Untitled-1**DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY  
AssessmenT DESCRIPTION 2019/20**

**MODULE DETAILS:**

|  |  |  |  |
| --- | --- | --- | --- |
| Module Number: | 600094 | Trimester: | 2 |
| Module Title: | Advanced Software Engineering | | |
| Lecturer: | Dr Gordon | | |

**COURSEWORK DETAILS:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Assessment Number: | 2 | of | | | 2 | | | |
| Title of Assessment: | Software Testing theory and advanced practice | | | | | | | |
| Format: | Written |  | | | | |  | |
| Method of Working: |  | | | | | | | |
| Workload Guidance: | Typically, you should expect to spend between | 10 | | and | | 20 | | hours on this assessment |
| Length of Submission: | This assessment should be **no** more than:  *(over length submissions* ***will be*** *penalised as per University policy)* | | 2000 words | | | | | |

**PUBLICATION:**

|  |  |
| --- | --- |
| Date of issue: | 21 April 2020 |

**SUBMISSION:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ONE copy of this assessment should be handed in via: |  | | If Other  (state method) |  |
| Time and date for submission: | **Time** | 2pm | **Date** | 5 May 2020 |
| If **multiple hand–ins** please provide details*:* | N/A. | | | |
| Will submission be scanned via TurnitinUK? |  |  | | |

**MARKING:**

|  |  |
| --- | --- |
| Marking will be by: |  |

**ASSESSMENT:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| The assessment is marked out of: | 50 | and is worth | 50 | % of the module marks |
| **N.B** If multiple hand-ins please indicate the marks and % apportioned to each stage above (i.e. Stage 1 – 50, Stage 2 – 50). It is these marks that will be presented to the exam board. | | | | |

**ASSESSMENT STRATEGY AND LEARNING OUTCOMES:**

The overall assessment strategy is designed to evaluate the student’s achievement of the module learning outcomes, and is subdivided as follows:

|  |  |  |
| --- | --- | --- |
| LO | Learning Outcome | Method of Assessment  *{e.g. report, demo}* |
| ***2***  ***3*** | *Demonstrate comprehension of the key concepts of formal specification and its applicability in the software engineering lifecycle.*  *Identify, select and employ systematic testing strategies and techniques for automated testing.* | Written answers demonstrating understanding and application of knowledge.  Written answers demonstrating understanding and application of knowledge |

|  |  |  |
| --- | --- | --- |
| Assessment Criteria | Contributes to Learning Outcome | Mark |
| Application of appropriate method to solve the given problems | 2, 3 | 50 |

**FEEDBACK**

|  |  |  |  |
| --- | --- | --- | --- |
| Feedback will be given via: | Rubric/annotations in Canvas |  |  |
| Exemption  (staff to explain why) |  | | |
| Feedback will be provided no later than 4 ‘teaching weeks’ after the submission date. | | | |

This assessment is set in the context of the learning outcomes for the module and does not by itself constitute a definitive specification of the assessment. If you are in any doubt as to the relationship between what you have been asked to do and the module content you should take this matter up with the member of staff who set the assessment as soon as possible.

Please be reminded that you are responsible for reading the University Code of Practice on Academic Misconduct through the Assessment section of the Quality Handbook. This governs all forms of illegitimate academic conduct which may be described as cheating, including plagiarism. The term ‘academic misconduct’ is used in the regulations to indicate that a very wide range of behaviour is punishable.

In case of any subsequent dispute, query, or appeal regarding your coursework, you are reminded that it is your responsibility to produce the assignment in question.

**Assessment Instructions**

This assessment is a written assignment.

You should answer the questions below. Consider the specific points raised: some ask you to compare, contrast or explain your reasoning. Others are applications of your knowledge.

You should type this (preferred). However, if you choose to handwrite it you shuld then scan/photograph and upload the images. In the event you handwrite it, ensure that you write clearly and that the scanned version is legible/readable.

**1** (i) Explain what white box testing is, and contrast it to black box testing.

[2 marks]

|  |
| --- |
| White box testing involves the process of creating and deploying tests based on the internal structure of the program, with the tester having knowledge of the inner workings of the program. This process is usually performed by the software developers themselves.  Contrary to this, with black box testing, the tester does not have knowledge of the inner workings of the program i.e. the internal structure/code is hidden to the tester. This process is usually performed by specialized software testers and is done to ensure software functionality. |

(ii) Given the following code fragment, explain the distinction between statement testing and decision (or branch) testing.

if (x > 3) {

if (y > 5) {

return 1;

}

return 0;

}

Calculate the statement coverage for inputs x = 2 and y = 6.

Identify suitable branch test cases for a 100% coverage.

[5 marks]

|  |
| --- |
| Statement testing involves testing with specific inputs and analysing the results each statement within the program. Whereas decision (branch) testing involves ensuring each possible decision point outcome has been explored and each branch has been executed at least once.  With the inputs x=2 and y=6, the first conditional (X > 3) is executed and evaluates as false, meaning that the nested (Y > 5) conditional will never be met. Therefore, as this first line is the only line within the code fragment to be executed, we can fill into the following equation to calculate the statement coverage. The total number of statements includes the two if conditionals and the two return statements, giving us 4 total statements.  *Statement coverage = (statements executed / total statements) \* 100*  *Statement coverage = (1 / 4) \* 100*  *Statement coverage = 25%*  To achieve 100% branch coverage requires the following test cases:  **Test 1:** (x = 3) – With an *x* value of 3 and any *y* value, the first if conditional evaluates to false, and therefore none of the following nested code fragment is executed. Thus, no return statement is called.  **Test 2:** (x = 4, y = 6) – With these input values, the first if statement will evaluate to true, meaning that the second if statement can be executed. As the *y* value is 6, the conditional *(y > 5)* evaluates to true, and thus this test returns the value 1.  **Test 3:** (x = 4, y = 5) - With these input values, the first if statement will evaluate to true, meaning that the second if statement can be executed. As the *y* value is 5, the conditional *(y > 5)* evaluates to false, and thus the nested code is never executed, and the program returns the value 0. |

(iii) Consider the following program fragment where the input domain for x is {2, 3, 5} and the input domain for y is {0, 1, 2}, i.e. nine inputs in total.

n = x;

if (x > 3) { // if-a

if (y == 2) { // if-b  
 return n;

}

}  
 if ((x < 5 && y == 0)

|| (x == 2 && y < 2)) { // if-c  
 return 4;  
 }

return 6;

(a) Complete the table below to indicate the branch executed at each if-statement in the program above when the test at the head of each column is executed (column has been completed for you).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Tests | | | |
|  | (x = 2, y = 0) | (x = 3, y = 2) | (x = 5, y = 2) | (x = 5, y = 0) |
| if-a | F | F | T | T |
| if-b |  |  | T | F |
| if-c | T | F |  | F |
| returns | 4 | 6 | 5 | 6 |

The bottom row of the table gives the value returned for each test.

[4 marks]

(b) Explain and identify a minimal set of tests that achieves coverage of all the feasible branches in the program.

[3 marks]

|  |
| --- |
| The minimal test set that will achieve coverage of all the feasible branches in the program with the specified input domains is as follows:  {(X = 5, Y = 2), (X = 5, Y = 1), (X = 3, Y = 0)}  **Test 1:** (X = 5, Y = 2) – The first if-a statement will evaluate as true due to 5 > 3, meaning the second nested if-b conditional can be met, which also evaluates as true as 2 == 2, thus leading the program to return n (in this case n = x = 5)  **Test 2:** (X = 5, Y = 1) – The fist if-a statement will evaluate as true due to 5 > 3, meaning the second nested if-b conditional can be met, however this returns as false due to 1 != 2. The if-c conditional is exexuted, however neither conditions are met and this evaluates to false, meaning the program returns 6.  **Test 3:** (X = 3, Y = 0) – The first if-a statement will evaluate as false due to 3 not being greater than 3, the if-c conditional is then met and evaluates as true due to 3 being less than 5 and the y value being equal to 0. This means that the program can then return 4.  These tests will identify each individual possible path within the program. |

(c) Explain what is meant by partition coverage in software testing. For the above example, partition the input domain into subsets of inputs that produce the same output, and identify a minimal set of tests that can achieve partition coverage for the program.

[3marks]

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Partition coverage involves the process of grouping all possible inputs into their respective data classes based on a behavior change in the program (in this case being the outputs), and acquiring this involves selection of a test set that results in each output once.   |  |  | | --- | --- | | **Output** | **Input Domain(s)** | | 4 | {(2,0), (3,0), (2,1)} | | 5 | {(5,2)} | | 6 | {(2,2), (3,1), (3,2), (5,0), (5,1)} |   To achieve partition coverage, this requires a set of three tests, one with output 4, one with output 5, and another with output 6. For example:  Test 1 (X = 3, Y = 2) -> output 6  Test 2 (X = 5, Y = 2) -> output 5  Test 2 (X = 2, Y = 1) -> output 4 |

(d) Consider a mutation of the program in which the expression y < 2 is replaced with n < 2. Give a test that will distinguish this mutant from the original program.

[3 marks]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| The mutation on this program is to the if-c statement. As this statement initially contains (X == 2 && Y < 2), which is changed to (X == 2 && n < 2), with the knowledge that N = X. As an X value cannot be equal to 2 and also less than 2, we can remove (disconsider) this section from the if-c statement, leaving us with ‘if (X < 5 && Y == 0)’.  The original program when tested with (X=2, Y=1) would reach the if-c statement and return 4 due to the if-c statement being true. However, in the mutation, the if-c statement would be reached but evaluate to false, and the program would return 6, hence distinguishing the mutatant.  The following is a table to display the outcomes of the input (X=2, Y=1) on both the original program and the mutated program.   |  |  |  | | --- | --- | --- | |  | **Original (Y < 2)** | **Mutation (N < 2)** | | If-a | F | F | | If-b |  |  | | If-c | T | **F** | | Return | 4 | **6** | |

(e) Insert your own mutation of the program in the line of the if-b conditional. Describe how this has altered the program. Create your own test (i.e. your own x and y values) and evaluate this for the original and the mitated program. Describe whether your test distinguishes your mutant from the original program.

[5 marks]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| For my mutation of the original program, I have decided to alter the if-b conditional by changing ‘y == 2’ to ‘y != 2’. Which means the conditional now evaluates to true if the y value does not equal 2 instead of when it equals 2.  **Test 1:** For the test values of (X=4, Y=1):  The original program will return 6, as although the if-a conditional is true, the if-b conditional is false and so is the if-c conditional.  With the mutant program however, the if-b conditional will be true and so the program will return n (in this case 4), distinguishing it from the original program.  **Test 2:** For the test values of (X=5, Y=2):  The original program will return n (in this case 5) as both if-a and if-b conditionals are true.  The mutant program will return 6 however as although the if-a is true, the if-b and if-c are both false, distinguishing it from the original program.  **Test 3:** For the test values of (X=3, Y=0):  The original program will return 4 as the if-a and if-b conditionals are false but the if-c conditional evaluates as true.  The mutated program will also return 4, as although the if-b conditional would be true, the required if-a conditional is false, and the if-c conditional remains true as this is unchanged between the original and mutant program. This test does not distinguish between the two programs.   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | |  | **Original (y == 2)** | | | **Mutation (y != 2)** | | | | Test Case | 1 | 2 | 3 | 1 | 2 | 3 | | If-a | T | T | F | T | T | F | | If-b | F | T | F | **T** | **F** |  | | If-c | F |  | T |  | F | T | | Return | 6 | 5 | 4 | 4 | 6 | 4 | |

**2**  (i) Describe some of the potential issues if a formal specification is long and complex, and why it may not be beneficial in terms of reducing errors.

[5 marks]

|  |
| --- |
| A formal specification is a way of detailing the program workings in a clear and concise format. If a formal specification becomes long and complex, it may not be beneficial in terms of reducing errors and can actually cause confusion regarding the aims and objectives of the program required.  Having a long specification is essentially like creating two programs which can be considered significantly time and money inefficient. A specification that is as long as or longer than the program being created provides little actual benefit. This specification may even contain errors itself, thus must be validated in some way.  Long specifications can often be hard to check, the simpler the specification the easier it would be to check and hence less likely to contain errors.  Given complex mathematical implementations stated in a specification to prove the correctness of a design, this may often lead to confusion when translating this into the program, and thus being more concise could have reduced potential for errors. |

(ii) Given the following program

t = 0;

while (x != 5) {

x = x - 1;

t = t + 2;

}

where x and t are integer variables.

1. Identify the program input variable or variables. [2 marks]

|  |
| --- |
| The input variable is x. The variable x is not defined within the code fragment. |

(b) Identify the program output variable or variables. [2 marks]

|  |
| --- |
| The program output variable is t.  X is not an output variable because when the program terminates, it is a constant value of 5. |

(c) Work out and explain the weakest pre-condition on the input that ensures that the program terminates.

[5 marks]

|  |
| --- |
| The program will terminate in the scenario where the loop is not executed once (input value of x == 5) or subsequently the loop executed a certain number of times until the x value reaches 5.  If input X is initially greater than 5 (X>5), this X value will be decremented by 1 every time the loop is executed and until the loop condition becomes false.  If the initial X value input is less than 5 (X<5), X will continually be decremented and the loop will execute an infinite amount of times as the exit clause will never be met.  Therefore, we can conclude the weakest pre-condition on the input that ensures that the program terminates is X = 5 or X > 5, otherwise stated as:  X >=5 |

(d) Find and explain the weakest pre-condition on the input that ensures the program post-condition t < 20.

[6 marks]

|  |
| --- |
| In order for a post-codition to be satisfied, the program must terminate, therefore the precondition must include X>=5  Take X=5, the loop is not entered, and the t value is never changed and remains at 0, therefore this satisfies the post condition of t < 20.  Take X>5, with X being decremented until it reaches the value of X=5 and the loop terminates. With each iteration of the loop, the t value will be incremented by 2. Provided the statement is executed up to a maximum of 9 times (output value of t = 18), then the post condition of t < 20 will be met. The value of X in this case will have been decremented no more than 9 times before the reaching the loop exit clause, thus the upper bound for the input X value will be 5 + 9 = 14.  We can combine these values to give the pre condition that ensures the post condition of t < 20 as the following:  X >= 5 and X <= 14. |

(e) Write out a specification for the program itself. E.g. for the program **i = 2\*i**; then a specification could be “the program doubles an integer”

[5 marks]

|  |
| --- |
| X is an input integer that is assigned a value, whilst another integer T is assigned a value of 0 at the beginning of the program.  For every iteration while the X value does not equal 5, the X value is then decremented by 1 whilst the T value is incremented by 2.  Upon program termination, the t value is output and the x value is at a constant of 5.  For the program to terminate and and output to be produced, the program requires the pre condition of X >= 5 as otherwise the loop would iterate on a constant cycle with no exit clause being met.  The post conditions of X = 5 and T >= 0 must be true upon termination of this program. We can calculate the final output value of T by filling the initial X value into the equation (2 \* (X-5)). |

**End of Questions**

`