Project Report Part 2

CS362_400 F16

Due: 12/5/2016

Erin Sullens
Jason Ghiraldini
Keith Adkins
Markus Woltjer

Testing a Buggy Version of the Apache-URLValidator

1.0 An Overview of the Techniques We Used

The techniques we chose were intended to support a testing strategy of exploration for bugs. We were faced with creating a new test suite for a set of Java classes with which we were largely unfamiliar. We did not know what we were going to find, or what the best approach might be to help us find bugs. An initial review suggested that using Maven and jUnit in the Linux environment were tools well suited to testing these Java classes. Each class is self contained and only uses the other classes. This all led us to three test techniques we felt offered enough versatility, flexibility, and breadth for exploration to come up with a suite of effective tests. Namely, the techniques we chose where unit testing for line coverage using the jUnit testing tool in the Maven build manager, random testing for covering domain partitions also using jUnit, and mocking for isolating individual classes from other classes using Mockito and jUnit. Furthermore, we chose an additional technique to evaluate the quality of our test suite which is mutation testing. The mutation testing tool we used is called PIT, which we will discuss in section 2.0. We divided these techniques amongst our team and started exploring.

1.1 Unit tests (Erin)

I created a test suite of unit tests for the functions in URLValidator.java, RegexValidator.java, InetAddressValidator.java, and DomainValidator.java. The goal was to achieve good line coverage. I was able to find a few bugs in RegexValidator in the validate() function, and the match() function by testing them with some simple regular expressions. (Please see more detailed bug reports in the Appendix). They both returned either an empty string or an empty array when they both should have returned something. I also found a bug in DomainValidator in the isValidLocalTld() function. I found this bug by testing one of the localTld values, “localdomain”. This function returned false when it should have been true. Here is my unit test that failed for isValidLocalTld():

```
    @Test
       public void testDomainValidator6() {
         DomainValidator validator = DomainValidator.getInstance(true);
         boolean valid = validator.isValidLocalTld("localdomain");
         assertEquals(valid,true);
     }
```

1.2 Random Testing (Jason)

One of the methods that was a great benefit for this project was Random Testing. This method allowed us to set the bounds for the input of the function we would like to test, and run many variations of that input. This method allows us to essentially test thousands of unit tests using automation. One test class that was easily implemented for this method was the InetAddressValidator.java class. This class was supposed to test whether the entered ip address was valid. Given the bounds had to follow a certain pattern, we could set the Random Tester to produce inputs inside and outside that bounds. Below is an example of how this Random Tester was implemented for this class.

```
private static int ipGoodSegment(){
    final int MAX_RAND = 255;
    final int MIN_RAND = 0;
    Random r = new Random();
    return r.nextInt((MAX_RAND - MIN_RAND) + 1) + MIN_RAND;
  }
```

Example 1 - Random Test Helper Method

```
  public void validIpAddr1() {
    InetAddressValidator myInetAddr = new InetAddressValidator();

    // Number of times to test random ip address
    for(int j = 0; j < numLoops; j++){
      // Get random ip address
      for(int i = 0; i < 4; i++){
        segment1 = ipGoodSegment();
        ipSeg = Integer.toString(segment1);
        ip += segment1;
        if(i < 3)
        ip += dot;
      }
      if(!myInetAddr.isValid(ip)){
        System.out.print("ERROR - IP ADDR NOT CONSIDERED VALID: ");
        System.out.println(ip);
      }
      assertTrue(myInetAddr.isValid(ip));
    }
  }
```

Example 2 - Random Test IP Validity Code Snippet

This example shows that the helper function, Example 1, will return a random number between 0-255. This helper method will be called four times to concatenate a full ip address in the form of xxx.xxx.xxx.xxx where xxx can be 0-255. This ip address is then used as the input of the validator class. The benefit of using

random testing here is that we are able to test thousands of possibilities. Another great advantage of using this method is that we can change the bounds of the input very easily, which is exactly what was done for the next test. We simply changed the upper and lower bounds of the numbers that would be considered part of the ip address from 0-255 to 256-999. Changing these bounds should cause the ip validator to fail, however, the class returned a value of true for these addresses, thus uncovering a bug in this class. Random testing was also used in all other classes for this project. It was used to create random strings of random lengths for the DomainValidator, RegexValidator, and UrlValidator classes.

1.3 Mocking (Keith)

Mocking with Mockito was chosen as a test technique to help with isolating test classes from other classes. Focus was placed on the UrlValidator class for mocking since all of the other accompanying classes are used in this class. Mocking all the other classes would effectively isolate the UrlValidator class and help highlight bugs located within just its code. But, after some exploration, the only class that could be mocked was the RegexValidator class since this is the only class in the Apache-URLValidator that is passed into the UrlValidator constructor. All the other accompanying classes are contained in the UrlValidator class but not accessible, therefore not mockable.

This test suite can be found in the UrlValidatorMockTest.java file. It successfully helped identify three bugs. See the bug reports found in the Appendix for additional details.

1. A query string bug where the introduction of a valid query strings in an url results in the url being flagged as invalid. For example, this valid url with an added query string should be flagged as a valid url, but it is being flagged as invalid: “http://www.abc.com/path?abc=123&def=345”. The following code snippet was used to find this bug.

```
@Test
    public void mockTestIsValid11() {
        RegexValidator rv = mock(RegexValidator.class);
        UrlValidator uv = new UrlValidator(rv,0L);
        // Mock the optional regex authority validator to false.
        when(rv.isValid(anyString())).thenReturn(false);
        // query string added
        assertTrue(uv.isValid("http://www.abc.com/path?abc=123&def=345"));
    }
```

Code Snippet - Query String Bug

2. A port number bug where the introduction of a port number of 1000 or greater in the authority portion of an url results in the url being flagged as invalid. For example, the following valid url with an added port number of 1000 is being flagged as invalid when it should be valid. “http://www.abc.com:1000”. Valid port numbers can go from 0 to 65535. The following code snippet was used to find this bug.

```
    @Test
    public void mockTestIsValid12() {
        RegexValidator rv = mock(RegexValidator.class);
        UrlValidator uv = new UrlValidator(rv,0L);
        // Mock the optional regex authority validator to false.
        when(rv.isValid(anyString())).thenReturn(false);
```

```
        assertTrue(uv.isValid("http://www.abc.com:1000"));
        assertTrue(uv.isValid("http://www.abc.com:65535"));
        assertFalse(uv.isValid("http://www.abc.com:65536"));
}
```

Code Snippet - Port Number Bug

3. An IPv4 address bug where an IPv4 address with decimal values of 256 or higher flags an url as being valid when is should be invalid. For example, the following invalid url is being flagged as valid when it should be invalid: "http://256.256.256.256". Valid decimal numbers in IPv4 dotted decimal notation can be from 0 to 255. The decimal number 256 is invalid. The following code snippet was used to find this bug.

```
@Test
    public void mockTestIsValid13() {
        RegexValidator rv = mock(RegexValidator.class);
        UrlValidator uv = new UrlValidator(rv,0L);
        // Mock the optional regex authority validator to false.
        when(rv.isValid(anyString())).thenReturn(false);
        // invalid ip address
        assertFalse(uv.isValid("http://256.256.256.256"));
}
```

Code Snippet - IPv4 Address Bug

1.4 Bug Verification (Markus):

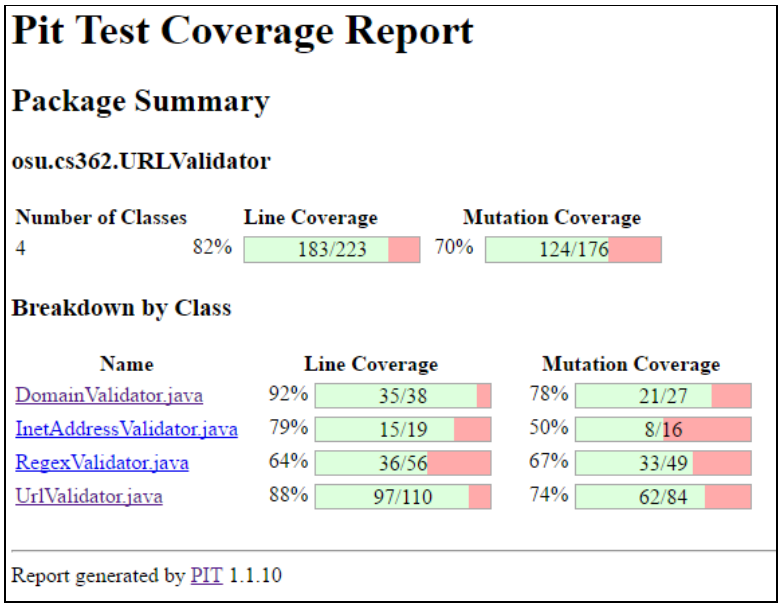
The random testing suite validated the String Validation Bug, the Top Level Domain Validation Bug, and the Large IPv4 Byte Rejection Bug were independently verified through the random testing suite. Unit tests were repeated with modified inputs and assertion expectations to confirm the extent and underlying failure of each bug to be that which was originally reported.

2.0 Evaluation of the New Tool We Used: PIT, Mutation Testing (Keith):

The additional tool we used is PIT, a mutation testing tool for use with Java (http://pitest.org/). Basically, PIT provides a measure of how good a test suite is by randomly inserting bugs, called mutations, into the source code that is being tested. If a test suite finds a mutation, PIT will kill it. Otherwise, the mutation lives. A living mutation is an indication of coverage that a test suite is missing. PIT provides a good addition to a test suite in that it helps identify areas where test coverage is missing, not only in terms of line coverage, but also in terms of domain partition coverage.

PIT goes beyond other coverage test tools we have used, namely Cobertura, in that PIT can change a line of code in various ways that may require more than one test case to catch. Cobertura simply measures if a test case "touches" a line. PIT in this way provides a better measure of a test suite's quality compared to Cobertura. But PIT does not replace Cobertura. Cobertura is still a very useful tool to highlight lines of code that have been completely missed.

PIT also provides a measure of line coverage in its output report for comparison to the coverage measured through mutation testing. The following screenshot are the results we achieved with our current test suite. While we achieved 82% line coverage, our mutation coverage was 70%.



Pit Test Coverage Report

Package Summary

osu.cs362.URLValidator

Number of Classes	Line Coverage		Mutation Coverage	
4	82%	183/223	70%	124/176

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
DomainValidator.java	92%	35/38	78%	21/27
InetAddressValidator.java	79%	15/19	50%	8/16
RegexValidator.java	64%	36/56	67%	33/49
UrlValidator.java	88%	97/110	74%	62/84

Report generated by PIT 1.1.10

PIT Report of Our Test Suite

Appendix. Bug Reports (Markus):

Bug Report #1

Bug Title: Regular Expression Matching Bug

Class: RegexValidator

Date: November 27, 2016
Reported By: Erin Sullens
Email: sullense@oregonstate.edu

Product: Apache URL Validator
Platform: OSU flip server (Linux).

Is it reproducible: Yes

Steps to Produce/Reproduce

1. Direct from without our groups submitted GitHub repository Apache-URLValidator to the subdirectory BuggyURLValidator.
2. Run the unit test script found there by entering: run-unit-tests

Expected Results

The regular expression “a*b” and the string “aab” should match, so the expectation written into the assertion is that the boolean “valid” would be true, indicated that the array of matched regular expressions is not empty.

Actual Results

However, the assert at the end confirmed that the boolean flag had been lowered due to the length of the array of matched regular expressions being zero. This indicates that the match method in the RegexValidator class failed to identify that the string “aab” matches the regular expression “a*b”.

(Note about failed test Maven reports: Maven states that it was expecting false, but the value in the assert was actually true, which isn’t intuitive, but that’s just because assertEquals tests the second value against the first, whereas the test case was written with the expected value second. It isn’t a problem in such simple cases as this and the following failed tests cases, but when writing JUnit tests on more complex projects, it may be helpful to keep in mind that the failures of assertions are reported specific to their format.

Snippet of a test case that fails

```
@Test public void testRegexValidatorMatch() {
        String[] regs = {"a*b", "ab*"};
        RegexValidator regex = new RegexValidator(regs, true);
        String[] matched = regex.match("aab");
        boolean valid = true;
        if(matched.length == 0)
                valid = false;
        System.out.println("matcher: " + matched.length);
        assertEquals(valid,true);
}
```

Justification for why this test case should pass.

The test case creates an array of regular expressions for any nonzero number of a’s followed by b and an a followed by any number of b’s. The value to match is the string “aab”, which matches the first regular expression in the array. Hence we would expect it to add that regular expression to the matched array, and increase its length beyond zero, which would not flip the boolean flag “valid”, and the assert should pass.

Bug Report #2

Bug Title: String Validation Bug

Class: RegexValidator

Date: November 7, 2016
Reported By: Erin Sullens
Email: sullense@oregonstate.edu

Product: Apache URL Validator
Platform: OSU flip server (Linux).

Is it reproducible: Yes

Steps to Produce/Reproduce

1. Direct from without our groups submitted GitHub repository Apache-URLValidator to the subdirectory BuggyURLValidator.
2. Run the unit test script found there by entering: run-unit-tests

Expected Results

The RegexValidator class' Validate method claims to also match a given string against the array of regular expressions, but instead of returning an array of strings like the match method does, validate aggregates the strings into a buffer and then returns the aggregation as a string. Hence, validating the same string "aab" against the same regular expressions "a*b" and "ab*" should return the string "a*b".

Actual Results

The test method returns an empty array, meaning that none of the regular expressions matched. This is likely due to the same logical error in regular expression matching, since it is failing the same match as the Regular Expression Matching Bug in the same way.

Snippet of a test case that fails

```
@Test public void testRegexValidatorValidate() {
        String[] regs = {"a*b", "ab*"};
        RegexValidator regex = new RegexValidator(regs, true);
        String valid = regex.validate("aab");
        assertEquals(valid,"a*b");
}
```

Justification for why this test case should pass.

"aab is a string that matches the regular expression "a*b", so the validate method should stringify a single "component" of "a*b" in the buffer and return it as a string.

Bug Report #3

Bug Title: Top Level Domain Validation Bug

Class: DomainValidator

Date: November 28, 2016
Reported By: Erin Sullens
Email: sullense@oregonstate.edu

Product: Apache URL Validator
Platform: OSU flip server (Linux).

Is it reproducible: Yes

Steps to Produce/Reproduce

1. Direct from without our groups submitted GitHub repository Apache-URLValidator to the subdirectory BuggyURLValidator.
2. Run the unit test script found there by entering: run-unit-tests

Expected Results

The isValid method in the DomainValidator class is expected to return true when passed strings that parse to top level domains. One tested domain was “google.com”, which was expected to be a valid domain, and return true.

Actual Results

Snippet of a test case that fails

```
@Test public void testDomainValidator() {
        DomainValidator validator = DomainValidator.getInstance();
        boolean valid = validator.isValid("google.com");
        assertEquals(valid,true);
}
```

Justification for why this test case should pass.

Clearly, “google.com” is a valid top level domain. Hence, the isValid must return true, and have the (true) return value assigned to the boolean “valid.” If valid is a boolean truth, it is identical to the “true” tested for in the assertion, which should then pass.

Bug Report #4

Bug Title: Query URL Bug

Class: UrlValidator

Date: November 28, 2016
Reported By: Keith Adkins
Email: adkinske@oregonstate.edu

Product: Apache URL Validator
Platform: OSU flip server (Linux).

Is it reproducible: Yes

Steps to Produce/Reproduce

1. Construct an UrlValidator object with default options.
2. Call the isValid method within the UrlValidator object, passing it a valid url that includes a valid query string.
3. Check the returned boolean value.
4. End the test.

Expected Results

The isValid method in the UrlValidator object should return true.

Actual Results

The below mock test falls, indicating that the UrlValidator's isValid function, which in other similar mock tests passed for other abc website URLs, did not accept the use of a query string, because the regular expressions for query strings are not present our or not being matched to the query during correctly due to regular expression logic.

Snippet of a test case that fails

```
@Test
    public void mockTestIsValid11() {
        RegexValidator rv = mock(RegexValidator.class);
        UrlValidator uv = new UrlValidator(rv,0L);
        // Mock the optional regex authority validator to false.
        when(rv.isValid(anyString())).thenReturn(false);
        // query string added
        assertTrue(uv.isValid("http://www.abc.com/path?abc=123&def=345"));
    }
```

Justification for why this test case should pass.

The given URL is typical form for a query on many websites and should be acceptable to the UrlValidator for valid domains. The abc website is clearly a valid domain so there is another instance of faulty regular expression logic surfacing, possibly rooted in the regular expression logic errors underlying previous bugs.

Bug Report #5

Bug Title: High Port Number Rejection Bug

Class: UrlValidator

Date: November 28, 2016
Reported By: Keith Adkins
Email: adkinske@oregonstate.edu

Product: Apache URL Validator

Platform: OSU flip server (Linux).

Is it reproducible: Yes

Steps to Produce/Reproduce

1. Construct an UrlValidator object with default options.
2. Call the isValid method within the UrlValidator object, passing it a valid url that includes a four digit port number.
3. Check the returned boolean value.
4. End the test.

Expected Results

The isValid method in the UrlValidator object should return true.

Actual Results

Ports 1000 and 66535 both failed assertions, and upon further investigation so did all ports from 1000 to 66535.

Snippet of a test case that fails

```
@Test
    public void mockTestIsValid12() {
        RegexValidator rv = mock(RegexValidator.class);
        UrlValidator uv = new UrlValidator(rv,0L);
        // Mock the optional regex authority validator to false.
        when(rv.isValid(anyString())).thenReturn(false);
        assertTrue(uv.isValid("http://www.abc.com:1000"));
        assertTrue(uv.isValid("http://www.abc.com:65535"));
}
```

Justification for why this test case should pass.

While this is the authority part, the use of the RegexValidator instance was determined correct in passing tests of the same mock test with ports under 1000. Ports 1000-65535 are supposed to have authority, meaning isValid for the UrlValidator instance should return true for the abc domain specified to that port.

Bug Report #6

Bug Title: Large IPv4 Byte Rejection Bug

Class: InetAddressValidator

Date: November 28, 2016
Reported By: Keith Adkins
Email: adkinske@oregonstate.edu

Product: Apache URL Validator
Platform: OSU flip server (Linux).

Is it reproducible: Yes

Steps to Produce/Reproduce

1. Construct an UrlValidator object with default options.
2. Call the isValid method within the UrlValidator object, passing it an url made up of an IPv4 address in dotted decimal notation. The url should be constructed in a valid format, other than the IPv4 address should be 256.256.256.256.
3. Check the returned boolean value.
4. End the test.

Expected Results

The isValid method in the UrlValidator object should return false.

Actual Results

This assertion failed, meaning that isValid determined the IPv4 address to be valid.

Snippet of a test case that fails

```
@Test
    public void mockTestIsValid13() {
        RegexValidator rv = mock(RegexValidator.class);
        UrlValidator uv = new UrlValidator(rv,0L);
        // Mock the optional regex authority validator to false.
        when(rv.isValid(anyString())).thenReturn(false);
        // invalid ip address
        assertFalse(uv.isValid("http://256.256.256.256"));
}
```

Justification for why this test case should pass.

The assertion should have failed because IPv4 dotted notation represents each dot-delimited segment as a byte, for which decimal representations are restricted to 0-255. Hence, isValid should have failed, which would pass the assertion.