Part A: Review of Existing Test Case(es)

I. UrlValidator - Explanation of testIsValid() function - Figure 1

A. Function Explanation

1. Function signature:

```
public void testIsValid(Object[] testObjects, long options)
```

2. Parameters:
   a) **`Object[] testObjects`**: An array of URL parts, each of which contains an array of the available options for each URL part.
   b) **`long options`**: (long type) Options that can be 'turned on' during testing, eg, 'ALLOW_2_SLASHES' allows two slashes in the path component of the URL.
3. Description (Note that all printed output assumes that 'printStatus' and 'printIndex' are set to 'true'.)
   a) First, the testIsValid function creates a new instance of the UrlValidator class (Figure 1, line 86).
   b) Next, the function tests two URLs that are known to be correct are tested and assert 'True' (Figure 1, lines 87 and 88).
   c) Within a do-while loop, the actual URL testing begins:
      (1) A 'for' loop is used to loop through the five parts of the URL, building various test cases based on the five different arrays for URL parts (Scheme, Authority, Port, Path, and Query). Figure 1, lines 97 - 102.
      (2) The test cases built in (1) are tested for validity and the URL is printed out (Figure 1, 105 - 108).
      (3) The results are then compared to the expected result, to assert that the test result and expected result are equal (line 109).
      (4) If the results are as expected, the function prints '.'. If results are not as expected, the function prints 'x' (Figure 1, lines 114- 120)
   d) The testIsValid() function will continues to loop until all possible URL combinations are tested, then exits.

B. Total number of URLs tested

1. Number of URL schemes: 9
2. Authorities: 19
3. Ports: 7
4. Paths: 10

5. Queries: 3
6. All possible combinations of these five categories = 9 * 19 * 7 * 10 * 3 = **35,910 URLs**.

```
85    public void testIsValid(Object[] testObjects, long options) {
86       UrlValidator urlVal = new UrlValidator(null, null, options);
87       assertTrue(urlVal.isValid("http://www.google.com"));
88       assertTrue(urlVal.isValid("http://www.google.com/"));
89       int statusPerLine = 60;
90       int printed = 0;
91       if (printIndex)  {
92          statusPerLine = 6;
93       }
94       do {
95          StringBuffer testBuffer = new StringBuffer();
96          boolean expected = true;
97          for (int testPartsIndexIndex = 0; testPartsIndexIndex < testPartsIndex.length; ++testPartsIndexIndex) {
98             int index = testPartsIndex[testPartsIndexIndex];
99             ResultPair[] part = (ResultPair[]) testObjects[testPartsIndexIndex];
100            testBuffer.append(part[index].item);
101            expected &= part[index].valid;
102         }
103         //System.out.println(testPartsIndex[0]);
104         String url = testBuffer.toString();
105         boolean result = urlVal.isValid(url);
106
107         if(result == true)
108             System.out.println(url);
109         assertEquals(url, expected, result);
110
111         if (printStatus) {
112            if (printIndex) {
113               //System.out.print(testPartsIndextoString());
114            } else {
115               if (result == expected) {
116                  System.out.print('.');
117               } else {
118                  System.out.print('X');
119               }
120            }
121            printed++;
122            if (printed == statusPerLine) {
123               System.out.println();
124               printed = 0;
125            }
126         }
127      } while (incrementTestPartsIndex(testPartsIndex, testObjects));
128      if (printStatus) {
129         System.out.println();
130      }
131   }
```

Figure 1. testIsValid() function in file 'UrlValidatorTest.java'.

II. Explain how the *testIsValid()* function is building all the URLs

The testIsValid() function builds the URLs by taking the array of the URL and going through all the permutations of the different combination of the array components. According to the comments on the URLValidatorTest.java file, there are 5 parts to the URL as shown below.

```
/**
* The data given below approximates the 4 parts of a URL
* <scheme>://<authority><path>?<query> except that the port number
* is broken out of authority to increase the number of permutations.
* A complete URL is composed of a scheme+authority+port+path+query,
* all of which must be individually valid for the entire URL to be considered
* valid.
*/
```

The function uses 9 schemes, 19 authority, 7 port, 9 paths with a single slash, 15 paths with a double slash, and 3 query URL parts. All these are value,boolean pairs (for example: *new ResultPair("h3t://", true)* or *new ResultPair("3ht://", false)*. And in the combination tested, if any of the URL parts are false, then the URL is invalid.

A valid url:

http://www.google.com:80/test1*?action=view*

An invalid url:

http://www.google.com:65a/test1?action=view

(the underlined portion is false)

III. URL Validator's testIsValid() function vs in-class Unit Tests

When comparing and contrasting URL Validator's testIsValid (real world test), with the unit tests that we have written so far in this class, a few similarities and differences can be observed. Overall, the test's characteristics appear very similar to the unit tests we have written. Nonetheless a few items are worthy of explicit mention. The most glaring similarity is the structure. testIsValid builds and implements unit testing in files separate from the files implementing the underlying class and objects being teste. This format is desirable and similar to the unit tests that we have written. Structuring unit tests in this fashion facilitates proper modular design, thus allowing additional unit tests to be added easily, and existing unit tests to be quickly adjusted to achieve better coverage, or simply to improve their testing approach.

Tests executed by testIsValid produce significant text output, allowing the user to use this output during their debugging efforts, or simply to confirm that tests executed without any bugs identified. The author of testIsValid provided two main toggles (booleans), which allow the user of testIsValid to manipulate the quantity and type of output they see. Specifically, booleans printStatus and printIndex can be changed to 'true' to vary the test output. printStatus provides a visual indicator of the test result, while printIndex generates output specific the indices of the various array options input during the test. This provides the user with different, but helpful, information to trace and follow test results. Printing information is critical during testing. Our unit tests to date have focused

on providing clear output messaging. The information printed must be clear and sufficiently detailed, so the tester is able to quickly identify what portion of the test failed, and what inputs generated that failure. Insufficient output is barely more valuable than no output. If test outputs simply state, “test failed”, the tester would know a bug might exist, but they would have no further recourse to investigate and isolate the issue. testIsValid provides ways in which the user can see valuable output, and manipulate the type of output to their preference. However, we believe the output generated could be strengthened. During a failed test, testIsValid prints ‘x’, a passed test outputs ‘.’, but it is not always abundantly clear what test the character printed correlates. Nor is it clear what portion of a test is causing a failure. Furthermore, the output is blocked on consecutive lines, implementing the use of more white space, could be beneficial. Adding this detail would improve the readability of the tests, which is in line with what we have discussed is best practice, not just in theoretical testing, but also in ‘real world’ testing.

As part 1 mentions, testIsValid produces a high volume of tests by testing every permutation of the input parameters given. This is a good practice, as generally more testing correlates to higher test coverage, however, higher test coverage does not guarantee better testing. To achieve better testing, and at the same time increase coverage, the input parameters should be interesting and logical. They should test for something specific. Much like partition and boundary testing in our own unit tests, we would expect the inputs to vary. It would be reasonable to include all commonly used url schemes (http://, https://, ftp://), a valid partition. It would also be reasonable to include a bucket of bad url schemes (fake://, etc). This would capture an invalid partition. Finally, it would be appropriate to then include in the testing some boundary cases. These would likely be slight variations on valid url schemes, which are likely to occur due to typos or bad memory recollections (http:\\, http//, http:/). testIsValid includes most of these categories of inputs, as we would expect from our unit testing experience thus far.