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. The project selected by Team 5 was the Open Remote software platform, a Java/Cloud/Mobile software integration platform for home automation.

2. OpenRemote Project Summary

Overview

The OpenRemote software suite is a software integration platform for residential and commercial building automation. 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

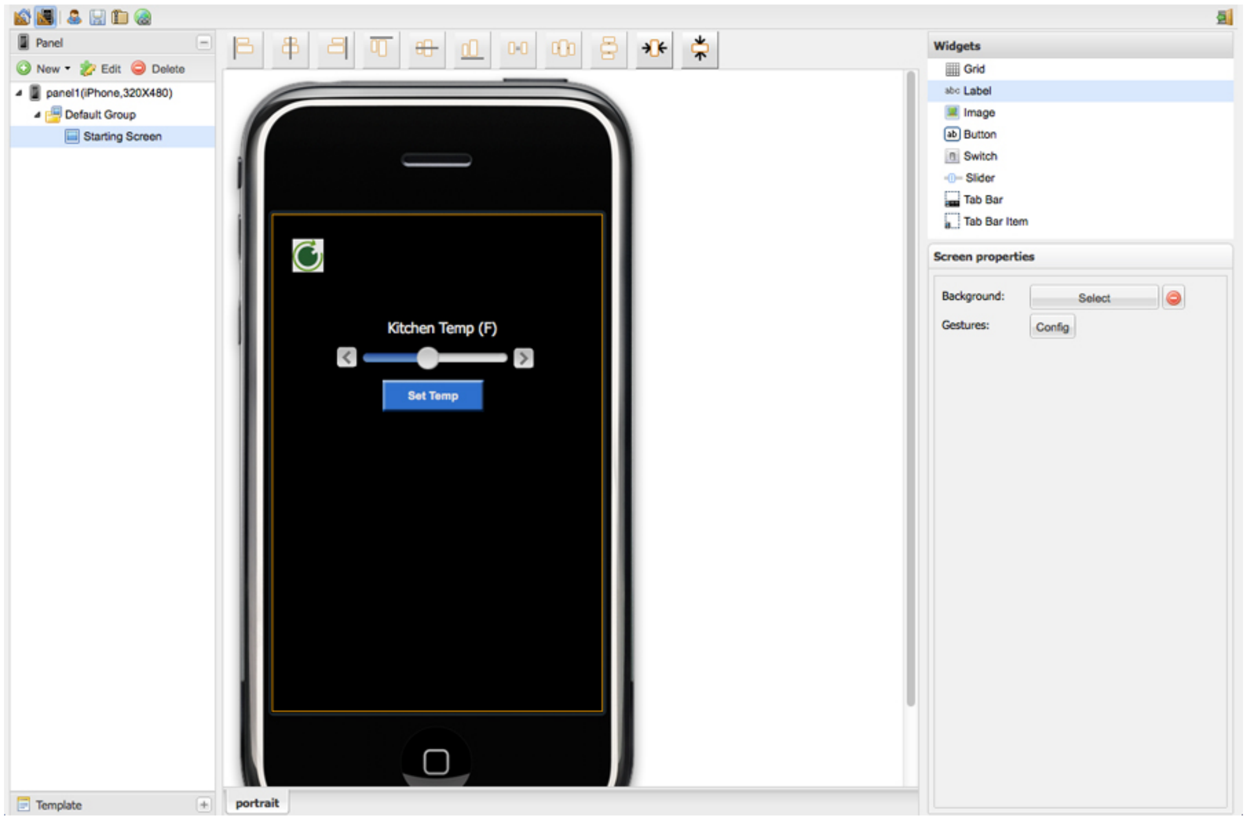
The OpenRemote Designer is a web-based software application that allows for easy and rapid creation of customized control panels for users. Managers can remotely update customer preferences, perform system updates or diagnostics, or change and add hardware controls. Users can create multiple user profiles 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
- Java 6 SDK
- GWT 2.4

○ Apache ANT

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.

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 http://localhost:8080/controller, and a web console can be found at http://localhost:8080/webconsole.

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 or higher 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 due. 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. The following is an example test case:

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: .format(String hex)
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.

SourceFile:
The source file provides the script with the location of class to be tested so it can be compiled.

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.

Requirement:
The specified requirement is a text description of what the tested method should do.

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.

Method:

The method is what is being directly tested inside the class identified in the source file. The format of the input is provided as well.

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.

5. Fault Injection

This is where the reported experiences of inserting faults into the original code will go, with an explanation of how we inserted faulty code and what the results were. This will be completed for Deliverable #5.

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.

Possible Errors

To be determined.

7. 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. Further development of the project will follow the same model until the final product is delivered.

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.

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 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 a 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 will require injecting faults into tested code and demonstrating that the testing platform will detect errors that will be caused by these faults. The rest of this timeline will be completed when this stage is done.