
Random Test Generator and Executor

for Java code

Project Report
Nicola Gilberti 198739

University of Trento
MSc Computer Science

Contents

Abstract	ii
1 Background	1
1.1 Spoon	1
2 Test generation Algorithm	3
2.1 Master-slave architecture	3
2.2 Class Instrumentor	4
2.3 Primitive input generator	5
2.4 Test generator	8
3 Usage	12
3.1 Project Object Model	12
3.2 Properties	13
4 Empirical evaluation	15
5 Conclusion	18
Bibliography	19

Abstract

Nowadays, software has a big impact in all aspects of our society. More and more companies develop software and testing plays a crucial role for ensuring its quality.

Web applications present new testing challenges with respect to traditional software testing, due to their dynamic and heterogeneous nature. In this document, a tool (Web Random Generator) able to create a test suite for a given application following a random testing approach is presented. WRGen first instruments Page Objects developed for the given application to measure coverage. Then it generates random test cases as sequence of Page Object methods.

Finally, a comparison with a state-of-the-art random generator tool (Evosuite) on 5 web applications is carried out. In terms of coverage, WRGen outperforms Evosuite random generator over all subjects.

Nicola Gilberti
<nicola.gilberti at studenti.unitn.it>

1. Background

The project is developed in Java [1] and is based on basic Java programming technologies such as, a Java compiler (Javac [2]), Reflection [3], to dynamically analyze classes at runtime, and Maven [4], to manage dependencies and for the cross-platform compatibility.

In addition to those elements, a Java parser, called Spoon, is used. Next section introduces Spoon and its application in this project.

1.1 Spoon

Spoon [5] is a Java library that gives the ability to create and modify Java source code. Therefore, with this library, a programmer can transform or analyze Java code dynamically.

Spoon takes advantage of the Java Reflection [3] feature to build the abstract syntax tree (AST) of the code under analysis, but also to give a programming interface (intercession API) that let the user modify and generate Java source code.

Regarding the code that a programmer can insert, Spoon gives different options to check its compliance with the Java grammar. **The first option leverages generics to manipulate the AST that give to the programmer a feedback in case of bad code.** Another option that Spoon implements is the template engine with static checks that let the programmer insert code automatically ensuring that it is well formed.

How it works

Spoon works primarily on the AST, giving the possibility to manipulate Java code elements. The *Processor* and the *Factory* classes are the key-components to use to analyze and transform the code. The Processor lets the programmer analyze the AST, while the Factory permits to modify the AST, adding and/or removing elements from the syntax tree. The concept is similar to the read and write operations, where the first operation (Processor) is read-only while the second (Factory) can also write.

In particular, the Processor class is utilised for querying Java code elements. That operation is possible thanks to the visit pattern applied to the Spoon model (AST). All the code elements have an *Accept* method, therefore each element can be visited by a visitor object. For instance, in Listing 1.1, a Processor is used to search for an empty catch block. As the code shows, the Processor works on a *CtCatch*,

which is the Compile Time Catch given by the Spoon Metamodel, and checks for statements inside the CtCatch body.

```
1 public class CatchProcessor extends AbstractProcessor<CtCatch> {  
2     public void process(CtCatch element) {  
3         if (element.getBody().getStatements().size() == 0) {  
4             getFactory().getEnvironment().report(this, Level.WARN, element  
5                 , "empty catch clause");  
6         }  
7     }  
}
```

Listing 1.1: Processor example taken from Spoon documentation

The Factory class gives the coder the ability to create new elements, and add them to the Syntax Tree under analysis. There is more than one Factory class, where each one is specialized to facilitate the creation of specific code elements.

```
1 Factory factory = this.getFactory();  
2 String snippet = this.getLogName() + ".testOut(Thread.currentThread())";  
3 CtCodeSnippetStatement snippetFinish = factory.Code().  
4     createCodeSnippetStatement(snippet);  
5 CtBlock finalizerBlock = factory.Core().createBlock();  
6 finalizerBlock.addStatement(snippetFinish);  
7 ctTry.setFinalizer(finalizerBlock);  
8 CtBlock methodBlock = factory.Core().createBlock();  
9 methodBlock.addStatement(ctTry);  
10 element.setBody(methodBlock);
```

Listing 1.2: Factory example taken from Spoon Projects

In Listing 1.2, a factory object is used to create a *try* block inside an existing Java method. The *Code* method of the factory object is used to create code elements (for example, *snippets*) while the *Core* helps in creating blocks of code, that will eventually contain *Code* generated elements.

2. Test generation Algorithm

This chapter explains all the implementation details and the key-points of the test generation algorithm. WRGen generates Selenium-based end-to-end (E2E) test cases for web applications. Researchers [6] have used *Evosuite* [7] to generate E2E test cases for web applications. Evosuite is a unit test case generator for Java classes. Although Evosuite is a stable tool that is shown to be quite effective in generating test cases for Java classes, it is very complex and its extension to other test generation scenarios (as web applications) is not straightforward. The goal of this project is to implement a test case generator that performs only those operations that are needed for generating Selenium-based (E2E) test cases for web applications.

2.1 Master-slave architecture

The program performs different steps to achieve the final result and each step is performed by a different *worker*. Specifically, the program uses two distinct threads. This architecture, that other test case generators (as Evosuite [7]) implement, can be used to balance the work-load on different processors to increase the efficiency.

The algorithm use threads for specific purposes, defining distinct operations and scopes for each of them. Firstly the main thread start, managing all the imports from resources (see section 3.2) to prepare the environment for the other thread. In fact, the main thread updates dynamically the Java *classpath* in order to recover all the classes needed for the runtime execution.

The second thread (slave) is activated by the main thread (master) once the environment is set up. The second thread generates and executes test cases and sends messages to the main thread with the results of its computations.

The second thread instruments the input class, a Java class containing Page Object methods for a given web application, and generates tests as sequences of those methods (see section 2.4). Those tests are sent back to the primary thread through a *LinkedBlockingQueue* [8], that checks their validity and creates a well-formed test suite. The choice of a *LinkedBlockingQueue* is a default option to reduce the impact on the memory load during inactivity and to avoid errors due to concurrency at run-time. Even if there can't be simultaneous access from both threads, because the second one generates the message and it restarts creating a new one, the implementation follows the best-practices hints for correct data exchange between threads.

2.2 Class Instrumentor

This section explains how the input class is instrumented. The main dependency for this package is Spoon, explained in section 1.1.

The *ClassModifier* class prepares the Spoon environment. Specifically, it generates all the variables that are necessary to create/modify elements in the Java input class. The program needs to insert extra code in the input class, therefore a *factory* object must be instantiated. In addition, Spoon requires a data model of the class under analysis, as seen in section 1.1. This is generated by a Spoon object that returns, given the source-path of the input class under analysis, the AST of the input class.

```

1 protected void buildModel(File sourcePath) throws IOException {
2     builder = new JDTBasedSpoonCompiler(factory);
3     try {
4         builder.addInputSource(sourcePath);
5         builder.build();
6     } catch (Exception e) {
7         throw new RuntimeException(e);
8     }
9 }

```

Listing 2.1: The function to create the AST from the ClassModifier class

Once the AST is created then the class under test can be instrumented. WRGen can instrument the class under test to measure both branch and line coverage. In both cases, when the target (line or branch) is reached during the execution of a test, WRGen uses a list that collects the identifier of the line/branch covered.

In case of branch coverage, the instrumentor searches for *If* structures, and adds a statement (Listing 2.2) in the first line of each *then* and *else* block. A counter is used to uniquely identify a branch.

```

1 factory.Code().createCodeSnippetStatement("checker.add("+ counter++ +")");

```

Listing 2.2: Snippet creation

In case of line coverage the statement added is the same, while the analysis of the class under test is different. In fact the instrumentor takes as input the lines of the class under test to cover and performs a bottom-up scan searching for the lines. When there is a match it appends the instrumentation snippet shifting one line down the rest of the code (????).

```

1 private void modConstructors(CtClass cc) {
2     List<?> tmp;
3     CtConstructor<?> constructor;
4     Set<?> constructors = cc.getConstructors();
5     Iterator<?> itc = constructors.iterator();
6     while(itc.hasNext()) {
7         constructor = (CtConstructor<?>) itc.next();
8         CtBlock<?> ctb = constructor.getBody();
9         tmp = ctb.getStatements();
10        List<CtStatement> newStatements = new ArrayList<CtStatement>();
11        CtCodeSnippetStatement newStatement = factory.Code().
createCodeSnippetStatement("checker = new ArrayList()");
12        newStatements.add(newStatement);
13        for(int j = 0; j < tmp.size(); j++){
14            newStatements.add((CtStatement) tmp.get(j));
15        }
16        ctb.setStatements(newStatements);
17    }
18 }

```

Listing 2.3: The function to add the ArrayList instantiation snippet

The instrumentor (see Listing 2.3) creates extra methods in the class under test that are needed during test case execution, like getting data from the list of targets and resetting it after every test execution.

The last operation that the instrumentor performs is an analysis of the class under test to understand all the default fields, in order to have those values as options when instantiating variables for the class methods.

2.3 Primitive input generator

The process of the generator works hand by hand with the *Test Generator* 2.4, and it does multiple works. The goal of this chapter is to define all the primitives types this project can random-generate and how values already instantiated could be reused during their lifecycle.

This work is based on the Java language, so its well-known primitives are maintained. There is an extra types directly generated from the code and it is the *String*. About the Java primitives, they can be used in a reflection environment only thanks to their object representations like *Integer* for *int*, *Character* for *char* and so on.

When the program require the generation of a new variable, and the process find that it require a primitive, it start instantiating the proper types with a default constructor, assigning a fixed value. Then it is checked if an older generated value exist and, in case of a true response, a first random choice is performed.

With an equal probability, it is possible that the program try to generate a new value or choose to pick an old ones. There is a third options for object types that is

the *null* value, but has a smaller probability to appear.

When an older value is chosen, the data to link the two references are arranged for future steps. On the other case, a random value is required, and the Random [9] class enter in action. For some primitives, this dependency is able to generate directly a value, while in other cases an extra work is necessary. In fact, in case of int, float, long, double, and boolean, there are already functions.

```

1 case "java.lang.Integer":
2     obj.setValue(random.nextInt());
3 case "java.lang.Float":
4     obj.setValue(random.nextFloat());
5 case "java.lang.Long":
6     obj.setValue(random.nextLong());
7 case "java.lang.Double":
8     obj.setValue(random.nextDouble());
9 case "java.lang.Boolean":
10    obj.setValue((random.nextBoolean()));

```

Listing 2.4: Random default generator

However, in case of short, the random-generator can be done thinking on how the type is constructed.

```

1 case "java.lang.Short":
2     obj.setValue(random.nextInt(65536) - 32768);

```

Listing 2.5: Random short generator

Char and string require more operations, thus an extra function enter in action.

```

1 public static String generateRandomChars(String candidateChars, int length
2     , Random random) {
3     StringBuilder sb = new StringBuilder();
4     for (int i = 0; i < length; i++) {
5         sb.append(candidateChars.charAt(random.nextInt(candidateChars.
6             length())));
7     }
8     return sb.toString();
9 }

```

Listing 2.6: Random String/char generator

The difference between String and char is the length input, which is 1 for character, and random generated for strings. Variable *candidateChars* is a sequence of accepted character.

For example, "ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890abcdefghijklmnopqrstuvwxyz" can be used if upper/lower case letters and numbers are the only accepted values we want to generate with.

In the Java language, there are also different types of objects, and one of them is the Enumerator [10].

This program consider Enums as a primitive type and when one is found, it generate one instance of it with a random value accordingly with the requirement of randomness.

With this particular object type, the equal operation work differently and so, the links for the usage of old values are not necessary. In case of it is required an instantiation that do not belong to the previous cases, a recursive approach is adopted. The 'unknown' class is analysed, collecting all its constructors. Then one is randomly picked and, if variables are required to execute the constructor, the random generation process is re-called on each element, line 14 in 1. If the constructor does not require any input variables, Java reflection can instantiate it directly.

There is also the option in which the class was already instantiated with random values and the randomness choose to take an old option instead of a new one. This is automatically done as in the case of primitives, and the links between variables are saved accordingly.

Algorithm 1 Instantiator

```

1: procedure INSTANTIATE(params)
2:   for all params do
3:     param  $\leftarrow$  params.next
4:     if isPrimitive(param) then
5:       param  $\leftarrow$  INSTANTIATEWITHREFLECTIONOROLDVALUE
6:     else if isEnum(param) then
7:       param  $\leftarrow$  PICKRANDOMENUMCONSTANT(param)
8:     else if isInterface(param) then
9:       param  $\leftarrow$  INSTANTIATE(param.getSubclass.getParameter())
10:    else
11:      if directlyInstantiable(param) then
12:        param  $\leftarrow$  INSTANTIATEWITHREFLECTIONOROLDVALUE
13:      else
14:        requireInstances  $\leftarrow$  INSTANTIATE(param.getParameter())
15:        param  $\leftarrow$  INSTANTIATEWITHREFLECTIONOROLDVALUE(requireInstances)
  return params  $\triangleright$  All the instantiated objects

```

For example, there is a class Clazz that has to be instantiated. It is nor primitive neither an enum, so a random constructor is picked (line 10 in 1). The one taken require two variables, an int and another unknown class Clazz2. So the generator is recalled on the int variable and on the Clazz2 (line 14 in 1). About the primitive typos, the program could generate a new value or pick an old ones if exist (line 5 in 1). Clazz2 has only the default empty constructor, thus Reflection can instantiate it directly (1 line 12). Now there is the backtrack on the recursion(line 15 in 1 can now be executed) and the Clazz can be instantiated with the constructor chosen

and the input variables created just before.

With this approach the program is able to instantiate any kind of object, which are referenced in the environment, that are required for tests.

As an example of the environment requirement, in case of a class X is required to be instantiated and its source is not in the class-path, the instantiation fail and the creation of tests *is* compromised. Until now, the program has take care only of 'directly' instantiable objects, avoiding Interfaces.

The management of those classes need an extra work, that consist in finding one of its sub-classes that can be instantiate in its place.

The operation is done filtering all the classes related to the interface's project and searching for *Beans* that extend the class.

```
1 ClassPathScanningCandidateComponentProvider provider = new
   ClassPathScanningCandidateComponentProvider(false);
2 provider.addIncludeFilter(new AssignableTypeFilter(target));
3 Set<BeanDefinition> components = provider.findCandidateComponents("./");
4 for (BeanDefinition component : components){
5     Class cls = Class.forName(component.getBeanClassName());
6     // use class cls found
7 }
```

Listing 2.7: how to find a sub-class of an interface

The *ClassPathScanningCandidateComponentProvider* and *BeanDefinition* are taken from the Spring Framework [11].

Interfaces are commonly used to make reference of it that refers to the Object of its implementing class, and this is why the program manage them.

2.4 Test generator

This chapter cover a big part of the program because the whole test generation process start from the secondary thread 2.1, pass through the generator 2.3 and end in the next subsections 2.4.1 2.4.2.

This program depends on an external service that given a graph representing the class under test, an ending point and a length, it return a random path of at most the size given that finish in the target. Given all the proper data collected by the main thread, see 2.1 and 3, the second thread recreate a possible correct sequence of action that can be performed on the class. This operation is executed before the creation of each test.

Before the creation of a test, the sequence generated is manipulated for the next steps. This sequence consist in a list of strings where each string is the name of the method to execute, the node in which it can be executed and the ending node if the method run throwing no error.

For each method, an instance of a `MethodTest` class is instantiated, and thanks to that, all the supporting variable are created. In this class there is the list of the attributes that are generated with the random generator 2.4, and all the data to reconstruct them. Those values are created once for each method and they are maintained until they need. Then, the test creation start.

2.4.1 Creation

Till now there is a list of methods and for each of them there is a possible input.

```

1 private static void AddToFinalTestCase(TestCase newTest) {
2     HashSet<Integer> tmp1 = newTest.getBranchCov();
3     boolean flag=true;
4     if(finalTests.isEmpty()) {
5         finalTests.add(newTest);
6     } else {
7         for(int i=0;i<finalTests.size();i++) {
8             TestCase oldTest = finalTests.get(i);
9             if(sameValues(oldTest.getBranchCov(),newTest.getBranchCov())) {
10                flag=false;
11                if(oldTest.getMethList().size()>newTest.getMethList().size()) {
12                    finalTests.set(i, newTest);
13                }
14                break;
15            } else {
16                if(newTest.getBranchCov().containsAll(oldTest.getBranchCov())) {
17                    finalTests.set(i, newTest);
18                    flag=false;
19                    break;
20                } else if (oldTest.getBranchCov().containsAll(newTest.getBranchCov())
21                ){
22                    flag=false;
23                    break;
24                } else {
25                    tmp1.removeAll(oldTest.getBranchCov());
26                }
27            }
28            if(!tmp1.isEmpty() && flag) {
29                finalTests.add(newTest);
30            }
31        }
32    }

```

Listing 2.8: Check-test method

Here can be done extra controls on the list and on their input to reject or accept the list as a possible test case. After that an instance of the instrumented class is generated and the method's list is executed in order.

When an error is thrown, the data is collected and saved for the next step 2.4.2.

Executing the methods, instrumentation can be invoked and, at the end of the list execution, all the covered element are saved.

Each ran list, its input and outcomes are then considered as a Test-case.

This blob of data contains everything that permit the validation of the test and also the re-execution of it, maintaining the same outputs: test-case are idem-potent.

Each test as to be checked in order to add it or not to the final test-suite. The final test-suite is then created dynamically 2.8.

If a test is the only one to cover a specific path, it is added to the test-suite. This work also in case of empty suite.

If a new test cover the same as another inside the test-suite, then the length or possible extra path covered by the tests decide for who will stay in the test-suite.

2.4.2 Pretty printing

Final test-suite has to be printed out in order to be executed. Then all the data collected before are used to recreate the formula to instantiate and execute methods in a JUnit fashion.

As in a Java class, the first area is about the imports. This is recreated thanks to the import that are inside the Instrumented class, which are managed automatically by Spoon, if it is setted accordingly.

```
1 factory .getEnvironment().setAutoImports(true);
```

Listing 2.9: How to let Spoon automatically manage imports during class generation

The same auto-generated import list is then recovered and used in the printing part. After that, a *Tester* class is generated and the real tests are written inside. Each single test start with a common pattern that is as following. First it came the JUnit annotation used to identify tests, *@Test*. As second element, a comment that give some information about the test, like the coverage. Third element is the test's header followed by the instantiation code for the class under test.

```
1 public class Tester{
2
3     @Test
4     //test case 2 coverage: 0.07906976744186046
5     //branch covered: [96, 161, 98, 163, 133, 201, 170, 171, 109, 206,
6     112, 49, 178, 88, 121, 122, 156]
7     public void test0(){
8         main.ClassUnderTest obj = new main.ClassUnderTest();
9         ...
10    }
```

Listing 2.10: Example test case, first part

Here, tests could differ: if the class has multiple constructors, the instantiation could be preceded by extra variables depending on the constructor parameter requirements.

```
1      int var3920 = 1599533343;
2      Id var3910 = new Id(var3920);
3      WidgetFeedTitle var3911 = WidgetFeedTitle.PAGEKITNEWS;
4      WidgetFeedUrl var3912 = WidgetFeedUrl.PAGEKIT;
5      int var3920 = var120;
6      WidgetFeedNumberPosts var3913 = new WidgetFeedNumberPosts(var3920);
7      WidgetFeedPostContent var3914 = WidgetFeedPostContent.SHOWALLPOSTS;
8      try {
9          obj.editFeedWidgetDashboardContainerPage(var3910, var3911, var3912,
10             var3913, var3914);
11     } catch (po_utils.NotInTheRightPageObjectException e) {}
```

Listing 2.11: example of some possible printing cases

Now each test differ for the method lists and the executions outputs, however the printer is able to manage all the variables for each instantiation and also try-catch block when methods thrown errors.

In the listing 2.11 we can see how cases similar to the example at the end of 2.3 are managed.

Tabulation is used to define the depth level of the variables. For example at line 1 there is a 2-Tab variable meaning that it this is required for a 1-Tab variable instantiation, which is in line 2.

All the 1-Tab depth variables are the input for the method in line 9.

During the execution, this method generate a *NotInTheRightPageObjectException* so the test take care of that with a try-catch block around it.

Another element is the reference to an old variable in line 5. The test end with an assertion, that check the correct generation of the test, checking if the branch covered are effectively executed by the test.

This can be done because the instrumentation during tests still work and the list of executed branch is maintained, so the equality can be assessed.

3. Usage

This chapter explain all the features that are necessary for the project to run. Specifically it will show some *not* well-known tags in the POM, and the properties file necessary for the Environment.

The project on Github contains also a .sh script with an example of how the program can be runned.

3.1 Project Object Model

The POM contains information about the project and various configuration detail used by Maven to build the project.

To compile classes at runtime, a specific *JDK* library has to be imported as a dependency inside the project.

Because it depends on *JDK* and so on the machine architecture, a `<profile>` tag is necessary to identify all the possible options, to maintain the code independent from the machine.

```
1 <profile>
2   <id>windows_profile</id>
3   <activation>
4     <os>
5       <family>Windows</family>
6     </os>
7   </activation>
8   <properties>
9     <toolsjar>${java.home}\..\lib\tools.jar</toolsjar>
10  </properties>
11 </profile>
```

Listing 3.1: Windows default profile for jdk lib

There are specific path for each OS type, and Maven accept only specific *OS family*. `<toolsjar>` is an ad-hoc tag created to define a new property, in this case the path to the tools.jar lib.

This variable will be used during a plugin execution.

In the next listing 3.2, there are few element to consider. The phase in which this operation work is decidable by the programmer, in this case the *install* phase is selected.

The configuration of the library is based on the one installed on your machine, but in this case the *JDK* library has no different option if the OS change.

The version is the value that the runner of the program *MUST* verify.

```

1 <plugin>
2   <groupId>org.apache.maven.plugins</groupId>
3   <artifactId>maven-install-plugin</artifactId>
4   <version>2.5.2</version>
5   <executions>
6     <execution>
7       <id>install-external</id>
8       <phase>install</phase>
9       <configuration>
10        <file>${toolsjar}</file>
11        <repositoryLayout>default</repositoryLayout>
12        <groupId>com.sun</groupId>
13        <artifactId>tools</artifactId>
14        <version>1.8.0</version>
15        <packaging>jar</packaging>
16        <generatePom>true</generatePom>
17      </configuration>
18      <goals>
19        <goal>install-file</goal>
20      </goals>
21    </execution>
22  </executions>
23 </plugin>

```

Listing 3.2: How to 'mvn install' the tools.jar

3.2 Properties

A .properties is a file extension for files mainly used in Java related technologies to store the configurable parameters of an application.

This project require information about the project that has to analyse and about the specific class to instrument.

- **FileName:**

It is required the name of the Class to test, with the complete package route.
E.g.:FileName = com.main.ClassUnderTest

- **ProjectName:**

It is required the name of the Project that contains the class.

- **PathToProjectDir:**

The program has to know where the project you are referring to is placed.

- **LineToCover:**

The line-coverage has to know which line to instrument.
E.g.:LineToCover = 33:47:60:72:85:99:115:132:144:157

- **Separator:**
Lines in the previous tag are a list of line numbers divided by a separator. The default accepted is ':' but it can be used a different one defining it here.
- **ExecutionTestTimer:**
This is the time the algorithm has to find tests. It is in millisecond, so 60000 is equal to 1 minute.
- **MaxNumberOfMethodXTest:**
Each test can contain a random number of method calls. This attribute is an upper bound to it.
- **GraphName:**
As seen in 2.4, this project work generating a sequence of method following a graph.
- **GraphDirPath:**
The program has to know where the graph you are referring to is placed.
- **StartNodeGraph:**
It is the starting point of the graph path analysis.
- **RequiredPath:**
The program use reflection to execute the external project written in the first item, but it is not able to know all the dependencies it need.

4. Empirical evaluation

This chapter focus on the evaluation of the proposed algorithm, making a comparison between this and the Evosuite method. The comparison is done executing 10 time both programs on the applications and analyzing the output in terms of coverage and number of test executed to define the final testsuite. The evaluation is performed by statistical tests.

The distribution of the result cannot be assessed, caused by the random approach followed, so a t-test can't be performed. Furthermore the number of tests make useless the use of a t-test or similar parametric analysis.

However, the *Wilcoxon rank sum test*, also known as Mann–Whitney U test, avoid those limitations. It quantifies significant differences between the results, considering a significance level of 0.05.

To ensure the correctness of results, the analysis use also the *Vargha and Delaney* statistic. This test emphasize the magnitude of the differences, given by the U test, between the two algorithms.

Table 4.1: Evaluation of the application on web services

Test details			Statistic		Wilcoxon rank sum test		Vargha and Delaney
WebSite	Analysis	Application	Mean	Standard Deviation	W	p-value	A
Dimeshift	Coverage	WRGen	35	6.236096	82	0.01684	0.82 (large)
		Evosuite	28.9	3.813718			
	# Test	WRGen	24	1.414214	79.5	0.02631	0.795 (large)
		Evosuite	21.8	2.347576			
Retroboard	Coverage	WRGen	68.9	4.72464	100	0.0001433	1 (large)
		Evosuite	53.9	2.330951			
	# Test	WRGen	26.1	2.233582	43.5	0.6436	0.435 (negligible)
		Evosuite	26.3	2.110819			
Pagekit	Coverage	WRGen	19.7	3.093003	93	0.001218	0.93 (large)
		Evosuite	14.1	2.378141			
	# Test	WRGen	4	0.8164966	17.5	0.01108	0.175 (large)
		Evosuite	5.2	0.9189366			
Phoenix	Coverage	WRGen	59.2	8.283853	86.5	0.00599	0.865 (large)
		Evosuite	47.9	8.238797			
	# Test	WRGen	9.2	1.032796	23.5	0.04131	0.235 (large)
		Evosuite	10.3	1.567021			
Splittypie	Coverage	WRGen	40.4	4.526465	96	0.0005501	0.96 (large)
		Evosuite	26.2	7.568942			
	# Test	WRGen	17.2	3.224903	6.5	0.0008922	0.065 (large)
		Evosuite	21.9	2.078995			

The evaluation is also executed limiting the number of methods callable for each unit test, *@Test*, but letting the test suite to grow in number of unit test infinitely. The execution time is also restricted, so the program has not to run endlessly and eventually halt when it is reached the total coverage.

Respectively, the maximum number of method is 40 while the time is 60 s, and they are given as input with the properties file 3.2.

The table 4.1 show the results for each application under analysis.

The results on the web applications, give us some interesting results about the algorithm. In all the coverage analysis, the algorithm presented has a greater performance compared to the counterpart. In fact, the p-value is below the agreed risk of 5 percent (0.05), so one significant difference can be assumed.

The Vargha and Delaney value confirm each result, pointing the magnitude of the differences to *large*, in favour of the Web Random Generator. The result, in case of number of tests, are slightly different than before.

Figure 4.1: Dimeshift Analysis

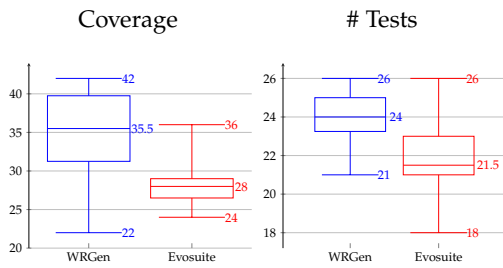
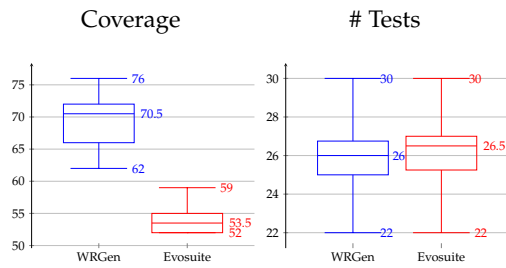


Figure 4.2: Retroboard Analysis

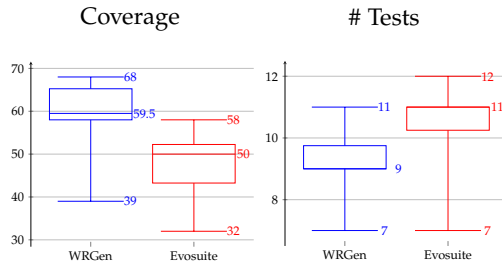
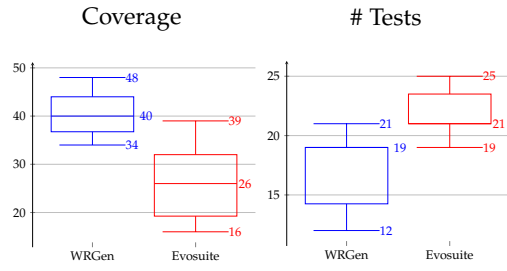
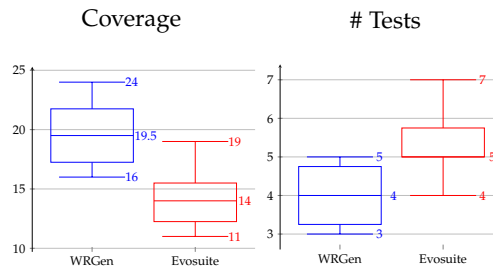


The first service, *Dimeshift*, point out that the WRGen has an higher value, however the number of test in each test-suite is preferable to be as thin as possible. Thus, even if the WRGen has a capacity to cover better the Web-app, in this case it require also a greater test-suite to pursue the scope.

We can check this results also with the box-plot representation 4.1. For the coverage is simple to find how the WRGen has a greater variance, possible synonym of a correct random generation. The number of tests, also, show that Evosuite on average require fewer tests to pursue the goal.

Nevertheless, the *Retroboard* application results on the test analysis are quite different. The p-value is higher than the risk, so the Wilcoxon test tells us nothing. Moreover the A value see no differences on the result of the two under test programs. With the box-plot 4.2, in fact, we can see that the behaviour of the two algorithm is the same.

Still, the WRGen is more performer, with a higher coverage compared to the same number of test executed.

Figure 4.3: Phoenix Analysis**Figure 4.4: Splittypie Analysis****Figure 4.5: Pagekit Analysis**

With these last three services, the results are similar. As defined before, The U test accordingly with the A value tell that the test-suite created by the WRGen has a higher coverage in respect of the counterpart. Additionally, the Vargha and Delaney method on the number of tests define that the Evosuite approach leads to generate a higher number of test. However, the higher number of possible final tests do not help the algorithm to find out a better test-suite. As before, the results are easily checkable with the graphical representation.

5. Conclusion

Starting from the evaluation, the method proposed has a better behaviour in terms of coverage and also number of test necessary. This can be caused by the great number of operation the Evosuite approach has to overcome, while this technique is ad-hoc for the purpose.

The evaluation is performed only on five application, and the constraints limit both programs, hence an higher number of Web-services or a different set of restriction can be defined to test new behaviours.

Despite this, the results are very positive and give solid feedback on the project. Since now the algorithm proposed can afford the problem of the automatic test generation, providing an adequate test-suite as output.

The technology used have a great impact on the performances, so a different approach on the instrumentation part could lead to different results.

In case of line coverage the technology used can be overwhelming, while an ad-hoc parser could perform the same operation with cheaper time/computation effort. The library used is a kind of parser, indeed, and a major number of external dependencies can be expensive too.

About the general instrumentation phase other techniques, like byte-code manipulation or other source-code handler, could make differences.

Moreover, the printing phase require extra workload during the instantiation steps, and other solution can be adopted. The use of a complex structure can reduce some computational/memory costs for a giant number of instantiations per methods. Nonetheless, the complex structure has to be chosen wisely, or the benefit are no longer available.

In conclusion, despite the limited tests, results are still impressive and the proposed algorithm seems to work correctly. As always, future work can be done to improve the performances and the efficiency on test generation phase.

So, in case you have questions, comments, suggestions or have found a bug, please do not hesitate to contact me. You can find my contact details below.

Nicola Gilberti
UniTN MSc student
nicola.gilberti at studenti.unitn.it

Bibliography

- [1] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 3rd ed., 2000.
- [2] Wikimedia Foundation Inc., “*javac*.” Available on <https://en.wikipedia.org/wiki/Javac>.
- [3] Oracle, “*Reflection in Java*.” Available on [java.lang.reflect](#) package description.
- [4] R. Bharathan, *Apache Maven Cookbook*. Packt Publishing Ltd, 2015.
- [5] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, “Spoon: A library for implementing analyses and transformations of java source code,” *Software: Practice and Experience*, vol. 46, pp. 1155–1179, 2015.
- [6] M. Biagiola, F. Ricca, and P. Tonella, “Search based path and input data generation for web application testing,” in *International Symposium on Search Based Software Engineering*, pp. 18–32, Springer, 2017.
- [7] G. Fraser and A. Arcuri, “Evosuite: automatic test suite generation for object-oriented software,” in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE ’11*, (New York, NY, USA), pp. 416–419, ACM, 2011.
- [8] Oracle, “*LinkedBlockingQueue in Java*.” Available on [LinkedBlockingQueue<E> Docs](#).
- [9] Oracle, “*Random in Java*.” Available on [Random Docs](#).
- [10] Oracle, “*Enum in Java*.” Available on [Enum Docs](#).
- [11] Pivotal.io, “*Spring Framework*.” Available on [Spring Framework Docs](#).