Automated Testing Framework

for

OpenMRS

A Software Engineering Project

Designed and Completed by

Daniel Hurlburt, Avery Douglas, and Benjamin Buckwalter

Table of Contents

Overview

This project involved the creation of an automated testing framework to be run on a certain existing H/FOSS project. We chose to work with OpenMRS, which is an open-source Medical Record System that can be used for the storage, update, and reference of medical records. This is of particular importance today as we are transitioning into a digital society. It facilitates the transfer of medical records as patients move between doctors and states, and also helps with diagnostics.

The testing framework we created is designed to run a series of user-created tests on the OpenMRS software. A more detailed description of the particulars of the framework is given in the *Testing Framework* section, however here we make mention that the testing framework functions by running a series of test cases which were written by the user. The results of these tests are recorded and displayed in an HTML page.

The testing framework is designed to be used in a Linux OS. There are no precise system requirements. The testing is done at run-time, test case by test case.

TESTING FRAMEWORK

The testing framework has the top-level directory of TestAutomation. Below is a description of the various directories within the TestAutomation directory and their proper use

Scripts

The *scripts* folder contains the *runAllTests.sh* script, as well as any other scripts necessary to the execution of the testing framework. The *runAllTests.sh* script will remove the old reports from previous tests and create a new html report. Once all the test cases have been run, the script opens the html file in a browser.

testCaseExecutables

The *testCaseExecutables* folder contains all files needed to run the test cases with the *runAllTests.sh* script. These will be various drivers. Included is the *CohortUnionDriver.java* file, which facilitates testing the union function of the Cohort class. It transforms the array inputs and outputs which are typed in the file into Integer arrays which can be used in java and calls the union method and sends the output of the test to the script.

project

The *project* folder contains the source code from the OpenMRS system.

reports

The *reports* folder contains an HTML file named *report.html*. This file contains the Test ID, Requirement, Inputs, and Expected lines from the test case and further indicates whether or not the test was passed or failed.

docs

The *docs* folder contains a txt file named *README.txt*. This will describe various aspects of the project and testing framework

The test cases are described in great detail in the following section. Below is a representation of the testing framework.

```
/TestAutomation
        /project
                /src
                /bin
                /...
        /scripts
                runAllTests.sh
                other helper scripts ...
        /testCases
                testCase1.txt test, Case2.txt ...
        /testCasesExecutables
                testCase1...
        /temp
        /oracles
        /docs
                README.txt
        /reports
                report.html
```

TEST CASES

The requirements of various aspects of the project are tested using test cases. Each test case will be in its own test case file and will test only one requirement. The filename for the test cases should be testCase#.txt, where # is replaced by the Test ID number of the particular test case. The test cases will be in the *testCases* folder and should have all of the following lines in this order:

Test ID: *the ID unique number of the test*
Requirement: *the requirement being tested in the test case*
Component: *the component being tested in the test case*
Method: *the signature of the method being tested in the test case*
Inputs: *the inputs for the test case*
Expected: *the results after the test*
Driver: *the name of the driver class without the extension*
Notes: *any notes or additional information about the test case*

Below is an example test case.

Test ID: 1
Requirement: The union of two Cohorts has no repeated values when the intersection of the inputs is non-empty
Component: Cohort
Method: public static Cohort union(Cohort a, Cohort b)
Inputs: [1,2,3,4,5] [3,4,5]
Expected: [1,2,3,4,5]
Driver: CohortUnionDriver
Notes:

FAULT INJECTION

Part of the process was—funny as it sounds—was to test the testing framework. The purpose of the testing framework was to detect errors in the program where it did not meet the program requirements. So, a simple way to test the testing framework would be to insert errors into the project's source code which would cause certain tests to fail. If the report shows that the proper tests passed and failed, then we would know that we are catching the errors which we are trying to catch.

The following errors have already been written into the code and are simply commented out. Removing the comments, and they will inject the errors mentioned hereafter.

Adding the line *ret.getMemberIds().remove(5);* to the Cohort Class in the *intersect* method after the conditional at line 138 will cause any test case which has a 5 in the result to fail. Specifically, this is test case 8.

In the *union* method of the Cohort class, adding the line *ret.getMemberIds().add(5);* after the conditionals at line 119 will put a 5 in the resulting Cohort. This will cause any test case which does not have a 5 in the returned Cohort to fail. Or, more precisely, test case 3 will fail.

If the line *ret.getMemberIds().remove(4);* is added to the *subtract* method of the Cohort class after the nested conditionals at line 158 will cause any test case which has a 4 in the result to fail. Hence, test case 15 fails.

Inserting the error *ret.getMembers.remove(-3);* to the *union* method of the Cohort class after the conditionals at line 120 will remove -3 from the returned Cohort, and thus any test case which expects a -3 in the result will fail. For this reason, test case 22 will fail.

Injecting the line *ret.getMembers().add(1);* to the *subtract* method of the Cohort class after the nested conditionals at line 159 will add 1 to any resulting Cohort. Any test case which does not have a one in the expected output will therefore fail. These would be test cases 13, 15, 17, and 23.

Experience

In the beginning, there was quite a large learning curve. We were totally unfamiliar and inexperienced with a great many of the aspects of the project. We had never written in bash and had never worked with anything quite so large; on the whole, our experience had revolved around those assignments given in class, which served to help us master certain techniques or understand certain algorithms. These were usually only a few hundred lines—and that is on the upper end. But here the size of the project was not counted in hundreds of lines, but hundreds of files, most having several hundred lines. This added a bit of complexity to the project to which we were relatively unaccustomed, and in the beginning it took quite a bit of work to even get the published code to compile.

Once these initial hurdles were overcome, the project on the whole got much easier. The general trend of the difficulty of the project was that it got easier as we progressed, further, each new task and deliverable followed a similar pattern—it appeared quite difficult in the beginning, but then became very manageable as we got into it. This was generally because each aspect of this was something with which we were heretofore totally unfamiliar. We had never written a test case before. Or a testing framework. Or a script which interacted with files and called methods. It was all totally novel. This was part of the difficulty of the project, but it was also part of the value.

Designing and creating allowed us to learn and to do things which are necessary and relevant to computer science and software engineering but which we had never done before. This, it may be said, could apply to any part of the field with which one is unfamiliar, whether it be AI, or algorithms, or compiler design. However, this seemed to be different from those in that creating this framework and testing already written software is quite a departure from writing code and working with someone else's software is quite different from tinkering with that program which is the creation of your own mind.

More than just programming skills, however, were involved in this project, and thus more skills were refined and developed. Because the project spanned the entire semester, and because the project itself was quite large, planning was involved—it was not the sort of thing that could be knocked out the night before. We had to work with others in our group and learn from each other as well as coordinate schedules to meet together and present a finished product. This was also a new experience for most of us, since traditionally the projects assigned in previous computer science classes were to be done alone and usually had a shorter time frame in which to work on them because the assignments were themselves shorter. This helped us develop those team-skills which are necessary for the workplace and, more often than not, not always natural—especially to people who are used to getting a task and hunkering down and completing a task.