Team NBA

Project Sigmah: Testing Framework

Noah Griffin, Brian Steele, and Andrew Miller

Table of Contents

Deliverable #1 An Introduction to the Project

- Choosing a project: Sigmah
    - For our testing framework project, we were instructed to select a project from a list of HFOSS projects. After assessing our team's strengths and weaknesses as related to software engineering and with which language we felt most comfortable coding with, we selected the project Sigmah.
    - Sigmah is essentially an open source information management software. Sigmah is backed and funded by several French companies who funded this software as an international aid handler, if you will. With the surplus of data that comes in from international aid projects, this software will help with ease of access and consistent data management.
    - Sigmah is coded in Java with an SQL database. What luck for our team as two out of our three members have just recently had a course in using databases with SQL, and we all three would have preferred to code in Java. This paired with one of our member's previous knowledge and experience with scripts, we felt as if we were one step ahead of the game by choosing project Sigmah.
- Cloning and Building Sigmah
    - When we first encountered problems building Sigmah, initially, we thought we had just missed a step somewhere in the instructions provided with the Sigmah API. However, upon further investigation we found out what the real problem was. It seems as if there are several different sets of instructions for how to compile and build Sigmah in several locations. Somehow in the mix of following these instructions over a course of a few days we had mixed up our instruction sets. Clearly we should have kept better track of which instruction set we had used, but at the time we were unaware that there was more than one. For the future, we would recommend that the developers of Sigmah be more specific and consistent with their instruction sets, as one set found on their GitHub was different from an "equivalent" set found on their website. Consistency is key in a situation where you have an open source software that you have a need to be committed to, otherwise no one will be able to contribute.
- Testing and Results
    - Sigmah came pre-loaded with their own testing framework accessible in the source files. We ran their tests, or rather their tests were run automatically upon completion of the build and all of their test cases outputted a passing result. We were then able to visualize what our final project would have to entail by examining how they approached their own testing framework.

Deliverable #2 The Testing Process

- Testing process
  - The testing process will begin with manual testing, during which individual methods will be tested for expected outputs via the terminal. After our initial approach here with five test cases, we will then be able to expand our testing selection to encompass a larger part of Sigmah to ensure that all of their source code performs as it is intended.
- Requirements traceability
  - truncate (Number n): This method truncates the parameter input, n, to two decimal places.
  - truncate (Number n, int decimals): This method truncates the parameter input, n, to parameter input, decimals, length.
  - truncateDouble (final Double value): This method truncates the parameter input (Double), value, to two decimal places. This method calls truncate.
  - ratioAsString (Number n, Number in): This method returns parameter input, n and in, as a ratio in the format of a percent. This method calls ratio and truncate.
  - ratio (Number n, Number in): This method returns a ratio as a double from the parameters n and in.
- Tested Items
  - The tested items consist of several methods within the NumberUtils.java file. Some of these methods to be tested are two instances of truncate, truncateDouble, ratioAsString, and ratio. Each of these methods performs a crucial task to ensure that user inputs match the expected input, and if not they are truncated to match.
- Testing schedule
  - October 3 --> Specified 5 test cases (*.txt).
  - October 10 --> Performed our 5 test cases with Sigmah's code (standalone).
  - October 17 --> Ran into dependency issues with our Drivers.
  - October 23 --> Solved dependency issues with our Drivers, finalized scripting code to run all 5 test cases.
  - October 31 --> Have 5 test cases plus testing framework operational.
  - November 7 --> Have 15 test cases developed.
  - November 14 --> Have 25 test cases developed and ready for presentation.
  - November 21 --> Have 5 fault injections ready that will cause at least 5 tests to fail, but not all; ready for presentation.
  - November 28 --> Have final write up, PowerPoint, and poster ready as well as the 25 test cases ready to present.
- Test Recording Procedures
  - The test recording procedure states that tests must be systematically recorded. As of this deliverable we have specified 5 of our test cases and we currently plan to have the outputs (pass or fail) be recorded to a text file of some sort with details of the test case and whether it passed or failed. Each of our test cases will be in a text file with the test case number, the purpose of the method tested, the name of the class tested, the name of the method tested, sample input, and expected output. Our script

will read in from this text file and perform the test, comparing the actual to expected output and reporting whether the test case passed or failed.

- Hardware and software requirements
  - Java JRE, Java JDK 1.6, PostgreSQL 9, Maven 3, and Git 2.7 are required to compile and run the Sigmah software.
- Constraints
  - Our constraints in this testing process will be mostly due to time constraints with our group members. All three of us are seniors here at the college and we have very different schedules. Between homework and projects for all of our other classes, and one of our group members having two kids and a part-time job, we have already experienced issues with finding a time to schedule meetings. However, we have already made plans to accommodate these constraints by having a very in-depth and regularly updated plan to split up the workload and meet up remotely if necessary. Some other restraints that are worth mentioning are the limitations of the size of our group, as such we are limiting our testing framework to 25 test cases for now.
- System tests
  - Test Case 1
    - Test #0001
    - This method truncates the parameter input, n, to two decimal places.
    - NumberUtils.java
    - truncate(Number n)
    - 3.883
    - 3.88
  - Test Case 2
    - Test #0002
    - This method truncates the parameter input, n, to two decimal places.
    - NumberUtils.java
    - truncate(Number n)
    - 3.888
    - 3.88
  - Test Case 3
    - Test #0003
    - This method truncates the parameter input, n, to two decimal places.
    - NumberUtils.java
    - truncate(Number n)
    - 3
    - 3.0
  - Test Case 4
    - Test #0004
    - This method truncates the parameter input, n, to two decimal places.
    - NumberUtils.java
    - truncate(Number n)
    - string
    - java.lang.NumberFormatException: For input string: "string"
  - Test Case 5
    - Test #0005

- This method truncates the parameter input, n, to two decimal places.
- NumberUtils.java
- truncate(Number n)
- 3.88
- 3.88

Deliverable #3 Automated Testing Framework

- Experiences
  - As per our constraints that were detailed in Deliverable 2, we have our limitations as far as meeting up in person outside of class. However, we did manage to meet up twice since the last deliverable. Our main issue moving forward are dependencies. Sigmah has thousands of dependencies throughout our code, which we are finding is creating issues when building specific java files. Imports and packages in scripts are new to us so we faced several issues with this. In short, we figured it out but in a cumbersome way using jar files and we chose to only test classes that had minimal dependencies for now. The way we understand dependencies is that it's sort of like importing something that you require, but that has already been done. So, on the surface, it seems as if much of the code that Sigmah requires was just repurposed from something else which begs the question if it is really necessary or if they are all just a series of shortcuts that were taken.
- How-To Documentation
  - Requirements: Linux OS (Ubuntu 16.04.3 LTS), Java 1.8, Git
  - Clone GitHub Repository:
    - → git clone https://github.com/CSCI-362-01-2017/NBA
  - Navigate to the TestAutomation file,
    - → cd TestAutomation/scripts
  - Execute scripts/runAllTests.sh to run the source framework and open the html file in the default browser.
    - → ./runAllTests.sh
  - Execute scripts/runAllTests.sh to run the faulted framework and open the html file in the default browser.
    - → ./runAllFaultedTests.sh
- Architectural design
  - This testing framework is built using classes from Sigmah's source code. We are using a bash script, runAllScripts.sh, to create our automated framework. The script reads in all test case files, as they all follow the same format, then tests these test cases. Our test case files are stored as "testCase0#.txt" so that many more can be added; it leaves the framework open to additional tests. We have a Java Driver for each method that we are testing, which is compiled and ran with our script. The script is essentially the director of our testing framework, the brains if you will. The script compiles the driver, then runs the driver with the correct parameters which it reads in from the test case files. The output of the driver is then recorded in a report to show whether or not the test has passed or failed.
- File Structure

```
/TestAutomation
        /project
                /src
                        FaultedNumberUtils.java
                        NumberUtils.class
                        NumberUtils.java
```

```
            CleanNumberUtils.java
            FaultedNumberUtils.java
    /scripts
            ratioAsString_NumberUtilsDriver.java
            ratio_NumberUtilsDriver.java
            runAllTests.sh
            truncateDouble_NumberUtilsDriver.java
            truncate_NumberUtilsDriver.java
    /testCases
            Format.txt
            testCase01.txt
            testCase02.txt
            ...
            testCase24.txt
            testCase25.txt
    /testCasesExecutables
            ratioAsString_NumberUtilsDriver.class
            ratioAsString_NumberUtilsDriver.java
            ratio_NumberUtilsDriver.class
            ratio_NumberUtilsDriver.java
            truncateDouble_NumberUtilsDriver.class
            truncateDouble_NumberUtilsDriver.java
            truncate_NumberUtilsDriver.class
            truncate_NumberUtilsDriver.java
    /temp
            .gitignore
    /oracles
            .gitignore
    /docs
            README.txt
            TeamProjectsSpecifications.pdf
    /reports
            NBA-TestReport.html
```

❖ Test Cases

- ➢ The template for our test cases are as follows:
    - ■ <test case ID>
    - ■ <description of the method's purpose>
    - ■ <class file name *.java>
    - ■ <method name with any parameters>
    - ■ <input>
    - ■ <expected output>
- ➢ Test Case Format Description
    - ■ The test case ID is just the number of the test case, starting at 00 and ending at 05 (for now) it leaves the potential for endless test cases to be added.
    - ■ The description is what the method itself is actually doing.

- The class file name is the name of the java class file in which the method is contained.
- The method name is the name of the function within the class that is being tested. Parameters (input) should be listed here as well.
- Input is what the user enters when this method is called.
- The expected output is what should be returned when this method is called with this specified input.

Deliverable #4 Completed Automated Testing Framework

- Experiences
  - For this deliverable, our primary goal was to make any changes that were requested by the customer and to continue testing in earnest. We were told to update our script to use relative class paths instead of absolute class paths, in order to ensure that it would work on a different machine. We updated our HTML output file to include a table to display our output data horizontally instead of being text based and vertical. This will increase readability of our testing framework if we had a larger test case base. From Deliverable 3, we had compiled our Drivers using a jar file for our imports and dependencies, this is something we were asked to fix. To go from a jar file as an import, we just had to change the code in our script to make this change. At first it was giving us a hard time but we eventually figured it out.
- Testing Framework Changes
  - We changed classpath dependencies to be relative so that they will run successfully on other machines. We finalized our test report outputs, the html file, to show our outputs in a table to make it more readable. That way if we added say, ten thousand more tests, it would be easy to read through them all in a single html file, rather than have them listed vertically and almost impossible to search through.
- Test Case Samples
  - We did not change our test case format. Our last deliverable showed our first five test cases, all of which were performed on one method inside of one class. Here are a few other test cases that are shown to test a separate method in the NumberUtils class. We now have 25 test cases for our testing framework. The test case format has been left below to aid in understanding each sample test case:
    - <test case ID>
    - <description of the method's purpose>
    - <class file name *.java>
    - <method name with any parameters>
    - <input>
    - <expected output>
  - Test Case #6
    - Test #0006
    - Returns a ratio from a number and decimal ratio.
    - NumberUtils.java
    - ratio(Number n, Number in)
    - 2,0.0
    - 0.0
  - Test Case #10
    - Test #010
    - Returns a ratio from a number and decimal ratio.
    - NumberUtils.java
    - ratio(Number n, Number in)
    - 1,1.5

    - 66.66
- Test Case #13
    - Test #013
    - Returns ratio as a String percentage.
    - NumberUtils.java
    - ratioAsString(Number n, Number in)
    - 2,0.5
    - 400.0 %
- Test Case #21
    - Test #021
    - Returns a truncated number.
    - NumberUtils.java
    - truncate(Number n, int decimals)
    - 2.1234,2
    - 2.12

Deliverable #5 Fault Injection

- Experiences
  - The fault injections were fairly straightforward in that we had to make a small somewhat unnoticed change in the source code that would cause that method to fail a previously passed test. We injected five faults into the source code. The first fault caused all tests for truncate (Number n) to fail. This method is hard coded to truncate to two decimal values, we changed this "2" to a "3" and now it truncates to three decimal values while the expected is still two.
  - Our second fault was to truncateDouble (final Double value) and this fault also caused all tests to fail. This fault is a bit more serious considering that it will not output a number at all now. This method first checks to see if the input value is null, and if so it output null (because you cannot truncate a null value). It uses: if (value == null), which we have changed to: if (value != null), which causes it to now attempt to truncate a null value, which will throw an error and also will output null if the input is actually a valid number.
  - For our third, fourth, and fifth fault we changed the method truncate (Number n, int decimals). The third fault changed a logical operator in an if statement that checked to see if the decimal value was positive or negative. We changed: if (decimals < 0) to if (decimals > 0) which now causes all positive decimal values to trip this if statement and throw an illegal arguments exception while a negative decimal value will then proceed to be truncated. For our fourth fault, we changed a negative one to a one which should cause at least 3 of the tests for truncate to fail. The fifth and final fault we simulate a simple typographical error, changing a "1" to a "2" in an integer value. This change will throw off the expected output by 1.
- Injected Faults
  - NumberUtils.java -> truncate (Number n)
    - Tests #001-005
    - Line 49, Source code: return truncate (n, 2);
    - Line 50, Fault injection code: return truncate (n, 3);
    - Expects a double truncated to 2 decimal places.
    - Tests #1, #2, #3, #5: FAIL
    - Test #4: PASS
  - NumberUtils.java -> truncateDouble (final Double value)
    - Tests #015-020
    - Line 62, Source code: if (value == null)
    - Line 63, Fault injection code: if (value != null)
    - Looks for a null input parameter. Now truncates null, and returns null on valid input.
    - Tests #16, #17, #18, #19: FAIL
    - Tests #15, #20: PASS
  - NumberUtils.java -> truncate (Number n, int decimals)
    - Tests #021-025
    - Line 85, Source code: if (decimals < 0)
    - Line 86, Fault injection code: if (decimals > 0)

  - Expects to throw error if decimal value is negative, and proceed if positive. This fault flips that and now throws error for positive and proceeds if negative.
  - Tests #21, #23, #24, #25: FAIL
  - Test #22: PASS
- NumberUtils.java -> truncate (Number n, int decimals)
  - Tests #021-025
  - Line 100, Source code: if (index == -1)
  - Line 101, Fault injection code: if (index == 1)
  - Checks to see if the double has a -1 index value, then returns a truncated double as a string. The fault causes this to occur when the index value is 1.
  - Tests #21, #23, #24, #25: FAIL
  - Test #22: PASS
- NumberUtils.java -> truncate (Number n, int decimals)
  - Tests #021-025
  - Line 112, Source code: final int last = index + 1 + decimals;
  - Line 113, Fault injection code: final int last = index + 2 + decimals;
  - DESCRIPTION HERE
  - Tests #21, #23, #24, #25: FAIL
  - Test #22: PASS

Deliverable #6 Reflections

- Software planning
  - All three of us had prior experience with the workflow of a software engineering team with our Software Architecture and Design class the previous semester. In this class, we learned how planning is paramount in software design and that even with an amazing plan and design architecture it can change over the course of development in the software process. The same goes for testing, this aspect of the process that we have learned about this semester. We learned how important it is to maintain a functional testing framework throughout the development of the software project so that you know that the code has been well tested since creation.
  - Our group this semester suffered serious time constraints and availability in group meetings. We conquered this constraint by careful delegation of work that played to each of our individual strengths. Andrew's strength lies in scripting as he has some prior experience dealing with them. Noah's strength lies in keeping dependency directories in check as well as keeping up with customer relations. Brian's strength lied in structure and making sure everyone was up to date with ongoing deadlines and ensuring that certain frameworks upheld the customer's expectation.
- Constant customer interaction
  - Having customer (Dr. Bowring) interaction throughout the development of the testing framework was very helpful in ensuring that we were developing exactly what the customer wanted. At first we struggled to deliver what the customer wanted because we did not communicate as well as we should have. After our first deliverable we made sure we knew exactly what the customer wanted to ensure that we did not lose the contract.
  - Customer interactions are vital to a software project, even in the testing framework. Without this you could proceed through the planning process without ensuring that the code performs up to the customer's standards. Before this project, we had somewhat limited experience with this; moving forward we all three now realize the paramount importance of open communication with the customer.
- Bash Scripting
  - In the college's computer science curriculum there is only one class in which we have worked with a Linux platform and used bash scripting. That being said, it was very limited as most of the class was limited to C programming using bash to compile and run our programs. Needless to say that this project was the most bash programming that any of us had ever experience and it was a great learning experience. From generic script creation to creating a script that will compile and test Java code, we have tackled bash scripting thanks to this project.
- Open-Source Quality
  - This project has opened our eyes to the openness of open-source software. This was most ours first experience with open-source software and we didn't know what to expect. Our first troubles experience with the Sigmah project were the instruction sets provided. We found 3 different sets of instructions, one for users, one for administrators, and one for developers. On the surface this would seem to make sense, but after proceeding with the developer instruction set, we saw how poorly

crafted it was. On one seamless document of instructions for Mac, Linux, and Windows users, there were links for specific sets of the instructions that were "uniform" for all three platforms that linked back to the Windows set of instructions. This was confusing at first and caused a bit of delay while setting up the Sigmah software.

- ➢ Our second trouble was in the high number of dependencies in the Sigmah source code. While this created a serious pain for us while compiling our script and ensuring that our class paths were correct, one would beg the question: Why so many? We understand that open-source software is just that, open-source but there were hundreds of dependencies scattered throughout the source files and after a while it just seemed entirely unnecessary and honestly, a bit rushed and poorly done.

❖ Conclusion

- ➢ To be completely honest, we had no idea what a testing framework was until we took on this project. After learning about the concepts of a testing framework, and what test cases and drivers actually were, things started to make sense. We all three now feel comfortable saying that if required to do so by our jobs we could implement a testing framework and test a software proficiently. Our experiences and conquered troubles in this project have more than qualified us as Testing Engineers.