Test Splitter - Status Report[1]

Oguz Demir

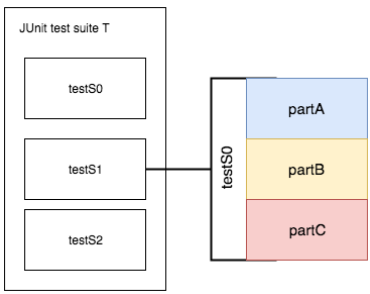
https://github.com/oguzdemir/Test-Splitter

June 26, 2018

This document describes the project status. Currently, the development is splitted into 2 branches. The "Master" branch contains the core functionality for test splitting using compile-time analysis and dynamic analysis (Section 1). The "Instrument" branch contains the core functionality for tracing test executions also using compile-time analysis and dynamic analysis (Section 2).

1. Master Branch

In the master branch development, our goal is to split JUnit tests into smaller parts while preserving the actual functionality. While the test code is splitted, the actual source code that is being tested is not modified. A JUnit test, which is defined as a Java method with *@Test* annotation, will be splitted into multiple methods from predefined or user selected points. Currently, we are considering test assertions as the split points and we are assuming that the assertions can only be at the outer-most block level, i.e, there may not be any assertions inside branches, loops, or calls to helper methods. Consider the following diagram:



In this diagram, the testS1 has 3 parts (A, B, and C). Assume that each part ends with a test assertion. In this project, we are trying to create 3 different JUnit tests from parts A, B and C, which can be executed separately. The main goals of this aim are the following.

- The latter parts of the tests can be executed without spending time and resources for executing previous parts. For example, assume that parts A, B and C take 10s, 5s, 10ms respectively so testing part C requires 15 seconds of "preprocessing" while running C itself requires only 10 milliseconds. Also, it may be the case that when a change is made to the code-base, the change affects only part C, and parts A and B do not need to be re-executed for that change. Hence, it would be too inefficient to test part C together with part A and B.

[1]We gratefully acknowledge Kaiyuan Wang's help in the design and implementation of this project.

- The splitted tests may be transformed into parameterized tests so that they can be tested for various cases. Therefore, the test coverage can potentially be improved by spending minimal effort on test code.

To perform splitting in the current state of our implementation, we are recording the objects' states in the method scope and restore these states back at the beginning of splitted tests. For example, we record the states at the end of part A, and the part B starts with first recovering this states. We are currently using the XStream library[2] for serialization/deserialization of objects and the serialized information is stored on the disk. Here is the XStream representation of a sample SingleLinkedList object, which has size 2 and nodes 0 (header) and 1.

```
<object-stream>
  <TestSplitter.Samples.Sample1.SingleLinkedList>
    <header>
      <next>
        <elem>1</elem>
      </next>
      <elem>0</elem>
    </header>
    <size>2</size>
  </TestSplitter.Samples.Sample1.SingleLinkedList>
</object-stream>
```

As mentioned, the test assertions are considered to be split points for now. In order to acquire the information of split points and variables that are accessible at each split point, the test code is parsed via JavaParser[3]. Basically, each variable definition in method scope is recorded and the set of variables defined up to the split point is taken into consideration. If there are consecutive assertions, the last one of them is taken as the split point.

For recording, a copy of original test method is created and code blocks for recording the object states are inserted. This copy test should be executed once to store the necessary information to disk for splitted tests. Then, splitted tests may be executed individually by restoring the states from disk. The code blocks for recording the object states are inserted before the split points, and it is assumed that the objects are not changed in test assertions. Each splitted test code(except the initial one) starts with restoring the previous state and followed by the next statements of previous assertions, and ends with the test assertion(s).

As an example, consider the following code snipped for *SingleLinkedList* class and related test:

```
public class SingleLinkedList{
    static class Node{
        Node next;
        int elem;
    }

    Node header;
```

[2] http://x-stream.github.io/

[3] https://github.com/javaparser/javaparser

```
    int size;

    public void add(int x) {
        Node n = new Node();
        n.elem = x;
        n.next = header;
        header = n;
        size++;
    }
}

public class SingleLinkedListTest {
    @Test
    public void testS0() {
        SingleLinkedList l = new SingleLinkedList();
        l.add(1);
        assertTrue(l.size == 1
                   && l.header.elem == 1
                   && l.header.next == null);
        l.add(0);
        assertTrue(l.size == 2
                   && l.header.elem == 0
                   && l.header.next.elem == 1
                   && l.header.next.next == null);
    }
}
```

In this example, the JUnit test, *testS*0 creates a *SingleLinkedList* object and asserts properties of this object's fields after 2 method calls. The current program generates the following code:

```
public class GeneratedTest {
    @Test
    public void testS0() {
        SingleLinkedList l = new SingleLinkedList();
        l.add(1);
        Transformator.ObjectRecorder.writeObject(1);
        Transformator.ObjectRecorder.finalizeWriting();
        assertTrue(l.size == 1
                   && l.header.elem == 1
                   && l.header.next == null);
        l.add(0);
        Transformator.ObjectRecorder.writeObject(1);
        Transformator.ObjectRecorder.finalizeWriting();
        assertTrue(l.size == 2
                   && l.header.elem == 0
                   && l.header.next.elem == 1
                   && l.header.next.next == null);
    }

    @Test
    public void generatedU0() {
        SingleLinkedList l = new SingleLinkedList();
        l.add(1);
        assertTrue(l.size == 1
                   && l.header.elem == 1
                   && l.header.next == null);
    }
```

```
    @Test
    public void generatedU1() {
        SingleLinkedList l = (SingleLinkedList)
                Transformator.ObjectRecorder.readObject(1, "1");
        l.add(0);
        assertTrue(l.size == 2
                    && l.header.elem == 0
                    && l.header.next.elem == 1
                    && l.header.next.next == null);
    }
}
```

The copy of actual test method *testS*0 is generated and code blocks for recording the object states are added before each split point. For this example, the only accessible object is the *SingleLinkedList l*. In the *generatedU*1, the previous state of the object is loaded back before continuing from where it is left of at first part. When the *generatedU*1 is being executed, the generated *testS*0 should have been executed at least once to store the state information needed for the test *generatedU*1.

To sum up, the current process is the following:

1. The source code of the test is parsed with JavaParser
2. From the source code, variable declarations and test assertions are collected.
3. For each test assertion, the code for recording the object states is added to the main test for each reachable object reference. Currently, we are assuming that all the tests assertions are located at the outer-most block level (no branching or loops) in the original test methods.
4. The test code is splitted to generated tests based on each test assertion point. At the beginning of each generated test, the code for recovering the reachable object references is added.
5. Once the modified version of the main test is run, the objects are recorded into disk and generated tests are ready for execution.

For a desired test, the program can be used as following:

```
mvn compile exec:java \
   -Dexec.mainClass="TestParser" \
   -Dexec.args="path-to-file JUnitClassName MethodName"
```

2. Instrument Branch

In Instrument Branch development, we instrument test and source-code, and do dynamic analysis over the test runs. Currently, we do our analysis based on the method calls, and we are tracking which methods are called in which order. Using this trace knowledge, crucial information about test method can be acquired such as calling frequency of the source methods, number of method calls in each method, methods that are safe to split or not etc. Based on the assumption that the methods that do not lead to an external operation (file IO, web communication etc.) are safe to split, we are working on marking the split-safe

methods according to their call stacks. Leading to an external operation makes a method unsafe to split because the state of external source is unknown during test execution.

As the first step of the instrumentation process, the *TestSplitter.TestMonitor* class is added to the classpath in order to control the instrumentation information. Then, a code block which includes a call to a static method of *TestSplitter.TestMonitor* is inserted at the beginning and end of the instrumented methods. In this method call, class name, method name and method description are passed to *TestMonitor*. With that information, *TestMonitor* will be able to execute one the following objectives according to user choice.

1. Determining the safe methods to split and printing them onto the console. (In progress)
2. Printing the execution trace into a file.
3. Printing the execution trace onto the console.

For instrumentation, the ASM library[4]is used. In order to add the *TestSplitter.TestMonitor* calls at the beginning and end of the methods, *visitCode* and *visitInsn* methods are overriden. Since this method call should be located before the last statement of a method, we are checking the "return" or "throw exception" statements and we are inserting the method calls before these statements.

```
//Beginning of each method
@Override
public void visitCode() {
    mv.visitLdcInsn(className
        + "#" + methodName + "#" + description );
    mv.visitMethodInsn(Opcodes.INVOKESTATIC,
        "TestSplitter/TestMonitor", "visitMethod",
        "(Ljava/lang/String;)V", false);
    super.visitCode();
}

//Determining whether this operation is end of the method or not.
//Then inserting the method call before the end of the method.
@Override
public void visitInsn(int opcode) {
    //Method exits
    if (opcode == Opcodes.IRETURN || opcode == Opcodes.FRETURN
        || opcode == Opcodes.ARETURN || opcode == Opcodes.LRETURN
        || opcode == Opcodes.DRETURN || opcode == Opcodes.RETURN
        || opcode == Opcodes.ATHROW ) {
        mv.visitMethodInsn(Opcodes.INVOKESTATIC,
            "TestSplitter/TestMonitor", "finalizeMethod", "()V",false);
    }
    super.visitInsn(opcode);
}
```

When a test method is executed with this instrumenter , our *TestSplitter.TestMonitor .visitMethod*(*String*) and *TestSplitter.TestMonitor.finalizeMethod*() methods are called at the beginning and end of each method. Recall that the *finalizeMethod*() does not need to include the method information as the finishing method will always be the lastly called

[4]https://asm.ow2.io/

method.

In the test monitor, the information passed is used for keeping the execution trace or determining the safe methods.

2.1. Execution Trace Information

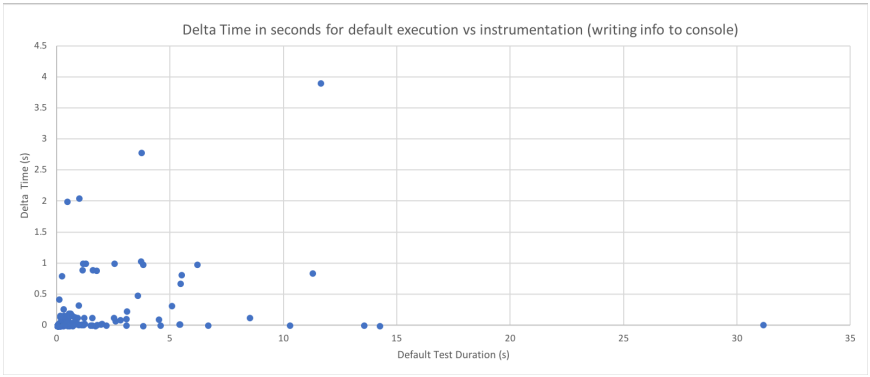
The method information which is passed to $visitMethod(string)$ is either printed directly to the console or printed to a file by a $java.io.FileWriter$ object stored in a static field in TestMonitor class. The writer is initialized with the first method name written to the file and closed with a shutdown hook which is defined in instrumenter in order to prevent data loss. Here is a code block from a class and execution trace of this method (Java, JUnit and Maven classes are not instrumented):

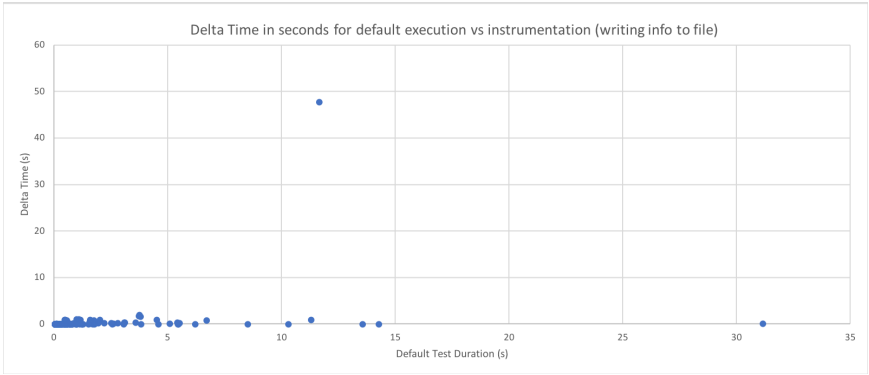
```
public class BankAccount {
    public int accountId;
    public String owner;
    public int accountBalance;
    public void addMoney(int amount) {
        accountBalance += amount;
    }
    public void withdrawMoney(int amount) {
        accountBalance -= amount;
    }
}

public class BankAccountTest {

    @Test
    public void testS0() {
        BankAccount account = new BankAccount();
        account.owner = "SampleOwner";
        account.accountBalance = 0;
        account.accountId = 1001;
        account.addMoney(599);
        account.withdrawMoney(100);
        assertTrue(account.accountBalance == 499);
    }
}

Execution trace:

   TestSplitter.Samples.Sample2.BankAccountTest.<init>.()V
   TestSplitter.Samples.Sample2.BankAccountTest.testS0.()V
      TestSplitter.Samples.Sample2.BankAccount.<init>.()V
      TestSplitter.Samples.Sample2.BankAccount.addMoney.(I)V
      TestSplitter.Samples.Sample2.BankAccount.withdrawMoney.(I)V
```

For calculating the overhead for collecting the execution trace information, we analyzed the running time impact of instrumentation for the Activiti[5]project. Our results show that the effect of instrumentation to runtime is relatively low. Only in one test, namely *org.activiti.engine.test.api.v6.Activiti6Test*, the number of method calls are so huge that

[5]https://github.com/Activiti/Activiti

it caused an unexpected slowdown (Default time: 11.612s, Printing to console:15.527s, Printing to file: 59.446s) since the file size becomes more than a gigabyte. In total, executing the test with printing to console option takes 269 seconds, while the default execution takes 258 seconds. Here are the charts for time differences for both options:





2.2. Split-Safe Methods

This section is currently in progress and our final goal is to determine the methods which are safe to split. With the visitMethod and finalizeMethod methods, a call stack similar to Java's call stack is constructed under TestMonitor. In this call stack, whenever an external operation method is called such as *FileReader.read*(), all the methods in the call stack are determined as unsafe. Here is an example of a Java class and safe - unsafe methods.

```
public class MathOps {
    public void allOperations() throws Exception{
        int a = readFromFileSquare();
        int b = givenSquare(1);

    }
    public int readFromFileSquare() throws Exception{
        BufferedReader br = new BufferedReader(new FileReader("somePath.txt"));
        return Integer.parseInt(br.readLine());
    }
```

```
    public int givenSquare(int x) {
        return x*x;
    }
}
```

In this example, the only safe method to split is *givenSquare*(*int*), the other methods *readFromFileSquare*() and *allOperations*() lead to external operation as reading data from file.

The running time analysis will be done for determining the split-safe methods after the implementation is completed.

2.3. Usage Information

The agent can be attached to a Java execution with following command line arguments:

```
-Xbootclasspath/p:\
    /path/Test-Splitter-1.0-SNAPSHOT-jar-with-dependencies.jar   \
    -javaagent:\
    /path/Test-Splitter-1.0-SNAPSHOT-jar-with-dependencies.jar:<option>
```