CS 362

Final Project

Team Members:

Jack Holkeboer
Kyle Collins
Shaine Afzal

Division of Work

When we read the assignment instructions we noticed a natural division between three types of testing - manual, partition, and programmatic. We decided that since we had 3 team members, each one of us would concentrate on one of these areas. We started out working separately on each area, running our tests and finding bugs. Then we came together and compared our results. In most cases, bugs that were found in one testing methodology were also found in the others. In other cases, we were able to point out elements of each other's tests that could be improved. Once we agreed upon what the bugs were, we then simultaneously dug into the UrlValidator source code to isolate the causes of them.

Agan's Principles

Of the nine principles mentioned by Agan, the following were the most relevant in our testing.

#1. Understand the System:

Just as we did with Dominion, it is important to understand the rules you're dealing with. Otherwise it is impossible to debug effectively. For Dominion, this meant playing the game and understanding what the rules are (i.e. what happens when you play each card). For UrlValidator, we first had to understand what makes a URL valid in the first place. This involved first reading online about URL validity and then thoroughly examining the UrlValidator class to see how the program works.

#2. Make It Fail:

Once we had developed an understanding of UrlValidator and what a valid URL should be we began creating tests with the intention of causing a failure in UrlValidator. We also recorded any URL components that return unintended results so that we could mark these for further testing.

#4. Divide and Conquer:

Using deduction to determine why UrlValidator proved to be extremely beneficial, especially for input partitioning. Examination of UrlValidator indicated that it tests individual components sequentially, and that if the test for a particular component fails, UrlValidator will return false and not test any remaining components. This would affect construction of the partitioning test which is described in methodology.

#5. Change One Thing at a Time:

Manual testing was conducted to isolate any issues occurring with UrlValidator. By slightly altering previous URL strings, it was easy to determine when an error occurred and what caused the error.

METHODOLOGY

Manual Tests:

For the manual tests I created many targeted values to test specific parts of the URL. I started with testing multiple different correct URL's to make sure they were correct. Next I tried different errors in the http:// part of the URL, including not including it at all. I tried including different errors in the main part of the URL. Then I tried introducing different extensions to the URL. One thing that I did while doing my manual testing was to run the tests I had often, then used the information I got from running those tests to target specific things that may have been causing

errors. I found that this was a very easy way to identify a specific thing that was causing the issue. For example a bug that I found during manual testing was that if a question mark was added to the extension of the URL it would fail every time. First I did a URL extension with multiple special characters and it failed, then I was able to do separate tests with each of the special characters and it ended up that the only time it failed was if it was the question mark that was used, and then I tested multiple different ways of including the question mark and they all failed. Some examples of URL's that I tested are:
http://www.amazon.com/
http://www.msn.com/
htp://www.amazon.com/
http//www.amazon.com/
http:/www.amazon.com/
http://www.msn.com/valid123/
www.amazon.com
Yo
91872398u8r9jiej
http://stuff.amazon.com/231?get=h/
http://www.msn.com/valid=valid
http://www.msn.com//valid?valid

Partition Tests:

The Partition Tests consists of several classes and a method in UrlValidator. The first class, UrlComponents consists of five ResultPair arrays. Two of the ResultPair arrays (Address and Port) are combined to make a list (Authorities). This was done so that new components could easily be written by adding them in the UrlComponents class and that URL components would be available in other classes/methods if necessary. The GenerateURL consist of five methods. The first method, randArr generates an array of 4 random integers that are based on the size of the components mentioned in UrlComponents. The second method, urlComponentArray selects the strings to be used in creating the URL and stores them in an array for later use. The third method, createUrl, simply creates the URL based on the strings from urlComponentArray. The final two methods, isValidArray and isValid generate a boolean array based on the strings in urlComponentArray and a final boolean value based on the AND of all the values in the boolean array. The actual partition testing occurs in the partitonTest for the UrlValidatorTest class. It is conceptually a partitioned random test, forming random URLs based on its random components created in the GenerateURL class. However, because of the addition of the boolean arrays and urlComponentArrays it is able to partition results based on components and report back on what unintended results occurred. For instance, if the URL ftp://255.255.255.255:65635?query=this is created and tested, the following output is shown.

MISMATCH DETECTED! Test URL is ftp://255.255.255.255:65635?query=this
Expected Result: true Actual Result: false
Authority error detected for 255.255.255.255:65635 Expected: true Actual: false
Querry error detected for ?query=this Expected true Actual false

As can be seen, this gives detailed information about what exactly failed and what the result should be. The final feature of the partition test is that it reports on all components that failed after the set number of random tests have been completed.

Programming Based Tests:

We used random testing in order to generate the widest possible variety of test cases. We used the ResultPair object that is supplied with the URLValidator code, which contains a string of text (the property "item") and a boolean value (the property "valid"). Each pair is meant to be a certain part of the URL.

The idea behind constructing tests this way is that each URL segment is either valid or invalid. If a URL string contains one invalid segment, the entire URL should be invalidated. Therefore, we take the "and" value of the validity of each resultPair like this:

```
// How we derive URL validity
Boolean testURLValidity = schemeValidity && addressValidity && portValidity
                                                && pathValidity && queryStringValidity;
```

Each validity boolean corresponds to one part of the URL:

- Scheme (http, ftp, etc)
- Address (e.g. yahoo.com, www.google.com, bbc.co.uk, etc)
- Port (e.g. no port, :80, :63353, etc)
- Path (e.g. /path, /path1/path2/file.txt, etc)
- Query String (e.g. ?value=1)

To generate a random test case, first we supply a list of possible strings for each parameter. You can see these in our UrlValidatorTest.java file in the testIsValid() function. We then choose a random element from each list and concatenate them. Parameters that are not mandatory will have an empty string ("") as one of the possible options, so not every test URL will contain every part. The code that generates our test cases looks like this:

```
// generate test cases
        for(int i = 0;i<numberOfTests;i++)
        {
                // get random scheme
                int randomSchemeIndex =
generator.nextInt(testSchemes.length);
                String scheme = testSchemes[randomSchemeIndex].item;
                Boolean schemeValidity =
testSchemes[randomSchemeIndex].valid;

                // get random address
                int randomAddressIndex =
generator.nextInt(testAddresses.length);
                String address = testAddresses[randomAddressIndex].item;
                Boolean addressValidity =
testAddresses[randomAddressIndex].valid;

                // get random port
                int randomPortIndex = generator.nextInt(testPorts.length);
                String port = testPorts[randomPortIndex].item;
                Boolean portValidity = testPorts[randomPortIndex].valid;

                // get random path
                int randomPathIndex = generator.nextInt(testPaths.length);
                String path = testPaths[randomPathIndex].item;
                Boolean pathValidity = testPaths[randomPathIndex].valid;
```

```
                                        // get random query string
                                        int randomQueryStringIndex =
generator.nextInt(testQueryStrings.length);
                                        String queryString =
testQueryStrings[randomQueryStringIndex].item;
                                        Boolean queryStringValidity =
testQueryStrings[randomQueryStringIndex].valid;

                                        // build test case
                                        String testURL = scheme + address + port + path + queryString;
                                        Boolean testURLValidity = schemeValidity && addressValidity
&& portValidity
                                                                        && pathValidity && queryStringValidity;

                                        // test the test case
                                        UrlValidator urlVal = new UrlValidator(null, null,
UrlValidator.ALLOW_LOCAL_URLS);
                                        boolean result = urlVal.isValid(testURL);
                                        if (result != testURLValidity) {
                                                System.out.format("Failed for URL %s\n", testURL);
                                                testsFailed += 1;
                                        }
                        }
```

The benefit of this approach is that it allows us to run a large number of tests. We set it to run 10,000 tests and on average had around 340 tests failing. This lets us know that the program is accurate the majority of the time but there are definitely some buggy cases that come up consistently.

We found that although random testing is great for achiving broad code coverage, it was easier for us to find bugs by constructing manual and partition cases and concentrating on the differences between those. The same bugs happen in the random tests, but there is extra work required to sift throught the results, detect patterns, and figure out what is causing them.

Bug Report

Bug #1: isValid() invalidates any URL with a question mark

How it was detected:

In our manual and programmatic tests, we noticed a pattern that every url with a question mark was invalidated, even if it was otherwise valid. Question marks are a common part of URLs so we knew this was a significant bug.

Cause:

The query string part of the url is validated by this code in UrlValidator.java:

```
  protected boolean isValidQuery(String query) {
     if (query == null) {
        return true;
     }
     // error is on the following line
     return !QUERY_PATTERN.matcher(query).matches();
```

```
  }
```

The error is on line 446 in UrlValidator.java. The exclamation mark before QUERY_PATTERN reverses the boolean value returned by the matches() method. This means that all valid query strings return false and vice versa. This could be fixed by removing the exclamation point.

Bug #2: isValid() does not detect invalid IP addresses

How it was found: We noticed that in our programmatic tests, all tests with an invalid IP address were failing. Any IP address with a number over 255 is invalid. Here are some examples of otherwise valid URLS with invalid IP addresses that should have been invalidated:
ftp://255.256.255.255:80/i12iro3i
http://300.168.0.110:0/testpath
http://300.168.0.110:80/testpath1/testpath2
To confirm this, we added the following test to the manual tests, to confirm that this wasn't caused by some other bug:

```
  System.out.println(urlVal.isValid("http://256.256.256.256"));
```

The above test evaluated to true when it should have been false. Since everything else about the URL is valid, we can confirm that this is caused by a bug in IP address evaluation.
Cause:
The following lines of code show where the "authority" (combination of hostname and port) is checked by isValid()

```
        // check if authority is hostname or IP address:
        // try a hostname first since that's much more likely
        DomainValidator domainValidator =
DomainValidator.getInstance(isOn(ALLOW_LOCAL_URLS));
        if (!domainValidator.isValid(hostLocation)) {
        // try an IP address
       InetAddressValidator inetAddressValidator =
        InetAddressValidator.getInstance();
        if (!inetAddressValidator.isValid(hostLocation)) {
        // isn't either one, so the URL is invalid
        return false;
        }
        }
```

We can see that first it checks to see if the host is valid, and if it's not a vaid hostname, it then uses InetAddressValidator to check if it's a valid IP address.
The faulty section of the code starts on line 94 in InetAddressValidator.java:

```
        if (iIpSegment > 255) {

                return true;

        }
```

This is wrong because any IP segment greater than 255 is not valid. This could be fixed by setting this condition to return false instead of true.

Bug #3: isValid() does not detect invalid Top Level Domains

How it was found:

We noticed in our programmatic tests that a lot of urls with invalid top level domains were failing. For example, the url "http://google.kjfwoei" was evaluating to true.
To investigate further, we added the following manual tests:

```
System.out.println(urlVal.isValid("http://google.kjfwoei"));
System.out.println(urlVal.isValid("http://google.BLAHBLAH"));
System.out.println(urlVal.isValid("http://www.google.123HFJ"));
```

These were all expected to return false, but instead they all returned true. Since these urls are valid except for the TLD, we can confirm that this is an issue with the TLD validation
Cause:
Domain validation is handled by the DomainValidator class. We focused on this class file to isolate the problem.
We then focused on the isValidTld() function contained within DomainValidator. This is the part of the code that evaluates the TLD, starting on line 153 in DomainValidator.java:

```
public boolean isValidTld(String tld) {
if(allowLocal && isValidLocalTld(tld)) {
return true;
}
return isValidInfrastructureTld(tld)
|| isValidGenericTld(tld)
|| isValidCountryCodeTld(tld);
}
```

We notice that isValidLocalTld() is only evaluated if allowLocal is true, and we are running with the ALLOW_LOCAL_URLS option. So we were able to isolate this effect to the isValidLocalTld function, starting on line 202 in DomainValidator.java:

```
public boolean isValidLocalTld(String iTld) {
// the bug is in this line
        return !LOCAL_TLD_LIST.contains(chompLeadingDot(iTld.toLowerCase()));
}
```

The bug here is the exclamation mark before the evaluation of LOCAL_TLD_LIST.contains(). In the other similar functions, this exclamation mark is not present. This is important beacuse it reverses the Boolean value we are expecting. In other words, if our TLD is not in the list, it will return true, which is the opposite of the behavior we want. When we remove this exclamation point, we get the expected behavior.

Bug 4 - UrlValidator returns false for scheme h3t:// when it should return true.

Class: UrlValidator
Variable: DEFAULT_SCHEMES
Line: 180
Methodology: Partitioning tests revealed that every URL string containing the scheme h3t:// produced an unintended result when all other components were true.
Cause: Examination of the isValidScheme shows that it should return false if scheme is NULL, if an invalid scheme is detected or if an invalid scheme pattern is detected. Otherwise, isValidScheme should return true. It was determined in the isValidScheme method a call to this.allowedSchemes.contains(scheme) was made. However, in the UrlValidator(String[], RegexValidator, long) constructor, allowedSchemes is formed from DEFAULT_SCHEMES which only contain “http”, “https”, and “ftp”. H3T should be considered a default scheme and added to DEFAULT_SCHEMES, as opposed to having to create an exception through the use of the aforementioned constructor.