Brian Laccone (ONID: lacconeb)
Jeff Luong (ONID: luongje)
Jesse Chan (ONID: chanjess)
CS362 Final Project: Part-B

1) Methodology Testing
    a) Manual Testing
        i) The first attempts at manual testing was to choose various port numbers as shown in the following:

```
class Main{

    public static void main(String[] args){

        UrlValidator urlVal = new UrlValidator(null, null, UrlValidator.ALLOW_ALL_SCHEMES);


        System.out.println(urlVal.isValid("http://www.amazon.com"));
        System.out.println(urlVal.isValid("http://www.amazon.com:10000"));
        System.out.println(urlVal.isValid("http://www.amazon.com:20000"));
        System.out.println(urlVal.isValid("http://www.amazon.com:30000"));
        System.out.println(urlVal.isValid("http://www.amazon.com:40000"));
        System.out.println(urlVal.isValid("http://www.amazon.com:50000"));
        System.out.println(urlVal.isValid("http://www.amazon.com:60000"));
```

The first statement passed when it was an URL without a port number, but the rest of the urls with port numbers all failed. Although those port numbers all seem valid, it was definitely odd that all failed validation.

The natural next thing to test were additional port numbers as shown in the following:

```
System.out.println("Hello1");


System.out.println(urlVal.isValid("http://www.amazon.com:0"));
System.out.println(urlVal.isValid("http://www.amazon.com:10"));
System.out.println(urlVal.isValid("http://www.amazon.com:200"));
System.out.println(urlVal.isValid("http://www.amazon.com:3000"));
System.out.println(urlVal.isValid("http://www.amazon.com:65535"));
System.out.println(urlVal.isValid("http://www.amazon.com:65536"));
System.out.println(urlVal.isValid("http://www.amazon.com:70000"));


System.out.println("Hello2");

System.out.println(urlVal.isValid("http://www.amazon.com:0"));
System.out.println(urlVal.isValid("http://www.amazon.com:52"));
System.out.println(urlVal.isValid("http://www.amazon.com:99"));
System.out.println(urlVal.isValid("http://www.amazon.com:100"));
System.out.println(urlVal.isValid("http://www.amazon.com:999"));
System.out.println(urlVal.isValid("http://www.amazon.com:555"));
System.out.println(urlVal.isValid("http://www.amazon.com:723"));

System.out.println("Hello3");


System.out.println(urlVal.isValid("http://www.amazon.com:1"));
System.out.println(urlVal.isValid("http://www.amazon.com:12"));
System.out.println(urlVal.isValid("http://www.amazon.com:123"));
System.out.println(urlVal.isValid("http://www.amazon.com:1234"));
System.out.println(urlVal.isValid("http://www.amazon.com:12356"));
```

The results were interesting. All of the above failed testing and it got me thinking that the bug is probably causing all URLs with port numbers to fail. This was a good place to start in localizing the bug.

b) Partitioning

Partitioning for this program was split up into 5 different sections to test all the URL parts. The five sections were divided into scheme, authority, port, path/alternative path, and query. Partition testing was very useful for this program because the it was easy to isolate each part of a URL and if one part was invalid then the whole URL will be invalid. Each partition contained tests that checked valid and invalid inputs to determine the accuracy of the program.

1. Scheme:
   Section: http://
   Example URL: http://www.google.com

```
Scheme Test

Testing - www.google.com
Failed
Testing - ftp://www.google.com
Failed
Testing - http://www.google.com
Passed
Testing - htt://www.google.com
Failed
```

2. Authority:
   Section: www.google.com
   Example URL: http://www.google.com

```
Authority Test

Testing - http://google.com
Passed
Testing - http://www.0.0.0.0.com
Passed
Testing - http://256.256.256.256.com
Passed
Testing - http://255.255.255.255.com
Passed
Testing - http://900.6.789.255.com
Passed
```

3. Port:
   Section: :5000
   Example URL: http://www.google.com:5000

```
Port Test

Testing - http://www.google.com:80
Failed
Testing - http://www.google.com:3500
Failed
Testing - http://www.google.com:65535
Failed
Testing - http://www.google.com:65636
Failed
Testing - http://www.google.com:-1
Failed
```

4. Path:
   Section: /test
   Example URL: http://www.google.com/test

```
Path Test

Testing - http://www.google.com/test
Passed
Testing - http://www.google.com/test/file
Failed
Testing - http://www.google.com/test/
Failed
Testing - http://www.google.com/test//file
Failed
Testing - http://www.google.com/..
Failed
```

5. Query:
   Section: ?action=view
   Example URL: http://www.google.com?action=view

```
Query Test

Testing - http://www.google.com?action=view
Passed
Testing - http://www.google.com?action=edit&mode=up
Passed
Testing - http://www.google.comaction=view
Passed
Testing - http://www.google.com???????????
Passed
```

c) Programming Based

This portion of testing involved more breadth in testing as multiple permutations in a “testURL” were used to determine if a URL was valid. Two unit tests initially written, one to test the ALLOW_ALL_SCHEMES (testIsValidAllSchemes()) and ALLOW_2_SLASHES (testIsValid2Slashes()). I used the following list of potential inputs for the testURL:

```
String schemes[] = new String [NUM_SCHEMES];
schemes[0] = "http://";      // Valid
schemes[1] = "ftp://";       // Valid
schemes[2] = "http:/";       // Invalid
schemes[3] = "localhost:";   // Invalid
schemes[4] = "fsociety";     // Invalid

String authority[] = new String [NUM_AUTHS];
authority[0] = "www.amazon.com"; // Valid
authority[1] = "amazon.com";     // Valid
authority[2] = "255.255.255.255";// Valid
authority[3] = "abcdef";         // Invalid
authority[4] = "12345";          // Invalid
authority[5] = "abc.d1f";        // Invalid
authority[6] = ".abc";           // Invalid
authority[7] = "123.";           // Invalid
authority[8] = "abc.d1f";        // Invalid
authority[9] = "";               // Invalid

String port[] = new String [NUM_PORTS];
port[0] = ":0";      // Valid
port[1] = ":8080";   // Valid
port[2] = "";        // Valid
port[3] = "port";    // Invalid
port[4] = "#";       // Invalid
port[5] = ":-8080";  // Invalid

String path[] = new String [NUM_PATHS];
path[0] = "/taco";           // Valid
path[1] = "";                // Valid
path[2] = "plus+path";       // Valid
path[3] = "underscore_path"; // Valid
path[4] = "at@path";         // Valid
path[5] = "dot.path";        // Valid
path[6] = "percent%path";    // Valid
path[7] = "forslash/path";   // Invalid
path[8] = "tilda~path";      // Invalid
path[9] = "colon:path";      // Invalid
path[10] = "//";             // Invalid
path[11] = "/double//path";  // Invalid

String query[] = new String [NUM_QUERIES];
query[0] = "";                              // Valid
query[1] = "?phrasing=Lana";                // Valid
query[2] = "?12345";                        // Valid
query[3] = "?rick=sanchez&morty=gazorp"; // Valid
query[4] = "/";                             // Invalid
query[5] = "phrasing";                      // Invalid
```

I added the parameters that were expected to pass first into the array and left the rest as invalid. With this many variables to use I incorporated nested for loops to test 21600

possibilities.

```
UrlValidator urlVal = new UrlValidator(schemes, null, UrlValidator.ALLOW_ALL_SCHEMES);

System.out.println("GOOGLE CHECK " + urlVal.isValid("http://www.google.com"));

for(int a = 0; a < NUM_SCHEMES; a++) {
    for(int b = 0; b < NUM_AUTHS; b++) {
        for(int c = 0; c < NUM_PORTS; c++) {
            for(int d = 0; d < NUM_PATHS; d++) {
                for(int e = 0; e < NUM_QUERIES; e++) {
                    testURL = schemes[a] + authority[b] + port[c] + path[d] + query[e];
                    if(urlVal.isValid(testURL)){
                        System.out.println(testURL + " is valid!");
                        numValid++;
                    } else {
                        System.out.println(testURL + " is NOT valid!");
                        numInvalid++;
                    }

                }
            }
        }
    }
} // for
System.out.println("Total VALID: " + numValid);
System.out.println("Total NOT VALID: " + numInvalid);
```

The “GOOGLE CHECK” was not initially there when I began the tests but was added since I saw that my ALL my tests in testIsValid2Slashes() failed despite using what had passed in the previous partitioning section to confirm that there was a bug with ALLOW_2_SLASHES. With the first run I am expecting 21600 total VALID/NOT VALID results to come back while the second should contain 54180. I added more test cases to the second test to try to determine where the path bug was manifesting. These allowed me to start looking into where things went wrong.

2) Bug Report

Bug 1 - A bug occurs when the URL contains a port number. This bug was found through manual testing when testing various different port numbers, such as 0, 10, 200, and many others. It seems that all port numbers lead to failure regardless of the port number length, for example a single digit like 0 versus a three digit number like 200. Partition testing also revealed this as each port test failed. Programmatic testing also discovered this as every test after the blank port test failed.

It appears that the cause of the bug is in the UrlValidator.java file starting on line 316, which is the following:

```
if ("http".equals(scheme)) {// Special case - file: allows an empty authority
    if (authority != null) {
        if (authority.contains(":")) { // but cannot allow trailing :
            return false;
        }
    }
```

Reading this, it does not allow the use of ":" in your URL. Reading the comment about the Special case, it makes sense that this was meant to be used with the "file" scheme instead since file's do not have ports, thus the inclusion of the ":" would not be valid.

Bug 2 - Calling isValid("http://www.google.com") returns false when it should return true because it is a valid url under any option that is NOT ALLOW_ALL_SCHEMES. Every test run under any other option than ALLOW_ALL_SCHEMES fails regardless of whether or not it should have. The programmatic unit tests that did not include ALLOW_ALL_SCHEMES discovered this. Because of this we suspected that the root of the problem had to do with this validation option not being properly set. Starting at the beginning where we create the call UrlValidator() we can see that ALLOW_ALL_SCHEMES is properly working the way it should which is reinforced by the first test we created. Looking further down where it begins to set the other schemes we see that the *allowedSchemes* is the culprit as it is contains the schemes selected AFTER they have been converted to all *uppercase*.

```
public UrlValidator(String[] schemes, RegexValidator authorityValidator, long options) {
    this.options = options;

    if (isOn(ALLOW_ALL_SCHEMES)) {
        allowedSchemes = Collections.emptySet();
    } else {
        if (schemes == null) {
            schemes = DEFAULT_SCHEMES;
        }
        allowedSchemes = new HashSet<String>(schemes.length);
        for(int i=0; i < schemes.length; i++) {
            allowedSchemes.add(schemes[i].toUpperCase(Locale.ENGLISH));
        }
    }

    this.authorityValidator = authorityValidator;
}
```

This causes problems down stream in the function isValid() since isValidScheme() checks that the schemes are matched against their *lowercase* counterparts. Because of this it never finds the correct schemes and automatically returns false in line 366.

```
protected boolean isValidScheme(String scheme) {
    if (scheme == null) {
        return false;
    }

    // TODO could be removed if external schemes were checked in the ctor before being stored
    if (!SCHEME_PATTERN.matcher(scheme).matches()) {
        return false;
    }

    if (isOff(ALLOW_ALL_SCHEMES) && !allowedSchemes.contains(scheme.toLowerCase(Locale.ENGLISH))
        return false;
    }

    return true;
}
```

Bug 3 - A bug occurred when testing the path of urls. Valid paths under the ALLOW_2_SLASHES scheme were all returning false. The bug was found through partition testing when testing various paths. The first test contained the path “/test”, which correctly passed. However, the following tests, which contained more than one “/”, failed. “/test/file”, “/test/”, and “/test//file” all failed the test. This suggests that the system is not allowing more than one “/” per url. The cause of the bug seems to be occurring on line 451 in UrlValidator.java when isValidPath() is called. The function returns false when it tries to match the the url’s path with the system’s accepted character pattern.

```
if (!PATH_PATTERN.matcher(path).matches()) {
    return false;
}
```

3) Debugging
    a) Agan’s Principles Used:
        i) Understand the System
            (1) Bug 1 - It definitely took a good number of hours to understand the code and especially the isValid and RegexValidator methods that contained bugs. It was moot point to start debugging from the get go without understand the source code.
            (2) Bug 2 - This required an understanding of how the scheme was parsed and compared in the program. Without knowing that the schemes are stored in *allowedSchemes* we would not be able to make the comparison to where the fault is occurring.
        ii) Make It Fail
            (1) Bug 1 - This concept was practiced when initially conducting the manual testing. Many URLS with different port numbers were tested to reproduce the fail result.

(2) Bug 2 - This one was fairly easy since the failure occurred quickly and allowed no tests to pass. By having it fail earlier in the process it granted us a quicker path to determining where in the flow the error was occuring (in this case near the beginning of the isValid() function).

iii) Quit Thinking and Look

(1) Bug 1 - Based on the manual testing results, the evidence was all all URLS with port numbers failed. The URLS without port numbers passed. This meant that we had to be looking for code and bugs pertaining to the port number. This was a good starting point.

(2) Bug 2 - The immediate assumption was that there was a problem with the ALLOW_2_SLASHES option. It wasn't until the same problem was observed with the other option NO_FRAGMENTS was set and had the same result that we realized this had to do with the options setting as a whole.

iv) Change One Thing at a Time

(1) Bug 1 - Code was changed in a sequential manner. First, fixes were made on UrlValidator.java and then on RegexValidator.java.

v) Get a Fresh View

(1) Bug 1 - Consultation with team members provided more insight in understanding the bugs and in debugging. The concept of how files work was explained and the source of the bug was validated.

(2) This bug was difficult to find during the Manual and Partition testing and it wasn't until we realized that all tests would fail regardless of minute changes that we were able to take a step back and go through the code from the beginning.

vi) If You Didn't Fix It, It Ain't Fixed

(1) Bug 1 - URLS with port number now pass the test after debugging and implementing fixes.

(2) Bug 2 - Fixing this *allowedSchemes* issue to search for toLowerCase versions of the scheme still did not resolve the issue of allowing valid options. This could be the result of another bug further downstream.

b) Bug 1 - A bug occurs when the URL contains a port number.

i) The bug roots in the UrlValidator.java file on line 316.
Using a debugger, it's clear that when the program is run, the program goes through the first "if" statement where it is checking if(authority != null). This consequently leads to the test failing. On line 316, "http" should be "file" because otherwise it does not allow the use of ":" in the URL. This was tested with the option ALLOW_ALL_SCHEMES.

```java
if ("http".equals(scheme)) {// Special case - file: allows an empty authority
    if (authority != null) {
        if (authority.contains(":")) { // but cannot allow trailing :
            System.out.println("hello1");
            return false;
        }
    }
    // drop through to continue validation
} else { // not file:
    // Validate the authority
    if (!isValidAuthority(authority)) {
        System.out.println("hello2");
        return false;
    }
}
```

After changing this, further error messages appeared:

```
Caused by: java.lang.IllegalArgumentException: Regular expressions are missing
        at RegexValidator.<init>(RegexValidator.java:121)
        at RegexValidator.<init>(RegexValidator.java:96)
        at RegexValidator.<init>(RegexValidator.java:83)
        at DomainValidator.<init>(DomainValidator.java:107)
        at DomainValidator.<clinit>(DomainValidator.java:96)
        ... 3 more
```

Based on this, the RegexValidator.java file was perused through, specifically line 121 (shown below).

```java
public RegexValidator(String[] regexs, boolean caseSensitive) {
    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);
    for (int i = 0; i < regexs.length-1; i++) {
        if (regexs[i] == null || regexs[i].length() == 0) {
            throw new IllegalArgumentException("Regular expression[" + i + "] is missing");
        }
        patterns[i] =  Pattern.compile(regexs[i], flags);
    }
}
```

Based on the error message, this led us to think there was another bug in the first “if” statement since the error message printed out “Regular expressions are missing”.

c) Bug 2

This failure manifested itself in line 282 of **UrlValidator.java.** When a breakpoint was set on line 271 to watch when this was called I saw that if flowed into the else branch at line 276 and skipped the if statement. During the for statement I then inspected the *allowedSchemes* variable and saw that each of the schemes

added was stored in all uppercase. Continuing the program the tests that ran the isValid() function is called on line 299 and when checking the scheme at line 311 happens, we see that the error occurs in line 365 where it attempts to search the *allowedSchemes* for the lowercase version of the schemes entered. This set off the “return false;” line on 366 and caused the isValid() function to prematurely return with false preventing any additional checking to continue.

4) Teamwork

   a) We used Slack for main means of communication and Google Docs in compiling the report. Within Google Docs, we also placed comments for items that required further discussion. This assignment had three different testing methodologies making it convenient to separate the work. The work was divided in the following manner: Jesse: Manual Testing, Brian: Partitioning Testing, and Jeff: Programmatic Testing.
   b) Although each member worked on their designated methodology testing, everyone helped each other out in debugging.