Methodology of Testing

Our team employed functional testing using manual- and unit-testing techniques that were modelled after (but not copied from) a test suite utilized when working through part A of the project.

A function was authored that builds URLs from most of the standard URL elements as defined in RFC 3986 ("Uniform Resource Identifier (URI): Generic Syntax"). For each URL element covered by the function, variations in strings were added to test known-good URLs as well as edge cases and boundaries (i.e. as outer limits of ports, upper ranges of IP address octets).

Manual testing was employed to test the test suite and to get a high level "feel" for what passes/fails in UrlValidator.java. From there, automated unit testing was established to quickly build and test all possible variations of our URL partitioning. This was by far the most revealing of the processes, presenting failures that would have taken days to reach had we stuck with manual testing.

When bugs were found, Eclipse debugging and console output were employed to investigate where bugs were located as well as the nature of the bugs. Eclipse debugging was found to be subpar in many ways but still provided some useful hints.

JUnit was used as the testing system. JUnit is useful but has a long, contested history of shutting down tests when the first assertion fails. Trying/catching assertions works but then assertions are not registered by JUnit so even when tests fail, JUnit reports success. Attempts to establish an error collection (which JUnit has created to get around this issue) were ultimately futile. For these and other reasons, simple printed output indicating pass/fail status was occasionally employed. (Note: we could have gotten past JUnit assertion failures by fixing the bugs, but that was not part of our task.)

Manual Testing

Manual tests were performed with a variety of inputs. This generated a base-level of test cases to better inform us on the current state of the URLValidator method.

Testing included a variety of valid (http://www.amazon.com) and invalid schemas (htt://ww.amazon.com). We opened the scheme testing to include all known schemes (**UrlValidator.ALLOW_ALL_SCHEMES**) as well as a restricted set of schemes (only http, https, and ftp, enabled by specifying the restricted set to the **UrlValidator** object constructor) and passed in valid cases for these sets as well as URLs with schemes outside of the sets. Example URLs for schemes outside the restricted set include: ssh://www.tower.com and rtsp://www.boing.net.

Manual testing continued with local URL testing enabled (**UrlValidator.ALLOW_LOCAL_URLS**). These tests included valid local URLs such as http://localhost, http://localhost/index.php#contact, and http://127.0.0.1 as well as invalid local URLs such as http://localhost.mymachine.

Testing then moved on to typos in paths. The allowance of extra slashes in path strings (**UrlValidator.ALLOW_2_SLASHES**) is, of course, invalid, but it is a common typo in systems where anchors are generated by scripts. Tests for the allowance of 2 slashes in paths included http://www.example.com//blog.

Fragment testing in UrlValidator is not very robust. Essentially, you can test whether a fragment is allowed in the URL or not (**UrlValidator.NO_FRAGMENTS**). URL fragments are governed by few rules and are generally hard to pin down, so it makes some sense to not go deep with their tests. We tested URLs with and without fragments including http://www.foobar.com/file.php#aboutus.

Results in every case were evaluated from the return values of the assertTrue and assertFalse methods.

Partitioning

For partitioning, we first separated a URL into the components being tested by the isValid() method: scheme, authority, path, query, and fragment. And then we were able to further partition the URL within each of these components based on what we expected to be a valid or invalid component. So, for instance, knowing that valid port numbers are in the range of 0 to 65535, we tested for URLs with port components inside of this range (80 and 8080, for instance), as well as edge cases (65535 and 65536), and outside of the range (negative numbers, or port numbers containing letters). Or for valid schemes, knowing that the default UrlValidator() constructor accepts only “http”, “https”, and “ftp”, we knew that we could test against scheme components that are not these three. We also included the empty string in each of the components for testing URLs with a particular component absent (e.g., URLs without a scheme (like “www.amazon.com”).

Each of the pieces of these five components were assigned to an array of ResultPairs where the “valid” member variable was assigned a true or false value based on whether or not we expected the piece to “break” the URL, and URLs for testing were concatenated from pieces of these arrays, where any false “valids” marked the URL as an expected failure. The component arrays were then looped through in a sequence of nested loops, testing all combinations against isValid() each time a new URL was compiled.

Through this process, we were able to identify a number of bugs, like for instance that some port numbers we expected to pass were failing. Or that query parameters we expected to pass were failing.

Unit Tests

Our tests were contained within a single file: **UrlValidatorTest.java**. This was modelled after the similarly named test suite that we encountered in part A of this project.

For the initial test-of-the-test-suite, two partition test functions were created: **testYourFirstPartition()** and **testYourSecondPartition()**. These functions check gave us two simple tools to check how well the test system works by offering quick and definitive feedback on whether or not assertions were passing/failing correctly. Confidence in the results from these functions allowed us to move forward with unit testing.

A unit test loop was set up—**testAnyOtherUnitTest()**—that loops through all combination of the URL elements once. Tests continue through the set until either all tests pass or an assertion fails. The looping starts with variations of common, uncommon, and invalid URL schemes (i.e. http:// and https:// as well as http:/// and foobar://), then it moves through multiple variations of valid and invalid hosts (i.e. www.amazon.com as well as botched hosts like 127.1), then on to ports, paths, and query strings. Best we

could tell, fragment testing was toothless (about as complex as “is there a fragment” or “is there no fragment”) so there were really no variations of fragments to test.

In addition to helping “debug” our test suite (we originally thought empty schemes were valid, but after multiple failed tests with empty schemes we investigated and discovered that they are actually invalid) the automated unit tests quickly revealed issues with local URLs and query strings. As mentioned earlier, unit testing (JUnit in particular) is made more difficult when assertion failures halt program execution and cancel subsequent tests. For this reason, the automated loop was not as useful at first as we had hoped, it did reveal very quickly where testing in general was failing, but ultimately we decided to move this version of the test into the **testIsValid()** method, where, instead of asserting failures, we logged to the console any time isValid()’s returned a result that differed from the result we expected. In this manner, we were provided with a log of failures that could be studied for the identification of consistent failure patterns. This proved much more useful for identifying bugs.

Failure Localization and Debugging

Agans’ Rules

1. Understand the System
2. Make it Fail
3. Quit Thinking and Look
4. Divide and conquer
5. Change One Thing at a Time
6. Keep an Audit Trail
7. Check The Plug
8. Get a Fresh View
9. If You Didn’t Fix it , it Ain’t Fixed

We first started by reviewing the isValid() method in the URLValidator to better understand the structure of the method, how it works, and what the expected outputs are. This follows closely to Agans’ Rule #1 of “Understanding the System”. This gave us the foundation of knowledge we needed to build our test cases for debugging. Based on what we learned of the isValid() method, we built cases that we knew should pass and fail the various checks in place. This follows closely to Agans’ rule #2 of “Make It Fail”, by giving us a starting point of failure in the code. Upon running the test cases through the code, we noticed certain failures that required further investigation. At this point, we entered into debug mode to analyze those points more closely. This mimics Agans’ rule #3 of “Quit Thinking and Look”. At this point, the team would communicate to one another when a potential bug was found, and their efforts to isolate the source. Further discussion between team members would help to determine if it is indeed a bug in the code, if the unit tests are working as expected, and other possible sources of failure. These steps demonstrate Agans’ rules #6-8. Upon further investigation, we were able to point to the source of the bug, and at times test it by fixing the error and rerunning the tests. This allowed us to confirm that the failures were no longer occurring, and if done with no other changes to the code, effectively removed. This follows Agans’ rules #4 and #5 of “Divide and Conquer” and “Change One Thing at a Time”.

Team Collaboration

The team was highly collaborative and shared in all aspects of testing. Brian started us off with the URL builder function and an initial set of partitioning strings as well as some skeleton code for future test functions. Erik continued by adding partition strings, setting up some manual tests, and creating up the initial automated unit test loop. Susan contributed code and testing/debugging efforts throughout. Although we initially divided work three ways, in the end the work was distributed evenly.

Email was the primary mode of team-wide communication. Brian and Susan share a study group so some of their collaboration took place in-person, but any information that was discussed was forwarded along so we could all benefit.

Bug Reports(1)

Bug 1
==========
Title: isValidQuery()
Class: High
Date: 12/05/2015
ReportedBy: Susan Lee
Email: leehw@oregonstate.edu

File name: URLValidator.java
Line Number: 446
Is it reproducible: Yes
==========

Description

The isValidQuery() method fails when passing in a valid query, and passes when passing in an invalid query.

Steps to Produce/Reproduce

Pass in a valid URL such as “http://www.amazon.com” appended with a valid query, such as “?lvalue=rvalue”. Additionally, a longer query may be used by including an “&” between queries, such as “?lvalue=rvalue&lvalue2=rvalue2”

Alternatively, pass in a valid URL appended with an invalid query, such as “?=rvalueonly”.

Expected Results

Expected boolean return valid of true for valid queries, and false for invalid queries.

Actual Results

Boolean return valid of false for valid queries, and true for invalid queries.

Fault Localization

Return statement on line 446 in URLValidator.java, negates the return value of the query pattern matcher. This negates failures to successes and successes to failures. Remove the “!” from the return statement.

Bug 2

==========

Title:	isValidInet4Address()
Class:	Medium
Date:	12/05/2015
Reported By:	Erik Ratcliffe
Email:	ratclier@oregonstate.edu
File name:	InetAddressValidator.java
Line Number:	96
Is it reproducible:	Yes

==========

Description

This bug appears when verifying the validity of an IPv4 address subgroup. In particular, whether or not the value falls within the 0-255 range. The method returns true when an IP address with a value outside this range is passed.

Steps to Produce/Reproduce

Pass in a value outside the valid 0-255 range. An example of an invalid URL is ftp://256.0.0.1.

Expected Results

Boolean result of false.

Actual Results

Boolean result of true.

Fault Localization

The bug is located on line 96 in the InetAddressValidator.java file. The statement evaluates if an ip segments has a value greater than 255, and returns true if it does. Change the return value to false.

Title: Bug 3
==========
Title: error validating proper port numbers in isValidAuthority()
Class: High
Date: 12/05/2015
Reported By: Brian Lamere
Email: lamereb@oregonstate.edu

File name: UrlValidator.java
Line Number: 393
Is it reproducible: Yes
==========

Description

This method returns a false value for any port number greater than 999.

Steps to Produce/Reproduce

Test any valid URL with a port number having more than 3 digits, e.g. running the following code:

UrlValidator uval = new UrlValidator();
uval.isValid("https://www.amazon.com:8080");

...we would expect a return value of "true". However, it returns "false."

Expected Results
---
true

Actual Results
---
false

Fault Localization
---
line 393 attempts to match the URL String's port group against a compiled regex version of the PORT_REGEX pattern on line 158. Inspection of this pattern reveals that it tests for patterns that start with a colon and includes up to 3 digits. Because of this, any port number greater than 999 will fail.

(1) Bug Report template revised from the following source:

www.noverse.com/images/novers-bug-reporting-template.txt