CS362 - Final Project Part B

Fall Quarter 2015

Charles Jenkins (jenkinch), Joseph Barlan (barlanj), Ty Hatton (hattont)

## Three Found Bugs:

### Bug 1: Valid URLs with port numbers above 999 do not pass the isValid method.

**Class:** Serious Bug

**File Name:** src/UrlValidator.java

**Line(s) where failure manifested itself:** 158

**Severity:** 1 - Critical Issue

**Date:** 11-29-2015

**Reported by:** Group

**Email:** barlanj@oregonstate.edu, jenkinch@oregonstate.edu, hattont@oregonstate.edu

**Product:** URL Validator **Version:** 1128446
**Platform:** Windows **Version:** 10
**URL example input:** https://google.com:1000
**Is it reproducible:** Yes

**Testing methodology used:** Manual and Input Testing (of the port schema partition)

**Description:**

1. What is the failure?
   a. Any valid URL with a port number above 999 will fail the isValid() method. For example: https://google.com:1000 will fail.
2. How did you find it?
   a. Manual test cases initially found this error. To better understand the bug, a partition input test was implemented to test all possible known valid port numbers from 0 to 65535. This was done to ensure that this bug wasn't just a corner case. The unit test function is testPortPartition()
3. What is the cause of that failure? Explain what part of the code is causing it?
   a. The cause of the issue is the PORT_REGEX in line 158
      i. Buggy: PORT_REGEX = "^:(\\d{1,3})$"
      ii. Fixed: PORT_REGEX = "^:(\\d{1,5})$"

**Steps to produce/reproduce:** Pass a valid URL with a port number above 999 to isValid().

**Expected results:** isValid should return true

**Actual results:** isValid returns false

**Workarounds:** None

**Attachments:** None

**Other Information:** None

**Bug 2: Any valid URL with a valid query returns as an invalid URL, and any valid URL with an invalid query returns as valid.**

**Class:** Serious Bug

**File Name:** src/UrlValidator.java

**Line(s) where failure manifested itself:** 446

**Severity:** 1 - Critical Issue

**Date:** 11-30-2015

**Reported by:** Group

**Email:** barlanj@oregonstate.edu, jenkinch@oregonstate.edu, hattont@oregonstate.edu

**Product:** URL Validator **Version:** 1128446

**Platform:** Windows **Version:** 10

**URL example input:** https://www.google.com/path1?auth=1

**Is it reproducible:** Yes

**Testing methodology used:** Unit Test

**Description:**

1. What is the failure?
   a. Any valid URLs with a query returns as an invalid URL via the isValid() method. For example: https://www.google.com/path1?auth=1 will fail validation. Similarly, any invalid URLs with a query return as valid.
2. How did you find it?
   a. This bug was found upon running the testIsValid() method, which was implemented with partitioned input domains and running through every possible combination of each partition.
3. What is the cause of that failure? Explain what part of the code is causing it?
   a. The cause of the failure is due to the return value of isValidQuery() method in UrlValidator.java file. The buggy version contains a not operator in the return which invalidates all valid query and validates all invalid query.

i. Buggy: return !QUERY_PATTERN.matcher(query).matches();
ii. Fixed: return QUERY_PATTERN.matcher(query).matches();

**Steps to produce/reproduce:** Pass a valid URL with a valid query to isValid() like https://www.google.com/path1?auth=1

**Expected results:** Should return as valid (true)

**Actual results:** Will return as invalid (false) in buggy version

**Workarounds:** None

**Attachments:** None

**Other Information:** None

## **Bug 3: A URL with a space in front of the path can pass the isValid method.**

**Class:** Serious Bug

**File Name:** src/UrlValidator.java

**Line(s) where failure manifested itself:** 411 (isValidPath)

**Severity:** 1 - Critical Issue

**Date:** 11-30-2015

**Reported by:** Group

**Email:** barlanj@oregonstate.edu, jenkinch@oregonstate.edu, hattont@oregonstate.edu

**Product:** URL Validator **Version:** 1128446

**Platform:** Windows **Version:** 10

**URL example input:** https://www.google.com   /test1234

**Is it reproducible:** Yes

**Testing methodology used:** Unit Test

**Description:**

1. What is the failure?
   a. Urls with spaces in the front of a path can pass the isValid method. For example:
      i. “https://www.google.com   /test1234”
2. How did you find it?
   a. This bug was found upon running the testIsValid() method, which was implemented with partitioned input domains and running through every possible combination of each partitions.
3. What is the cause of that failure? Explain what part of the code is causing it?

a. The cause of the failure is due to the validation logic of the isValidPath() function starting on line 411. This method does not check for leading spaces because it uses a Regex pattern matcher, which ignores the leading spaces.

   i. Fix: A check should be made before the PATH_PATTERN.matcher() for any leading spaces. If it exist, then return false immediately.

**Steps to produce/reproduce:** Pass a valid URL with a path and introduce spaces in front of the path, like so: “https://www.google.com   /test1234”

**Expected results:** Should return as invalid (false)

**Actual results:** Will return as invalid (true) in buggy version

**Workarounds:** None

**Attachments:** None

**Other Information:** None

## Testing Methodologies:

Test code can be found in the file: src/UrlValidatorTest.java:

- Our test functions were named: testManualTest(), testPortPartition(), and testIsValid().

### Manual Testing [testManualTest()]:

Our group did manual testing by creating a list of hardcoded valid and invalid URLs and looped through each list to run the isValid method on each item. In doing this manual testing, we found the port number bug which lead us to further investigate by creating an Input Partitioning test. Below is an example of our manual test cases:

```
// Valid URLs
String validUrls[] = { // add valid urls here
          "http://www.amazon.com",
          "http://www.amazon.com/",
          "http://www.amazon.com:0",
          "http://www.amazon.com:65535",
          "https://google.com:1000",
          "https://www.google.com/path1?auth=1"
};

System.out.println("Running Manual Test: Valid Urls | Count: " + validUrls.length);
for(int i = 0; i < validUrls.length; i++) {

   if(urlVal.isValid(validUrls[i]) == false) {
          System.out.println( "testManualTest: Actual: " +urlVal.isValid(validUrls[i])
              + " | Expected: true  || " + validUrls[i] );
          System.out.println("testManualTest: FOUND ERROR in above url.");
   }
   assertEquals(urlVal.isValid(validUrls[i]), true);
}
```

Invalid Urls were also tested in the same fashion (see testManualTest()).

## Input Partitioning [testPortPartition()]:

testPortPartition() is an Input Partitioning test implementation to thoroughly investigate the port number bug found in the manual testing. This method is written such that it will test every possible known valid port concatenated to a valid url. The port numbers run from [0 to 65535] inclusive - these numbers represent the 16-bit unsigned integer valid port numbers [src: https://en.wikipedia.org/wiki/Port_(computer_networking)].

A set of valid example test cases are:

- "http://google.com:20"
- "http://google.com:200"
- "http://google.com:2000"
- "http://google.com:6535"

And invalid examples are:

- "http://google.com:string"
- "http://google.com:-1234"
- "http://google.com:@1234"
- "http://google.com:~1234"

## Unit Tests [testIsValid()]:

We implemented testIsValid() to encompass all permutations of the various partitions making up a URL. These include the partitions {urlScheme, urlAuthority, urlPort, urlPath, urlQuery}. A nested for-loop was used to combine all ResultPair objects belonging to each partition to ensure that all possible permutations were tested:

```
for(i = 0; i < urlScheme.length; i++){
   for(j = 0; j < urlAuthority.length; j++) {
          for(k = 0; k < urlPort.length; k++) {
                 for(l = 0; l < urlPath.length; l++) {
                        for(m = 0; m < urlQuery.length; m++) {

                               // concatenate partitions
                               // do testing  of concatenated result

                        }
                 }
          }
   }
}
```

`(see testIsValid() in UrlValidatorTest.java for detailed implementation)`

In this way we are able to test every partition to find bugs that may occur. This Unit Test was able to find every bug found in the Manual and Input Partitioning test cases given specific inputs as well as similar inputs (by changing the value of the partition) which confirms the existence of those bugs. This method of testing also allows us to give varying inputs. We have documented three such bugs found: (1) the port number bug, (2) the query bug and (3) the space in path bug. All three bugs are discussed in detail above (see sections Bug 1, Bug 2 and Bug 3).

Some example generated inputs from the partitions that triggered the aforementioned bugs are:

- https://www.google.com:8080/path1?user=SomeUser123&pass=123456
- https://google.com:8080/123//789?user=SomeUser123&pass=123456
- https://www.google.com:8080
- https://255.255.255.255/path1?authentication=true
- https://255.255.255.255   /test1234
- ftp://google.com:8080/path1
- ftp://wwwwwwwwww.google.com:1
- ftp://wwwwwwwwww.google.com   /test1234

## Did you use any of Agan's principle in debugging URLValidator?

Our team made use of most of Agan's principles, with some notes on each below.

- Rule #1: Understand the System
  - Between part A of this project and the lecture materials we all were able to get a strong enough grasp on the use cases and source code of URLValidator to design effective test cases.
- Rule #2: Make It Fail
  - Our test cases tested many valid and invalid inputs and succeeded at causing the system to fail numerous times reliably and reproducibly.
- Rule #3: Quit Thinking and Look
  - Our test code shows that we actually "looked" for bugs rather than just hypothesizing they were there. A good example of this was when we hypothesized there were issues coming out of our manual testing with port numbers, so we actually wrote a test that partitioned the port number input space to check.
- Rule #4: Divide and Conquer
  - We narrowed down the sources of our problems to effectively identify what would cause bugs to occur. An example of this is when we identified the issue with leading spaces by

    observing our resultant test output and noticing a pattern where the leading space-filled invalid input was showing up very consistently. We then knew that the test was failing when calling isValidPath(), which we then investigated and realized its regular expression checking was incorrect.

- Rule #5: Change One Thing at a Time
    - We methodically made singular changes to our test suite before each execution as we went along to avoid confusion and help us target buggy lines of code.
- Rule #6: Keep an Audit Trail
    - We could have kept a more robust set of notes as we debugged, however, the scale of this project and the intuitiveness of the code and system made it generally not worth the added overhead.
- Rule #7: Check the Plug
    - Our team often questioned each other's assumptions as we worked to ensure our test suite was robust. For example, one person questioned the validity of the "leading spaces" error fix, only to be directed to the fact his test input's ResultPair object was malformed and that when fixed it indeed properly showed that the "spaces" bug fix worked.
- Rule #8: Get a Fresh View
    - We were without proper experts to ask for help during this project, though it was helpful to have the correct URLValidator code handy to review as a check and balance for some of our assumptions, similar to how asking someone for help would.
- Rule #9: If You Didn't Fix It, It Ain't Fixed
    - For every bug we identified we were able to find a fix and execute that fix to successfully eliminate the bug.

## Teamwork:

How did you work in the team? How did you divide your work? How did you collaborate?

- We broke up the project into roughly three main duties: the report, coding and bug identification, and cleanup/consistency checking/assumption questioning. Since the scope of this project was relatively narrow, we chose to have all three of us work on all aspects to a certain extent such that our overlapping efforts would drive out more bugs and a more finely tuned report. However, each of us did champion certain parts of the project more so than others.
- For example, the basis of the test code would be built by one person, then we'd share that base, individually build onto it to find different bugs, and then settle on the best combination of bugs and test suite to submit. For the report, another person would build its structure and the bug report format to be filled out as everyone found bugs. He would also handle most of the writing duties with

input from everyone else. We’d all check each other’s work and let the code and report “templates” help drive consistency, but everything would be put into the final “voice” and package by the third member.

- In terms of how we collaborated technically, we first had a team meeting using Google Hangouts to level set on the project requirements and each other’s schedules. We were then able to work together effectively via email, Google Docs, and GitHub.