CS362_400_S17

Final Project - Part B

6/6/2017

Group Members:

- Alex Giesbrecht - giesbral@oregonstate.edu
- Michael Sterritt - sterritm@oregonstate.edu
- Ian Stanfield - stanfiei@oregonstate.edu

Methodology Testing -

Test Methodology

Our group used a combination of manual testing and unit testing to test and identify bugs in the URL Validator application from Apache Commons. During manual and unit testing, we attempted to provide test cases with good boundary coverage as well as broad coverage of the variety of different URL components and their combinations. Boundary coverage included testing scheme, authority, path, and query URL parts that were empty, included unescaped reserved characters, escaped reserved and unreserved characters, and parts with otherwise invalid formatting according to ISOC's generic syntax standards for URIs[1]. Broad test coverage of mainstream URLs included a variety of test cases and combinations of different URL components, including: scheme, hostname, port number, IP literal, country code, path, and query. Additionally, we used input partitioning to focus the test cases for our unit tests and to ensure.

Manual Test Cases

The following are several example inputs used during our manual testing:

- *http://www.google.com*
- *ftp://www.google.com*
- *http://0.0.0.0*
- *http://255.255.255.255*
- *http://256.256.256.256*
- *http://www.google.ca*
- *http://www.google.jp*
- *http://www.google.com:65535*
- *http://www.google.com:65536*
- *http://www.google.com/path/to/go*
- *http://www.google.com//path/to/go*
- *http://www.google.com/path?key=value*

[1] Berners-Lee, et al., "Uniform Resource Identifier (URI): Generic Syntax" RSC 3986, The Internet Society, January 2005, 6 June 2017, <https://tools.ietf.org/html/rfc3986>

Partitioning
We partitioned our input (URL strings) into the following sections (with example URLs) when creating test cases for our unit tests:

- **Scheme** - `https://`, `http://`, `aaa:`, `ftp://`
- **Authority -** `255.255.255.255`, `google.com`, `localhost`, `google.jp`
- **Port -** `:-1`, `:0`, `:65535`, `:100000`
- **Path -** `/file`, `/%20`, `//file`, `/file1/file2`
- **Query -** `?key=value`, `?key1=value&key2=value`, `?key = value &`

We used these partitions because they closely followed the URL syntax form outlined by ISOC and because we believe they adequately compartmentalize and cover the concerns of the URL Validator application. Each partition, scheme, authority, port, path, and query, is parsed by the URL Validator from the raw input and evaluated separately by the program. If any one of these pieces is invalid, the URL validator returns invalid for the entire URL. By matching our partitions to the concerns and structure of the application, it was easier for us to address and narrow down the source of our bugs.

Test Names
Our manual tests are contained in the class URLValidatorTest.java in the **TestManualTest** method. Calling this class as a JUnit test will run all of our manual test cases through the URL Validator application. Each manual test will print the results of calling URL Validator with the corresponding test case to the console.

Our test cases for our unit tests are built and evaluated in the class URLValidatorTest.java in the **testIsValid** method. Calling this class with JUnit will run all of our unit tests on the URL Validator application. Unit tests that fail will print out the test results as well as system state at the time of failure, the set of expectations per each partition, and the expected vs. the actual results from URL Validator.

Bug Reports -

1. **Bug #1** - Expected length for a valid port is too short
    - **Error Text & Input Examples:**

```
FAIL 1: "https://google.com:1000" was validated incorrectly.
        ---> expected: true, actual: false
FAILURE EXPECTATIONS ---
        SCHEME: true,
        AUTHORITY: true,
        PORT: true,
        PATH: true,
        QUERY: true

FAIL 2: "https://google.com:10000" was validated incorrectly.
        ---> expected: true, actual: false
FAILURE EXPECTATIONS ---
        SCHEME: true,
        AUTHORITY: true,
        PORT: true,
        PATH: true,
        QUERY: true
```

- **Error Description:** Otherwise valid URLs that include a port number longer than three characters are evaluated as invalid by the UrlValidator class. Port numbers in URLs should be 1 to 5 digits in length. URLs that include port numbers that are three characters or shorter in length are correctly evaluated as valid. URLs that have more than 5 characters are correctly validated as invalid.
- **Test Case & Identification Process:** The failure was identified when testing a url with the maximum valid port, 65535, in manual tests ended up returning false. However, in order to identify the nature and cause of the bug we ended up using white-box testing and looked at the code to identify how it was validating ports. This led to us discovering that the regex expression governing ports was only validating numbers 3 digits long instead of up to 5.
- **Suspected Cause in Code:** The bug is caused by the urlValidator's regex expression for ports at line 158 in UrlValidator.java. The regex expression only flags ports between 1 and 3 digits as being correct, when it should flag ports between 1 and 5 digits correct as well in order to validate all valid ports.

2. **Bug #2** - Local URLs are not validated correctly
   - **Error Text & Input Examples:**

```
FAIL 12: "http://localhost" was validated incorrectly.
        ---> expected: true, actual: false
FAILURE EXPECTATIONS ---
        SCHEME: true,
        AUTHORITY: true,
        PORT: true,
        PATH: true,
        QUERY: true

FAIL 13: "http://machine" was validated incorrectly.
        ---> expected: true, actual: false
FAILURE EXPECTATIONS ---
        SCHEME: true,
        AUTHORITY: true,
        PORT: true,
        PATH: true,
        QUERY: true
```

   - **Error Description:** When a valid local URL authority is passed into urlValidator, it evaluates the URL string as invalid.
   - **Test Case & Identification Process:** A valid URL authority is passed into urlValidator, alongside partitions that have have previously evaluated correctly. In our test cases, our URL strings uniquely contain `localhost` and `machine`, along with `http://` as a scheme, which has evaluated as true, and and a blank port, blank path, and blank query, all of which are valid.
   - **Suspected Cause in Code:** When the Local URL is evaluated by isValidLocalTld(), the function returns the opposite boolean value of what is expected, thus a valid Local URL is evaluated as invalid.

3. **Bug #3** - IP addresses exceeding 255 are valid in dot-decimal notation
    - **Error Text & Input Examples:**

```
FAIL 1: "https://256.256.256.256" was validated incorrectly.
       ---> expected: false, actual: true
FAILURE EXPECTATIONS ---
       SCHEME: true,
       AUTHORITY: false,
       PORT: true,
       PATH: true,
       QUERY: true
```

    - **Error Description:** When validating urls, the url validator returns true for values greater than 255 when IPv4 addresses are written in quad-dotted notation (xxxx.xxxx.xxxx.xxxx). Dotted-decimal has a numeric range of 0-255 due to each number representing 8-bits in memory and is shown in RFC 790[2] tables.
    - **Test Case & Identification Process:** The bug was identified during manual testing when we tested the url listed above. The test was meant to test the upper-range of dotted-decimal notation and the expected result was false from urlValidator. However, the test returned and continued to return true for numbers exceeding 256 as well in other tests.
    - **Suspected Cause in Code:** The cause of this bug is found at line 91 in InetAddressValidator.java. At that location, the if statement checking if an IP address's numeric value exceeds 255 returns true instead of the expected false value for numbers that exceed 255.

4. **Bug #4** - Some country code top-level domains result in invalid urls
    - **Error Text & Input Examples:**

```
FAIL 1: "https://www.google.jp" was validated incorrectly.
       ---> expected: true, actual: false
FAILURE EXPECTATIONS ---
       SCHEME: true,
       AUTHORITY: true,
       PORT: true,
       PATH: true,
       QUERY: true
```

    - **Error Description:** When validating a urls domain, all registered country code top-level domains alphabetically after .it (Italy) return invalid.
    - **Test Case & Identification Process:** The bug was first identified in manual tests when the .jp domain returned false for a url that was expected to be valid, but other country domains, such as .ca, returned true. Further testing was performed during partition and unit tests, where all country code domains were tested and they showed that all domains alphabetically following .it were returning invalid while those prior returned true.
    - **Suspected Cause in Code:** The cause of the bug is that the COUNTRY_CODE_TLDS array in DomainValidator (located at line 248 in DomainValidator.java) is incomplete and only contains entries up to .it instead of all registered country domains. Therefore, domains not included in the array end up returning false since they are not hard-coded as valid domains.

[2] J. Postel, "Assigned Numbers" RSC 790, September 1981, 6 June 2017, <https://tools.ietf.org/html/rfc790>

5. **Bug #5** - Urls containing queries are always invalid
    - **Error Text & Input Examples:**

```
FAIL 1: "https://google.com:0/path?key=value" was validated incorrectly.
       ---> expected: true, actual: false
FAILURE EXPECTATIONS ---
       SCHEME: true,
       AUTHORITY: true,
       PATH: true,
       PORT: true,
       QUERY: true
```

    - **Error Description:** Whenever a url contains a query, which identified as beginning with a '?', the url always returns invalid even if the query is valid and all other url components are valid.
    - **Test Description & Identification Process:** The bug was first identified in manual testing when all urls containing with queries ended up failing. The bug then continued to show for all partitions with valid queries and in unit tests.
    - **Suspected Cause in Code:** The cause of the bug is that the member function isValidQuery in the UrlValidator class (line 446 in UrlValidator.java) returns the negative of the query_pattern matches function found at line 446. Therefore, when the query matches the provided query pattern, indicating a valid query, the function returns invalid (false) instead of valid (true).

Debugging -

In each case, bugs were investigated and identified using the debugging tools provided with the Eclipse IDE.

- **Bug #1** - Expected length for a valid port is too short
    - **Bug Location -** Line 158 in UrlValidator.java
    - **Debugging Details -**
        - We began the debugging process by creating a breakpoint at the failed test; the test input we used for debugging was `https://google.com:1000`; which we expected to be a valid URL but for which the URL Validator returns invalid.
            - Agan's Rule #2 "Make it Fail" & Agan's Rule #4 "Divide and Conquer" - we used a number of test cases, isolating otherwise valid URLs and giving them port numbers with different lengths and values, making sure we could reliably make the system fail on this single issue.
        - Because this was the first bug we encountered, we plan on stepping through each part of the code until we find unexpected behavior
            - Agan's Rule #1 "Understand the System" & Agan's Rule #3 "Quit Thinking and Look" - The first thing we do step through each part of the code so we can make sure that we know how the UrlValidator works, what it is doing with the test cases it's being given, and to understand and pinpoint which part of the code is failing.
        - Then we started the test in debug mode; debug mode allowed execution to stop at our breakpoint.
        - We stepped into the call to the UrlValidator.isValid function in our test, which took us to the isValid function in UrlValidator.java.

        - First, the isValid function parsed out the scheme part of the URL, using a watch on the scheme variable, we found that the scheme obtained from the function was `https`.
        - Then we stepped into the function call isValidScheme, a function that validates the scheme part of the URL and returns a boolean value.
        - isValidScheme returned true, that the scheme is valid, matching the expected behavior
            - Agan's Rule #4 "Divide and Conquer" - here, we're checking to make sure that each part of the code previous to the bug is behaving correctly.
        - Then the isValid function parses out the authority part as `google.com:1000`
        - We then step into isValidAuthority, a function that validates the authority part of the URL (including the hostname, IP address, and port).
        - Stepping through isValidAuthority, we found that it confirms that the authority part as a whole matches the expected authority pattern (AUTHORITY_PATTERN.matcher on line 372), which matches the expected behavior.
        - Then, isValidAuthority parses out the domain as `google.com`.
        - The domain part is valid according to DomainValidator.isValid on line 381, which matches the expected behavior.
        - Finally, the port number is parsed from the authority part as `:1000`.
        - The port is invalid according to the PORT_PATTERN.matcher(port).matches function on line 393, which makes the entire authority part invalid, so isValidAuthority returns false. This does not match the expected behavior, so this narrows down where our bug should be.
        - Then because isValidAuthority returns false, the entire URL is invalid according to isValid, and isValid returns false, indicating that the URL Validator program thinks that this is an invalid URL.
        - We then went to examine the declaration of the PORT_PATTERN object, which is the object that was showing error behavior, this object uses a regex definition to define a port pattern.
        - The regex definition referred to by PORT_PATTERN limits the length of a port to 3 on line 158; we believe that this is the location of our bug.
        - To verify that this is the cause of the bug, we changed the Regex definition to length of 5 (the expected behavior) and found that our previously failed test passes
            - Agan's Rule #9 "If You Didn't Fix It, It Ain't Fixed", Agan's Rule #7 "Check the Plug", & Agan's Rule #5 "Change One Thing at a Time" - by changing just a single character in the regex definition (3 to 5) we were able to see that the test would pass if we changed this one factor and verify that our test passing or failing hinged on this part of code.

- **Bug #2** - Local URLs are not validated correctly
    - **Bug Location -** Line 137 in DomainValidator.java
    - **Debugging Details -**
        - Using the debugger over the URL string "http://localhost", I proceed stepping through the primary function `isValid()`. Since I suspect the bug to be located in the authority parsing logic, I proceed to line 305 of `isValid()`. On this line, the function `isValidAuthority()` is invoked.

        - At this point, our authority is equal to `www.localhost`, so the function does not immediately return false as it would if the string was equal to null. I proceed to line 380 of `isValidAuthority()` to verify that the `DomainValidator` object is accepting local URLs.
        - When the `DomainValidator` checks that it is allowing local URLs, it returns true, which is expected behavior. After this, I step into the `isValid` function on the `DomainValidator` object, where I suspect the meat of the issue lies.
        - The value passed into this function continues to be `www.localhost`. The regular expression parser in this function teases out `localhost` to be a valid domain, and stores the string `localhost` in the groups variable.
        - At line 136, the isValid method on the DomainValidator objects determines the string to be neither `null`, or of length 0, and so in the following line it invokes the function `isValidTld` on the groups strings. This function asks whether the `allowLocal` variable is set to true (it is in this case), and then invokes `isValidLocalTld` on the string passed into the outer function.
        - This function converts the string to lowercase (it already is in our case), chomps the leading dot from the string (there is no leading dot in this case), and then tests whether or not that resulting string is present in a list called `LOCAL_TLD_LIST`. It then returns the boolean opposite value, as indicated by the presence of a `!` at the start of that line of code.
        - When I use the watch functionality on the evaluation without the ! operator, it evaluates to true, meaning that the string value `localhost` is present in the list `LOCAL_TLD_LIST`. When the `!`operator acts on it, it changes the evaluation to `false`, propagating that `false` value back to the test results.
    -
- **Bug #3** - IP addresses exceeding 255 are valid in dot-decimal notation
    - **Bug Location -** Line 96 in InetAddressValidator.java, Line 139 of DomainValidator
    - **Debugging Details -**
        - This bug is interesting in that if local URLs are allowed, the bug is introduced at line 139 of `DomainValidator.java`. In the event that local URLs are not allowed, the bug is introduced at line 96 of `InetAddressValidator.java` when it is invoked from `DomainValidator.java`. When local URLs are allowed, we begin by invoking `UrlValidator`'s `isValid()` method on `https://256.256.256.256`.
        - We know this URL is invalid because 256 exceeds 255. From UrlValidator's `isValid()` method, we progress through the function to line 305, where `isValid()` invokes `isValidAuthority()` on our authority (`256.256.256.256`). `isValidAuthority()` then evaluates the string up to line 381, where it then invokes the `DomainValidator`'s `isValid()` method on the authority string.
        - The `domainRegex` object's `match()` method finds no matches on the string, and evaluates to `null`. It then determines if `allowedLocal` is set to true(it is in this case), and runs `hostnameRegex`'s `isValid()` method on the domain string, where returns false, and then the opposite boolean value is evaluated by the if statement, and so the function returns true, which propagates back up into the test results.
        - When local URL's are not allowed, the `allowLocal` value on line 138 of `DomainValidator.java` is evaluated as `false`, and the function continues to

return `false`. Since the `DomainValidator`'s `isValid()` function returns false at line 381 of `UrlValidator.java`, the inverse of that statement evaluates to true, and the `InetAddressValidator` is allowed to run.

- The `InetAddressValidator` object runs the `isValidInet4Address()` function from the isValid function on the same object, and for each chunk of the four numbers, it tests whether that number is greater than 255 and returns true if it is (this is at line 96) and propagates that number back to the test results.
- This evaluation should be returning false, since 255 is exceeds 256, and is thus invalid. The function returns true, and returns that true value to the `UrlValidator` `isValid()` function which evaluates the inverse of that value which is false. It then does not return the false value within that logic branch.

- **Bug #4** - Some country code top-level domains result in invalid urls
  - **Bug Location -** Line 248 (beginning of COUNTRY_CODE_TLDS array) in DomainValidator.java
  - **Debugging Details -**
    - We began the debugging process by creating a breakpoint at the failed test; the test input we used for debugging was `https://www.google.jp` (The country domain code for Japan), which we expected to be a valid URL, but for which the URL Validator returns invalid.
    - Then we started the test in debug mode; debug mode allowed execution to stop at our breakpoint.
    - We stepped into the call to the UrlValidator.isValid function in our test, which took us to the isValid function in UrlValidator.java.
      - Agan's Rule #1 "Understand the System" & Agan's Rule #3 "Quit Thinking and Look" - The first thing we do step through each part of the code so we can make sure that we know how the UrlValidator works, what it is doing with the test cases it's being given and to understand and pinpoint which part of the code is failing.
    - First, the isValid function parsed out the scheme part of the URL, using a watch on the scheme variable, we found that the scheme obtained from the function was `https`.
    - Then we stepped into the function call isValidScheme, a function that validates the scheme part of the URL and returns a boolean value.
    - isValidScheme returned true, that the scheme is valid, matching the expected behavior
      - Agan's Rule #4 "Divide and Conquer" - here, we're checking to make sure that each part of the code previous to the bug is behaving correctly.
    - Then the isValid function parses out the authority part of the URL as `www.google.jp`
    - We then step into isValidAuthority, a function that validates the authority part of the URL (including the hostname, IP address, and port).
    - Stepping through isValidAuthority, we found that it confirms that the authority part as a whole matches the expected authority pattern (AUTHORITY_PATTERN.matcher on line 372), which matches the expected behavior.
    - Then, isValidAuthority parses out the domain as `www.google.jp` (the same as the authority).

- We then step into DomainValidator.isValid function on line 135 of DomainValidator.java. This function parses out the domain "group" using the domainRegex.match function. This function parses out the `jp` part of the domain (the top-level domain) and stores it in an array called "groups".
- The DomainValidator.isValid function then verifies that the top level domain is valid by calling the function isValidTld. The isValidTld function makes three different function calls, if any one of them returns true, it will return that the top-level domain group is valid.
    - The first function call is to isValidInfrastructureTld which checks if `jp` is in a list of infrastructure top-level domains; it isn't, so the function returns invalid.
    - The second function call is to isValidGenericTld, which checks if `jp` is in a list of generic top-level domains; it isn't, so the function returns invalid.
    - The third function call is to isValidCountryCodeTld, which checks if `jp` is in a list of country-assigned top-level domains, called COUNTRY_CODE_TLDS. We expect that the country code `jp` should be in this list. However, the function does not find `jp` in the list of country codes, so it returns false.
- All three function calls return false, so the top-level domain part of the domain is invalid, which invalidates the authority part of the URL, which invalidates the entire URL, so URL Validator returns false. This does not match the expected behavior, so the test fails.
    - Agan's Rule #4 "Divide and Conquer" - here, we checked the behavior of each function call separately because their return values are all a part of the same truth statement. This way we made sure we could narrow down which function call was causing the bug.
- We then went to examine the declaration of the COUNTRY_CODE_TLDS array, which is where we expected to find the country code `jp`. We found that this list is in alphabetical order, and contains all the country codes for countries beginning with the letters A through I, however, the list terminated where country codes starting with "J" would have began [3].
    - Agan's Rule #8 "Get a Fresh View" - in this case, we consulted with IANA, the coordinating body for top-level domain country codes, to make sure that what we suspected was missing from Url Validator's country code list was actually missing.
- To verify that this is the cause of the bug, we added the country code "jp" to the COUNTRY_CODES_TLDS array and found that our previously failed test passes.
    - Agan's Rule #9 "If You Didn't Fix It, It Ain't Fixed", Agan's Rule #7 "Check the Plug", & Agan's Rule #5 "Change One Thing at a Time" - by adding just a single entry in the COUNTRY_CODES_TLDS array, we were able to see that the test would pass if we changed this one factor and verify that our test passing or failing hinged on this part of code.

- **Bug #5** - Urls containing queries are always invalid
    - **Bug Location -** Line 446 in URLValidator.java

[3] "IANA root zone database". Iana.org. Retrieved 2015-11-10.

- **Debugging Details -**
  - We began the debugging process by creating a breakpoint at the failed test; the test input we used for debugging was `https://www.google.com?key=value`, which we expected to be a valid URL, but for which the URL Validator returns invalid.
  - Then we started the test in debug mode; debug mode allowed execution to stop at our breakpoint.
  - We stepped into the call to the UrlValidator.isValid function in our test, which took us to the isValid function in UrlValidator.java.
    - Agan's Rule #1 "Understand the System" & Agan's Rule #3 "Quit Thinking and Look" - The first thing we do step through each part of the code so we can make sure that we know how the UrlValidator works, what it is doing with the test cases it's being given and to understand and pinpoint which part of the code is failing.
  - First, the isValid function parsed out the scheme part of the URL, using a watch on the scheme variable, we found that the scheme obtained from the function was `https`.
  - Then we stepped into the function call isValidScheme, a function that validates the scheme part of the URL and returns a boolean value.
  - isValidScheme returned true, that the scheme is valid, matching the expected behavior
    - Agan's Rule #4 "Divide and Conquer" - here, we're checking to make sure that each part of the code previous to the bug is behaving correctly.
  - Then, the isValid function parses out the authority part as `www.google.com`
  - We then step into isValidAuthority, a function that validates the authority part of the URL (including the hostname, IP address, and port).
  - The isValidAuthority returned true, that the authority is valid, matching the expected behavior.
  - Then, the isValid function parses out the path part as an empty string.
  - We then step into isValidPath, a function that validates the path part of the URL. This function returns true, that the path is valid, which matches the expected behavior.
  - Then, the isValid function parses out the query part as `key=value`
  - We then step into isValidQuery, a function that validates the path part of the URL.
  - The query is invalid according to the QUERY_PATTERN.matcher(query).matches() function on line 446, which makes the entire query part invalid, so isValidQuery returns false.
  - Then because isValidQuery returns false, the entire URL is invalid according to isValid, and isValid returns false, indicating that the URL Validator program thinks that this is an invalid URL. This does not match the expected behavior.
  - We then went to examine the declaration of the QUERY_PATTERN object, which is the object that was showing error behavior, this object uses a regex definition to define a port pattern.
  - The regex definition referred to by QUERY_PATTERN accepts any number of any characters (all queries match the pattern), which should match our query without issue. However, we see on line 446 that when the QUERY_PATTERN.matcher(query).matches() function returns to the isValidQuery function, isValidQuery immediately returns the *opposite* truth result (using the "not" operator, "!") to isValid, invalidating valid queries when it does so.

- To verify that this is the cause of the bug, we removed the not operator from the return statement on line 446 and found that our previously failed test now passes.
    - Agan's Rule #9 "If You Didn't Fix It, It Ain't Fixed", Agan's Rule #7 "Check the Plug", & Agan's Rule #5 "Change One Thing at a Time" - by removing a single operator in the return statement on line 446 we were able to see that the test would pass if we changed this one factor and verify that our test passing or failing hinged on this part of code.

Team Work -

- **Working as a Team**
  Each week we scheduled a meeting time through email or at the end of a meeting, and then met at that time in a Google Hangouts chat room. In our meetings, each team member would take turns driving, while the remaining members co-piloted or researched additional information regarding our test cases. We used GitHub to manage and share source code.

- **Division of Labor**
  Work was divided in an ad hoc manner. We used a pair programming method, where each team member piloted at least once, and we each took initiative to expand tests outside of our regularly scheduled meeting times.

- **Collaboration Methods**
  We collaborated using Google Docs for our reports, Google Hangouts for meetings, and shared source code through GitHub.