these.

Given these conclusions, we have enough information to continue the broader work we have mentioned in Chapter 1, made by Silva and Cota (2020), which aims to help developers in creating and maintaining an effective test suite. That work uses a knowledge discovery in database (KDD) framework and a data set built from 12 Java projects to create predictive models for prime-path prediction. Now that we know the redundancy identified in the Java methods comes mainly from tests with different kinds of inputs and do not consist of a real redundancy, we can safely use the present data set to generate the predictive models.

One question that still remains open is as to what is the impact of this redundancy in the test suite's effectiveness? Does it improve it, or just causes the suite to be larger? This, along with the modifications that were not made to the PPC generation tool (shown in Table 4.2), remains as ideas for future work.

REFERENCES

AMMANN, P.; OFFUTT, J. **Introduction to Software Testing**. Cambridge: Cambridge University Press, 2016. ISBN 9781107172012.

EVOSUITE. 2020. Acessed on 2021-03-01. Available from Internet: <https://github.com/EvoSuite/evosuite>.

GLIGORIC, M. et al. Comparing non-adequate test suites using coverage criteria. In: **Proceedings of the 2013 International Symposium on Software Testing and Analysis**. New York, NY, USA: Association for Computing Machinery, 2013. (ISSTA 2013), p. 302–313. ISBN 9781450321594.

GOPINATH, R.; JENSEN, C.; GROCE, A. Code coverage for suite evaluation by developers. In: . New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014). ISBN 9781450327565.

INOZEMTSEVA, L.; HOLMES, R. Coverage is not strongly correlated with test suite effectiveness. In: **Proceedings of the 36th International Conference on Software Engineering**. New York, NY, USA: Association for Computing Machinery, 2014. (ICSE 2014), p. 435–445. ISBN 9781450327565. Available from Internet: <https://doi.org/10.1145/2568225.2568271>.

MYERS, G. J.; SANDLER, C.; BADGETT, T. **The Art of Software Testing**. 3rd. ed. [S.l.]: Wiley Publishing, 2011. ISBN 1118031962.

NAMIN, A. S.; ANDREWS, J. H. The influence of size and coverage on test suite effectiveness. In: **Proceedings of the Eighteenth International Symposium on Software Testing and Analysis**. New York, NY, USA: Association for Computing Machinery, 2009. (ISSTA '09), p. 57–68. ISBN 9781605583389. Available from Internet: <https://doi.org/10.1145/1572272.1572280>.

NIEMEC, W. **Execution Flow**. 2020. Acessed on 2021-03-02. Available from Internet: <https://github.com/williamniemiec/ExecutionFlow>.

SILVA, K.; COTA, E. Predicting prime path coverage using regression analysis. In: **Proceedings of the 34th Brazilian Symposium on Software Engineering**. [S.l.: s.n.], 2020. p. 263–272.

WOLFART, M.; LEE, S. H.; PLATT, E. **TRGeneration (adapted)**. 2020. Acessed on 2021-05-18. Available from Internet: <https://bitbucket.org/mwolfart/trgeneration>.

ZHANG, J. et al. Predictive mutation testing. **IEEE Transactions on Software Engineering**, v. 45, n. 9, p. 898–918, 2019.