URLValidator Testing

Final Project

Bradley Mader

Manual Testing:

Tested by making the following manual calls:

urlVal.isValid(“http://www.google.com”); //PASSED

urlVal.isValid(“www.google.com”); //FAILED

urlVal.isValid(“http://www.google.com:80”); //PASSED

urlVal.isValid(“http://www.google.com/test/test”); //PASSED

urlVal.isValid(“http://www.google.com?pid=10”); //FAILED

Input Partitioning:

Using information acquired by manual testing, the ‘http://www.google.com’ was used as the base to individually test each aspect of the URL independently. In the first partition, various inputs are used to test the authority/protocol portion of the URL. This is done by testing a number of different protocols in place of ‘http://’. These inputs included both valid values (‘ftp://’, ‘h3t://’) and invalid values (none, ‘3ht’, ‘http:/’, ‘http:’, ‘://’).

The second partition tested the host utilized by switching ‘www.google.com’ out for various alternative inputs. Alternative inputs were again structured to be both valid values (‘hulahoop.edu’, ‘www.gusto.com’, ’0.0.0.0’, ‘255.255.255.255’) and invalid values (‘256.256.256.256’, ‘ew_fd.au’, ‘fs:.ru’, ‘Puffle&w.com’). The ‘htttp://’ protocol is used in all of these cases as it has been proven to work, and so the only change in each of these individual cases is the host address.

Similar partitions are included for ports, paths, and queries. In each of these cases, the base of 'http://www.google.com' is used as it has been proven to work by itself. In each test case of these partitions a query, path, or port identity is concatenated to the existing address in order to test validity.

Program Based Testing:

Files for testing isValid() are included in maderb/URLValidator/ . The code contains all unit test followed by random testing which outputs whether to console only if the results are not valid. Tests are contained within the testIsValid function, which in turn calls testYourFirstPartition( ) and testYourSecondPartition( ) as well as testAnyOtherUnitTest( ) in order to execute all unit tests. Random testing is located within the testIsValid( ) function.

Bug Reports:

Bug #1: Missing protocol not recognized as valid

Error occurs on isValid line 296; should not pass into 'if' statement. isValidScheme( ) returns false when it should return true. Incorrect value returned on line 336.

Bug #2: '256.256.256.256' recognizing as valid despite invalid IP address.

Error occurs on line 324 when execution reaches the end of isValid( ) without returning false.

Bug #3: ':65535' not recognized as valid port despite being within valid range.

Fails in isValid( ) function at line 305. Call to isValidAuthority( ) returns True at line 394 when it should return False. Failure causes isValid( ) to return False at line 306.

Bug #4: ‘/#’ and ‘/#/file’ not recognized as invalid paths.

Returns at line 324 when execution reaches end of function. Should likely return false at logic switch on line 310.

Bug #5: All queries return invalid input

Returns at line 315. Calls isValidQuery( ) on line 314 in if-statement returning false, which in turn causes isValid( ) to return false at line 315.

Application of Agan’s Rules:

Several of Agan’s Rules were utilized to debug this system:

Understanding the system is very important in order to determine both where breaks are likely to occur (which helps with input partitioning) and because it is necessary to know what the system is expected to return in order to determine when these values are not accurate.

The second of Agan’s Rules (Make It Fail) is essential, as observing the fail state is essential to diagnose where breaks occur. Observing the break allowed me to utilize Eclipse, in this case, to determine when the incorrect path was followed, and at what point to incorrect values came to be.

‘Quit thinking and look’ can sometimes be difficult, as it is commonly the case that we know where the fault should occur and test only those areas, when that is many times not the case; when testing, I was sure to identify a fail case, and then walk through the steps that led to that fail case step-by-step until the source of the fault was observed, rather than only testing specific areas.

‘Divide and Conquer’ is important, because there just is not time to walk through every step of code executed, however, if you narrow it down to a single function that is returning the incorrect end result, it is possible to then break that function down and observe how it is perpetuating the error.

The subsequent steps of Agan's Rules apply more strongly to fixing the errors following testing than testing, although the final rule, 'Question Assumptions…" is always a greatly applicable concept when testing. It is important to test all values, as errors may come from an unexpected source.

Group Work:

Not really applicable in this case, I had overlooked the group forming aspect of this assignment until after the deadline, and as a result I ended up working by myself on this one. On the positive side, there was no difficulty with correspondence and I am very happy with all of my team members' contributions (although I'm sure I would have been anyway).

EXTRA CREDIT: TARANTULA Used on Dominion

The first mechanism required is a random test schema, by which the program determines whether a set of random inputs generates a valid or invalid output. Then, for each iteration through the random tester, GCOV is run and for each line either the valid or invalid counter is incremented for that line to indicate that it was part of either a successful or failed run through the program.

Tarantula then works by, after a large number of executions, determining which statements/branches/methods have the highest ratio of failed executions and this is most likely the statement/branch/method in which the fault occurs.