JUnit Per-Test-Case Coverage Tool

Research Thesis

Lawrence Yu

Introduction

Test cases reveal information about how a program has been executed. This can be useful for both failing test cases to help find and diagnose bugs, as well as in situations where we are trying to understand the behavior of the program. In black box testing, test cases are designed to test functionality without knowledge of the implementation details. Therefore, in order to extract useful information from these test cases, we need code coverage information, which shows what lines of code are either executed or not executed. To be able to differentiate the behavior of different test cases, we need to gather and compare the coverage information for each one. This information can help with both debugging and comprehension issues, as well as potentially allow programs to be more efficient in their debugging and their development.

There are currently two solutions for test case coverage information. The first is to use coverage tools that currently exist, which only provide aggregate coverage information for an entire test suite, not on a per test case basis. The second is to write a custom built scripting framework that starts up the existing coverage tool for one test case then brings down the environment, repeatedly, for every single test case.

The aggregate coverage information for an entire test suite is not sufficient because it does not allow one to differentiate the effects of each test case, so it is less effective at revealing behaviors of test cases. In the case of using custom scripts to achieve per test case coverage information, it is inconvenient for the developer, who is trying to understand the program and test case behavior, to have to manage the scripts not pertinent to their program. It is also

inefficient because running each test case requires all of the overhead that would normally be encountered once per test suite. This includes tasks such as starting and stopping the Java VM, as well as setting up and tearing down infrastructure, like databases and network connections. Needing to run into this overhead for every test case can prove to be very computationally expensive. Finally, the scripted approach does not allow for individual execution of "parameterized" test cases. Instead, the coverage of all individual test case parameterizations would be aggregated.

To overcome this problem, we extended an existing coverage framework to provide the desired per-test-case information. Our solution is effective in the sense that it provides the ability to differentiate the effects of each test case, unlike the aggregate version. It is convenient, unlike the scripted version, in that the user just runs a Java program and does not need to modify their test suites or manage a scripting framework, and per-test-case coverage is gathered. Thirdly, it is efficient because the startup and teardown of the whole system happens only once for the entire test suite, like when normally running the test suite. Finally, it handles parameterized test cases properly. We conducted a test to compare gathering per-test-case coverage information using the scripting framework approach and using our tool, and we observed that our tool executed the test cases over 100 times faster than the scripting framework.

With this tool in existence, we now have a convenient way to gather per-test-case information in a way that is easily accessible to and usable for people. The remainder of the paper will first give background information about JUnit testing and the JaCoCo code coverage framework. Then we present our approach to solving the problem, followed by empirical validation of the approach and possibilities for extending the tool in the future.

Background: JUnit

A central concept in our problem is unit testing, which is implemented by JUnit, a popular unit testing framework for Java. Typical usage of JUnit involves one or more test suites, each of which consists of many independent test cases. A simple example of a JUnit test suite and its individual test cases can be seen in Figure 1 below. Each test case typically focuses on one small piece of functionality, and information about which tests succeed and which tests fail usually gives an idea of what aspects of the program are working. Per-test-case code coverage would allow developers to gain a deeper understanding of the relationship between test cases and their code. For example, if testCase2 in Figure 1 failed, then a developer could look at the coverage information for testCase2 to aid their debugging.

```
public class TestSuite1 {

    @Test
    public void testCase1() {
        // Test case body...
    }

    @Test
    public void testCase2() {
        // Test case body...
    }
}
```

Figure 1: A JUnit test suite and its test cases

A major challenge when exploring existing tools for code coverage was that few of them were designed specifically for use with JUnit tests. Rather, most of them, such as Cobertura and Eclemma, focused more on code coverage for the entire execution of a program. Therefore, when attempting to apply these tools in a JUnit environment, the result was simply aggregate

coverage information spanning across all test cases; this was insufficient for the problem at hand. When looking at the code coverage Eclipse plugin called Eclemma, we found out that Eclemma was built on top of a Java code coverage library called JaCoCo. After investigating the documentation and goals of JaCoCo, it was clear that the way forward was to build a custom tool on top of this library.

Background: JaCoCo

JaCoCo is a flexible, lightweight Java library for producing code coverage information in Java VM environments. It provides a well documented API to produce and analyze different types of coverage information, such as instructions, lines, branches, methods and cyclomatic complexity. JaCoCo is designed for integration in different contexts, and proved to be an ideal starting point for our tool.

JaCoCo produces execution coverage information on the fly using a Java agent. When running a Java application, the jacoco.jar that is included with a JaCoCo distribution may be specified as the Java agent. The usual syntax for a running a Java application is seen below.

```
java -cp . MyProgam
```

In comparison, the syntax used for running the same Java application but with the JaCoCo Java agent is as follows.

```
java -javaagent:jacocoagent.jar -cp . MyProgam
```

Specifying the jacoco.jar as the Java agent causes JaCoCo to record coverage information and dump the data to an .exec file when the JVM exits. Various options may be passed along with the Java agent to override various jacocoagent.jar default options, such as the name of the output file and what Java classes to include or ignore when recording coverage information. The

following example shows the syntax for specifying the output file name and which classes to exclude in the code coverage.

```
java -javaagent:jacocoagent.jar=destfile=MyCoverageData.exec,
excludes=ClassA:ClassB -cp . MyProgam
```

All of the coverage information produced by JaCoCo is stored in an .exec file. JaCoCo provides classes and methods in its API to open and analyze the .exec file. The information that is stored in the file includes coverage information for each class, with coverage counters at different levels. These counters can range from Java byte code instructions to branches to source code lines. Provided in the API is the ability to read the .exec file, load all of the counters into memory, and also produce coverage reports in different formats such as HTML, CSV and XML.

Like other coverage tools, JaCoCo is not designed specifically for JUnit. However, since it is a library, the flexibility this entails provided the freedom to adapt JUnit in a way that the existing tools could not. Of particular note is the ability to make JaCoCo API calls at runtime, which made much functionality of the tool possible. These details will be discussed in the approach section.

Approach

To generate per-test-case coverage information, we needed to tie JaCoCo's code coverage collection with the execution of each individual JUnit test case. Normally, JaCoCo would not differentiate between test cases and collect code coverage information over the entire execution. Fortunately, JUnit provided the necessary hooks at the start and end of each test case for our tool to notify JaCoCo that different test cases were being executed. This is shown in Figure 2 below. What follows is a more detailed look at the Java classes used to implement this workflow.

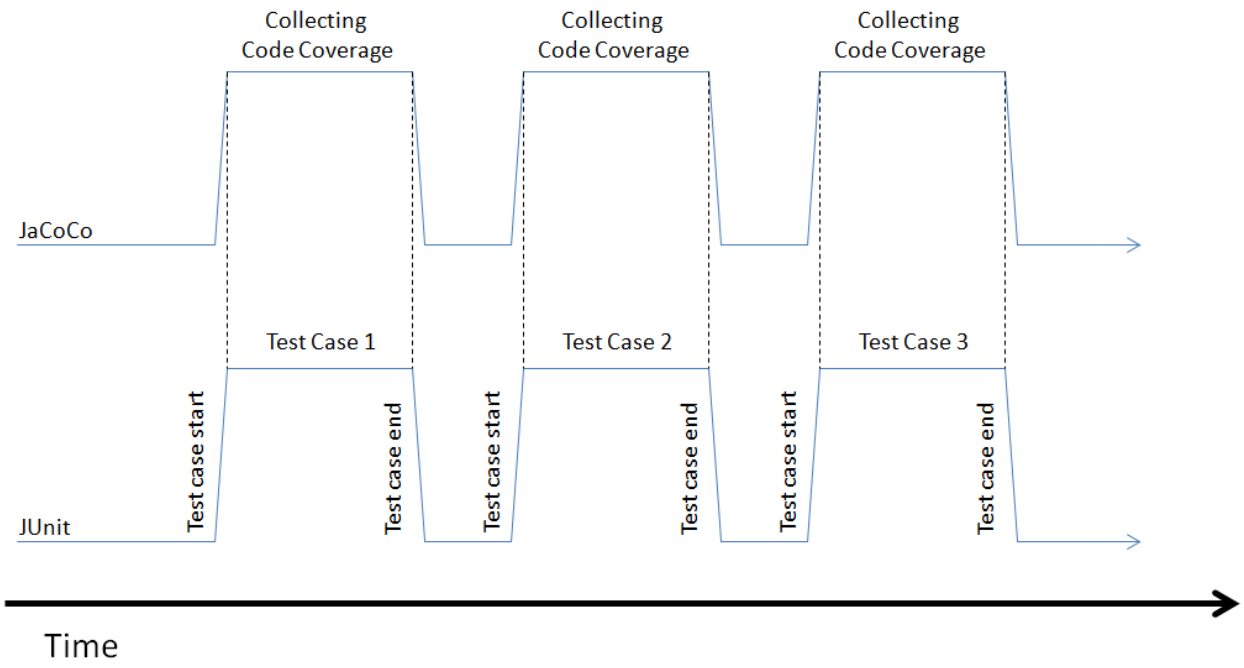

Figure 2: Timing diagram of JaCoCo and JUnit

The Java classes that were delivered are responsible for two separate workflows, generating the per-test-case coverage information, and reading the coverage information. The two classes that are responsible for generating the per-test-case coverage information are PerTestCaseCoverageRunner and JacocoListener. The two classes that are responsible for reading the coverage information are PerJUnitTestCaseVisitor and ExecAnalyze.

For generating the per-test-case coverage information, the main method resides in the PerTestCaseCoverageRunner class. This class is responsible for inserting our custom JUnit listener, JacocoListener, into the JUnit test execution. It first creates a JUnitCore instance, adds our custom listener to it, then begins execution of the desired JUnit test suites. The custom listener, JacocoListener, is responsible for generating per-test-case information during test case execution. As a child of JUnit's RunListener class, JacocoListener has methods that are called by

JUnit whenever a test case begins or finishes. By making calls to the JaCoCo API at these hooks in between test cases, the coverage information that would normally be aggregate for the entire test suite ends up being separated for each test case. After this first workflow, per-test-case coverage information has been gathered in the form of an .exec file.

The classes ExecAnalyze and PerJUnitTestCaseVisitor are responsible for reading an .exec file and extracting the coverage information for further analysis. The way that coverage information for multiple test cases is stored in a single .exec file is that each test case's information is stored as a session in the file. The PerJUnitTestCaseVisitor class implements JaCoCo's ISessionInfoVisitor interface, which means that it can implements the methods that are called whenever a new session is read in the .exec file. The PerJUnitTestCaseVisitor class maintains a map of SessionInfo objects to Collection<ISourceFileCoverage> objects. Therefore, each entry in the map is a mapping from one test case to all of the coverage information for that test case's execution. Whenever the PerJUnitTestCaseVisitor class reads the content of a session, it adds a new entry to the map. The ExecAnalyze class instantiates the PerJUnitTestCaseVisitor object and adds it as a visitor to use when reading the execution data. Once PerJUnitTestCaseVisitor returns the map to ExecAnalyze, ExecAnalyze iterates through the data in the map and prints out the coverage information to standard out.

```
Session "testTriple(ClassATest)": Sun Mar 02 15:49:11 UTC 2014 - Sun Mar 02 15:49:13 UTC 2014
ICoverageNode name: ClassA.java
       lineCounter: 1/4
instructionCounter: 4/15
      classCounter: 1/1
     methodCounter: 1/4
     branchCounter: 0/0
 complexityCounter: 1/4
         First Line: 2
          Last Line: 16
             Line 2:    0/3 (NOT_COVERED)
             Line 6:    4/4 (FULLY_COVERED)
            Line 11:    0/4 (NOT_COVERED)
            Line 16:    0/4 (NOT_COVERED)
```

Figure 3: An example of coverage information for one test case

Figure 3 above shows part of the output of ExecAnalyze method. We can see that the name of the test case is "testTriple", and that we are viewing the coverage information in the class ClassA. Figure 3 shows that in ClassA, the test case "testTriple" only executed line 6. This output is only a simple dump of the coverage information; the tool can be extended to do much more with the coverage information.

Compiling and running this tool is all easily done at command line. The recommended syntax for compilation is as follows.

```
javac -d /bin/ -cp .:lib/jacocoagent.jar:lib/junit-4.11.jar:lib/org.jacoco.core-
0.6.2.20130126-2101.jar src/PerTestCaseCoverageRunner.java
```

The same syntax is used for compiling ExecAnalyze.java.

Running the tool to generate per-test-case coverage information follows the following syntax.

```
java -javaagent:lib/jacocoagent.jar=destfile=bin/jacoco.exec,dumponexit=false -cp
bin:lib/jacocoagent.jar:lib/hamcrest-core-1.3.jar:lib/junit-4.11.jar Per
TestCaseCoverageRunner ClassATest
```

The first thing to note about the above command is that the jacocoagent.jar is being passed as the Java Agent, so that JaCoCo will automatically collect coverage information. Our tool makes calls to the JaCoCo API to cause it to collect coverage information for individual test cases. Provided as an argument to the main method is the name of the test suite that will be executed; in Figure 2 this is "ClassATest".

Lastly, the following is the syntax for dumping the information out of an .exec file.

```
java -cp bin:lib/asm-all-4.1.jar:lib/org.jacoco.core-0.6.2.20130126-2101.jar
ExecAnalyze
```

A Makefile is also provided with the source code for easier compilation and invokation. The user can also study the Makefile to better understand the syntax used.

Empirical Validation

Given this new technology, we wanted to validate that our tool was faster than preexisting solutions. Therefore, we conducted an experiment between our tool and one preexisting solution, which was the approach of using a scripting framework. Our independent variable was technique used, either our tool or the scripting framework. Our dependent variable was the execution time of all test cases. All other variables, including, but not limited to, the platform, processor and number of test cases, which was 2000, were controlled.

Our results are provided in table 1 below. As you can see, the scripting framework approach took approximately 7.5 minutes to finish executing all 2000 test cases, while our tool took less than 5 seconds to execute all 2000 test cases. Our tool experienced over a 100 times speedup compared to the framework approach. We believe that the primary reason for this execution time difference between the two techniques is that the scripting framework needs to start and stop the Java VM for each test case, while our tool only needs to start and stop the Java VM once for all the test cases. This result demonstrates that, our tool is faster than the preexisting solution of using a scripting framework.

Technique	Execution Time
Scripting framework (each test case individually)	7 minutes 29 seconds
Our tool (all test cases at once)	4 seconds

Table 1: Execution times for running 2000 test cases

Future considerations

The tool in its current state is very general and has a lot of room to grow or be extended upon. For instance, currently, after the tool generates the per-test-case coverage information, the data is still in JaCoCo's .exec file format. This file format is disadvantageous for our purposes because it must be parsed using JaCoCo's API, and that it is a binary file and is therefore not human readable. Extending the tool to convert the coverage information from an .exec file to a new, human readable file format would prove useful for integration with other tools.

Another possibility for extension lies in the visualization of the generated per-test-case coverage information. Users of code coverage information benefit greatly from being able to visualize the code coverage directly in the source code. This is even more so for per-test-case coverage information, as the ability to interpret and understand the coverage for various test cases would be greatly increased by visualization. For example, one feature specific to per-test-case coverage would be the ability to toggle source code visualization for specific test cases on and off, allowing the user to focus on specific test cases. This visualization could be implemented in two ways. One would be to create an Eclipse IDE plugin, while the other would be to create a standalone application using Java or HTML.

Conclusion

Unit test cases provide valuable insight into the behavior and inner workings of a program. While success and failure information for test cases is useful to some degree, code coverage information for individual test cases is very useful for developers and researches when trying to understand their programs. However, previous methods of producing per-test-case code coverage information, whether this was using existing tools or custom scripts, proved to be

inconvenient, inefficient and ineffective. We extended an existing code coverage information to provide a tool to produce per-test-case code coverage information that is both more convenient and efficient than previous solutions, and also allows for individual recording of parameterized test cases. With the existence of this tool, we now have an easy way to gather per-test-case code coverage information.