OSU CS 362, Fall 2018
Final Project Part A
Christopher Piemonte, OSU ID: 933283391
Christopher Seay, OSU ID: 933014621

Part-A : Review of Existing Test Case(s) (**75 points)** For Part- A, you will be provided a correctly working version of URL Validator. You need to answer the following questions:

- Explain testIsValid function of UrlValidator test code. It is available under **URLValidatorCorrect** folder. (20 points)
    - From main, testisValid() is called with no parameters, which in turn calls testisValid with an array containing sub arrays of URL bits and a long that determines which schemes are allowed. They are combined and checked for validity. See our answer below for a more detailed explanation of how the URL is built and how the URL is validated. If any part of the URL is invalid (indicated by the boolean associated with each URL part), an “X” is printed. This is repeated for every possible permutation.

- Give how many total number of the URLs it is testing. Also, explain how it is building all the URLs. (20 points)

  https://math.stackexchange.com/questions/1641793/calculating-number-of-combinations-of-multiple-sets-each-containing-different-n

  8x20x9x10x15 = 216,000 possible combinations

  Its is building out possible urls by cycling through six permutations to build the four parts of a url in the form of <scheme>://<authority><path>?<query>. Only five permutations are used to fill the four parts, but a sixth permutation is present for fragments. All used permutations must be valid for the url to be considered valid. The six permutations are:
  1. **testUrlScheme** - assigns a scheme to random testUrls. These schemes configure options such as whether two slashes are valid or url fragments are allowed. Some of the applied are valid (http://), while others are not (3ht://).

2. **testUrlAuthority** - this provides the hostname or IP address. The authority is actually comprised of both the hostname and the port, but the port is configured in the next permutation. Both must be valid for the authority to be valid and the authority cannot be null. Some of the applied are valid (www.google.com), while others are not (1.2.3.4.).
3. **testUrlPort** - provides the port number, which can be empty in the form of an empty string, unlike the hostname. Some of the applied are valid (:80), while others are not (:-1).
4. **testPath** - configures the path to the desired location. The path can be empty in the form of an empty string, but cannot be null. Some of the applied are valid (/test1), while others are not (/../).
5. **testUrlPathOptions** - configures a fragment path if fragments are allowed. This is not used when constructing testUrlParts, only testUrlPartsOptions.
6. **testUrlQuery** - configures a possible query, which can be null. There are no bad queries generated in this test.

- Give an example of valid URL being tested and an invalid URL being tested by testIsValid() method. (20 points)
  Valid
  “ ”
  Scheme + authority + port + path + query
  http:// + "www.google.com" + :80 + /test1 + ?action=view

  Invalid (only a single element needs to be invalid for the URL to be invalid):
  Bad scheme example:
  3ht:// + "www.google.com" + :80 + /test1 + ?action=view
  Bad authority example:
  http:// + go.a1a + :80 + /test1 + ?action=view
  Bad Port example:
  http:// + "www.google.com" + :-1 + /test1 + ?action=view
  Bad path example:
  http:// + "www.google.com" + :80 + /../ + ?action=view
  No Bad query examples

- UrlValidator code is a direct copy paste of apache commons URL validator code. The testfile/code is also direct copy paste of apache commons test code. 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)? Explain in few lines. (15 points)

- No, they operate on very similar principles and practices. Identify what you want to test, identify the pass and fail criteria, then execute the code and compare it against the known outcome paths. Another reason this code is similar to real world is it's adaptability and "re-usability". If the specific use case of the end user had a very detailed requirement specification it wouldn't require much modification other than adding those known positive and negative outcomes, all with minimal code refactoring. In my (Chris S) current role as R&D Software Intern I occasionally have to write unit tests and test suites for various code, and while the code tested is very different, the overall principles are very similar. Try and limit the test to one item at a time, so you can identify what broke, test known failures, corner cases, and then some positive outcomes.

Team Work: (**25 points**)

We divided up the work evenly between setting up various tools and resources to answering and refining different questions. Chris Seay handled setting up the canvas group and creating a google doc to collaborate on while Chris Piemonte setup a slack workspace to make communicating easier with our time zone differences. Chris S. handled Q3 and a rudimentary answer to Q2. Chris P expanded upon, refined, and generally improved Q2 to the form we are turning in. Chris S converted the google doc into a PDF and turned it in.