Random Test Generator and Executor

for Java code

Project Report

Nicola Gilberti 198739

University of Trento
MSc Computer Science

Contents

Abstract

Nowadays, a considerably high amount of software is present in the commercial scenario. For this reason, the testing phase plays a crucial role for the software developing companies.
Thanks to new technologies, which are continuously facing those problems, other scenarios can be explored and tested accordingly using the code (to be modified). More specifically, the testing phase on web pages can be carried out making use of a graph, where each depicted node represents a state of the application, transitions correspond to links, buttons, text boxes or every kind of operation which can lead to a movement from an application state to another.
Despite the solution to the illustrated situation can be exploited with other SoA processes, web pages can be extremely demanding in terms of test generation.
In this document, a program able to create a test suite for a given application following a random testing approach is presented. This program pursues the main goal following these steps: first, an instrumentation phase is performed in order to insert extra code for analysis purpose, then test cases for the line-coverage/branch-coverage analysis are created.
The illustrated program has been coded using Java language, supporting Java for the representation of the graphs.

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 Javac [2], a Java compiler, 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, it is used an external library, called Spoon, that will be properly explained in the next section [1.1].

1.1 Spoon

Spoon [5] is a Java library that gives you the ability to create/modify the source code. Therefore, with this library, a programmer can transform or analyze Java code dynamically.
This routine takes advantage of the Java Reflection [3] technique to represent the Java abstract syntax trees (AST) of the code under analysis, but also to give a programming interface (intercession API) that let the coder modify/generate Java source code.
Moreover, it implements a native method to integrate and process Java annotation [6], to embed metadata into the code.
About the code that a programmer can insert, Spoon give different option to check its correctness, in order to avoid troubles during compile time.
The first method leverage on generics to manipulate the AST that give to the programmer a feedback in case of bad code.
Another option that Spoon implement is the template engine with static-checks that let the programmer insert code ensuring automatically its correctness.
In case of those technique are not used, you can still check the well-formedness analyzing the stack trace error at compile time.

How it works

Spoon works primarily on the AST, giving elements to analyze and modify the syntax tree.
In fact, the *Processor* and the *Factory* features are the key-components to use to examine and to transform the code.
The Processor let the programmer analyze the AST, while the Factory permit to modify the AST, adding and/or removing elements from the syntax trees.
The concept is similar to the read and writes operation, where the first read-only

while the second can also write.
In particular, *Processor* is utilised for the analysis and the querying onto the code. Those operation are possible thanks to the visit pattern applied to the Spoon model. All the elements have an *Accept* method that give the permission to be visited by a visitor object.
For instance, in the example 1.1 a Processor is used to analyze the code, searching for empty catch block. As we can see, the Processor works on a CtCatch, which is the Compile Time Catch given by the Spoon Metamodel, and check for the Statments inside the CtCatch body.
As shown before, to generate new elements, a *Factory* instance is necessary.

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

Listing 1.1: Processor example taken from Spoon documentation

Factory give the coder the ability to create new elements, fill their data and add them to the Syntax Trees under analysis. There are more than one Factory, where each one is specialized to facilitate the creation of elements. As shown before, to generate new elements, a *Factory* instance is necessary.

```java
Factory factory = this.getFactory();
String snippet = this.getLogName() + ".testOut(Thread.currentThread())";
CtCodeSnippetStatement snippetFinish = factory.Code().
    createCodeSnippetStatement(snippet);
CtBlock finalizerBlock = factory.Core().createBlock();
finalizerBlock.addStatement(snippetFinish);
ctTry.setFinalizer(finalizerBlock);
CtBlock methodBlock = factory.Core().createBlock();
methodBlock.addStatement(ctTry);
element.setBody(methodBlock);
```

Listing 1.2: Factory example taken from Spoon Projects

In this example, a Factory object is used and different method, in this case only *Core* and *Code*, are utilised to specialize the factory on the needs.
It is evident, from case 1.2, that the *Code* method is necessary in creating code elements, called Snippet, while *Core* help creating blocks of code, that will eventually contain *Code* generated elements.

2. Test generation Algorithm

This chapter explain all the implementation choices and the key-points of the algorithm.

2.1 Master-slave architecture

The program execute different steps to achieve the final result, where each of them are performed by different 'workers'.
Specifically, the program use two distinct threads to pursue its goal.
This architecture balance the work-load on multiprocessor, increasing the efficiency. A similar approach is also used by EvoSuite [7], the counterpart of this project.
Although EvoSuite is a better choice because it perform more different operations, some of them are not required for the scope of the project. Thus, it is demanded a resource affordable project that perform only the required operations, avoiding not necessary tasks.
The algorithm use threads for specific purposes, defining distinct operation and scopes for each one. Firstly the main thread start, managing all the import from resources 3.2 to prepare the environment for the other thread.
In fact, the values imported need to be manipulated, to became effective elements. For example, class-path must be updated dynamically in order to recover all the classes necessaries in future steps.
The second thread begins after the allocation of the environment. This new thread, that can be seen as the slave, execute most of the complex algorithms and send messages to the master all the time it find something.
In fact, second thread instrument the input class 2.2, and generate tests on it 2.4.
Those tests are sent back to the primary thread, messages pass through a LinkedBlockingQueue [8], that check their validity to create 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 concurrent mistakes 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 explain the classes inside the package *spoon*. The main dependency for this package is Spoon, explained in Section 1.1.
The *ClassModifier* class prepare the Spoon environment. In detail, it generate all the variable that are necessary to create/modify elements in a java class.
The program need to insert extra code inside an existing class, so a *factory* object must be instantiated. In addition, Spoon requires a data model of the class under analysis, as seen in 1.1. This is generated by a Spoon Object that return, given the source-path, the AST where the root is the given path.

```
protected void buildModel(File sourcePath) throws IOException {
    builder = new JDTBasedSpoonCompiler(factory);
    try {
        builder.addInputSource(sourcePath);
        builder.build();
    }catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

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

Now the Spoon requirements are setted and the *SpoonMod* class, can be instantiated to effectively instrument the class under test.
The instrumentation depends on the scope of the analysis required. Program can generate extra code to pursue a line or branch coverage.
In each case, target is reached using an ArrayList [9] that collect the identifier of the lines/branches when the execution pass through them.
To work, the ArrayList has to be defined, instantiated and then it can be used to collect the required data.
The algorithm work with random choices, so the instantiation has to be done carefully.
To be sure that there will be no runtime exceptions, the program modify all the constructor of the class. In case the target doesn't implement a constructor, the empty one is still available and reachable from the code below 2.3.
After that, we need to place all the add operation on the arraylist where required.
In case of branch coverage, the instrumentor search for *If* structures, and add a snippet 2.2 in the first line of each block of execution, *then* and *else* blocks.

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

Listing 2.2: Snippet creation

As we can see, a counter is used to identify uniquely a branch.

In case of line coverage the snippet is the same, while the placing task no. In fact extra data are collected from 3.2 that provide the lines to cover.
The instrumentor does a bottom-up scan searching for the lines, and when there is a match it append the snippet shifting one line down the rest of the code.

```
private void modConstructors(CtClass cc) {
    List<?> tmp;
    CtConstructor<?> constructor;
    Set<?> constructors = cc.getConstructors();
    Iterator<?> itc = constructors.iterator();
    while(itc.hasNext()) {
        constructor = (CtConstructor<?>) itc.next();
        CtBlock<?> ctb = constructor.getBody();
        tmp = ctb.getStatements();
        List<CtStatement> newStatements = new ArrayList<CtStatement>();
        CtCodeSnippetStatement newStatement = factory.Code().
    createCodeSnippetStatement("checker = new ArrayList()");
        newStatements.add(newStatement);
        for(int j = 0; j < tmp.size(); j++){
            newStatements.add((CtStatement) tmp.get(j));
        }
        ctb.setStatements(newStatements);
    }
}
```

Listing 2.3: The function to add the ArrayList instantiation snippet

The SpoonMod class generate extra methods in the class under test for the run-time management tasks, like getting data from the ArrayList or resetting it for the next execution. The last operation that this class provide is an analysis of the class 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 con-

structor, 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 [10] 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.

```
case "java.lang.Integer":
    obj.setValue(random.nextInt());
case "java.lang.Float":
    obj.setValue(random.nextFloat());
case "java.lang.Long":
    obj.setValue(random.nextLong());
case "java.lang.Double":
    obj.setValue(random.nextDouble());
case "java.lang.Boolean":
    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.

```
case "java.lang.Short":
    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.

```
public static String generateRandomChars(String candidateChars, int length
    , Random random) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < length; i++) {
        sb.append(candidateChars.charAt(random.nextInt(candidateChars.
    length())));
    }
    return sb.toString();
}
```

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, *"ABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890abcdefhijklmnopqrstuvwxyz"*

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 [11].
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 ← params.next
4:          if isPrimitive(param) then
5:              param ← INSTANTIATEWITHREFLECTIONOROLDVALUE
6:          else if isEnum(param) then
7:              param ← PICKRANDOMENUMCONSTANT(param)
8:          else if isInterface(param) then
9:              param ← INSTANTIATE(param.getSubclass.getParameter())
10:         else
11:             if directlyInstantiable(param) then
12:                 param ← INSTANTIATEWITHREFLECTIONOROLDVALUE
13:             else
14:                 requireInstances ← INSTANTIATE(param.getParameter())
15:                 param ← INSTANTIATEWITHREFLECTIONOROLDVALUE(requireInstances)
        return params                                  ▷ 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.

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

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

The *ClassPathScanningCandidateComponentProvider* and *BeanDefinition* are taken from the Spring Framework [12].
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.

```
private static void AddToFinalTestCase(TestCase newTest) {
  HashSet<Integer> tmp1 = newTest.getBranchCov();
  boolean flag=true;
  if(finalTests.isEmpty()) {
    finalTests.add(newTest);
  }else {
    for(int i=0;i<finalTests.size();i++) {
      TestCase oldTest = finalTests.get(i);
      if(sameValues(oldTest.getBranchCov(),newTest.getBranchCov())) {
        flag=false;
        if(oldTest.getMethList().size()>newTest.getMethList().size()) {
          finalTests.set(i, newTest);
        }
        break;
      }else {
        if(newTest.getBranchCov().containsAll(oldTest.getBranchCov())) {
          finalTests.set(i, newTest);
          flag=false;
          break;
        }else if(oldTest.getBranchCov().containsAll(newTest.getBranchCov()
    )){
          flag=false;
          break;
        }else {
          tmp1.removeAll(oldTest.getBranchCov());
        }
      }
    }
    if(!tmp1.isEmpty() && flag) {
      finalTests.add(newTest);
    }
  }
}
```

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.

```
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.

```
public class Tester{

    @Test
    //test case 2 coverage: 0.07906976744186046
    //branch covered: [96, 161, 98, 163, 133, 201, 170, 171, 109, 206,
    112, 49, 178, 88, 121, 122, 156]
    public void test0(){
        main.ClassUnderTest obj = new main.ClassUnderTest();
        ...
```

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,
        var3913,var3914);
10  }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.

```
<profile>
    <id>windows_profile</id>
    <activation>
        <os>
            <family>Windows</family>
        </os>
    </activation>
    <properties>
        <toolsjar>${java.home}\..\lib\tools.jar</toolsjar>
    </properties>
</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.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-install-plugin</artifactId>
    <version>2.5.2</version>
    <executions>
        <execution>
            <id>install-external</id>
            <phase>install</phase>
            <configuration>
                <file>${toolsjar}</file>
                <repositoryLayout>default</repositoryLayout>
                <groupId>com.sun</groupId>
                <artifactId>tools</artifactId>
                <version>1.8.0</version>
                <packaging>jar</packaging>
                <generatePom>true</generatePom>
            </configuration>
            <goals>
                <goal>install-file</goal>
            </goals>
        </execution>
    </executions>
</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. The evaluation is also executed limiting the number

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			

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

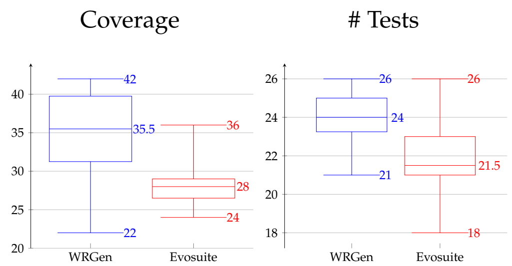
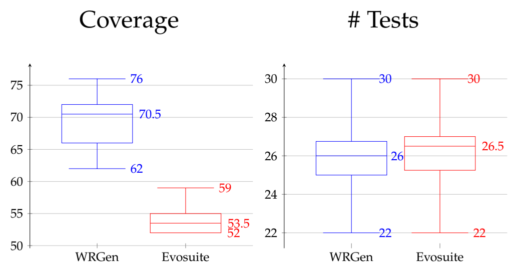


Figure 4.2: Retroboard Analysis



The first service,*Dimeshift*, everything point out, as before, 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 Wilcoxom 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

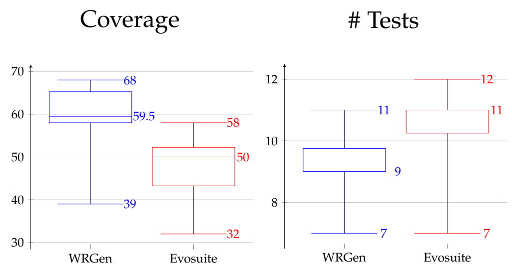


Figure 4.4: Splittypie Analysis

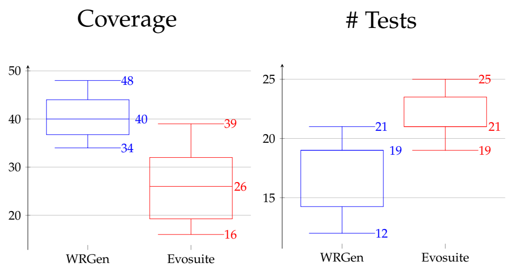
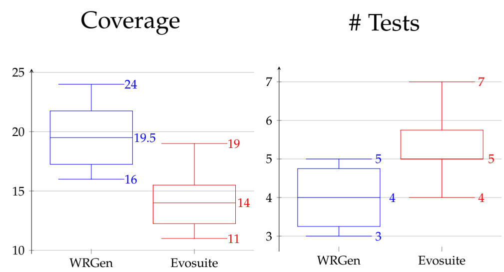


Figure 4.5: Pagekit Analysis



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

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] Oracle, "*Annotations*." Available on Annotations.

[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, "*ArrayList in Java*." Available on ArrayList<E> Docs.

[10] Oracle, "*Random in Java*." Available on Random Docs.

[11] Oracle, "*Enum in Java*." Available on Enum Docs.

[12] Pivotal.io, "*Spring Framework*." Available on Spring Framework Docs.