CS362 URL Validator Report

Group Members:

Kevin To <toke@oregonstate.edu>
Sean Harrington <harrings@oregonstate.edu>
Carol Toro <toroc@oregonstate.edu>

Testing Methodology

Manual Unit Tests

For the manual unit tests, I ran 5 manual tests that focused on the parts of the URL: scheme, authority, port, path and query. I first focused on testing with valid parts since the expected result would be TRUE and then I focused on testing with invalid parts since the expected result would be false.

URLS tested:

- http://www.amazon.com/ → Expected: valid, Actual: valid
- https://www.amazon.com/index_2_4.html → Expected: valid, Actual: valid
- http://www.amazon.com/t123?action=view → Expected: valid, Actual: invalid
- https://www.a-.com/index_2_4.html → Expected: invalid, Actual: invalid
- hst://www.amazon.com/ → Expected: valid, Actual: valid

The third test which had all valid parts of the url, I encountered a failure of the software under test when the result of the test came out as FALSE and I was expecting for it be true. In order to localize the fault, I put a breakpoint on the third test and stepped into the test for each part of the url. This allowed me to check the system state at the various checkpoints that verify the various URL parts. By doing so I was able to understand that up until the check for query, everything was behaving as expected and that the problem stemmed from the isValidQuery function.

Partitioning Tests

For input partitioning I broke it into four main sections to try to represent the infinite set of possible inputs. The first section tests out a range of different valid and invalid authorities. We first test invalid authorities, if one of them passes we know there is a possible bug with checking invalid authorities. We then test valid authorities, if one fails there is a possible bug with checking valid authorities.

We then test ports with a range of different port inputs. The first section again checks invalid port inputs, if one of them passes then we know there is a bug causing invalid ports to be marked as valid. The next section tests a set of valid ports and if one comes back as a fail then we know there is a bug causing valid ports to be read as invalid.

For the third section we are testing schemes. We again first test invalid schemes, if one of them comes back valid we know there is a bug with invalid schemes being recognized. We then test valid schemes, if one fails we know there is a bug with valid schemes being flagged as invalid.

The fourth section tests out paths. We test invalid paths if one of them comes back valid we know there is a possible bug with invalid paths not being spotted. The next section tests valid paths and if one returns false then we know there is a possible bug.

Run Results (third trial):

```
Running First Partition Testing...
invalid authority failed as expected
valid authority failed, possible bug

Running Second Partition Testing...
invalid port failed as expected
valid port failed possible bug

Running Third Partition Testing...
invalid input schemes failed as expected
valid input schemes passed as expected

Running Fourth Partition Testing...
invalid path failed as expected
valid path passed as expected
```

Partitioning Results Discussion:

The sections above in red indicate possible bugs. The first bug is due to an authority being valid when it should not be. The IP address parsing is not correct. The second bug is due to the port being recorded as valid when it should not be.

Programmatic Testing

Testing method:

I wanted to make the programmatic tests easier to understand in case future developers have to debug through the test to figure out new features or debug failing tests. Readability mainly why I broke the programmatic tests into 4 unit tests. Here are the descriptions of each test:

- testProgrammicSchemeAuthority()
    - Loops through different combinations of schemes and authorities.

- testProgrammicPort()
    - Loops with different combinations of ports with all other components being valid.
- testProgrammicPath()
    - Loops with different combinations of paths with all other components being valid.
- testProgrammicQuery()
    - Loops with different combinations of queries with all other components being valid.

This testing method allowed me to test a large number of URLs without writing an individual test for each URL. The test data is organized into 5 different arrays. The arrays hold result objects that contain two properties, the component string and a boolean valid specifying whether the component is valid. This allowed for easier iteration for each component.

Urls that Failed:
http://www.google.com:80/blah?key=value → Expected: valid, Actual: invalid
http://www.google.com:80/blah?key==value → Expected: valid, Actual: invalid
http://www.google.com:80/blah?key===value → Expected: valid, Actual: invalid
http://www.google.com:80/blah?key=?value → Expected: valid, Actual: invalid
http://www.google.com:80/blah?key=??value → Expected: valid, Actual: invalid

http://www.google.com:80/? → Expected: valid, Actual: invalid
http://www.google.com:80/?? → Expected: valid, Actual: invalid

http://www.google.com:8080 → Expected: valid, Actual: invalid
http://www.google.com:80808 → Expected: valid, Actual: invalid

Debugging:

All the bugs were debugged through to find the original cause, and all the proposed fixes are also within each bug report.

We did use some of Agan's principles in debugging URLValidator. Here is a list of the principles and how well we adhered to it:

1. Understand the System:
    a. We did not fully adhere to these since we do not know everything about URLs. The team had a working knowledge of URLs in order to find superficial bugs.
2. Make it fail:
    a. When we found a bug, we tried different variations of it. All of our bugs are consistently reproducible.
3. Quit thinking and look:
    a. All the bugs were relatively easy to pinpoint and we did not have to provide instrumentation to narrow down where the bug was happening.
4. Divide and conquer:

    a. We used breakpoints to narrow down our search area. We were usually correct in our assumptions of there the bug was predicted to be. The method names in URLValidator were easy to understand.

5. Change one thing at a time:
    a. Our programmatic tests provides a way for us to test scenarios by changing one thing at a time.
6. Keep an audit trail:
    a. Our bugs were not complicated, so the debugging call stack was all that was needed for an audit trail.
7. Check the plug:
    a. Not really utilized because we had superficial bugs.
8. Get a fresh view:
    a. Not really utilized because we had superficial bugs.
9. If you don't fix it, it ain't fixed:
    a. Not applicable.

Test Method Names:

public void testManualTest()
public void testYourFirstPartition()
public void testYourSecondPartition()
public void testYourThirdPartition()
public void testYourFourthPartition()
public void testProgrammicSchemeAuthority()
public void testProgrammicPort()
public void testProgrammicPath()
public void testProgrammicQuery()

Team Work

Our team used the following collaborative tools to communicate and effectively complete the final project: email, Google chat, Google Drive, Google Document, and GitHub. Email was mostly used to outline tasks, divide up work, communicate progress and communicate what other tools we would be using to complete the project. We used Google chat when we had specific questions to our partners or needed clarification. We used Google Drive and Google Document to store this shared document that would allow each team member to make additions and edits based on their assigned tasks. The team also setup a team repository on GitHub outside of the class' repository that contained the buggy URL validator files. As a member completed assigned tests, the team member would push their changes to the repository to ensure that all tests would be tracked in the same file. This would make it easier for us at the end of the project to hand in the team's shared files.

The actual testing of the validator was split amongst the 3 team members: Manual Testing, Partitioning Testing, and Programmatic Testing. Originally, we also agreed that each team member would write up 1 bug report. The person working on the manual testing would also take on additional duties like completing ½ of the report, the coordination of the report and integration of all tests. The person working on partitioning testing would also work on completing ½ of the report.

Bug Reports

URLs with a valid query are not valid

Details **Type:** ● Bug **Status:** Open **Priority:** ↑ **Major** **Resolution:** Unresolved **Affects Version/s:** 1.4.0 Release **Fix Version/s:** None **Component/s:** None **Labels:** None	**People** **Assignee:** Unassigned **Reporter:** Carol Toro **Dates** **Created:** 11/29/15

Description
isValid returns false for following URL: http://www.amazon.com/t123?action=view
isValidQuery(String query) returns false on valid query when query is "action=view" (id=89)

How did you find it?
I found the failure by debugging with above URL when I expected to receive a result of True but instead received a result of False. The bug was found during manual testing.

What is the cause of that failure?
The return expression inside the protected method boolean isValidQuery(String query) is incorrect. It returns the exact opposite result from calling the method QUERY_PATTERN.matcher(query).matches().

Filename: UrlValidator.java
Line Number: fault is located on line 446

Actual Code on that line:
return !QUERY_PATTERN.matcher(query).matches();

The fix should be to change this line of code to:

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

URLs with invalid IP address are valid

Details

Type: Bug
Status: Open
Priority: **Major**
Resolution: Unresolved
Affects Version/s: 1.4.0 Release
Fix Version/s: None
Component/s: None
Labels: None

People
Assignee: Unassigned
Reporter: Kevin To

Dates
Created: 11/30/15

Description
These fail when they should pass:
http://1.1.1.256 → Expected: false. Actual: true
https://1.1.1.256 → Expected: false. Actual: true
http://256.256.256.256 → Expected: false. Actual: true

The max number for an IP address should be 255.255.255.255. The numbers can range from 0 - 255. The validator should return a false because these IP addresses are invalid.

How did you find it?
This bug was found during input partitioning testing and programmatic testing.

What is the cause of that failure?
An if statement is returning false when the IP address number is greater than 255. It should return false.

Filename: InetAddressValidator.java
Line Number: 96

Actual Code on that line:
```
if (iIpSegment > 255) {
        return true;
}
```
Fix:
```
if (iIpSegment > 255) {
        return false;
        }
```

URLs with valid port is flagged as invalid

Details	**People**
Type: ● Bug **Status:** Open **Priority:** ↑ **Major** **Resolution:** Unresolved **Affects Version/s:** 1.4.0 Release **Fix Version/s:** None **Component/s:** None **Labels:** None	**Assignee:** Unassigned **Reporter:** Kevin To **Dates** **Created:** 12/04/15

Description
These fail when they should pass:
http://www.google.com:8080 → Expected: valid, Actual: invalid
http://www.google.com:80808 → Expected: valid, Actual: invalid

URLs with port number count between 1 and 3, inclusive, all pass as valid URLs. URLs with port number count 4 or 5, fail the URL validation. Valid port number counts should be between 1 and 5, inclusive.

How did you find it?
We found it through programmatic testing. This means that we iterated through many different combinations of URLs and checking if their valid.

What is the cause of that failure?
The regex expression checking the port number length is incorrect. It is saying saying that lengths between 1 and 3 are valid, when it should be saying that lengths between 1 and 5 are valid.

Filename: UrlValidator.java
Line Number: 158

Actual Code on that line:
private static final String PORT_REGEX = "^:(\\d{1,3})$";

The fix should be to change this line of code to:

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