Assignment 2
Due: Friday 11:59pm on Oct 11 (Extended to Oct 15), 2024

Group Members:

Member #1:

First Name: Eric
Last Name: Morse
Student ID: 1141504

Member #2:

First Name: Or
Last Name: Brener
Student ID: 1140102

Member #3:

First Name: Danindu
Last Name: Marasinghe
Student ID: 1093791

1. Chapter 3, Question 6 in the Textbook.
    a. A test that does not reach the fault [2 Marks]

Test the execution of computePrimes with n = 0.

    b. A test that reaches the fault, but does not infect [2 Marks]

Test the execution of computePrimes with n = 4.

    c. A test that infects the state, but does not propagate [2 Marks]

This is not possible, because when n is greater than or equal to 8, the test will always infect, and by doing so, the fault will always propagate to cause the final array (primes) to be different from the expected array.

    d. A test that propagates, but does not reveal [2 Marks]

Test the execution of computePrimes with n = 8, where we only assert there are no exceptions being thrown.

    e. A test that reveals the fault [2 Marks]

Test the execution of computePrimes with n=8, where we assert with the expected results (2,3,5,7,11,13,17,19)

2. Chapter 3, Question 7 in the Textbook.

Save your zipped source code as: **lastname_firstname_Assignment_2_Q2_Source_Code**

a. What is the first false positive? [1 mark]

The first false positive is when n = 9, because 9 is not a prime number but it is included in the output.

b. How many “primes" must a test case generate before encountering it? [2 marks]

The test case must generate 5 “primes” before encountering the first false positive. The first 4 are correct prime numbers (2, 3, 5, 7), and 9 is the 5th.

c. What does this exercise show about the RIPR model? [2 marks]

This exercise shows that we need to be very aware of faults that may not propagate or may not reveal themselves when testing. When we are creating a testing suite we need to optimize the level of revealability while also keeping in mind tests that might not be propagated or reached. In this exercise our fault is only reached on the 9th input for a false positive. So we must make sure to not only test for false negatives, and “happy paths” but also make sure to test for false positives and make sure that we test deep enough to actually reach and reveal the fault.

3. Chapter 3, Question 8 in the Textbook. [5 marks]

a. Create at least 10 test cases that include examples of passes, errors, and failures. Make sure to provide comments in your code explaining the purpose and expected outcome of each test. Your comments should clearly describe whether the test is expected to pass, produce an error, or fail, and why.
b. Additionally, capture and paste screenshots of your code and the test results below. These screenshots should clearly show your code, the executed tests, and their respective outcomes. Your goal is to demonstrate thorough testing and an understanding of the results for each test case.
c. Finally, save your source code in a zipped folder named: **lastname_firstname_Assignment_2_Q3_Source_Code.zip**

```java
/**
 * This test shows the method returns the right element when there's only one,
 * it should pass
 */
List<String> list1 = new ArrayList<>();
String result1;
list1.add("dog");
result1 = "dog";

/**
 * This test shows the method returns the smallest string, it should pass.
 */
List<String> list2 = new ArrayList<>();
String result2;
list2.add("cat");
list2.add("dog");
result2 = "cat";

/**
 * This test shows the method returns the right int when there's only one, it
 * should pass.
 */
List<Integer> list3 = new ArrayList<>();
Integer result3;
list3.add(2);
result3 = 2;

/**
 * This test shows the method returns the right int when theres more then one,
 * it should pass.
 */
List<Integer> list4 = new ArrayList<>();
Integer result4;
list4.add(1);
list4.add(2);
result4 = 1;
```

```java
/**
 * This test shows the method returns the right int when it's not the first, it
 * should pass.
 * element
 */
List<Integer> list5 = new ArrayList<>();
Integer result5;
list5.add(200);
list5.add(20);
list5.add(50);
list5.add(3);
list5.add(2);
result5 = 2;

/**
 * This test shows the method returns the right string even through, it should
 * pass.
 * capitilization
 */
List<String> list6 = new ArrayList<>();
String result6;
list6.add("cat");
list6.add("Dog");
result6 = "Dog";

/**
 * This test shows the method returns the right int even though it's negative,
 * it should pass.
 */
List<Integer> list7 = new ArrayList<>();
Integer result7;
list7.add(1);
list7.add(-1);
list7.add(5);
result7 = -1;
```

```java
/**
 * This test shows the method returns the right string even though the 2nd could
 * have been interpreted as a number, it should pass.
 */
List<String> list8 = new ArrayList<>();
String result8;
list8.add("cat");
list8.add("four");
result8 = "cat";

/**
 * This test shows the method doesn't always return true (the result is
 * incorrect), it should fail
 */
List<String> list9 = new ArrayList<>();
Integer result9;
list9.add("two");
list9.add("four");
result9 = 2;

/**
 * This test shows the method returns NPEs, it should show an error
 */
List<Integer> list10 = new ArrayList<>();
Integer result10;
list10.add(null);
list10.add(2);
result10 = 2;
```

You can see test 9 and 10 (8 and 9 in the harness) and produced errors or failed as expected.