SFWRENG 3S03 Software Testing and Measurement, Winter Term 2021-22
Assignment #2 – due 11 March 2022 by 1159pm Hamilton time (submit on Avenue)

This assignment contains three questions. Answer all questions. If you are working in a group of up to three people and submitting one assignment, please make sure **all** people are named clearly on the assignment so that marks are correctly allocated to everyone in the group!

1. [10 marks] Consider the following class *PrimeNumbers*, which has three methods. The first, *computePrimes()* takes one integer input and calculates that many prime numbers. *Iterator()* returns an Iterator object that will iterate through the primes, and *toString()* returns a string representation.

```
public class PrimeNumbers implements Iterable<Integer>
 {
   private List<Integer> primes = new ArrayList<Integer>();
   public void computePrimes (int n)
   {
     int count = 1; // count of primes
     int number = 2; // number tested for primeness
     boolean isPrime; // is this number a prime
     while (count <= n)
     {
       isPrime = true;
       for (int divisor = 2; divisor <= number / 2; divisor++)
       {
         if (number % divisor == 0)
         {
           isPrime = false;
           break; // for loop
         }
       }
       if (isPrime && (number % 10 != 9)) // FAULT
       {
         primes.add (number);
         count++;
       }
       number++;
    }
}

@Override public Iterator<Integer> iterator()
  {
    return primes.iterator();
  }
```

```
@Override public String toString()
   {
     return primes.toString();
   }
}
```

computePrimes() has a fault that causes it not to include prime numbers whose last digit is 9 (for example, it omits 19, 29, 59, 79, 89, 109, ...).

Normally, this problem is solved with the *Sieve of Eratosthenes*. The change in algorithm changes the consequences of the fault. Specifically, false positives are now possible in addition to false negatives.

(a) [4 marks] Reimplement the algorithm for calculating primes using the Sieve of Eratosthenes approach, but *leave the fault* in place.

(b) [3 marks] What is the first false positive and how many primes must a test case generate before encountering it?

(c) [3 marks] Remember Assignment 1 where you explained a fault in terms of the concepts of *reachable, infection, propagation and revealing*? We used the following terminology. A test must reach a location in a program that contains the fault (*reachability).* After the location is executed, the state of the program is incorrect (*infected)*. The infected state must propagate through the execution and cause some output of the final state of the program to be incorrect (*propagation)*. Finally, the tester must observe part of the incorrect portion of the final program state (*revealability).* (This all serves to illustrate that testing is actually really complicated!) What does this example illustrate about these four concepts?

2. Derive input space partitioning test inputs for the GenericStack class with the following method signatures:

- public GenericStack (); // constructor
- public void push (Object X);
- public Object pop ();
- public boolean isEmpty ();

Assume the usual semantics for the GenericStack. Try to keep your partitioning simple and choose a small number of partitions and blocks.

(a) [2 marks] List all of the input variables, including the state variables.

(b) [4 marks] Many testing methods involve first identifying *characteristics* that can help us determine desired behaviour, and then partitioning the input space into two or more *blocks*. For

example, if we were testing a method findElement(list, element) we might use the characteristics “list is null” and “list is empty”, and the blocks might be “null list” and “empty list”. Once we have blocks we can come up with values for the actual tests, e.g., list=null, list=List.empty(). A good set of characteristics and blocks should be disjoint and complete. For the example findElement(), the characteristics and blocks are indeed disjoint, but they are incomplete.

Define characteristics of the input variables. Make sure you cover all input variables.

(c) [2 marks] Partition the characteristics into blocks.

(d) [2 marks] Define values for each block.

3. Consider the following class.

```
/** ******************************************************
*  Rewraps the string (Similar to the Unix fmt).
*  Given a string S, eliminate existing CRs and add CRs to the
*  closest spaces before column N.  Two CRs in a row are considered to
*  be "hard CRs" and are left alone.
******************************************************** */
import java.io.*;

public class FmtRewrap
{
   static final char CR = '\n';
   static final int inWord      = 0;
   static final int betweenWord = 1;
   static final int lineBreak   = 2;
   static final int crFound     = 3;
   static private String fmtRewrap (String S, int N)
   {
      int state = betweenWord;
      int lastSpace = -1;
      int col = 1;
      int i = 0;
      char c;

      char SArr [] = S.toCharArray();
      while (i < S.length())
      {
         c = SArr[i];
         col++;
         if (col >= N)
            state = lineBreak;
         else if (c == CR)
            state = crFound;
         else if (c == ' ')
            state = betweenWord;
         else
            state = inWord;
         switch (state)
```

```
         {
         case betweenWord:
            lastSpace = i;
            break;

         case lineBreak:
            SArr [lastSpace] = CR;
            col = i-lastSpace;
            break;

         case crFound:
            if (i+1 < S.length() && SArr[i+1] == CR)
            {
               i++; // Two CRs => hard return
               col = 1;
            }
            else
               SArr[i] = ' ';
            break;

         case inWord:
         default:
            break;
         }  // end switch
           i++;
      }  // end while
      S = new String (SArr) + CR;
      return (S);
   }

   public static void main (String []argv)
   {  // Driver method for fmtRewrap
      int integer = 0;
      int []inArr = new int [argv.length];
      if (argv.length != 1)
      {
         System.out.println ("Usage: java FmtRewrap v1 ");
         return;
      }

      System.out.println ("Enter an integer: ");
      integer = getN();

      System.out.println ("The Result String is: " + fmtRewrap (argv[0],
integer));
   }

   private static int getN ()
   {
      int inputInt = 1;
      BufferedReader in = new BufferedReader (new InputStreamReader
(System.in));
      String inStr;

      try
      {
         inStr    = in.readLine ();
```

```
            inputInt = Integer.parseInt(inStr);
        }
        catch (IOException e)
        {
            System.out.println ("Could not read input, choosing 1.");
        }
        catch (NumberFormatException e)
        {
            System.out.println ("Entry must be a number, choosing 1.");
        }

        return (inputInt);
    }  // end getN
}
```

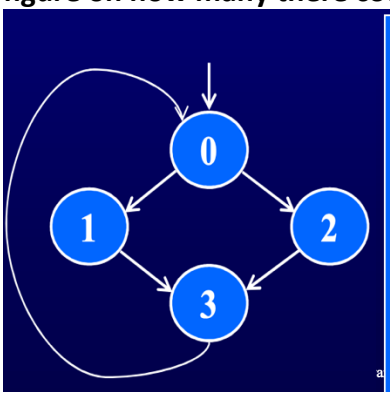
(a) [10 marks] Draw the control flow graph for the fmtRewrap() method.

(b) [5 marks] For this method, find a test case such that the corresponding test path visits the edge that connects the start of the *while* statement to the S=new String(SArr)+CR; statement, *without* going through the body of the while loop.

(c) [5 marks] State the test requirements for Node Coverage, Edge Coverage and Prime Path Coverage. That is, summarize what needs to be evaluated (nodes, edges etc) to achieve 100% coverage. See the note about prime path coverage below!

Node coverage is where for every reachable node in your CFG, your test set generates a path that visits that node. *Edge coverage* is similar but slightly stronger: your test set generates a path that includes every reachable path of length up to 1 (this allows for CFGs with one node).

You've not seen *Prime Path Coverage* before. Here's the definition: a path from node I to node J is simple if no node appears more than once (except possibly the first and last nodes can be the same). A prime path is therefore a simple path that does not appear as a proper subpath of any other simple path. For example, given the CFG shown on the left, the simple paths and prime paths for that CFG are shown on the right. **HINT: there are a VERY LARGE number of prime paths for this program. DO NOT ATTEMPT TO CALCULATE THEM ALL, but do attempt to give a ballpark figure on how many there could be. HINT 2: there are more than 100!**



Simple Paths : [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1], [0, 1, 3], [0, 2, 3], [1, 3, 0], [2, 3, 0], [3, 0, 1], [3, 0, 2], [0, 1], [0, 2], [1, 3], [2, 3], [3, 0], [0], [1], [2], [3]

Prime Paths : [0, 1, 3, 0], [0, 2, 3, 0], [1, 3, 0, 1], [2, 3, 0, 2], [3, 0, 1, 3], [3, 0, 2, 3], [1, 3, 0, 2], [2, 3, 0, 1]