SEMESTER 1, 2020 SAMPLE EXAM

CITS5501 Software Testing and Quality Assurance

This paper contains: 12 pages (including title page)

Time allowed: 135 minutes (including reading time)

Instructions:

- The exam has 5 questions with a total value of 50 marks.
- The time allowed to complete the exam is 135 minutes. (2 hour exam plus 15 minutes to allow for any technology overheads)
- This is an ExamSoft exam.
- Printed or handwritten notes are permitted.
- No electronic devices are permitted during the examination.
- Calculators are not allowed for this exam.
- Students may use blank paper for rough working during the exam.
- All answers must be typed in ExamSoft for marking.
- Any feedback for the examiner must be entered using the ExamSoft Notes feedback feature.

replace this page with official cover page 1

replace this 2nd page with official cover page 2
(that is, “this page has been left blank”)

`wordCount` method

Questions 1 and 2 refer to the following Java `wordCount` method.

```java
/** Returns the number of unique words in the string "aString"
 * (where a "word" is a contiguous sequence of non-whitespace
 * characters).
 * If "aString" is null, the result is undefined.
 */
public static int wordCount(String aString) {
  aString = aString.trim(); // trim whitespace from either end

  // handle case where string has no words
  if (aString.length() == 0) {
    return 0;
  }

  String words[] = aString.split("\\s+");

  ArrayList<String> uniqueWords = new ArrayList<>();
  int count = 0;
  for (int i = 0; i < words.length; i++) {
    String w = words[i];
    if (! uniqueWords.contains(w) ) {
      count += 1;
      uniqueWords.add(w);
    }
  }
  return count;
}
```

1. Consider the `wordCount` method described above.

   Using input space partitioning, describe some partitions from which test cases for this function could be drawn, showing your working. Describe three unit tests derived from those partitions. You need not present code for the unit tests, but should clearly describe the input and expected results.

[10 marks]

Solution:

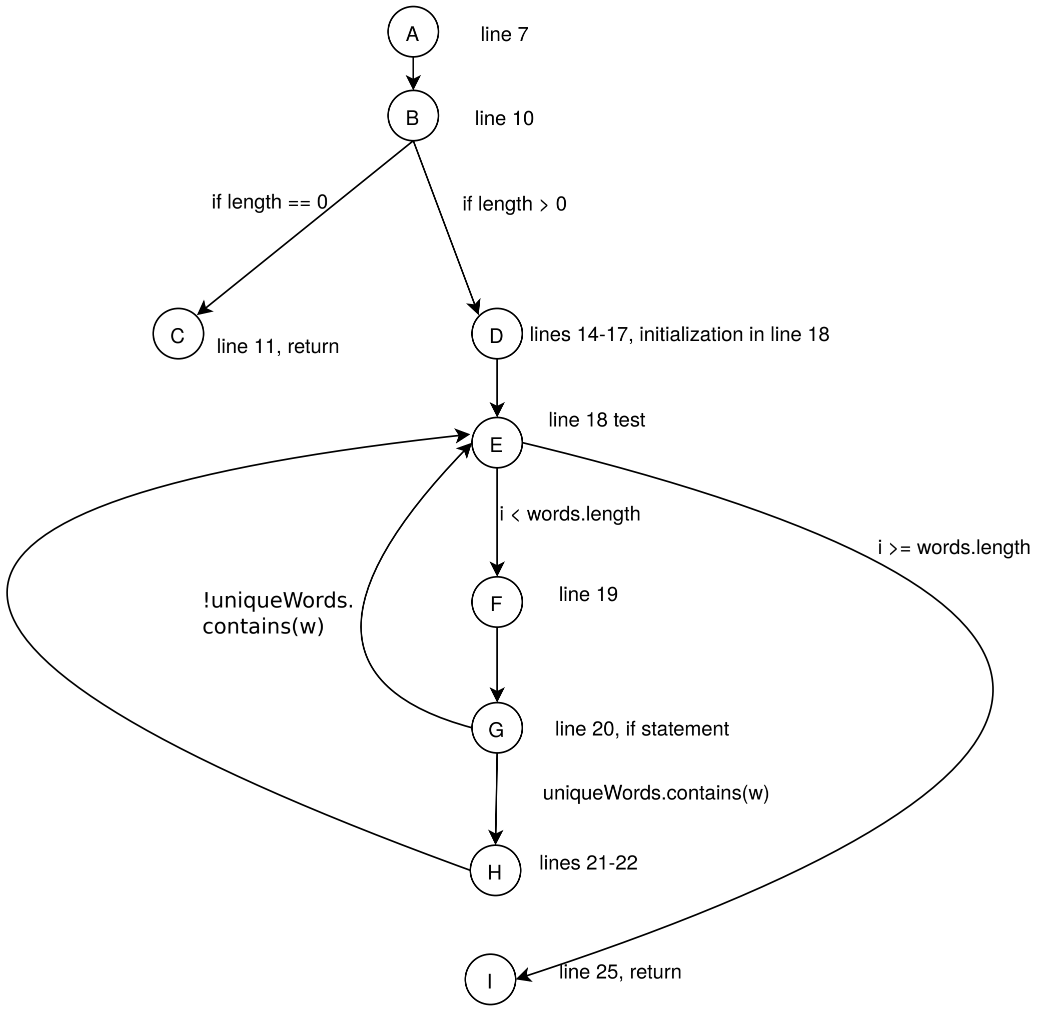
- The function has only one input parameter, “aString”, which ranges over the set of all strings. Since the result is undefined for null parameters, we do not consider those as part of the domain.
- Properties of “aString” which we could use for partitioning include:
  - whether it is empty, or non-empty
  - the number of words it contains (e.g. 0, 1, or more)
  - whether the string does or does not contain duplicates
- It follows that some possible partitions are:

  - `aString` is the empty string (from the "empty or non-empty" characteristic)
  - `aString` contains one word (from the "number of words contained" characteristic)
  - `aString` contains more than one word, and contains no duplicates
  - `aString` contains more than one word, and contains duplicates

- Unit tests derived from these partitions are:
  - An empty string, `""`. We should test that `wordCount("")` equals 0.
  - A string with one word, e.g. "aaa". We should test that wordCount("aaa") equals 1.
  - A string with two words and no duplicates, e.g. "aaa bbb". We should test that wordCount("aaa bbb") equals 2.
  - A string with two words and duplicates, e.g. "aaa aaa". We should test that wordCount("aaa aaa") equals 1.

2. Consider the Java `wordCount()` method described previously.

The following diagram is intended to represent the flow of control through the function.



i. Explain what a *simple path* and a *prime path* are.

ii. Identify three simple paths through the diagram that are *not* prime paths, justifying your answer.

iii. Identify two prime paths through the diagram, justifying your answer.

iv. Identify any simplifying assumptions which have been made in producing the diagram.

[10 marks]

Solution:

i. A simple path never intersects itself (except that the first and last nodes may be the same). A prime path is a maximal simple path - it cannot be extended without ceasing to be simple.

ii. Three non-prime simple paths are listed below; none of them intersect themselves, and all of them could be extended and remain simple.

- A B D (could be extended to ABCDE)
- E F G (could be extended to EFGE)
- E F G H (could be extended to EFGHE)

iii. Two prime paths are listed below; none of them intersect themselves, and none can be extended without ceasing to be prime:

- E F G E
- E F G H E

iv. Simplifying assumptions are that exceptions have not been considered. Line 7 would throw an exception if `aString` were null, but this is not shown. Likewise, calls to methods or constructors (such as `new ArrayList<>()` in line 15) could potentially throw exceptions, but these have not been shown.

Also, the graph only shows the control flow for this method: it does not follow the flow of control through any method calls or constructors.

3. You are on the software quality assurance team for Rook Capital, which specialises in executing trades for stock and futures brokers. The software development team is working on a program, "UBAR" ("Unbelievably Brisk Automatic Ratiocinator") for automatically executing trades. It takes as input sets of rules (effectively, programs described in a simple language) supplied by Rook Capital's clients, monitors stock prices, and automatically executes trades based on the rules. An incorrectly executed trade could result in enormous losses for clients, so it is important the system be highly reliable: it should function correctly, transmit data securely, and have minimal downtime.

   How would you develop a *risk management* plan for Rook Capital? Explain the steps involved. Identify at least three risks, assess their impact and chance of occurring, and explain how you would mitigate them.

[10 marks]

Solution:

Development of a risk management plan is one part of managing risk for a project. It assumes we have already done the following things:

- Identify possible risks
- Analyze each risk to estimate its likelihood and impact
- Rank the risks by probability and impact

The risk management plan then aims to reduce the risk involved in the highest-ranked risks. This can be done by risk avoidance, risk monitoring, and risk management and contingency planning.

Risk avoidance means attempting to ensure the risk does not occur. For instance, if ambiguity of specifications were identified as a risk, then a way of avoiding that might be to formally model the specifications, precluding ambiguity.

Risk monitoring means monitoring factors that could contribute to a risk eventuating. For instance, if staff turnover were assessed as a likely risk, then staff satisfaction might be such a factor.

Risk management and contingency planning attempt to reduce the impact of a risk if it does occur. For instance, if there were a risk involved in using some novel technology or component, then as a contingency, we might plan to use an alternative (e.g. one that maybe has less good performance, but is better known).

Three possible risks here are detailed below.

- Insecure transmission of data. We might assess the likelihood as *likely* (since it can be quite easy to make security errors), and the impact as severe (presumably, Rook highly values the confidentiality of its data).
- System failure, resutling in downtime. Again we might assess the likelihood as *likely* (since if extra care is not taken, it can be easy for bugs to result in

downtime), and the impact as severe (since downtime would presumably mean no trades could take place, which would be a severe impact). (As an aside, "minimal downtime" on its own is a vague requirement; presumably, the specification of the system contains a more precise requirement.)

- Failures (i.e., incorrect performance of the system) occurring when a trade is placed. We might assess the likelihood as *likely* (since it is easy to introduce bugs, and bugs in the relevant code could cause failures), and the impact as severe (since an incorrectly placed trade might result in significant financial loss).

To manage these risks, possibilities are:

- For critical parts of the system, develop formal specifications of those parts, and verify that implemented code adheres to those specifications. For instance, the code which interprets rules supplied by clients could be formally modelled and verified, to ensure that a badly-formulated rule is never executed by the system. We might also be able to verify security properties of the system.
  This application of formal methods could help avoid both risks 1 (insecurity) and (incorrect operation).
- Ensure that best practices are applied when securing the system. For instance, if data is transmitted to or from a web-site, it should use appropriate secure protocols. A penetration testing team (in-house or external) could be employed to identify possible weaknesses in system security.
  These would both help avoid the risk of insecurity.
- Put infrastructure in place so that the system has backups and redundancy. For instance, we might create a redundant system that can take over if the original fails.
  This would help avoid the risk of downtime.

4. You are designing tests for a back-end banking system, which takes as input a list of transactions to perform.

   The input format is specified by the following grammar:

```
<transaction> ::= <deposit> | <withdrawal>
<deposit>     ::= "DEP" <accountId> <amount>
<withdrawal>  ::= "WDR" <accountId> <amount>
<amount>      ::= "." <digit> <digit> | <digit><amount>
<accountId>   ::= <digit><digit><digit> "-" <digit><digit><digit>
<digit>       ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

   i. Explain how *terminal coverage* and *production coverage* differ.

   ii. If you were writing syntax-based tests for the system, based on the grammar above, how many tests would be required to achieve terminal coverage? And how many for production coverage? Explain your answer.

   iii. Provide one test based on the grammar: a description of the test, the inputs, and the expected output. If you need to make any assumptions, state what they are.

[10 marks]

Solution:

i. *terminal coverage* requires at least one test case for each terminal symbol – i.e. a symbol that is not defined in terms of other symbols. Examples of terminal symbols here are `"DEP"`, `"WDR"`, and `"0"`.

Production coverage, on the other hand, requires at least one test case for each production in a grammer - i.e., each possible alternative contained in a rule. For example, the `<transaction>` symbol is defined in terms of two productions, `<deposit>` and `<withdrawal>`, so production coverage would require at least one test case for each of these two.

ii. There are ten terminals for `<digit>`, and four others: `"DEP"`, `"WDR"`, `"."` and `"-"`. So terminal coverage would require 14 tests at minimum.

The number of productions is:

- two for `<transaction>`
- one for `<deposit>`
- one for `<withdrawal>`
- two for `<amount>`
- one for `<accountId>`
- ten for `<digit>`

for a total of 17 tests.

iii. A test based on this grammar would operate on some string in the grammar: e.g. `"DEP 001-001 02.31"`. The exact test would depend on what kind of test we are doing. We might be writing a unit test (e.g. just testing that the string is parsed properly), or an integration test, or a system or subsytem test (testing that a system or subsystem operates according to specifications).

If we assume a subsystem test, then we might test that when the system is supplied with this string, it carries out the appropriate operation (presumably, making a deposit to account 001-011). We would using real, but non-production components such as databases, and still use mocks for external systems (for instance, obviously when tests are run, we would not want to *actually* make a trade).

There would be many input values, but most of these would be *prefix values*: commands or functions required to set up the database, log a user in, etc. before the final input (`"DEP 001-001 02.31"`). We would probably also have to supply postfix values to find out whether the deposit occurred (e.g., sending SQL commands to query a database).

5. You are part of a team developing an online, searchable movie database called "YAMDB" ("Yet Another Movie Database"), and you are using the Alloy analyzer to develop a formal model of your system.

   Your team has identified the following entities, relationships and constraints which your system should represent:

   - The system models relationships between entities called *Movies*.
   - A movie may have one movie which is a direct *sequel* to it. (Or, it may not have a sequel.)
   - A movie cannot be its own sequel. (As a result of this, certain genres of time-travel film are outside the scope of YAMDB.)
   - If we have two movies, a and b, then it cannot be the case that they are both each other's sequel.
   - In addition to its direct sequel, a movie has a set (possibly empty) of *sequels*, defined as follows:
     - If movie b is the direct sequel of movie a, then b is in the sequels of a.
     - If movie m is in the sequels of movie a, and movie n is the direct sequel of movie m, then n is in the sequels of a.
     - No other movies are sequels of a.

   Give Alloy definitions which model a "Movie" entity, with the constraints described above.

   [10 marks]

Solution:

One simple solution is as follows:

```
sig Movie {              // there are "Movie" things
  sequel: lone Movie,    // which may have a direct sequel
  sequels: set Movie     // and which have a set of sequels
}

// no movie is its own sequel
fact NoReflexiveSequel {
  no m : Movie | m = m.sequel
}

// movies a and b cannot be each others' sequel
fact NoSymmetricSequel {
  no a, b : Movie | (a = b.sequel) && (b = a.sequel)
}

// the set of sequels is the transitive closure of the "sequel"
// relation
fact SequelsDefinition {
  all m : Movie | m.sequels = m.^sequel
}
```

[NB:

This model doesn't prevent cycles in the solution. The rule `NoReflexiveSequel` prevents cycles of size 1 (i.e. self-loops), and the rule `NoSymmetricSequel` prevents cycles of size 2 (a back-and-forth loop between two movies), but larger cycles could still occur. However, the question does not explicitly prohibit those.

A *very* good answer might point this out, and suggest the requirements could potentially be improved to avoid this.]

END OF PAPER