URLValidator.txt

Team Members:
-David Profio
-Christopher French Loomis
-Nathan Thunem
CS362 Winter 2016

Manual testing - manualTest02.java
URLS tested:
-"http://www.google.com"
-"http://" -"https://www.google.com/webhp?sourceid=chrome-instant&ion=1&espv=2&ie=UTF-8#q=cat%20videos
-"http://www.google.com?action=view"
-"http://www.google.com/$23"
-”http://www.amazon.uk”

Random testing - URLValidatorTest.java
Unit testing - portTest01.java

Teamwork details:

Each member in our group proactively worked on the project. We discussed the details of the project, making sure that each member understood the requirements. After that, each member then took it upon themselves, as time allowed, to work on a different part of the project. Once one part was done, we communicated to the group through Google Hangouts what had been completed and the work was posted in a shared google Drive folder. We would collaborate on the work done and edit it if necessary based on group commentary. Once one part was done, we continued on towards other parts of the project, repeating the above process. Once all of the project requirements were met, we all edited the project individually then talked about the final submission as a group. This strategy allowed us to comfortably complete the project.

Unit Tests:

For our unit tests instead of writing individual unit tests we decided to utilize a semi-random tester. In order to do this we created a new test object that stored a value, a description, and a boolean for if it is a valid object. These objects are actually parts of the url broken up into three sections. The following is the format: 1[http://] 2[www.google.com] 3[:80/hello/?action=hello]. We created several objects of each of these three sections testing various elements and boundaries. We add these test objects to three different lists. Each list represents a separate section of the url. We then use their size to as an upper limit for random number generation.

This allows us to create a valid url randomly while also maintaining knowledge of if it is valid and a description. We create our expected result using that is valid information. We then build the test url out of the values of the objects. If result does not match our expected we print the result as well as the descriptions that we created.

We run the above code in a loop of 1000. After our initial run we realized that the descriptions are actually not as helpful as seeing the tested url. We left the descriptions in there in case we can think of another reason that they would be useful later. After making the changes to printing the test url's when they error we thought it would be helpful to see a count for how often a certain url piece is failing. For this reason we created a map to store this information.

The map works by storing the string value as the key, and then we store an integer as the value. Whenever we encounter an error we first check if the map contains the url piece. If it does we retrieve the integer and increase it by one. If it does not then we add it to the map with a value of 1. We repeat this two more times with the remaining pieces of the errored url. After the for loop has ended we iterate through the keys and values and print them. This was very helpful as seeing how often certain pieces error gave us a good direction for manual testing.

Partitioning testing:

For the partitioning we felt that splitting up the partitions into three would be logical as there are two major classes for valid url's and then we would have one partition test for an invalid url. We wanted to test for as many of the possible parts as possible so for our first partition test we test an http:// protocol with a path of www.amazon.com/test and then a query string parameter of ?hello=there. For the second partition we wanted to test the second major class of possible inputs and that is instead of having a path with domain we used an IP address with a port number. For this test we utilized ftp:// as the protocol, 55.253.251.255 as the IP address, and :80 as the port number. This means that we have tested with and without query string parameters, IP addresses and typical path with domain, with and without port number, with and without extra paths being specified, and multiple protocols.

For the final partition test we chose an invalid URL. We utilized a valid URL, but instead of a domain of .com we utilized a domain of .p which should cause a false to be returned. It is much harder to test for false values as once one value is false it makes the rest of the values false. This means that with each test you can only test one piece of a URL being false whereas with true we are able to test multiple pieces at the same time.

Agan's Principles

Several of Agan's principles were used while debugging URLValidator. Part A of the final project correlates to Agan's first principle, "Understand the System", as we looked over URLValidator and described how and why it worked.

Agan's second principle, "Make it Fail", is essential to proper debugging. Before looking for

where the failure occurs within the code, we had to show that the code reliably fails at a particular point by narrowing down what inputs were causing the error. Looking at the manual testing (manualTest02.java) you can see how after some basic tests, the query bug was discovered and confirmed by testing other URLs with query strings and URLs without query strings.

Agan's third principle, "Quit Thinking and Look", like the second principle, is essential. Once we believed a bug was properly identified we ran tests in Eclipse debugger to locate the line where the error occurred.

Agan's fourth principle, "Divide and Conquer", was utilized in Eclipse. As breakpoints were put at various points within the code, it allowed us to very identify the line of code that produced the failure.

Agan's fifth principle, "Change One Thing at a Time", goes hand in hand with the previous principles. Once you have made it fail, you must begin looking, but before that you must narrow down what you are looking for. Changing a single variable at a time allows us to narrow down what we are to look for.

Agan's sixth principle, "Keep an Audit Trail", was exercised out of necessity. Since this is a group project and we are working from different locations and times via the internet, a proper audit trail is produced to allow all members to stay on the same page.

Agan's eighth principle, "Get a Fresh View", much like the sixth, was exercised naturally due to our group working remotely. Have group members looking at the same code at different times produced fresh views every time any of us worked on the project.

Bug reports

Bug 1: queryBug

URLs with a query string marked invalid

UrlValidator.java
isValidQuery() function - return statement - line 446

Serious Bug

Can be reproduced everytime

Run UrlValidator.java
Pass any URL with a query (ie - "http://www.google.com?action=view") to method isValid()
Always returns false when passed anything except null (it is invalid)

isValid() should return true, as these are legitimate URLs

This bug does not result in a crash, but is noted as a serious bug as it's function is wrong.
It is marked serious as this product cannot ship with this bug, due to being fundamentally wrong.

Bug manifests at line 446 of URLValidator.java

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

This returns the opposite boolean it should.

Bug 2: PORT_REGEX
URLs with a port length greater than 3 digits are marked invalid

URLValidator.java
PORT_REGEX initialization - line 158

Serious Bug

Can be reproduced every time

Run URLValidator.java
Pass any URL with a port length greater than 3 (i.e. ":65535") to method isValid()
Always returns false when passed anything with porth length greater than 3

isValid() should return true, as these are legitimate URLs

This bug does not result in a crash, but is noted as a serious bug as it's function is wrong.
It is marked serious as this product cannot ship with this bug, due to being fundamentally wrong.

Bug manifests at line 393 due to line 158 of URLValidator.java

393 if (!***PORT_PATTERN***.matcher(port).matches())
159 private static final Pattern ***PORT_PATTERN*** = Pattern.compile(***PORT_REGEX***);
158 private static final String ***PORT_REGEX*** = "^:(\\d{1,***3***})$";

This causes port numbers greater than 3 to fail.

Bug 3:

URLs with a domain of uk fail incorrectty.

DomainValidator.java
Line 248 private static final String[] COUNTRY_CODE_TLDS = new String[] {

Serious Bug

Can be reproduced every time

Run URLValidator.java
Pass any URL with a domain of .uk
Always returns false

isValid() should return true, as this is a valid country domain

This bug does not result in a crash, but is noted as a serious bug as it's function is wrong.

It is marked serious as this product cannot ship with this bug, due to being fundamentally wrong.

Bug manifests at line 192 as a result of not being contained in list of possible options. It appears all of the country domains after Italy have been removed.

```
192    return COUNTRY_CODE_TLD_LIST.contains(chompLeadingDot(ccTld.toLowerCase()));
368    private static final List COUNTRY_CODE_TLD_LIST = Arrays.asList(COUNTRY_CODE_TLDS);
248    private static final String[] COUNTRY_CODE_TLDS = new String[] {
```