Final Project – Part B
Group:Eric Anderson, Michael Byrne, Matthew Ailes
Term: Winter 2016

Introduction:

To test URLValidator, we began by examining what the function actually does in a white box fashion. The black box is simple enough—hand it a string and it returns in affirmative or not whether the given string is a valid URL—but the internal view allows us to look at the URLs we are choosing to test in the composite. This is uniquely important as URLs are by their very nature composite entities, typically consisting of a path, scheme, host, and query string.

Within URLValidator, URLs are divided more specifically into scheme, authority, port, path and query or queries. This division provides clear guidance as to partitioning our inputs.

Thus, we begin by partitioning a URL into the parts identified above. Based on that input partitioning, we then write manual and random tests (based on what we learned in class) to test these constituent parts on both a good validator and a bad validator.

The good, or reference validator will yield the test results we expect (both positive and negative). We then take the same test file and run it on the bad validator. The results will be incorrect and our objective is to find out why.

Our plan to test the validator is to first create a set of manual tests to locate bugs in a full URL. Next, we create partition tests to indicate which part of the URL validation process fails. Following the first two tests, random testing further tests combinations to ensure that each correct URL segment works when integrated with its other components.

Finally, to determine root causes, we step through the code with the debugger and isolate the errant code. Based on this data, we are to identify bugs that indicate the issues we find.

Manual testing:

First, we define a root URL known to be good: http://www.google.com:80/test?query=test

Like the good UrlValidatorTest.java sample, for manual testing, we call the isValid() method given with URLValidator on different possible inputs, both valid and invalid. Each set of manual tests is run with a new UrlValidator instance.

For example:

```
 // test http://
    System.out.println("http://www.google.com" + "\t\t\t\tResult: " +
urlVal.isValid("http://www.google.com"));

    // test no http://
```

```
    System.out.println("www.google.com" + "\t\t\t\t\tResult: " +
urlVal.isValid("www.google.com"));
```

Based on the parts of our URL, we define both valid and invalid, complete and semi-complete URLS. The sets and results are illustrated in the following figures.

Scheme/Authority	Expected	Actual
http://www.google.com	True	True
h3t://www.google.com	True	True
://www.google.com	False	False
http:/www.google.com	False	False
“ “ www.google.com	True	False

Figure 1.1

We thus receive an unexpected, erroneous result in the final manual test, which tests for an empty string as scheme. From here, we manually tested other input partitions, including authority, port, path, and query. The results given default options can be seen in the tables below. Further results with varying options, including no-fragments, allow-all-schemes, and allow-local-URLs can be found in the attached text file.

Authority	Expected	Actual
http://www.google.au	True	True
http://<blank>	False	False
http://www.google~.com	False	False
http://www.google	False	False
http://my.hostname.com	False	True
http://1.2.3	False	False

Figure 1.2

Port	Expected	Actual
http://www.google.com:80	True	True
http://www.google.com:800	True	True
http://www.google.com:8000	True	False
http://www.google.com:80000	True	False
http://www.google.com:	False	False
http://www.google.com:-80	False	False
http://www.google.com:80a	False	False

Figure 1.3

Path	Expected	Actual
http://www.google.com/test	True	True
http://www.google.com/	True	True
http://www.google.comtest	False	False
http://www.google.com//test	False	False
http://www.google.com/test/. .	False	False
http://www.google.com/test/. ./	False	False
http://www.google.com/test//	False	True

Figure 1.4

Path Options	Expected	Actual
http://www.google.com/test/test	True	True
http://www.google.com/test/	True	True
http://www.google.com/testtest	False	False
http://www.google.comtesttest	False	False
http://www.google.com/test//test	False	False

Figure 1.5

Queries	Expected	Actual
http://www.google.com?query=test	True	False
http://www.google.com?query=test&hi=hello	True	False
http://www.google.com?query=	False	False
http://www.google.com=test	False	False
http://www.google.com??query=test	False	False

Figure 1.6

Thus, manual testing revealed errors within every partition except for path options.

Partition testing:

This is somewhat similar to manual testing except rather than building a complete URL string, we test the individual parts of a URL mutually exclusively. This was accomplished by

testing the constituent isValid() methods, including isValidScheme(), isValidAuthority(), isValidPath, and isValidQuery(). The port partition was tested using isValidAuthority() with a single valid authority and varying appended ports. To wit:

```
String[] testPort = {"www.google.com:80", "www.google.com:800", "www.google.com:8000",
"www.google.com:80000", "www.google.com:", "www.google.com:-80", "www.google.com:80a"};
    UrlValidator portVal = new UrlValidator(testPort, 0);
```

Using the same root string and the five partitions we defined above in the manual testing section the partition tests look like the following, with the following results:

Scheme	Expected	Actual
http://www.google.com	True	True
h3t://www.google.com	True	True
://	False	False
:/	False	False
“ “	True	True

Figure 2.1

Authority	Expected	Actual
www.google.com	True	True
my.hostname.com	False	True
www.google	False	False
www.google~.com	False	False
“ “	False	True

Figure 2.2

Port	Expected	Actual
80	True	True
80000	False	True
-80	False	False
80a	False	False

:	False	True

Figure 2.3

Path	Expected	Actual
/test	True	True
/	True	True
test	False	False
//test	False	False
..	False	True

Figure 2.4

Queries	Expected	Actual
?query=test	True	True
?query=	True	True
?query	False	False
??query=test	True	False
?	True	True

Figure 2.5

Programmatic testing:

For the programming-based testing we take valid URL parts from our partitions and create arrays for each. Then, in a loop, each URL component is selected from the array in the proper order to form a valid address. The address is then tested until all possible combinations with the proper order are completed. Each address is tested using isValid(). If *false* is returned the faulty URL is stored in an array. These arrays of programmatically-generated failed URLs are printed at the end of the testing runs to notify us of addresses that caused errors.

Here is how we set up the arrays (failure and sub-URL):

```
String[] badUrls = new String[length];
 String[] trueSchemes = {"http://", "", "h3tp://"};
 String[] trueAuthority = {"www.google.com", "google.com"};
 String[] truePort = {":80", ":9", ":100"};
 String[] truePath = {"/test", "/"};
 String[] trueQueries = {"?query=test", " "};
```

The array indices are selected at random, resulting in random combinations of predetermined URL segments. The scheme segment is randomized as such:

```
int schemeInt = (int) (Math.random() * 3);
```

After the random selections are complete, the test pastes them together and runs the target function on the final string:

```
 String url = trueSchemes[schemeInt] + trueAuthority[authorityInt] + truePort[portInt] +
truePath[pathInt] + trueQueries[queriesInt];
      UrlValidator validator = new UrlValidator();

      // check if it was marked valid
      boolean valid = validator.isValid(url);
```

Using this procedure, we returned 10 erroneous results. The following valid URLs were rejected by the isValid() method under test:

```
google.com:100/?query=test
www.google.com:100/
http://www.google.com:80/
http://google.com:80/
http://google.com:100/?query=test
http://google.com:9/?query=test
http://google.com:9/
h3tp://www.google.com:9/test
www.google.com:100/test
www.google.com:100/?query=test
```

To continue unit testing, we tested the validity of individual URL segments, a similar but programmatic approach to the segment testing done manually. In testing valid query segments, we returned three incorrect negative answers:

```
Testing query ?action=view    Result: false
Testing query null            Result: false
Testing query ?page=1&test=0        Result: false
```

For programmatically testing paths and authorities, we applied a somewhat different approach. Five preselected invalid paths/authorities and five preselected valid paths/authorities were added to string arrays, which were then fed into loops that ran each array entry against the isValidPath() and isValidAuthority() methods. Counts of valid/invalid segments were compared for discrepancies. This approach was taken to provide a framework that can more easily scale up for a fully randomized approach employing long sequences of URL segments.

NAME OF YOUR TESTS

public void testYourFirstPartition() - Tested isValidScheme(), scheme partition
public void testYourSecondPartition() - Tested isValidAuthority(), authority partition
public void testYourThirdPartition() - Tested isValidAuthority() with added port partition
public void testYourFourthPartition() - Tested isValidPath, path partition
public void testYourFifthPartition() - Tested isValidQuery, query partition

public void testIsValid() - loops through the url parts in proper combination, Failed addresses stored in an array and printed.
public void testisValidQueryUnitTest() - Similar to testYourFirstPartition, but looks at the complete whole string, therefore creating a more of a unit test of a URL string.
public void testIsValidSchemeUnitTest() - Like testisValidQueryUnitTest, this test eliminates partition errors, and directly tests isValidScheme.

How did we work together as a team
Communication for this project was handled exclusively through email and work was shared back and forth. Work was done and split up through email as well. It was all done in a type of volunteer system and everything was completed as time allowed.

Use of Agan's principles

Understand the System: Project Part A got the group familiar with the system at a base level. We then each spent time watching the instructional videos, reading the forums and, walking through the URLValidator with the debugger (adding watches for our local variables. We worked to understand when each function was called, what it was intended to do, and how it impacted the overall result. This was all written up in Part A.

Make it Fail: We started with what we knew was a buggy product! In our manual tests and our partition tests we purposefully put in both valid and invalid cases. Invalid cases were created to make the code fail to test if the failure was handled correctly, We adjusted our oracle when this was not the case. Similarly, we also included cases we knew should pass. Because of the wrong output, we were able to verify that there were bugs.

Divide and Conquer: Michael took the manual tests, Eric the partition tests, and Matthew the programing tests. The benefit if this was that we independently came up with similar tests and in some cases augmented test cases based on what we saw in each other's tests. When we saw issues, we each "owned" the bug and dug in with the debugger and determined root cause. Then each of us entered the bug. Way more coverage and results were achieved due to this principle.

Change One Thing at a Time: This was critical for the isValidQuery bug with the "?" separator. This made all non-null queries fail when they should have passed. We had to take a string at a time and then a part of the string at a time to find this subtle bug. Everything looked right but step by step discovery revealed what was happening.

Keep an Audit Trail: When we defined our tests, we wrote an audit trail of what we expected and why. When we ran the tests, we had the test generate an audit trail for us. This made it real easy to isolate buggy code (when the audit trail didn't match. We could improve this code by adding the expected results and doing asserts. The program then would have been self-auditing.

Bug Reports:

Title: [URLValidator] isValidQuery() function returns opposite result than intended
Project: URLValidator
Version: 1.4
Type: Bug, major
Reported by: Michael Byrne Email: byrnemi@oregonstate.edu
Created: 03/12/16
Updated: N/A
Resolved: N/A
Type: Bug
Status: Open
Priority: Major
Resolution: Not Resolved
Affected Version: 1.4
File name: UrlValidator.java
Environment: Windows 10, Eclipse Mars, Java 1.8
Description: The isValidQuery() function, which pattern matches the query segment of a URL input using regular expressions, returns the opposite result. While it should return true on a correct query, it returns false. This is because of an erronous logical NOT operation on the return value.
Code, line 446:
Correctly, the code should read:

```
return QUERY_PATTERN.matcher(query).matches();
```

However, the following occurs:

```
return !QUERY_PATTERN.matcher(query).matches();
```

Title: [URLValidator] Validator port regEx doesn't handle 4 or 5 digits port numbers correctly
Project: URLValidator
Version: 1.4
Type: Bug, major
Reported by: Eric Anderson Email: anderse7@oregonstate.edu

Created: 03/03/16
Updated: N/A
Resolved: N/A
Type: Bug
Status: Open
Priority: Major
Resolution: Not Resolved
Affected Version: 1.4
File name: UrlValidator.java
Environment: Mac OSX 10.10.5, Eclipse Mars, Java 1.7
Description: The isValid() method incorrectly determines that URLs with port lengths of 4 or 5 digits are invalid. Manual testing shows this bug when calling isValid() as follows:

```
System.out.println("http://www.google.com:80");
System.out.println(urlVal.isValid("http://www.google.com:80"));
System.out.println("http://www.google.com:800");
System.out.println(urlVal.isValid("http://www.google.com:800"));
System.out.println("http://www.google.com:8000");
System.out.println(urlVal.isValid("http://www.google.com:8000"));
System.out.println("http://www.google.com:80000");
System.out.println(urlVal.isValid("http://www.google.com:80000"));
```

For port 80 and 800, the expected values are true, and the actual values are true. However, for ports 8000 and 80000, the expected values are true, but the actual values are false.
Code Causing Bug:
URLValidator.java line 158

```
private static final String PORT_REGEX = "^:(\\d{1,3})$";
```

Correctly, the code should read:

```
private static final String PORT_REGEX = "^:(\\d{1,5})$";
```

Title: [URLValidator] Incorrect regex in query string pattern definition
Project: URLValidator
Version: 1.4
Type: Bug, major
Reported by: Michael Byrne Email: byrnemi@oregonstate.edu
Created: 03/12/16
Updated: N/A
Resolved: N/A
Type: Bug
Status: Open
Priority: Major
Resolution: Not Resolved
Affected Version: 1.4
File name: UrlValidator.java
Environment: Windows 10, Eclipse Mars, Java 1.8
Description: The validator attempts to match an incorrect query pattern. The addition of the regex “*” should cause the function to return a match on 0 characters.
Code Causing Bug:

URLValidator.java line 151

```
private static final String QUERY_REGEX = "^(.*)$";
```

Should be:

```
private static final String QUERY_REGEX = "^(.)$";
```