Term Project

URLValidator

Part II

CS362 - Winter 2016

Lucia Blackwell

Alicia Broederdorf

David Rigert

Karen Thrasher

Part II - URLValidator Testing Report

1 Introduction

Apache URLValidator is a Java library which facilitates validation of commonly-used Web address formats. It is comprised of four main features including the domain, inet address, regex, and url validators.

2 Testing Techniques

Unit tests were created for each of the four features using the JUnit framework. These unit tests were designed using input domain partitioning to maximize code coverage and edge cases. Cobertura was used to analyze line and branch coverage, while FindBugs was utilized to complete static testing of the code. In cases where the functionality of one feature depended on another, mocks were used to isolate the features.

Input domain partitioning was the driver behind creating the unit tests. As the features all validate some information, the main partition was using valid and invalid input. As both valid and invalid input are comprised of many options, several inputs were used for each domain according to the feature being tested. For example, the `InetAddressValidator` class tested the minimum and maximum range for a valid IP address as valid input, along with invalid input that included characters other than decimals or ranges that were outside of the maximum.

Test including valid, invalid, and bad input (from IDP_validInputTest):

```
String testMax = "255.255.255.255"; //Max range IP address
String testMin = "0.0.0.0"; //Min range of IP address
String badRange2 = "999.888.777.1111";
String badDot1 = "100a100.100.100";

testResult = inetAddrVal.isValid(testMax);
assertEquals(true, testResult);
testResult = inetAddrVal.isValid(testMin);
assertEquals(true, testResult);
testResult = inetAddrVal.isValid(badRange2);
assertEquals(false, testResult);
testResult = inetAddrVal.isValid(badDot1);
assertEquals(false, testResult);
```

Following this same approach, the domain validator unit tests tested the various `isValid` methods for infrastructure, generic, country, and local using both valid and invalid input. As an example, the following code snippet is for an invalid test of the `isValidCountryCodeTld` method in the domain validator. This test case uses the JUnit assertFalse to ensure the result returned from the validator method is false as the input is invalid:

```
@Test
public void invalidCountryTld(){
   DomainValidator dv = DomainValidator.getInstance();
   assertFalse(dv.isValidCountryCodeTld(".com"));
   assertFalse(dv.isValidCountryCodeTld(".localhost"));
   assertFalse(dv.isValidCountryCodeTld(".madeuptld"));
   assertFalse(dv.isValidCountryCodeTld(".123"));
   assertFalse(dv.isValidCountryCodeTld(".***"));
   assertFalse(dv.isValidCountryCodeTld(""));
}
```

The input domains of valid and invalid were also further subdivided for the regex validator into case sensitive and insensitive. Several potential inputs were tested against a regex with both a single value or an array of values. The unit tests were made to be case insensitive by passing a false boolean argument in the `RegexValidator` constructor. The following are code snippets from two of the unit tests for the regex validator using valid input, and thus, using the JUnit `assertTrue` assertion to ensure the result returned from the `isValid` method is true for valid input. Note the change in the boolean from case insensitive to sensitive.

Case insensitive test with single regular expression (from testRegex_ValidInsensitive_isValid):

```
String reg1 = "ab{1,5}a";
RegexValidator rv1 = new RegexValidator(reg1, false);
boolean isValBool = rv1.isValid("Aba");
assertTrue(isValBool);
```

Case sensitive test with array of regular expressions (from testRegex_ValidSensitive_isValid):

```
String[] regArr = {"ab{1,5}a", "d", "[m-z]+", "(cat) (.+?) ", "([0-9M-V]+)"};
RegexValidator rvMulti = new RegexValidator(regArr);
boolean isValBool = rvMulti.isValid("cat watches ");
assertTrue(isValBool);
```

The `UrlValidator` class was the largest with the most complex state, and as such required the greatest number of partitions and unit tests to ensure sufficient coverage. Furthermore, it has dependencies on both the `RegexValidator` and the `DomainValidator` classes.

The public interface does not provide a way of modifying the `DomainValidator` field, but one of the constructors allows you to specify a custom `RegexValidator` for the authority part of the URL. To test that the `authorityValidator` was being correctly invoked, a mock was created and then the `verify` method was used to confirm that the validator's `isValid` method was invoked the correct number of times. An assert was used to check if the `UrlValidator` object was returning the expected result based on the mocked authority validator.

```
RegexValidator rv = mock(RegexValidator.class);
when(rv.isValid(any(String.class))).thenReturn(false);
validator = new UrlValidator(rv, 0);
url = "http://www.test.com";
assertFalse(url, validator.isValid(url));
verify(rv).isValid(any(String.class));
```

As part of the testing process, a helper class, called `UrlHelper`, was created to consolidate the information needed to produce valid URLs. This class includes arrays of all valid top-level domains and various other metadata based on the syntax requirements for URLs. This information was also used to write a method that generates random valid URLs. Certain components, including the scheme, user information, port, path, query string and fragment, can be toggled on and off with arguments. The generated URLs respect all requirements defined in RFC 1034, RFC 2181, and RFC 3986, and only use schemes and top-level domains defined on the IANA website.

The pom.xml file was configured to generate a site report that includes the Cobertura coverage and the FindBugs static analysis results. The coverage report provides some insight into the quality of the test suite in examining what percentage of lines of code and conditional statements are covered by the unit tests. It also helped tailor some of the unit cases to ensure a higher branch coverage. The static analysis report includes potential bugs found in the classes classified by priority and type. There are three bug categories and further bug codes that can be found in the FindBugs documentation. For instance, the static analysis returned two medium priority bugs in the `ResultPair` class. The bugs were classified as dodgy with code UrF, or unread field. This means these lines of code are never read and could be considered for removal from the class. In this particular package, the class itself is never used and could potentially be removed.

3 Summary of Bugs

A summary of the twelve bugs found are listed below. *For detailed bug reports, please see section 5.*

- InetAddressValidator:
  - Allows range greater than 255+ for dotted decimal numbers
- DomainValidator:
  - Enabling local domains to be considered valid does not work
  - Does not recognize country codes in the last half of the alphabet
- UrlValidator:
  - Does not recognize URLs with user authentication information
  - Considers URL with invalid label (longer than 63 characters) to be valid
  - Does not recognize URLs with a query string
  - Does not recognize localhost with a fragment
  - Considers URL with invalid whitespace after port to be valid
  - Does not catch malicious URLs that use too many double-dots
  - Does not catch URLs with invalid characters in fragment
  - Does not recognize URLs with a valid port number of 65535
  - Does not respect return value of custom authorityValidator

4 Tool Comparison

A change from the test plan included using Cobertura for analyzing line and branch coverage, rather than using Pitest for line coverage and mutation testing. For one, URL strings have such a wide range of valid input that mutation testing is of limited utility. The tool also would not generate reports when failed test cases were encountered, and the test suite includes several which identify bugs. Instead, Cobertura was used to get an understanding of the quality of the test suite through its line and branch coverage reports.

FindBugs is a completely new tool that provides static analysis capability to identify potential bugs and security issues based on the identification of troublesome patterns in the code. There were no prior techniques that provided this same capability. One of the biggest differences between using this tool compared with other testing techniques is that it analyzes the uncompiled code without the need to write any unit tests.

This particular project is a modification of Apache URLValidator, which has a fairly mature code base. Therefore, static analysis did not find very many issues in the code. It did, however, correctly identify two unused public members of the `ResultPair` class. The fact that all public APIs were reported as unused means that the entire class is unused, and could therefore be removed completely without impacting any other functionality of the library.

5 Bug Reports

InetAddressValidator

Title: Allows range greater than 255+ for dotted decimal numbers

Class: Serious bug

Description: InetAddressValidator allows for input that exceeds the maximum allowed for IPv4 addresses written as dotted decimal. Specifically, it allows for the range of each segment of dotted decimal number to exceed 255, which is the maximum allowed in the numbering system.

Steps to Produce:

1. Instantiate the InetAddressValidator class
```
InetAddressValidator inetAddrVal = new InetAddressValidator();
```
2. Receive or create input that exceeds maximum allowed value of 255.255.255.255
```
String badRange = "256.256.256.256";
```
3. Run isValid() method on input
```
boolean result = inetAddrVal.isValid(badRange);
```

Expected Results: `result` should be false.

Actual Results: `result` is true.

DomainValidator

Title: Enabling local domains to be considered valid does not work

Class: Serious bug

Description: DomainValidator does not recognize .localhost and .localdomain to be valid when enabled.

Steps to Produce:

1. Get the DomainValidator singleton
```
DomainValidator dv = DomainValidator.getInstance(true);
```
2. Run isValidLocalTld() with a valid local domain name
```
assertTrue(dv.isValidLocalTld(".localhost"));
assertTrue(dv.isValidLocalTld(".localdomain"));
```

Expected Results: Test case should assert that the result is true.

Actual Results: Test case assertion is false.

Title: Does not recognize country codes in the last half of the alphabet

Class: Serious bug

Description: DomainValidator is only able to validate the first half of country codes listed alphabetically.

Steps to Produce:

1. Get the DomainValidator singleton
```
DomainValidator dv = DomainValidator.getInstance();
```
2. Run isValid with a domain including a country code from the latter half of the alphabet
```
boolean result = dv.isValid("xyz.zw");  // Zimbabwe code
```

Expected Results: `result` should be true

Actual Results: `result` is false

UrlValidator

Title: Does not recognize URLs with user authentication information

Class: Serious bug

Description: UrlValidator does not recognize valid URLs that include user authentication information as valid.

Steps to Produce:

1. Instantiate a UrlValidator with the default configuration
```
UrlValidator validator = new UrlValidator();
```
2. Call isValid on a valid URL that includes user authentication information.
```
String url = "http://username:password@www.test.com";
boolean result = validator.isValid(url);
```

Expected Results: `result` should be true.

Actual Results: `result` is false.

Title: Considers URL with invalid label (longer than 63 characters) to be valid

Class: Minor bug

Description: The isValid method considers a URL with a label longer than 63 characters to be valid. According to RFC 2181, each label can be no longer than 63 octets (characters).

Steps to Produce:

1. Instantiate a UrlValidator with the default configuration
```
UrlValidator validator = new UrlValidator();
```
2. Create a URL that has a label with 64 characters
```
StringBuilder sb = new StringBuilder();
sb.append("http://www.")
for (int i = 0; i < 64; ++i) {
    sb.append("a");
}
sb.append(".com");
```
3. Call isValid on the invalid URL.
```
boolean result = validator.isValid(sb.toString());
```

Expected Results: `result` should be false.

Actual Results: `result` is true.

Title: Does not recognize URLs with a query string

Class: Serious bug

Description: UrlValidator does not recognize URLs that include a valid query string as valid. Considering how common query strings are in modern dynamic websites, this would impact a large percentage of URLs.

Steps to Produce:

1. Instantiate a UrlValidator with the default configuration
```
UrlValidator validator = new UrlValidator();
```
2. Call isValid on a valid URL with a query string
```
String url = "http://www.test.com?param=value";
boolean result = validator.isValid(url);
```

Expected Results: `result` should be true.

Actual Results: `result` is false.

Title: Does not recognize localhost with a fragment

Class: Serious bug

Description: UrlValidator does not recognize a localhost with a fragment when the correct option for it is enabled.

Steps to Produce:

1. Instantiate a UrlValidator with the ALLOW_LOCAL_URLS option enabled
```
UrlValidator validator =
    new UrlValidator(UrlValidator.ALLOW_LOCAL_URLS);
```
2. Call isValid on a valid local URL with a fragment
```
String url = "http://localhost#fragment";
boolean result = validator.isValid(url);
```

Expected Results: `result` should be true.

Actual Results: `result` is false.

Title: Considers URL with invalid whitespace after port to be valid

Class: Minor bug

Description: URLValidator accepts a URL with whitespace after the port number although this should be invalid

Steps to Produce:

1. Instantiate a UrlValidator with the default configuration
```
UrlValidator validator = new UrlValidator();
```
2. Call isValid on invalid local URL with whitespace after port number
```
String url = "http://www.test.com:80  /path";
boolean result = validator.isValid(url);
```

Expected Results: `result` should be false

Actual Results: `result` is true

Title: Does not catch malicious URLs that use too many double-dots

Class: Vulnerability

Description: URLValidator accepts a URL with double dots that go above the root directory of the website. This is a security vulnerability in naive web server implementations.

Steps to Produce:

1. Instantiate a UrlValidator with the default configuration
```
UrlValidator validator = new UrlValidator();
```
2. Call isValid on invalid URL with multiple sets of double dots
```
String url = "http://www.test.com/../../../etc/passwd";
boolean result = validator.isValid(url);
```

Expected Results: `result` should be false

Actual Results: `result` is true

Title: Does not catch URLs with invalid characters in fragment

Class: Minor bug

Description: UrlValidator incorrectly detects a URL fragment that contains invalid characters as valid. According to RFC 3986, the URL fragment can only include the following characters: `!$&'()*+,;=-._~:@/?` plus alphanumerics. Any other character, such as the caret (`^`) is considered to be invalid and must be encoded.

Steps to Produce:

1. Instantiate a UrlValidator with the default configuration
   ```
   UrlValidator validator = new UrlValidator();
   ```
2. Call isValid on a URL with a caret in the fragment
   ```
   String url = "http://www.test.com#fr^gment";
   boolean result = validator.isValid(url);
   ```

Expected Results: `result` should be false.

Actual Results: `result` is true.

Title: Does not recognize URLs with a valid port number of 65535

Class: Serious bug

Description: UrlValidator does not recognize URL with a valid maximum port number of 65535 to be valid.

Steps to Produce:

1. Instantiate a UrlValidator with the default configuration
   ```
   UrlValidator validator = new UrlValidator();
   ```
2. Call isValid on a URL with the maximum port number
   ```
   String url = "http://www.osu.edu:65535";
   boolean result = validator.isValid(url);
   ```

Expected Results: `result` should be true.

Actual Results: `result` is false

Title: Does not respect return value of custom authorityValidator

Class: Serious bug

Description: When a UrlValidator instance is created with a custom RegexValidator for the authority part, the UrlValidator.isValid method can still return true even if the RegexValidator.isValid method returns false.

Steps to Produce:

1. Instantiate the UrlValidator class with a custom RegexValidator for the authorityValidator argument.

```
RegexValidator rv = new RegexValidator("foo");
UrlValidator validator = new UrlValidator(rv, 0);
```

2. Call isValid with a valid URL that fails the custom RegexValidator.isValid method.

```
String url = “http://www.test.com”;
boolean result = validator.isValid(url);
```

Expected Results: `result` should be false.

Actual Results: `result` is true.