Random Test Generator and Executor

for Java code

Project Report

Nicola Gilberti 198739

University of Trento
MSc Computer Science

Contents

Abstract

Here is the Abstract.
I want a brief explanation of the whole project, using one/one and a half pages.

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.

```
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.

```
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 1.1, 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.
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.

2.2 Class Instrumentor

This section explain the classes inside the package *spoon*.

2.3 Primitive input generator

2.4 Test generator

2.4.1 Creation

2.4.2 Pretty Printing

3. Usage

How to use the program.
Maven command but also...

3.1 Project Object Model

POM.xml that define all the dependencies, how to set it properly.

3.2 Properties

What is a Properties file, followed by an explanation of the variables.

```java
File sourceFile2 = new File(root2, fileName+"Instr.java");
Files.write(sourceFile2.toPath(), k.getBytes(StandardCharsets.UTF_8));

String[] source = { "-d",destDirectory,"-cp",".;"+classpath,new String(
    filePath + "/" + fileName + "Instr.java")};
ByteArrayOutputStream baos= new ByteArrayOutputStream();
com.sun.tools.javac.Main.compile(source);
```

Listing 3.1: Testing how to print code!

4. 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 `https://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html`.

[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.