

Tim Kelly (kelltimo)
Bryce Hart (hartbr)
Parker Howell (howellp)

CS362 Software Engineering II
11-30-2018

Final Project Part B

Methodology Testing

Clearly, write about your methodology of testing (manual, partition and programming basic).

The **manual** tests were created with Assert functions that accept two boolean values and then the string URL. We compared the boolean values of true or false, to the validated URL value of true or false. The first set of tests we passed URLs that we felt would return true values like "<http://www.google.com>" and "<https://www.facebook.com>". The next set we passed urls that we felt should return false, like "http://google.com///test1". We compared the bool of false to URL validator and we printed out the results for both sets of manual tests.

Additionally, we tested some URLs created out of ResultPair object in order to more completely test our code base. The ResultPairs were created by know true or false components of a complete URL and from these ResultPairs we created URL strings. We then tested these URL strings and their expected boolean value with the isValid() function and our custom myAssert() function.

The **partition** tests will take each part of the URL and segment them into partitions, and we tested whether they are true or false. For example, say we tested the scheme partition when the scheme is correct and when all other portions are incorrect, also tested when the scheme is correct and all other parts are correct, when the scheme is incorrect and all other parts are incorrect, and finally when the scheme is incorrect and all other portions are correct. The partitions we are testing are scheme, authority, port and path. This created 16 tests total.

The **programming** based tests were created by first creating different valid and invalid schemes, authorities, ports, paths, and queries. Then we created a function to concatenate the complete URL and test it. We looped through schemes, authorities, ports, paths and queries and compared versus expected boolean.

For example the first test looped through and tested that there were valid schemes, authorities, ports, paths and queries and it should return true. Next we tested URLs with invalid schemes, but valid authorities, ports, paths, and queries, and these URLs should return false. We kept going down the line so we tested URLs with valid schemes, invalid authorities, and valid ports, paths and queries. We did this for all the combinations and compared versus the expected boolean, which was expected to be false for all except the first because the first contained all valid pieces to the URL.

In the report mention the name of your test functions that implement each methodology.

testManualTest() - manual testing for valid and invalid URLs.

- myAssert()
- isValid()

partitionTests - test all partitions of the URL

- testYourFirstPartition()
- testYourSecondPartition()
- testYourThirdPartition()
- testYourFourthPartition()

testIsValid() - tests for all parts of URL

- testURLs()
- myAssert2()
- isValid()

For manual testing, provide some of your (not all) urls.

"http://www.google.com"

"https://www.google.com"

"ftp://255.255.255.255/test1/file"

"www.google.com"

For partitioning, mention your partitions with reasons and some of the urls that represent each partition.

Scheme - On the scheme we tested a result what would be a good scheme, like "http://" and then we also tested a bad scheme like "http:". We then tested a good scheme with all other good parts, a good scheme with all other bad parts, a bad scheme with all other good parts, and a bad scheme with all other bad parts.

Authority - On the authority we tested a result what would be a good authority, like "www.google.com" and then we also tested a bad authority like "aaa.". We then tested a good authority with all other good parts, a good authority with all other bad parts, a bad authority with all other good parts, and a bad authority with all other bad parts.

Port - On the port partition, we tested a result what would be a good port, like ":8888" and then we also tested a bad port like ":-1.". We then tested a good port with all other good parts,

a good port with all other bad parts, a bad port with all other good parts, and a bad port with all other bad parts.

Path - On the path partition, we tested a result of what would be a good path, which would be a blank path, and then we also tested a bad path like "../". We then tested a good path with all other good parts, a good path with all other bad parts, a bad path with all other good parts, and a bad path with all other bad parts.

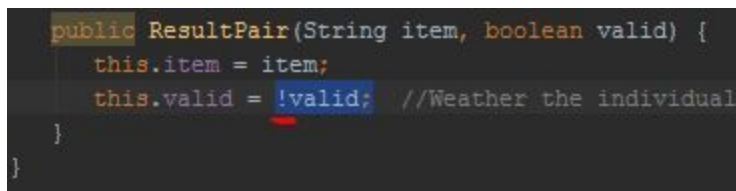
Bug Report

Write bug report for each of the bugs you found. You need to write about the following:

- 1. What is the failure?*
- 2. How did you find it and describe the test case that detected it?*
- 3. What is the cause of that failure? Explain what part of the code is causing it?*
- 4. You can attach a picture/snapshot of the faulty code.*

Bug 1 - In ResultPair class file:

- 1) The failure is that when using the constructor to create a ResultPair object the passed in boolean value, "valid" is assigned with the opposite value. E.G. - If a value of "true" is passed in, it is stored as "false".
- 2) This was discovered in the manual tests where two of the tests were created using ResultPair objects. The expected results from the isValid() function on these tests led me to manually/visually examine the ResultPair class file. That file being so small, I was quickly able to visually scan it and that is when I noticed the oddity.
- 3) In ResultPair.java, line 30. A logical not (!) is applied to the passed in "valid" argument.
- 4)



```
public ResultPair(String item, boolean valid) {  
    this.item = item;  
    this.valid = !valid; //Whether the individual  
}  
}
```

Bug 2 - In UrlValidator constructor:

- 1) It appears that values are improperly capitalized instead of being lowercase as expected.
- 2) With the bug (Bug 02) in ResultPair not fixing my manual tests, I then knew that we had to look deeper into the isValid() function from our UrlValidator.java file. I set a breakpoint right after entering the function and called it with the string, "<http://www.google.com>". Stepping through isValid, the value was not null so we passed that. The creation of urlMatcher on line 307 and its test passed, but I had no idea what urlMatcher should look like. The string "scheme" on line 312 returned with an unexpected answer.

```

urlMatcher = {Matcher@1008} "java.util.regex.Ma
scheme = "http"

```

I was expecting something along the lines of "http://" from our URL above. The test on scheme (isValidScheme) then failed, the incorrect results were therefore printed, and the debug session exited. So I'm thinking at this point that there could be a problem in either the creation of urlMatcher, or how it is used on line 312, or with isValidScheme.

So in figuring out how the urlMatcher is made, I need to figure out what a URL_PATTERN is. I then found its definition on line 112.

```

private static final String URL_REGEX =
    "^( ([^:/?#]+) : )? ( // ( [^/?#]* ) )? ( [^?#]* ) ( \\? ( [^#]* ) )? ( # ( . * ) )? ";
//      12          3 4          5          6 7          8 9
private static final Pattern URL_PATTERN = Pattern.compile(URL_REGEX);

```

I also see that it is created with the URL_REGEX defined above it. The URL_REGEX is preceded by a comment stating that this regular expression is defined in rfc 2396, so to the interwebs I go where I find the related rfc (<https://tools.ietf.org/html/rfc2396>) and the actual regex expression:

```

The following line is the regular expression for breaking-down a URI
reference into its components.

^( ([^:/?#]+) : )? ( // ( [^/?#]* ) )? ( [^?#]* ) ( \\? ( [^#]* ) )? ( # ( . * ) )?
12          3 4          5          6 7          8 9

```

It's identical, except one additional backslash. But this backslash is used as an escape character, so it looks fine.

So looking at line 312 where scheme is assigned a value from urlMatcher.group(PARSE_URL_SCHEME), even though scheme isn't what I expected, it's not too far off and the RegEx the function uses looks good based on our investigation above.

```

urlMatcher = {Matcher@1008} "java.util.regex.Matcher[pattern=^( ([^:/?#]+) : )? ( // ( [^/?#]* ) )? ( [^?#]* ) ( \\? ( [^#]* ) )? ( # ( . * ) )? "
parentPattern = {Pattern@1011} "^( ([^:/?#]+) : )? ( // ( [^/?#]* ) )? ( [^?#]* ) ( \\? ( [^#]* ) )? ( # ( . * ) )? "

```

Next I looked at isValidScheme. Following the debugger through this function, we get to the final "if" statement on line 367, which fails. We can see for the values of allowedSchemes:

```
▶ this = {UrlValidator@1005}
▶ scheme = "http"
▼ allowedSchemes = {HashSet@1009} size = 3
  ▶ 0 = "HTTPS"
  ▶ 1 = "HTTP"
  ▶ 2 = "FTP"
```

Now we are seeing a part of the problem, namely that our scheme is lower case and the allowed schemes are upper case. Which begs the question, “where is allowedSchemes set?” Searching for this we find that allowedSchemes is set in the constructor for the default UrlValidator object. When you call “new UrlValidator()” with empty arguments, the constructor then begins to call overloaded constructors until we get to the constructor on line 271. In here we see where allowedSchemes is set:

```
schemes = DEFAULT_SCHEMES;
}
allowedSchemes = new HashSet<String>(schemes.length);
for(int i=0; i < schemes.length; i++) {
    allowedSchemes.add(schemes[i].toUpperCase(Locale.ENGLISH));
}
```

Note that schemes assigned the value of DEFAULT_SCHEMES. Also that on line 282 as the schemes are added to allowedSchemes, they are capitalized.

Looking at DEFAULT_SCHEMES we see:

```
private static final String[] DEFAULT_SCHEMES = {"http", "https", "ftp"}; // Must be lower-case
```

Here we see lowercase schemes and additionally the comment, “Must be lower-case”

So at this point I think we have found a bug. Namely that when we create our UrlValidator object, it is improperly setting it's member variable, allowedSchemes, to values that are capitalized.

3) The fault in the code is caused by incorrectly capitalizing the values in allowedSchemes, in the constructor for UrlValidator, when they should remain as lowercase values. The toUpperCase() should be a toLowerCase().

4)

```
allowedSchemes = new HashSet<String>(schemes.length);
for(int i=0; i < schemes.length; i++) {
    allowedSchemes.add(schemes[i].toLowerCase(Locale.ENGLISH));
}
```

Bug 3 - In RegexValidator constructor

1) The bug is that in the RegexValidator constructor the variable “patterns[i]” is not properly set because of an off by 1 bug in the preceding “for” loop.

2) Having isolated the last bug our manual test with “<http://www.google.com>” now correctly validates. Adding “<http://www.google.com/>” to our manual test also validates. However we run into a problem when adding in “<https://www.google.com>”. When running with this URL we get an exception and we see that it is caused by RegexValidator() on line 121 in our RegexValidator.java file.

```
119 @ public RegexValidator(String[] regexs, boolean caseSensitive) {
120     if (regexs != null || regexs.length == 0) {
121         throw new IllegalArgumentException("Regular expressions are missing");
```

Setting a breakpoint here we look at what is being evaluated:

```
▶ this = {RegexValidator@1008} Method threw 'java.lang.NullPoin
▼ regexs = {String[1]@1011}
  ▼ 0 = "^(?:\p{Alnum}(>[\p{Alnum}-]{0,61}\p{Alnum})?\.)+(\p{Alpha}(>[\p{Alnum}-]{0,61}\p{Alnum})?|\p{Alnum})?\.?$"
    ▼ value = {byte[96]@1014}
      01 0 = 94
      01 1 = 40
      01 2 = 63
      01 3 = 58
      01 4 = 92
      01 5 = 112
```

We see that the triggering “if” statement evaluates two conditions. The first evaluates to true because regexs has 1 item in it, the string “^(?:\p{Alnum}(>[\p{Alnum}-]{0,61}\p{Alnum})?\.)+(\p{Alpha}(>[\p{Alnum}-]{0,61}\p{Alnum})?|\p{Alnum})?\.?\$”

So that is a problem. The other condition test that the size of regexs is not 0. Which it clearly isn't so that is ok. It seems that changing the first conditional check to “regexs == null” would make more sense because then we would be checking to see that regexs does in fact exist.

Making that change we do get past that part but run into another problem:

```
Exception in thread "main" java.lang.NullPointerException
    at RegexValidator.match(RegexValidator.java:165)
    at DomainValidator.isValid(DomainValidator.java:164)
    at UrlValidator.isValidAuthority(UrlValidator.java:413)
```

So again we set a breakpoint at line 164 in DomainValidator.java and re-run our test and see:

```

> this = {DomainValidator@1008}
> domain = "www.google.com"
> hostnameRegex = {RegexValidator@1011} Method threw 'java.lang.NullPointerException' exception. Ca
> domainRegex = {RegexValidator@1012} Method threw 'java.lang.NullPointerException' exception. Cann
> allowLocal = false

```

domainRegex evaluates to null, and it probably shouldn't be. So we need to check out how it is created. Searching for this we see:

```

public RegexValidator(String[] regexs, boolean caseSensitive) { regexs: {"^(?:\p{2}
    if (regexs == null || regexs.length == 0) {
        throw new IllegalArgumentException("Regular expressions are missing");
    }
    patterns = new Pattern[regexs.length];
    int flags = (caseSensitive ? 0: Pattern.CASE_INSENSITIVE); flags: 0 caseSens
    for (int i = 0; i < regexs.length-1; i++) {
        if (regexs[i] == null || regexs[i].length() == 0) {
            throw new IllegalArgumentException("Regular expression[" + i + "] is mi
        }
        patterns[i] = Pattern.compile(regexs[i], flags); patterns: Pattern[1]@101
    }
}

```

domainRegex is created by creating a new RegexValidator. Following the constructor for the RegexValidator we see that it is constructed in a similar manner to how UrlValidator object was created in the prior bug by a series of overloaded constructor calls. We eventually reach the constructor call of interest at line 119 so I set a break point there. Rerunning the debug test and stepping through the constructor I notice that on line 125 the “for” loop is evaluated but doesn't trigger, bypassing lines 126-129. This seems odd. Looking the “for” loop conditional I see that because regex.length has a size of 1 and because we subtract 1 from that the for loop never evaluates to true because our i value starts at 0.

removing the “-1” from the “for” loop seems to fix the problem as when we re-run our test we now see that that line works and that domainRegex and hostnameRegex are now populated with data and not null.

3) The cause of the bug is found on line 125 in the file RegexValidator.java. The error is an extra “-1” on the conditional check of the “for” statement.

4)

```
public RegexValidator(String[] regexs, boolean caseSensitive) { regexs: {"^"?:\p{2}
    if (regexs == null || regexs.length == 0) {
        throw new IllegalArgumentException("Regular expressions are missing");
    }
    patterns = new Pattern[regexs.length];
    int flags = (caseSensitive ? 0: Pattern.CASE_INSENSITIVE); flags: 0 caseSens
    for (int i = 0; i < regexs.length-1; i++) {
        if (regexs[i] == null || regexs[i].length() == 0) {
            throw new IllegalArgumentException("Regular expression[" + i + "] is mi
        }
        patterns[i] = Pattern.compile(regexs[i], flags); patterns: Pattern[1]@101
    }
}
```

Bug 4 - in isValidAuthority inetAddressValidator is null...

- 1) The bug is that inetAddressValidator is null and doesn't return anything.
- 2) The InetAddressValidator provides methods to validate an IP address, after completing manual tests and drilling down into the InetAddressValidator.java, there is a function for the InetAddressValidator that states it will return the singleton instance of this validator. However, on examining what is actually being returned it's clear that it's only returning null but it should be returning something else.
- 3) The cause of the failure relates to line 68 in InetAddressValidator, the InetAddressValidator function should in fact return VALIDATOR.

4)

```
public static InetAddressValidator getInstance() {
    return null;
}
```

Debugging

When you find out any failure, debug using Eclipse/IntelliJ debugger or any other tool and try to localize its cause. Provide at what line/lines in what file the failure manifested itself.

Bug 1 - ResultPair.java line 30

Bug 2 - UrlValidator.java line 282

Bug 3 - RegexValidator.java line 125

Bug 4 - InetAddressValidator.java line 68

Did you use any of Agan's principle in debugging URLValidator? (Provide your debugging details for each bug.)

Yes, Agan's Principles were used:

Understand the system: This wasn't an easy one to comply with at the beginning, because the source code for the project was fairly large and complex, but as time went on, and as we dug deeper into our test cases, our understanding of the system as a whole increased noticeably. After setting up the workspace and starting testing for bugs we got a much better understanding of the system.

Make it fail: This was trivially easy considering the amount of failed tests initially caused by our manual, partition, and programmatic test cases. Almost all of the tests we did made it fail, so we had no problem with this principle.

Quit thinking and look: This was particularly necessary as a lot of the code base was unfamiliar to us. We weren't even able to guess what the bug was in most cases. We really did need to use the debugger to step through our debugging efforts for basically every bug to see what was actually happening.

Divide and conquer: This came into play while using the debugger and its ability to set breakpoints and step through code. With these abilities we could jump to right before a known problem would arise, and manually examine the state of the program and how it changed in expected/unexpected ways. Additionally, it was helpful to comment out parts of the code not related to the current bug being explored. This helped with the speed of debugging. We just had to remember to remove the `/**` to ensure we didn't break prior tests, after fixing the current bug.

Change one thing at a time: This was applied to every bug fix. Once we figured out what we thought a bug was, one of us would make one change to fix the code, and then run our tests again. If the change worked for our test input, then we would test to check that the fix worked with other/more/prior input values. This helped to ensure that we didn't break other things with our current fix.

Keep an audit trail: As I was looking into different bugs, I kept a text editor open to document my path and provide a history of what was already examined or what was changed. This was particularly helpful if I had to step away from the computer for a while. Using this history I could quickly get up to speed with where I previously left off.

Check the plug: In Bug 3 I as I was exploring the bug, I initially thought that there was a problem in either the creation of `urlMatcher`, or how it is used on line 312, or with `isValidScheme`. However, by not cementing myself into these initial assumptions, I was able to later find the source of the bug in the `UrlValidator` constructor. It was helpful to not get pigeonholed into a certain area as our suspicions grew on certain bugs.

Get a fresh view: I found myself talking out loud to my cat when working on bug 2. It really helped me to get my head wrapped around what was happening. I did this again on the other bugs I worked on. This feels very similar to the well known and loved “Rubber Ducky” method of debugging. If you can explain the problem to a rubber duck, then it’s easier to wrap your head around it.

If you didn't fix it, it ain't fixed: For all tests, if we thought that we found a bug, we would test that fixing the bug resulted in the expected test output. Additionally, we would check that the current bug fix worked with our prior fixes.

Team Work

Write about how did you work in the team? How did you divide your work? How did you Collaborate?

Mostly individuals volunteered to assist with the project where help was needed. If individuals were having issues understanding, or completing, some aspect of the assignment, others stepped in to offer ideas or help.

The team used several tools to collaborate including, Google Hangouts, Google Docs, and codeshare (<https://codeshare.io>).

Team Contributions:

Tim Kelly - Code and ran part 1 (manual testing.) Wrote some content for methodology testing, and edited the final document. Wrote Bug 4.

Bryce Hart - Coded and ran part 2 (input partitioning). Added some content to the Agan’s principles as well as editing final document. Fixed bug 1.

Parker Howell - Coded and ran part 3 (programming), added some content to the Agan’s Principles, documented bug fixes 2 and 3, edited final document.