Final Project Report

Bradley Huffman, Caleigh Runge-Hottman, Jesse Hanson, Jonathan Nicolosi

The Buggy URL Validator did not have any test coverage when we began, so at the very least, we found it important to generate some amount of code coverage. We aimed for a goal of achieving 80% coverage in hopes of executing the majority of the code base. We planned to achieve this by writing tests for each of the classes in the URLValidator. We will also test various input types for the classes against correct and incorrect types and make sure they pass or fail accordingly. Moreover, we will report on any false-positives as well on ways we might resolve incorrect failures.

Techniques and code snippets:

For the implementation of the improvements outlined in our Test Plan document, we used JUnit as our testing framework. Moreover, we chose to make use of Cobertura to analyze our tests' code coverage. To obtain the coverage, we used functionality based input domain partitioning.

Cobertura Coverage Results:

Test Suite	Line Coverage	Branch Coverage
DomainValidatorTest	92%	75%
InetAddressValidatorTest	84%	80%
RegexValidatorTest	80%	76%
ResultsPairTest	100%	N/A
UrlValidatorTest	80%	55%
Overall	82%	65%

Table 1. Cobertura line and branch coverage for all five test suites.
Source: http://web.engr.oregonstate.edu/~nicolosj/report/cobertura/

DomainValidator:
We tested various valid and invalid country codes, generic top-level domains, infrastructure top-level domains, and local top-level domains via functionality based input domain partitioning (IDP) to ensure the valid codes and top-level domains return 'true', and the invalid codes and top-level domains return 'false'.

An example of our test code for the DomainValidator function *isValidGenericTld* which checks the validity of a given domain can be seen in the snippet below. This test exhibits our use of IDP by demonstrating how we partitioned the input domain into two different sections: 1) an

accepted input that will return valid, and 2) an accepted input that will return invalid. Since the given function checks the input string to see if it is within the list of accepted domains, we simply provide a domain that is accepted and one that is not in the list.

```
@Test
public void testGTld() {
   // Test GenericTld
   String validgeneric = "com";
   assertEquals("valid generic", true, domainval.isValidGenericTld(validgeneric));
   String invalidgeneric = "cats";
   assertEquals("invalid generic", false,
domainval.isValidGenericTld(invalidgeneric));
}
```

InetAddress Validator:

We confirmed that the function correctly identifies IPv4 addresses and that it accurately verifies that address subgroups are legal. Again, we used IDP to test both an accepted input that is a valid IPv4 address and an accepted input that is not a valid IPv4 address, as seen in the snippet below.

```
@Test
public void isValidInet4AddressTest() {
       assertTrue("68.199.224.252 is a real IP. This should return true.",
IAV.isValidInet4Address("68.199.224.252"));
       assertTrue("999.999.999.999 is not a real IP. This should return error.",
IAV.isValidInet4Address("999.999.999.999"))
}
```

UrlValidator:

For the url validator, we ensured that the returns of the url validator and testing if they return true for proper urls and false for bad urls and what inputs on the url might cause it to pass or fail incorrectly. We also ensured that null inputs returned the appropriate results.

An example of testing valid and invalid URLs can be seen below. Again, note the use of IDP similar to the tests previously discussed.

```
@Test
public void exampleUrlTest() {
  UrlValidator urlvali = new UrlValidator();
  String abc = null;
  String goog = "http://google.com/";
  assertTrue("google is a valid url", urlvali.isValid(goog));
  assertFalse("null isn't a valid url", urlvali.isValid(abc));
}
```

RegexValidator:

We ensured that the RegexValidator function throws exceptions when appropriate. We confirmed that caseSensitive flag was being set properly.

An example of testing a throwing an exception:

```
@Test(expected=IllegalArgumentException.class)
public void illegalArgTest() {
   String empty = "";
   RegexValidator noCaseReg = new RegexValidator(empty, false);
}
```

ResultPair:

To test ResultPair, we used pass fail tests to determine if the ResultPair function correctly matches the input to its respective part in the url.

Example:

```
@Test
public void ResultPairTest(){
   ResultPair RP = new ResultPair("item", true);
   assertTrue(RP.item.toString() == "item");
   assertTrue(RP.valid);
}
```

Bug reports

For the InetAddressValidator class, we designed a test to determine if the isValidInet4Address would return false if it was given a valid regular expression but an invalid IP (e.g “999.999.999.999”). We noticed that the code would return true even if the IP subgroup was out of range. The PIT test further illustrated the flaw in the design of the method by showing us that the negated conditional check for the validity of the subgroup would fail for any number it checked. This failure drove us to design a test to ensure that isValidInet4Address would properly throw a NumberFormatException (e.g “abc.def.ghi.jkl”) and indeed it did.

DomainValidatorTest contained bugs within the function that evaluates Top Level Domains. When a new DomainValidator is instantiated with a “true” argument, the variable “allowlocal” is set to true and therefore the function isValidLocalTld and isValidTld should return true. However, it appears that although the string “localhost” resides in the String[] LOCAL_TLDS, isValidLocalTld indicates that the string “localhost” is not contained within LOCAL_TLDS due to the “!” operator in the return statement. This causes isValidLocalTld and isValidTld to evaluate false where they should evaluate true.

New tool: PIT Mutation Testing

Description

The new tool our group used for this project was PIT mutation testing. Mutation testing is a form of testing that involves incorporating mutations into one's source code, and then running the corresponding tests for said source code to see if they fail as a result of these mutations. The quality of the tests can then be determined by analyzing how many of the mutations cause test cases to fail, since ideally no mutations should survive the tests. We will run each of the classes against 100 mutation tests and report on the results. The report will reflect on the tests that had the most failed mutations and why this might have occurred.

Comparison

The PIT tests helped to reveal a lot of potential pitfalls lurking within the use of conditional operators. One particularly enlightening mutation test was the NEGATE_CONDITIONALS_MUTATOR. This mutator allowed us to observe whether or not a branch was taken irrespective of the conditional operator being evaluated. Overall, the mutator was successful at revealing bugs. However, there were some less useful indications. For example, mutated greater-than/less-than signs in for-loops were flagged; however, these indications have little bearing on the overall integrity of the program. The purpose of this exercise was to discover if our mutation test suite could detect and kill the changes produced by the PIT test with respect to the assertions we've made about if-then statements and return values.

Conclusively, the PIT tests generated 83% line coverage and 55% mutation coverage. This means 55 of our 100 mutations passed the tests. Moreover, the test results show that the majority of the failed mutations came from flipping the expected input in the True/False assertions. To avoid this, more tests could be written that use raw input or check against Null or expected test values rather than the return values themselves. Applying different conditionals on our mutation tests, we received different results as shown in the table below.

Active Mutators	Mutation Coverage	Classes Covered
REMOVE_CONDITIONALS_EQUAL_ELSE_MUTATOR REMOVE_CONDITIONALS_ORDER_ELSE_MUTATOR REMOVE_CONDITIONALS_ORDER_IF_MUTATOR REMOVE_CONDITIONALS_EQUAL_IF_MUTATOR	55% out of 154 tests	4/4
CONDITIONALS_BOUNDARY_MUTATOR	67% out of 18 tests	2/4
CONSTRUCTOR_CALL_MUTATOR	56% out of 9 tests	3/4

Table 2. Mutation and class coverage for active mutators.
Source: http://web.engr.oregonstate.edu/~nicolosj/report/cobertura/