Team 5 Testing Platform Project

A Test of the Openremote Software Platform

Team Schfifty-Five
Andrew Rodman
Chad Hobbs
Brett Ostwalt
Jacob Wisse
Jason Daniel

Contents

1. Introduction

In the Fall of 2012, a team consisting of Andrew Rodman, Chad Hobbs, Brett Ostwalt, Jacob Wisse, and Jason Daniel were tasked with constructing a testing framework for an open source project. The requirements specified that the testing platform must operate on Ubuntu Linux, must be invoked by a single script from within the top level folder using the command "./ scripts/runAllTests.###", must perform tests based on test cases that contained all of the relevant information that would be needed to identify a testable component, supply or locate the necessary resources to perform the test, and then generate results in a .html format. The timeline for the testing framework project was eleven weeks from team formation to final deliverable reporting.

Teams were responsible for creating a software development team with appropriate roles and responsibilities across all members of the team. Each individual of the team was responsible for the project as a whole, but the team was left to decide how to carry itself through the software development process. The testing framework development was carried out in phases with deliverables required at each stage. The first stage required a selection of an appropriate open source project, the second stage required that a test plan for the project be developed, the third stage was the demonstration of the testing framework with 5 test cases, the fourth stage required twenty-five test cases and a more polished testing framework, and the final stage was the completed project with fault injection and full documentation. Team 5 selected the Open Remote project, a Cloud platform written in Java with mobile and web interfaces for use in home and building automation.

2. OpenRemote Project Summary

Overview

The OpenRemote software suite is a software integration platform for residential and commercial building automation. The software can be found at www.openremote.org. The individual components can be found on their Downloads page. The software has been developed in order to facilitate the often complicated process of getting different off-the-shelf automation hardware to work across multiple software platforms. The hardware can be controlled via custom end-user interfaces that are available for Android or iOS devices along with any device that has a modern web browser. OpenRemote also provides architecture that enables fully autonomous and user-independent buildings. This is all made freely available under an Open Source license and all software is developed by the community and for the community. OpenRemote consists of three major components, the cloud based Designer, the installation based Controller, and the user based panels.

Designer

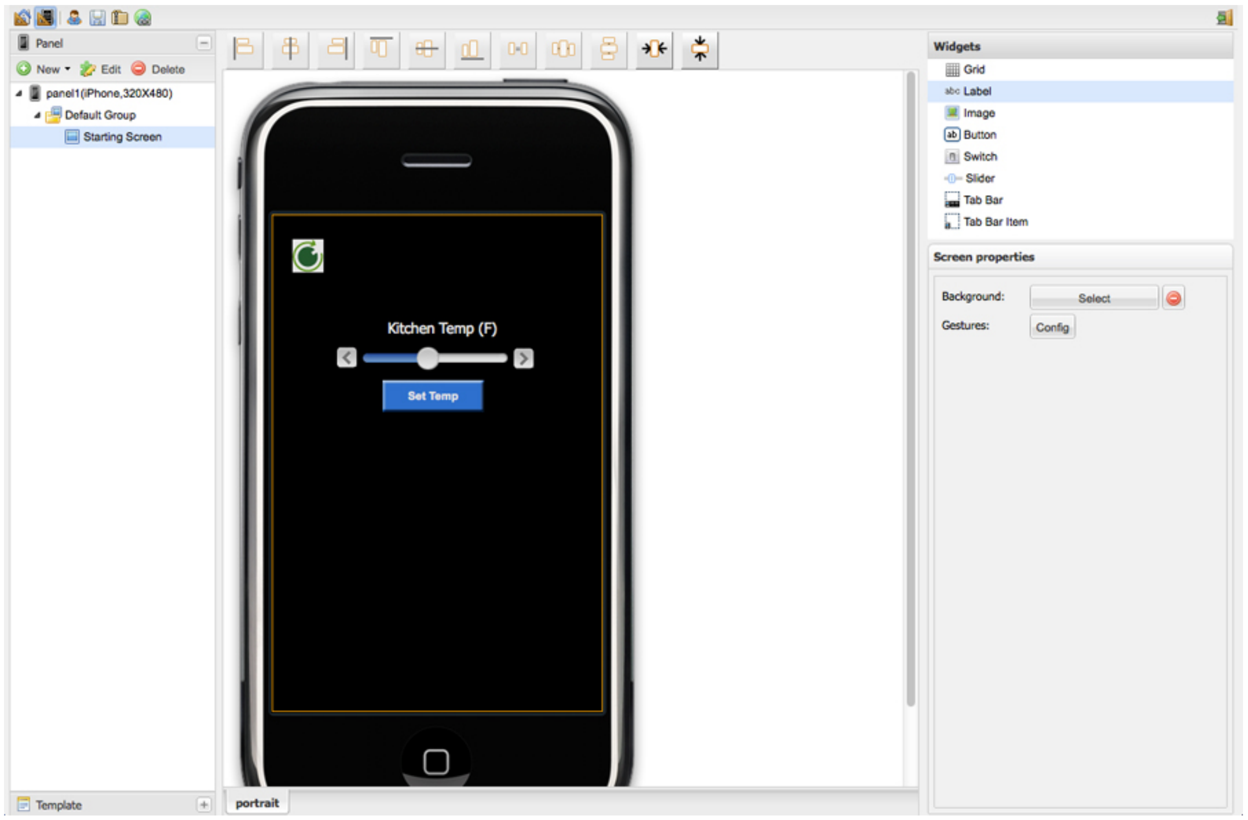
The OpenRemote Designer is a web-based software application that allows for easy and rapid creation of customized control panels for users. Managers can remotely update customer preferences, perform system updates or diagnostics, or change and add hardware controls. Users can create multiple user profiles and assign custom user interfaces to each profile, allowing for a diverse array of settings and specifications for any automation project.

The Designer software is an implementation of the Google Web Toolkit, and its dependencies are:

- Latest OpenRemote Controller Binaries (version 2.0.1 as of 26NOV12)
- Java 6 SDK

- GWT 2.4
- Apache ANT 1.8.x

The interface for Designer can be accessed through any modern web browser.



Panels

These are the most common user interfaces for the OpenRemote software suite. Panels are UI's that can be downloaded and installed via Apple's App Store, or as an Android APK via Google Play/Android Marketplace. For some basic, one-way control setups, it is possible to get up and running without use of Controller (introduction below) using only one of these control panels on a mobile device. The Panels are generated by the Designer and saved in the cloud.

Controller

The backbone of any OpenRemote installation is the Controller. The Controller functions as the server that accepts requests from users and issues commands to the hardware installed in the facility. Once installed [with default settings], the Controller can be found at on the server at http://localhost:8080/controller, and a web console on the server can be found at http://localhost:8080/webconsole. With proper permissions, they can also be accessed over the network so remote configuration can be done. Once installed. The controller can sync with the cloud account of the user and retrieve Panel information so any devices that try to interface with the Controller have the matching Panel for the server. Requests are received from user apps or web consoles, and the commands are issued to programmed devices.

3. Testing Framework

The testing framework for the OpenRemote project is written in Python. It consists of a single script called runAllTests.py which when executed, performs all the necessary steps to discover test cases and testable classes, compiles all code that needs to be compiled, runs the tests and generates results, and then removes all temporary files from the directory.

File/Folder Structure

The file and folder structure adheres to the requirements that were specified by the Term Project Specifications. The directory follows the following outline:

```
/[Team name in CamelCase]_TESTING (Note: No Spaces in any folder or file names)
        /project
                /src
                        /bin
                                /...(class specific folders)
        /scripts
                runAllTests.py

        /testCases
                testCase01.txt
                …
        /testCasesExecutables
                testCase01.java
                …
        /temp
                testCase01.class
                …
        /oracles
                testCase01Oracle.txt
                …
        /docs
                README.txt
        /reports
                testReport2012-11-29.html
```

Framework Requirements

The entire testing framework requires a Ubuntu Linux environment to meet project specifications, and the environment must have Python 2.5.x and Java 6 SDK installed.

Framework Architecture

The testing script is a memory-efficient program that can execute an indefinite number of test cases and compare/record outputs, limited only by the size of the user's hard drive. Below is a technical analysis of each method included in the script and a walkthrough of its execution.

compileTarget(target_path)

In previous versions, this script pre-compiled all project source code files into the "temp" directory (as denoted in the above section), however since the entire project is now included in the "src" directory this is no longer practical. This function takes a single argument, an absolute or relative path (from the root of the project directory) to a java source file, which is then compiled into the "temp" directory using Python's standard library subprocess.check_call method. Any errors during compilation will be reported to the terminal from which the script was executed, but a compilation failure will not cause the halt of the entire program; it will simply be skipped. This is intended to be used only with the Java test cases in the "testCasesExecutables" directory and not the project files themsevles since compiling one of these test cases will trigger the compilation of the project files.

executeTest(test_case)

After the compilation phase, the test cases need to be executed in order to compare outputs. This function accepts the name of a test case in "testCasesExecutables" to execute; an example might be "TestCase01.java" or something similar. Assuming that compileTarget has been called with essentially the same parameter, this function will then use the "java" command, included with the standard Java SDK, to execute the compiled class file. Note that it also uses Python's "subprocess" module and passes an additional "classpath" argument to the java command - this is the path to the “temp” directory where the compiled code for the test case should be residing. All test case executables return output to the standard output stream, which is gathered by the “check_output” function of subprocess and any errors are reported to the command line, similar to *compileTarget*.

compareOracle(output_file, oracle)

This method is used by the script to compare the output of a test case executable to a manually generated oracle file in the “oracles” directory. These files are expected to contain the expected/accepted output of a test to act as a point of comparison for the success of a test. The arguments *output_file* and *oracle* contain the relative names of their associated files; e.g. *output_file* might be “TestCase01.tmp,” containing the output of the first test case, and *oracle* might be “testcase01.oracle.” Again Python’s “subprocess” module is used to execute the “diff” command to determine whether the two files match and any discrepancies are recorded as a test failure.

buildHTMLReport(results_dir)

Rather than incorporate a fully fledged templating library into the project, *buildHTMLReport* will dynamically generate a report from the results of all tests and create a timestamped html document in “reports.” Its argument, *results_dir*, as currently implemented should point to the directory containing the serialized result data from each test case run. The function adds a simple header to the document and writes the headers of the tables as specified by a keys dictionary, which should contain all of the fields parsed by *parseTestCaseTemplate* method, discussed below. The function then uses the template data yielded by the *parseCaseResults* generator, also discussed below, to add a row to the table containing the field values of each template in *results_dir* and the results of the test execution. Each report is automatically opened by main for convenience. These reports are never cleared by the script, enabling users to accumulate these reports over time to support regression analysis and thorough testing. Each generated HTML report uses the same “report.css” Cascading Style Sheet to format output and maintain a consistent, professional look.

parseTestCaseTemplate(template)

This function is implemented in “scripts/parse.py” instead of “runAllTests.py” for a more logical grouping and to reduce the clutter of the latter. Its *template* argument should be the path of a template file in “testCases” that contains certain information about each and every test case. This information is as follows:

1. Id: The numeric test ID
2. Sourcefile: The project source file being tested (including the package)
3. Driver: The name of the Java source file in “testCaseExecutables” to be compiled and executed
4. Requirement: A description of the requirement being tested
5. Component: A description of the component being tested
6. Method: The name of the actual method being tested
7. Inputs: Any inputs that are provided to the method
8. Outputs: Expected outcomes; this can be a string containing any information about what the test executable is expected to return.

All of these specifications are included in the “README.txt” file included within the project’s root directory, as well as instructions for the execution of the script. This method contains a set of hard-coded keys that are required to be present in *every* template for it to be considered valid (see above); any templates containing keys that are not included in this list or failure to include one of these keys will cause the method to skip the template and report it as invalid. The function itself is essentially a small parser that will extract every key-value pair from the template file and store them all in a dictionary, which will be returned to the caller. Any errors or invalid template data will cause None to be returned, which is handled within *main*.

parseCaseResults(results_dir):

Main uses the *parseTestCaseTemplate* and *compareOracle* functions to build the results for each test case, which include the metadata provided for the test and the result of the actual execution of the test. Each result is then serialized and placed within the "temp/templates" directory, which needs to be retrieved for the final test report. This simple function is a generator that utilized Python's "glob" module in order to lazily retrieve the pathnames within this subdirectory and de-serialize each one to be returned to the user. It is intended to be used within a for-each loop within *buildHTMLReport* so that no more than one template at a time has to be kept in memory, eliminating any requirements that the number of test cases remain below a certain threshold. All it really does is use the "cPickle" module to *load* what was previously *dump*'d into each of these temporary result files and return it to the user, continuing until all of these files have been evaluated.

cleanTemp(temp_dir, results_dir)

This function is relatively self-explanatory, it expects the path of the temporary directory and the subdirectory containing the serialized results data (as mentioned above) in order to completely clear the "temp" directory of any previous test data. To do so it utilizes Python's "shutil" standard library module to remove directory trees and the *clearDirectory* method, discussed next.

clearDirectory(path)

A helper method to *cleanTemp* for use in removing all files from a directory so that a method may be called in *cleanTemp* to remove it. This function starts at a top-level directory and recurses down through the file tree, removing any files that are not directories.

main()

The method *main* is what drives the entire script and can be executed from the command line, per instructions detailed in this paper. It begins by clearing the temporary directory to ensure that results from past test cases have no chance to interfere with previous results. Further details will be included in an upcoming revision.

Expected Outcomes

In order to meet the project requirements, the testing framework is expected to generate testing results for each test case provided or report if no test cases are found. As discussed above in the *buildHTMLReport* function analysis, the framework will generate an HTML report containing all of the information for a test case including template information and the recorded result of a run.

4. Test Cases

Test cases are provided as a means to break down the project into small inputs and outputs that can be reliably compared against an oracle value. Oracles are generated by output from the original compiled code, while test cases are generated by isolating individual methods that produce the outputs. In order to define a test case, three items must be present: a test case template, a test case oracle, and the test case driver or executable.

The following is an example of a test case template, within which certain key-value pairs are defined in order to provide metadata about a given test. All test case templates included in the project must adhere to this format or will otherwise be interpreted by the script as being an invalid test.

testCase01.txt

```
Id:1
SourceFile: src.com.jpeterson.util.BinaryFormat
TestDriver: testCase01.java
Requirement: Utility for converting binary numbers works properly.
Component: Controller Utility
Method: public final String format(byte number)
Inputs:0x04
Outcome: "0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0"
```

Id:

The ID is a unique identifier inside the test case that provides a test number that can be readily identified and make the test unique. When generating new test cases, a unique number should be selected. The Id numbers do not have to be sequential.

SourceFile:

The source file provides the script with the location of the class to be tested so it can be compiled. The format is <source root folder>.<next folder>.<next folder>. … . <class name>

TestDriver:

The test driver provides a wrapper that can instantiate any classes that are necessary to test the source file and it also allows for a method to call the method with the appropriate inputs. An example for how to create a test driver can be found below.

Requirement:
The specified requirement is a text description of what the tested method should do. This is used to aid the tester in understanding what the inputs and outputs will be, and this description will be included in the final report.

Component:
The component is the part of the software system that is being tested. This can help identify where in the file/folder structure the original tested method is coming from. For most of our tests, the Controller or a subsystem of the Controller is what is being tested.

Method:
The method is what is being directly tested inside the class identified in the source file. The full method signature should be provided here.

Inputs:
The input specified is what is being provided to the method.

Outcome:
The outcome is what is expected out of the method and what is also found in the oracle.

This is the generic framework for all test cases and additional test cases can be created by following this template. When creating new test cases, an associated test case executable and oracle must be created as well, and the java class must be located in the src directory with original required folder structures. It is recommended that a test case, after manually being checked for validity, output directly to a file and that file become the oracle for the given test. Oracles are simply text files that contain a string output of some state within a test case. For example, the oracle for test case 01 is as follows:

testCase01.oracle

```
0 0 0 0 0 1 0 0
```

Test Case 01 passed the input 0x04 to the BinaryFormat class and the BinaryFormat.format() method within, and the method returns “0 0 0 0 0 1 0 0 “ as long as the code has not been broken.

The test case executable itself is slightly more complicated. Unlike traditional modern testing frameworks, test case classes do not need to extend any base unit test class; they need only provide a main method that can be executed as a “driver” to execute any testing code. The main method may either instantiate an instance of its enclosing class and call a method entitled testSomething, where something is the name of the test, or may simply define the test itself in main as many of the test cases do. Again, the first run of a given test, if known to be the correct output, can and should be used directly as its associated oracle. For a quick glance at an existing test case, we’re going to skip ahead to test case 13:

testCase13.java

```
        // Imports excluded for brevity
public class testCase13 {
        public String testSocketObjectPoolCheckOut() {
                try {
                        int port = 9090;
                        ServerSocket s_sock = new ServerSocket(port);
                        SocketObjectPool pool =
                                new SocketObjectPool(InetAddress.getLocalHost(), port);
                        Socket s = (Socket) pool.borrowObject();
                        String port_str = Integer.toString(s.getPort());
                        s.close();
                        s_sock.close();
                        return port_str;
                } catch (IOException e) {
                        return e.toString();
                }
        }

        public static void main(String[] args) {
                testCase13 test = new testCase13();
                System.out.print(test.testSocketObjectPoolCheckOut());
        }
}
```

Note in this test that the actual test functionality is defined in an instance method and that the class is instantiated to run the test. While not shown it is also important to note that because of the way the script handles the classpath and compilation phases, it is possible to directly import source files from the project without having to use the relative path. For example, in order to import the BinaryFormat class the line “import com.jpeterson.pool.SocketObjectPool;” is used within the omitted section - see the source file itself in the project for further clarification.

Examining the directory structure of the testing framework should exemplify the theoretical implementation as described above and throughout this document. The simple pattern described in this section may be used for adding each and every new test case to the framework, attempting to make organization of the tests as easy as possible.

5. Fault Injection

The fault injection process for the project is a fairly uncomplicated one. As mentioned earlier in the testing framework portion of the report, there are several components including: the files to be tested, the test case input and output, and the oracle. The idea of fault injection is to modify the code so that the output is no longer the same as the oracle. This can be done in a number of ways, whether it be changing array positions or causing errors while running the code. The code should still compile, but now it runs and produces unexpected outcomes.

For our first fault injection, simply modifying one line of code in the project directory will break three test cases since the method to be modified is used in code that is tested separately. The three in question all test methods of *com.jpeterson.util.pool.SocketObjectPool*, which is a utility class for improving the efficiency of allocations and using sockets in the core Open Remote Controller classes. In order to cause these tests to fail, one small change on line 129 of the *com.jpeterson.util.pool.SocketObjectPool* class was made. Commenting out the line that returns the created socket object and replacing it with a null return statement causes the method to cease functioning properly; it should look like the following:

```
protected Object create() {
   // ...Code omitted for clarity...

   //return(socket);
   return null;
}
```

Executing the test framework script per instructions in section six should now cause tests thirteen, fourteen, and fifteen to fail. Granted for proof of concept really only one of these classes needs to fail, but such results can be expected from tests that evaluate the same class.

A second fault injection can be made by modifying the file Condition.java, located at schfiftyFive_TESTING/project/com/jpeterson/util/Condition.java. If in line 57 of this file, if a `!` is inserted into the return statement so it reads, `return(!isTrue)`, test case 21 will now fail. This change in the project code also causes test cases 22 and 23 to fail, as the methods tested in those cases require that the isTrue() method be functioning properly.

Another fault injection involves the Unsigned.java class and specifically the unsigned(byte number) method. The class in general is a utility class that accepts various inputs and returns the input unsigned. If an input is positive then there is no change, but if an input is negative then the unsigned value is returned. The method in its entirety is listed below:

```
public static long unsigned(byte number)
        {
    long result = 0;
    BinaryFormat format = new BinaryFormat();
    String binary = format.format(number);

    StringBuffer buffer = new StringBuffer(binary);
    buffer.reverse();
    int length = buffer.length();
    for (int i = 0; i < length; i++) {
        result += (buffer.charAt(i) == '1') ? 1 << i : 0;
    }

    return(result);
        }
```

When the line inside the for() loop is commented out (line 50), result is never manipulated, and a 0 is always returned. This causes a test failure for all results except an input of a long formatted 0.

The next fault injection is a manipulation of an array in the XUtil class under src/com/x10. The x10Util class contains several methods and several arrays for the encoding of house code and bytes. One method, deviceCodeToByte takes a character, which corresponds to an int as defined in the code, and uses that int to find the index in a code2Value array. In test case 18, the input is the char 'b', which corresponds to the number 2. This means that the second index of code2Value will be returned. Originally the number in the index was 14, but in order to break the code it was changed to 22. Now, when the testing suite compares the values and expects a 14, it will no longer pass.

6. Instruction Manual

Prerequisites

- Access to the top level directory of the testing framework, schfiftyFive_TESTING, via terminal
- Python 2.5 installed
- Java 6 SDK installed
- Internet Explorer, Firefox, Chrome, Safari, or any other modern web browser.

The use of the provided testing framework is meant to be as automated as possible, per the requirements of the project. The user must be able to access a terminal screen on the computer that contains the testing framework. The script will be executed in python and the tested classes are written in java. The outputs will be displayed in a browser, hence all of the prerequisites.

Before execution

The user must open a terminal window. This is accomplished on Ubuntu by holding Control and Alt while pressing the T key. This will open a terminal window that starts in the home directory of the users computer. The user must then navigate to the top level of the testing directory. This is accomplished by using the command 'cd' (change directory) followed by the desired folder. In order to view files and folders in the current directory, use the command 'ls' (list). The top level directory is called schfiftyFive_TESTING. Once in this directory, it is generally good practice to grant execute permissions on the testing script by typing the following command into the terminal: 'chmod +x scripts/runAllTests.py'. This will allow the script to run on the users computer.

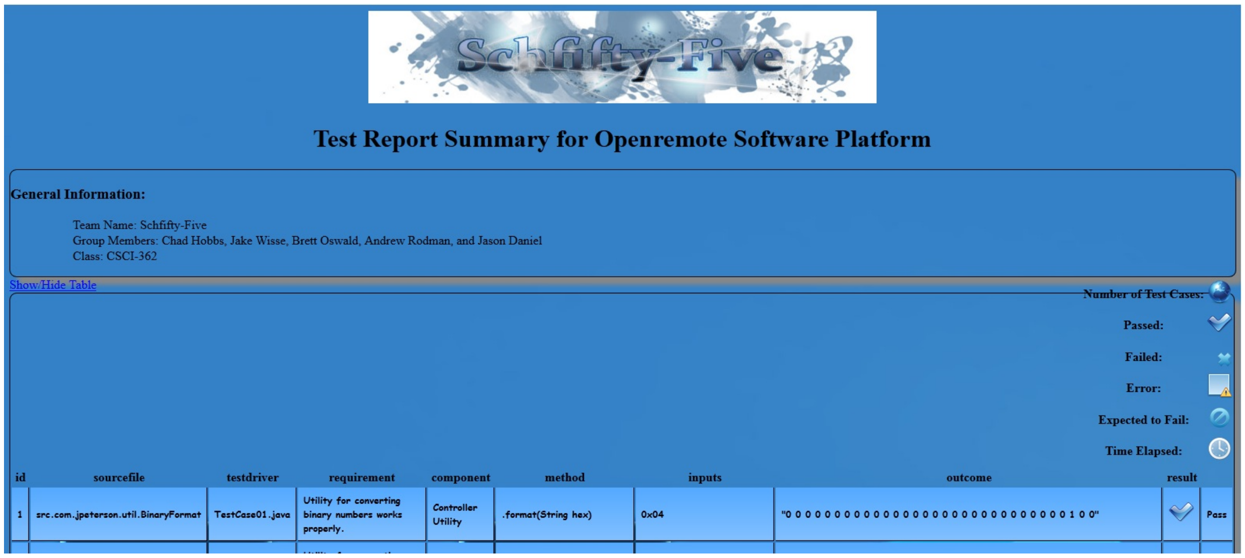
Execution

The testing script is executed by typing the following command into the terminal: './ scripts/runAllTests.py'. The script will generate status reports as it discovers directories and files for the testing process. At the completion of the status reports, the script will announce that the final report is viewable in a automatically opened browser window. If the user already has a browser open, a new tab with the results will be displayed, otherwise the default browser will be opened with the report.

Understanding the Report

The report will consist of a header that has general information about the project, including the team name, team members, and the class this project was completed for. Below this is an option to view the details of the report, including all of the test cases, their particular datasets, and the outcomes. The ID of the test case is included, along with the source file, the test driver name, the requirements of the method, the project component being tested, the particular method being tested, inputs, expected outcome, and the test result. If a test is reported as passed, then what was included in the test case matched what was in the oracle. If a test is reported as failed, then output did not match the oracle and further analysis of the cause is required.

Example report is below:


Test Report Summary for Openremote Software Platform

General Information:

Team Name: Schfifty-Five
Group Members: Chad Hobbs, Jake Wisse, Brett Oswald, Andrew Rodman, and Jason Daniel
Class: CSCI-362

Show/Hide Table

Number of Test Cases:
Passed:
Failed:
Error:
Expected to Fail:
Time Elapsed:

id	sourcefile	testdriver	requirement	component	method	inputs	outcome	result	
1	src.com.jpeterson.util.BinaryFormat	TestCase01.java	Utility for converting binary numbers works properly.	Controller Utility	.format(String hex)	0x04	"0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0"	✓	Pass

Possible Errors

Any failures due to compilation or running test cases should be reported by the script and ought to be relatively easy to track down. However there exist potential for more subtle errors that may cause a test case to fail that may not relate to the success of the test run. For example, oracles should include no gratuitous newline characters past the last line, although the last line should be terminated by hitting the “enter” key. The diff tool is very sensitive to small dissimilarities like this and it is worth noting that this could be a potential problem. Users should also make sure that all templates adhere exactly to the format specified in section four of this report, or else risk causing the template to be marked as invalid for lacking the required or possessing extra key-value pairs.

7. Definitions and Resources

Definitions

Apache ANT - A software tool for automating software build processes. Written in Java, ANT used XML to describe the build process and dependencies.

Fault Injection - A method of improving the dependability of a testing platform by probing all software logic pathways with broken code.

GWT 2.4 - Google Web Toolkit is an open source set of tools that help developers create Javascript front-ends in Java. This allows for web developers to create and maintain rich web user interfaces.

Java 6 SDK - Java is a very popular high level programming language developed and owned by the Oracle Corporation. It is an object-oriented programming language and is deployable on most platforms and on the web. SDK is an abbreviation for Software Development Kit.

Python 2.5 - Python is a high level programming language that includes object-oriented, functional, and imperative programming paradigms. It is lauded for its ease of use and readability and is popularly used for scripting.

Open Source - Open source is a term used to describe a project that can be created, edited, and maintained by a wide array of people. When speaking of open source software, the source code is freely available to users and methods to allow for bug fixes and feature upgrades are provided.

Oracle - An oracle is a method used by software developers to determine whether a particular test has passed or failed. This is usually done by providing expected results and comparing that with generated outcomes.

Terminal - A Linux Terminal that provides a serial method to issue commands to the computer.

Ubuntu - A popular Linux computer operating system. Ubuntu is a free and open source OS, and is widely used on both desktops and servers.

Resources relevant to the Openremote testing platform

Apache ANT - http://ant.apache.org/

GWT - https://developers.google.com/web-toolkit/

Java SDK - http://www.oracle.com/technetwork/java/javase/downloads/index.html

Openremote - www.openremote.org

Python - http://www.python.org/

Ubuntu - http://www.ubuntu.com/

8. Team Review

Development During Design and Build

Our primary development model started as a prototyping model and switched into an incremental development process once a basic framework was established. The initial prototype demonstrated most of the functionality that was desired in the testing framework and helped identify required methods and parameters of the scripts and drivers. Once the requirements were mostly developed, the prototype was cast aside and the first increment of the final product was created. Individual components of the script were completed individually, reviewed for functionality, and then committed to the project. This would in turn lead to another cycle where either a different component was developed or an existing component was refined. This incremental method of development lead to a steady march towards project completion.

Obstacles Encountered

A difficulty we predicted revolved around the fact that one of our team members was across the country for a week during a critical development period. However he was able to communicate via email and still work on the project, so this was easily resolved. However, an unforeseen obstacle stemmed from the nature of the OpenRemote project, which is a highly integrated and coupled system that requires substantial configuration and communication between classes. This limits the amount of classes that can be tested without providing a very deep and complicated scaffolding platform. However, the project was large enough to isolate enough individual classes that had minimal scaffolding and those were tested. While developing fault injections for the project, the integrated nature of Java again required an in-depth evaluation of how to break code while not breaking the entire project. This was overcome by carefully selecting appropriate edits and handling errors properly.

Timeline

The initial step in our project was to determine what open project would be best suited for testing based our familiarity with project structure, language, as well as the components that were integral to the software. The group eventually decided on the open source software, OpenRemote due to the fact that OR is mostly made up of java code, which each member is familiar with. Also, one team member already had quite a bit of experience with similar software and could act as somewhat of an authority on how OR (Openremote) works.

The next step in the process was to determine a testing plan for the project. This required an addressing of the items to be tested, a method for testing them involving the predetermined testing architecture, testing schedule and so forth. As far as tested items go, the team decided to hone in on the controller portion of the project and test some of the methods within. Some of the more straightforward formatting methods were chosen so that a proper script could be developed without having to troubleshoot the java classes. Then, the python script was developed that was able to recurse through testing directories and parse test cases in order to run and test them. The python script is also able to generate an html report with testing results in it for each of the cases.

The next step of the project was the generation of twenty-more test cases using various methods from the controller module. The remainder of the test cases were to be similar to the first five in that they tested particular methods from certain classes. For the most part, this was to be the most straightforward portion of the project. Unfortunately, as mentioned before, the controller classes were highly integrated and most required a plethora of dependencies, which was a bit of a challenge for developing simple test cases. The group worked through the dependency problem and was able to provide the appropriate number of test cases.

The final stage of the project required injecting faults into tested code and demonstrating that the testing platform will detected the errors caused by these faults. The team broke 5 different methods that were in the original testing suite and showed that the testing framework reported these test cases as failures.

Review of the Testing Framework Project and Assignments

It is safe to say that the testing framework project in this class, Software Engineering 362, contains many lessons for computer science students. At the forefront, any who have little or no experience working in a team setting are forced to adapt quickly to the new scenario. This is great experience for working in the real world since, unless one is unfortunate enough to be left coding in a back closet all day, the vast majority of development work will be performed in teams. For those who invested in the project and took on leadership roles, delegation of duties and team management are certainly skills that can be gained from this project and really the course as a whole. It could be said that assignments earlier in the semester were relevant and thought provoking, but there was an awkward transition between having assignments virtually every class and having none and being responsible for components of a team project.

One thing that Team 5 struggled with internally was sticking to a timeline. Too often the project was left until the last minute with members scrambling to meet deliverable requirements, which was undoubtedly frustrating. Despite a certain level of panicking all deliverables were met on time and satisfactorily, without the team being torn apart by the professor in front of the class.

Suggestions for Improvements

The testing framework, while a great project to work on in theory, suffered hugely from a high degree of ambiguity despite multiple discussions and project specification documents.

Team 5 in particular was often referred to the testing specifications but even after practically memorizing it members found that there were still many missing pieces required to reach a starting point, much less finer details of the assignment. It took probing and, at the expense of one of our members, a dissection of an email containing questions to solidify several core requirements. The lesson learned from this is that a lot of research and fact-checking is required before a project can even begin, with a full understanding of the requirements.

As for the IEEE 829 test plan format, it is reasonable to make the claim that such a framework is highly irrelevant in modern software testing suite implementations. The majority of open source software depends on the concept of a unit testing framework, which allows developers to extend base test case classes with various statements to assert the validity of certain states of an application. That said, if there is a feasible way to, perhaps, get students to implement their own base class for such assertions (such as Python’s unittest.TestCase) or to allow some level of usage of such a class then it would be a drastically more relevant project. The reason being that test cases in modern software must be robust and the layout of the current specifications allows for, if not forces, students to “hack together” tests instead of actually being able to step back and really test individual components of classes. With the existing approach, any software that requires scaffolding or configuration becomes dramatically more cumbersome to implement as students have to dedicate valuable time to figuring out how to write additional driver classes to perform setup work before they can even write tests. While this is a realistic expectation when developing a new testing suite for a large open source project, even then it is most likely going to be built by contributors with a great deal more knowledge and familiarity with the codebase than, say, students working incrementally as part of a Software Engineering class.

As a slightly more biased recommendation, one of our team members is of the strong opinion that, in his experience, Subversion is a fundamentally broken tool that is absolutely more trouble than it’s worth. The degree of ease with which one can corrupt a directory is horrifying and this happened twice throughout the duration of this project, despite having at least one member with a very high familiarity with Version Control Systems. That said, unless there is any specific requirement for using Subversion, students may benefit from at least having the option of using a Distributed Version Control System (e.g., Git) would be hugely beneficial in reducing the amount of time spent fixing environment errors. Git in particular is a very powerful and robust tool that allows for a dynamic workflow and is much more flexible and forgiving than Subversion. Despite these criticisms, however, the testing framework project is undoubtedly beneficial and leaves students having learned lessons from teamwork to time management, ranging beyond previous Computer Science classroom material on coding.