Group:
Josh Ribera
Ava Petley
Kyle Johnson

Final Part B

Manual Tests

For manual testing we used the testManualTest() method in the UrlValidatorTest.java file by replacing the default url with those the be tested. We called the isValid method on 30 distinct urls, both valid and invalid, and compared the actual result returned to the expected result for each manual test. The urls were not selected ahead of time. Finding a bug with one url would often inspire the selection of the next url. Which was along the lines of Agan's Principle "Make it Fail". We used Agan's Principle "Understanding the System" because we know how the code partitions and checks each piece so if we are getting faulty results when we alter the authority section then we know where in the code to look for errors and how to test to bring out those errors. The principle "Change on Thing at a Time" also applies here because we looked first at a valid url returning as valid change one part of the url and see what results.

Examples of urls used for manual testing:

Valid:
http://www.amazon.com
http://www.oregon.gov/pages/index.aspx
https://hangouts.google.com/

Invalid:
gttp://google.com
http:/amazon.com
http://www*.carnival.com

The manual test revealed four errors (see Bug Report below). Some urls that were valid were returned invalid and some that were invalid returned valid.

Input Partitioning

The input was partitioned by each individual URL component. This is because these are all of the parts used to build a valid or invalid url, and because there are validations for each

component in the code such as isValidScheme(), isValidAuthoritity(), etc. Each of these validations is used within the isValid() to test a given url, so it made sense to test each of these components as its own partition. The isValid() method was called against a URL with 1 of the 5 components using an invalid input so that all input components were tested.

A function called createPartition was written to generate a URL with flags for each component to be set at a valid or invalid value. For example, createPartition(1, 0, 0, 0, 0) would generate a URL with an invalid scheme and all other components valid.

The testPartitions function then generated urls where 1 of the 5 components was set as invalid with the others as valid so that each component was checked individually and then the results from isValid for the generated urls were checked:

----------Testing input partitions----------

PASSED - Invalid scheme: http:www.google.com:80/t1/?action=view failed.

PASSED - Invalid authority: http://go.a::80/t1/?action=view failed.

PASSED - Invalid port: http://www.google.com:65a/t1/?action=view failed.

PASSED - Invalid path: http://www.google.com:80/..?action=view failed.

PASSED - Invalid query: http://www.google.com:80/t1/?action=)view failed.

----------Testing of input partitions complete----------

Programming Based Testing

We built the URL by creating an array of strings for each component using the original validator test code. We broke them up into true-arrays and false-arrays where all of the values in those arrays are true or false. The next thing we did is build the string using a randomized component for all of the components. If the test passed -- meaning tests that should have failed returned false and tests that should have passed returned true, then it would simply print . pass . but if the test failed then it would print the URL that failed along with the actual and expected result.

Here are some examples of strings that are expected to be true but reported as false:

http://go.au:65535/test1?action=edit&mode=up
http://go.com:80?action=view
https://www.google.com:65636/test1/?action=edit&mode=up

https://www.google.com:65636/test1/?action=edit&mode=up
http://255.255.255.255:0?action=view
https://go.cc:65535/test1

Because the we ran this test 1,000,000 times it was easy to spot the possible bugs by looking at the URL strings. It seems that strings built with ports 65535 and 65636 were failing along with URLS with query parameters.

Since the strings are generated with random components if one part fails they all do so for debugging purposes we dissected a URL:

https://www.google.com:65636/test1/?action=edit&mode=up
First we tested https://www.google.com and it passed.
Then we tested https://www.google.com:65636 and it failed. So now we know that the bug is to do something with the port part. Now omitting the port we continued to test the string.
Now we tested https://www.google.com/test1/ and it passed
Lastly we tested https://www.google.com/test1/?action=edit&mode=up and it failed once again -- meaning that another problem lied within the query parameter.

Name of Tests

Manual:
testManualTest() method in the UrlValidatorTest.java

Input Partitioning:
createPartition(int schemeN, int authN, int portN, int pathN, int queryN) method in UrlValidatorTest.java

testPartitions() method in UrlValidatorTest.java

Unit Tests:

testIsValid() method in UrlValidatorTest.java
testAnyOtherUnitTest() method in UrlValidatorTest.java

Working Together as a Team

Our team worked together effectively for many reasons. First, when we created the group we all had the same schedule which allowed us to communicate effectively. We would use Google's

Hangouts as our primary means of communication by simply using the chat feature. If any problems, or questions arose, we would ask them on Hangouts.

To determine how to divide the work we all met on Hangouts and broke the assignment into sub-parts which we then volunteered to complete. Ava performed the manual test, Kyle the input partitioning and Josh the programming based testing.When our individual parts were completed we simply pushed it to Github to allow our other teammates to view the results, or code. We all worked together on the report and the debugging.

To begin working on our report, we created a Google Doc which served as our primary collaboration platform. We would all add our own testing results and findings and contribute to the report as time went on. By using a Google Doc other teammates would easily be able to begin where the other has left off.

Use of Agan's Principles

Understanding the System We definitely used this principle from Agan extensively because of part a in the project. We entered this project with little or no knowledge of the URL Validator and had to be brought up to speed on their code base. I think we gained a lot of knowledge of the URL validator by describing what and how the isValid function accomplishes.

Make it Fail One of the principles we used was to create URLS we knew to be invalid and use that URL to test the function. For example, in our manual and partition tests we tested code that we knew to be incorrect.

Quit Thinking and Look This principle was extremely helpful to find the exact location of our bugs through use of the debugger. However, it also had us go line by line which took a long time and often felt like we weren't being as productive as we liked.

Divide and Conquer We used this principle in all of our tests. It was also helpful with the partition tests to divide down to sub components to determine which part of the URL was failing. Additionally, when there was an error we would use the debugger to narrow down to the most likely part that failed.

Change One Thing at a Time We used this principle with our unit test. Once we got a URL that failed the test we broke that particular URL down into its components and tested those one at a time. This helped to isolate the area where the problem occurred.

Keep an Audit Trail Because we used collaboration tools like Github, Google hangouts, and Google Docs we always had an audit trail due to either our commits in github or our edits in

Google Docs. We also kept track of what was or wasn't working by communicating with each other through Google Hangouts.

Bug Report

Bug 1

File: UrlValidator.java
Method: isValid()
Line: 446

Summary: URL's with query parameters are marked false.
Priority: Severe
Reproducible: Yes
Steps: Create UrlValidator Object
System.out.println(validatorObject.isValid(www.google.com:80/?action=view));
Expected: True | Actual: False

Description
Any valid queries are rejected and returned as false the only query parameter returning true is NULL. Could also investigate the function that's being called within the code at line 441: isValidQuery().

Bug 2

File: UrlValidator.java
Method: isValid()
Line: 305

Summary: Port values > 3 in length are invalid. Should support values up to 5 digits (65535).
Priority: Severe
Reproducible: Yes
Steps: Create UrlValidator Object
System.out.println(validatorObject.isValid(www.google.com:50000));
Expected: True | Actual: False

Description
Any port values greater than 3 in length are determined invalid.

Bug 3:

File: UrlValidator.java
Method: isValidQuery()
Line: 446
Summary: valid urls with variables not returning as valid
Priority: Severe
Reproducible: Yes
Enter the following url's in the manual testing section:
https://oregonstate.instructure.com/courses/1568425/assignments/6656128?module_item_id=16591774
https://www.petfinder.com/pet-search?location=97401&animal=cat&breed=
expected: true
actual: false
Description: All url's with any not null query returned as invalid. Using Eclipse debugger we traced the failure to line 446: return !QUERY_PATTERN.matcher(query).matches(); If the query matches the pattern the opposite it returned. Removing the character ! from this line should solve this error.
Patch: Change line 446 to: return QUERY_PATTERN.matcher(query).matches();

Bug 4:

File: UrlValidator.java
Method: isValidAuthority()
Line : 367
Summary: letter or number in subdomain returning as valid.
Priority: Severe
Reproducible: Yes
Enter either of the following urls into the manual testing section
http://wwww.carnival.com
http://www0.carnival.com
expected: false
actual: true
Description: There is no checker on the length of the subdomain. I entered urls with several w's and it returned as true. Also with integers it returns true.

Bug 5:

File: UrlValidator.java
Method: isValid()
Line : 344
Summary: Incorrect protocol returning as valid

Priority: Severe
Reproducible: Yes
Enter the following url into the manual testing section.
gttp://google.com
expected: false
actual: true
Description: Any scheme is allowed so url that would be invalid are returned as valid. This would be solved if allowed schemes are specified.

Bug 6:

File: UrlValidator.java
Method: isValidAuthority()
Line: 399
Summary: Url's with whitespace at the end of the authority are returned as valid
Priority: Moderate
Reproducible:
Enter the following url in manual testing section
"http://www.robdixoniii.com /wp-content/uploads/2011/11/uri-segments.png"
Expected: False
actual: True
Description: The whitespace is trimmed off on line 399 so urls with whitespace in the aurthority are returned as valid instead in invalid. To be sure that the second part of the url (after the whitespace) was still being passed in and checked. The url
"http://www.robdixoniii.com */wp-content/uploads/2011/11/uri-segments.png"
was checked and was correctly defined as invalid because of the character.
Patch: Change line 399 to if (extra != null && extra.length() > 0){