CS362_400_S17
Final Project - Part A
5/18/2017

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

1. ***Total number of URLs*** `testIsValid()` ***is testing***
   Number of URLs: 31,920

   We estimated that the the total number of URLs the function will be testing is 35,910, which was obtained by calculating the total number of combinations of scheme, authority, port, path, and query that can be constructed for the testUrlParts object. The following equation was used to estimate number of urls:

$$\#URL = \#scheme \times \#Authority \times \#Port \times \#Path \times \#Query = 9 * 19 * 7 * 10 * 3 = 35,910$$

   However, when we actually ran the test with a counter, the number of times the do...while loop ran (i.e. the number of URLs that were constructed) came out to 31,920. When running the test, we also modified the code to print out all tested URLs (not just the valid ones), and after examining the constructed URLs, we believe this is because the test is not actually iterating through the last `ResultPair ("", true)` in the `testUrlScheme` array.

2. ***Explanation of how*** `testIsValid()` ***constructs URLs***
   For each URL construction loop, the `testIsValid()` function creates an array of `ResultPair` objects called `part`. This `part` array is populated with `ResultPair` objects from the `testUrlXXX` array members of `UrlValidatorTest` class. A `ResultPair` constitutes a specific example segment of a URL and contains two members: a URL component as a string literal and a corresponding boolean value that says whether or not that string literal invalidates the URL. Each string literal is then appended to a string called `testBuffer`. The number of `ResultPairs` that are appended to each `testBuffer` is equal the length of the `testPartsIndex` array. The `testPartsIndex` array is an array of indices, each of which represents the current index on each array that stores a component of the supposed ("so-called") URL. The indices in the `testPartsIndex` array are incremented by a function called `incrementTestPartsIndex` at the end of each construction loop.

   The components that are used in this test handler are provided by an argument called `testObjects` in the `testIsValid` function. Each component represents part of a URL: the scheme, the URL authority, the URL port, the path, (optionally) the path options, and URL query. In UrlValidatorTest.java, the testIsValid function is only ever called with the

`testUrlParts` array; the invocation of the function with `testUrlPartsOptions` as an argument is commented out in this particular file.

When the testable URL has been created in the 'testBuffer' string, a `UrlValidator` object instantiated by `testIsValid` calls a validation algorithm with 'testBuffer' as the argument (this is the algorithm being tested). The expected return value is derived by examining each corresponding boolean value for each `ResultPair` URL component. If even one URL component has a corresponding `false` boolean value, it should invalidate the entire URL, and `testIsValid` will expect the `UrlValidator.isValid` algorithm to return a 'false'.

3. ***Example of valid URL being tested and an invalid URL being tested by* `testIsValid()`**
    - *Valid:* "h3t://go.au:65535/$23?action=view"
        i. All components have a corresponding boolean value that is true, thus the URL is a valid URL.
    - *Invalid:* "http://1.2.3.4.5:80/../?action=edit&more=up"
        i. The component "/../" has a corresponding boolean value of false, thus the entire URL is an invalid URL.

4. ***Do you think that a real world test (URL Validator's* `testIsValid()` *test in this case) is very different than the unit tests that we wrote (in terms of concepts & complexity)?***

No, comparing the real world test to our unit tests, we did not notice anything significantly different between the two tests. For both sets of tests, the programs are only testing a discrete number of scenarios, with the hope of covering all edge cases, instead of testing every possible scenario or URL. For example, our Dominion unit tests generated several game states that attempted to cover all possible scenarios, such as no cards in deck. Similarly, URL Validator constructs URLs by combining a limited number of scheme, authority, port, path, and query values that are hard-coded into the program, as opposed to generating all possible random URLs (infinite) and testing if they are URLs. However, when comparing our unit tests and URL Validator's tests, we did notice that the validator's unit testing was more complex than ours. In general, our unit tests focused on a single domain for each argument, whereas URL Validator deconstructed each of its domains into individual components and then tests the combinations of those parts.