Deliverable I
Project: Martus

Deliverable:

Create a report based on checking out the Martus project from the repo. After Checking out attempt to build the project, and run any tests that came with the repo.

Checking Out:

This process has been restructured to reflect the actual procedure required learned via trial and error. Several initial dependencies must be brought into the system before the repo in order to successfully build the project. Hg tortoise version control shell is required for the mercurial plugin to the Eclipse IDE to function correctly. Next we must go into eclipse and install the e(fx)clipse plugin. The e(fx)clipse plug-in requires the Eclipse 4.4 SDK, which can be . This can be done by going to Help → Install New Software. Then add the address:

http://download.eclipse.org/efxclipse/updates-released/1.0.0/site

into the new address bar. e(fx)clipse will then display as an option, following the prompts will install the plugin. Now the environment is ready for import. Following the import steps for each project at

https://code.google.com/p/martus/wiki/DeveloperHOWTO

will bring all the components into the environment. Note that csv2xml auto refractors on import and that step can be omitted. Through trial an error it was discovered that the final step to add an external JRE is also not necessary, in fact it causes conflicts. If the project is cleaned there should be two projects with errors, client and common. Both of these are reliant on a JavaFX import that is not being resolved. By right clicking each project we can go in and configure the build path. There is a tab for libraries that allows us to add an external JAR. Inside the local Java installation lib folder is the file jfxrwt.jar which must be added as an external JAR to both projects. At this point the project should be error free. By pressing [CTRL]+[SHIFT]+T and typing "TestAllQuick" we can run the included test suite. About 1300 tests will execute, with one failure. To the best ability of our error tracing the failure is due to undefined SSL certificate, which would be needed to connect to a secure server. As per the annotation to the exception thrown, this failure can be disregarded.

Impressions:

Overall the process was more pain-free than expected. The several issues that popped up seem to be due to integration with newer technologies such as JavaFX 2.0. Also some issues can be attributed to Eclipse not integrating JavaFX well and not fully supporting Java 8 yet. Some other annoying problems also arose for some of us in trying to check out the project via the Mercurial Eclipse plug-in. There seems to be some setting in the Mercurial install that is misconfigured and will not allow me to proceed in checking out the project. Also, at times, I have been unable to reach the server that houses the Martus source files. In order to solve this problem, I had to revert and finally uninstall Eclipse, and reinstall a clean version of the software. This fixed the problem of reaching the Martus servers, but still did not allow me to successfully check out the project.

Complete Instructions:

After much trial and error the martus project has been successfully built and compiled, in addition all modules show green across the board based on the included test suite of 1269 tests. There is one failure but it is expected based on no connection to a SSL martus server. The following are the step by step instructions for creating a stable Martus environment. It should be noted that this was done on a fresh Ubuntu 14.0.8 installed VMware virtual machine.

1- install java8 JRE/JDK

```
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
```

2- install TortoiseHg

```
sudo apt-get install TortoiseHg
```

3- install Eclipse

4- install mercurial plugin via marketplace

5- install e(fx)clipse plugin via marketplace

6- import each of the projects via the links provided in the project wiki

7- IGNORE ALL OTHER INSTRUCTIONS IN THE WIKI!

8- right click on the client module

9- go to build path

10- click configure

11- go to libraries tab

12- click "add external JAR"

13- navigate to the bc-jce module

14- select the bc-jce.jar file

15- click ok

16- the workspace should rebuild and the error gone

17- [shift]+[ctrl]+t and type "TestAllQuick"

18- run as JUnit Test

Deliverable II
Testing Plan

Project
Martus

Team
Tom Evans
Erik Engstrom
Ryan Sprowles
Trevor Kirkpatrick

Deliverable

This deliverable will serve to outline a set of test cases, derived directly from the software requirements, which we will use to test segments of the Martus source code. The software was written to perform a set of functions based on how the end product is supposed to work. These functions are detailed in a list of requirements. For each of the requirements we choose to test, we will be exploring the input space and choosing inputs which illustrate important cases. We will then use the outcome of these tests to verify that the software is working as it is intended, at least according to the requirements.

The testing process

We will create a script which runs a suite of tests on a segment of the Martus source code, compares the outcomes to a set of expected outcomes, and displays the results in an HTML file. Our script needs to live in the same directory as the Martus source code. It will have a set of file dependencies which it must use to compile the driver. It will compile and run the driver, using pre-selected methods and inputs designed to illustrate test cases, listed in a text file. It will output the results to another text file and compare this outcome to the expected results of each test. It will then create a final HTML file displaying the context and results of each test.

Requirements traceability

1.The appendWithColonTerminator(StringBuilder existing, String toAppend) method in the martus-server module, org.martus.server.main package, MartusServer class, should append the input string, "toAppend" to the input string, "existing", and append a colon to the resulting string.

a) The method should be able to append two non-empty strings
b) The method should be able to append a string to an empty string.
c) The method should be able to append an empty string to an existing string.
d) The method should be able to append two null strings.

2. The setDelimiter(char c) method in the martus-utils module, org.martus.utils package, DatePreference class, should allow user defined delimiter to be set for processing dates.

a) The method should be able to set the delimiter to a character
b) The method should be able to set the delimiter to a symbol
c) The method should be able to set the delimiter to an integer
d) The method should be able to set the delimiter to a reserved character
e) The method should be able to set the delimiter to a special character

Test Cases

Test ID - **1a**

Requirement - The method should be able to append two non-empty strings
Component - Module: martus-server Package: org.martus.server.main Class:MartusServer.java
Method - appendWithColonTerminator(StringBuilder existing, String toAppend)
Input - ("Test String Existing", "Test String Appendix")
Expected Output - "Test String ExistingTest String Appendix:"

Test ID - **1b**

Requirement - The method should be able to append a string to an empty string
Component - Module: martus-server Package: org.martus.server.main Class:MartusServer.java
Method - appendWithColonTerminator(StringBuilder existing, String toAppend)
Input - (null, "Test String Appendix")
Expected Outcome - "Test String Appendix:"

Test ID - **1c**

Requirement - The method should be able to append an empty string to a string
Component - Module: martus-server Package: org.martus.server.main Class:MartusServer.java
Method - appendWithColonTerminator(StringBuilder existing, String toAppend)
Input - ("Test String Existing", null)
Expected Outcome - "Test String Existing:"

Test ID - **1d**

Requirement - The method should be able to append two empty strings
Component - Module: martus-server Package: org.martus.server.main Class:MartusServer.java
Method - appendWithColonTerminator(StringBuilder existing, String toAppend)
Input - (null, null)
Expected Outcome - ":"

Test ID - **2a**

Requirement - The method should be able to set the delimiter to a character
Component - Module: martus-utils Package: org.martus.utils Class:DatePreference.class
Method – setDelimiter(char c)
Input - ('X')
Expected Output - ('X')

Test ID - **2b**

Requirement - The method should be able to set the delimiter to a symbol
Component - Module: martus-utils Package: org.martus.utils Class:DatePreference.class
Method – setDelimiter(char c)
Input - (':')
Expected Output - (':')

Test ID - **2c**

Requirement - The method should be able to set the delimiter to an integer
Component - Module: martus-utils Package: org.martus.utils Class:DatePreference.class
Method – setDelimiter(char c)
Input - ('3')
Expected Output - ('3')

Test ID - **2d**

Requirement - The method should be able to set the delimiter to a reserved symbol
Component - Module: martus-utils Package: org.martus.utils Class:DatePreference.class
Method – setDelimiter(char c)
Input - ('_')
Expected Output - ('_')

Test ID - **2e**

Requirement - The method should be able to set the delimiter to a special symbol
Component - Module: martus-utils Package: org.martus.utils Class:DatePreference.class
Method – setDelimiter(char c)
Input - ('~')
Expected Output – ('~')

Testing schedule

1.[Due 9/23] We will outline a set of test cases based on the requirements to verify that the software is working as expected
- First we will produce 5 test cases
- We will then expand our list to 25 test cases

2.[Due 10/14] We will write a script which will take these test cases as input, runs the selected methods with these inputs, and compares them to the expected outcomes
- First we will include only the initial 5 test cases
- We will then expand our script to incorporate the additional 20 test cases

3.[Due 11/6] We will present a formalized, professional product, including documentation, which tests our 25 test cases

4.[Due 11/18] We will design and inject 5 faults into our test suite, in order to cause some of the tests to fail

5.[Due 11/20] We will present a final report on our testing suite

Test recording procedures

Every test phase will have a documentation in the form of testing requirement summary and sub test composition. At a higher detailed level, each sub test will be described in terms of what is being tested, the oracle being used, and the actual outcome of the test with any pertinent additional information. The script will compile the outcome of the tests with the predetermined inputs in a file and compare this file to the oracle, giving a rating of pass to every outcome that matches the corresponding element in the oracle, a rating of fail to every outcome that does not match, and a rating of error to every outcome which represents an unexpected problem in the execution of the test.

Hardware and software requirements

This testing suite requires the following to reflect the testing environment:

- •Ubuntu Linux v12+
- •TortoiseHG shell
- •Mercurial shell
- •Java runtime JRE/JDK v8

Constraints

Our ability to test the Martus source code will be limited by our understanding of how the code is written, and what each piece was written specifically to do. We will not be able to compile and run the code to get an idea of the intended function of some of the pieces of the software. In addition, effective knowledge of data structure implementation could offer complications in understanding how to test some requirements in phase one.

System tests

System testing will be defined as other phases of the test suite take form. Ideally each phase will focus on a different part of the Martus framework. Cohesion of all phases will reach an endpoint constituting a test of the framework as a whole.

Deliverable III
An automated testing suite

Project
Martus

Team
Erik Engstrom
Thomas Evans
Ryan Sprowles
Trevor Kirkpatrick

Deliverable

For this deliverable, we are required to develop and present an automated testing suite. We must write a script to gather and compile specified segments of the Martus source code, as well as a driver to instantiate classes from the code and call their methods. For this deliverable, we are only required to test 5 test cases. Later in the course, we will incorporate a full list of 25 test cases.

Test Cases

*****tests 1a-1d not yet added to test suite*****
Test ID - **1a**
Requirement - The method should be able to append two non-empty strings
Component - Module: martus-server Package: org.martus.server.main Class:MartusServer.java
Method - appendWithColonTerminator(StringBuilder existing, String toAppend)
Input - ("Test String Existing", "Test String Appendix")
Expected Output - "Test String ExistingTest String Appendix:"

Test ID - **1b**
Requirement - The method should be able to append a string to an empty string
Component - Module: martus-server Package: org.martus.server.main Class:MartusServer.java
Method - appendWithColonTerminator(StringBuilder existing, String toAppend)
Input - (null, "Test String Appendix")
Expected Outcome - "Test String Appendix:"

Test ID - **1c**
Requirement - The method should be able to append an empty string to a string
Component - Module: martus-server Package: org.martus.server.main Class:MartusServer.java
Method - appendWithColonTerminator(StringBuilder existing, String toAppend)
Input - ("Test String Existing", null)
Expected Outcome - "Test String Existing:"

Test ID - **1d**

Requirement - The method should be able to append two empty strings
Component - Module: martus-server Package: org.martus.server.main Class:MartusServer.java
Method - appendWithColonTerminator(StringBuilder existing, String toAppend)
Input - (null, null)
Expected Outcome - ":"

Test ID - **2a**
Requirement - The method should be able to set the delimiter to a character
Component - Module: martus-utils Package: org.martus.utils Class:DatePreference.class
Method – setDelimiter(char c)
Input - ('X')
Expected Output - ('X')

Test ID - **2b**
Requirement - The method should be able to set the delimiter to a symbol
Component - Module: martus-utils Package: org.martus.utils Class:DatePreference.class
Method – setDelimiter(char c)
Input - (':')
Expected Output - (':')

Test ID - **2c**
Requirement - The method should be able to set the delimiter to an integer
Component - Module: martus-utils Package: org.martus.utils Class:DatePreference.class
Method – setDelimiter(char c)
Input - ('3')
Expected Output - ('3')

Test ID - **2d**
Requirement - The method should be able to set the delimiter to a reserved symbol
Component - Module: martus-utils Package: org.martus.utils Class:DatePreference.class
Method – setDelimiter(char c)
Input - ('_')
Expected Output - ('_')

Test ID - **2e**
Requirement - The method should be able to set the delimiter to a special symbol
Component - Module: martus-utils Package: org.martus.utils Class:DatePreference.class
Method – setDelimiter(char c)
Input - ('~')
Expected Output – ('~')

These test cases and the requirements from which they were derived are outlined in more detail in Deliverable 2.

Architectural Framework

The main component of our testing suite is our script. The script has access to a list of dependencies, a list of specified methods and inputs based on our test cases, the Martus source code, and an oracle file that lists the expected outcome corresponding to each test. The driver is a Java class that compiles the necessary segments of the source code, and runs them with the inputs specified by the script. The script then gathers the results of each test from the driver, puts them into a readable context, wraps the results in HTML tags, and dumps them into a file.

Experiences

Creating a driver, and the test case methods themselves, for the martus project was relatively straight forward. Using the conceptual template of how Junit tests behave I started with a prototype driver. The prototype was merely to make sure I could instantiate an instance of the class I was testing in order to access the method I wanted to test. Once I affirmed that I could do this I drummed up some spoof test methods just to make sure I could interact with my instance of the test object. As this proved successful I then sat down and implemented the test methods according to the test case plan. I tested the ability to run the tests, and check against oracles stored in a text file, until I was satisfied my implementation was performing as expected. Note this was all done in the Eclipse IDE to separate the task of creating a driver and small test suite from command line execution. With task one complete I turned to figuring out how to run my driver from the command line, which meant learning about class path.

I made the process of learning class path much more difficult than it needed to be. My mistake can be attributed to using a spoofed Junit test as practice for simple execution from command line. This lead me astray very quickly as the intricacies of class path and Junit are much more complex than my needs. When it came time to apply what I had learned worked for Junit to my own test driver class, I was making things far more complicated than they needed to be. Through much trial and error I finally figured out that all I needed to do was specify the JVM to look in the current directory and the root directory of the package where the class I wish to test resides. This seems glaringly obvious once I got it working, as I said I misled myself initially and made things much harder than they needed to be.

This was the point where I realized, through some feedback from a classmate, that my implementation was incorrect. I failed to back check my progress in trying to figure out each step with the actual specifications for the project. I ended up creating a prototype based on what I had learned that did not fulfill the guidelines. It seemed a monumental screw up at first. As my first CS professor taught me the best tool in a developers kit is the ability to disassemble the problem into its simplest units. I pulled everything out of my driver but the ability to instantiate the class object I was testing. Then I implemented it so the driver looked to the command line for the inputs for the method test. This was incredibly simplistic compared to my previous iteration. I then got to work on building a bash script that would read a test spec text file for the needed information and call the necessary driver passing it the input. I attacked the implementation of the script's functions piecemeal, building on dealing with a single test case. Once I had one working I modified for processing every test case text file found in the testCase directory. At this point I am confident the script performs as it outlined in the specification for the project. The only needed change would be a decision structure in the script to determine which driver to call based on which method is being tested.

How-To

The testing framework can be executed by executing **bash ./scripts/runAllTests** in the TeaMate_TESTINGSUITE directory via terminal.