Office Use Only

MONASH University

Semester One 2015
Examination Period

Faculty of Information Technology

EXAM CODES: **FIT5171**

TITLE OF PAPER: **SYSTEM VALIDATION AND VERIFICATION, QUALITY AND STANDARDS - PAPER 1**

EXAM DURATION: 2 hours writing time

READING TIME: 10 minutes

THIS PAPER IS FOR STUDENTS STUDYING AT:(tick where applicable)

☐ Berwick ☐ Clayton ☐ Malaysia ☐ Off Campus Learning ☐ Open Learning
☒ Caulfield ☐ Gippsland ☐ Peninsula ☐ Enhancement Studies ☐ Sth Africa
☐ Parkville ☐ Other (specify)

During an exam, you must not have in your possession, a book, notes, paper, electronic device/s, calculator, pencil case, mobile phone, smart watch/device or other material/item which has not been authorised for the exam or specifically permitted as noted below. Any material or item on your desk, chair or person will be deemed to be in your possession. You are reminded that possession of unauthorised materials, or attempting to cheat or cheating in an exam is a discipline offence under Part 7 of the Monash University (Council) Regulations.

No exam paper or other exam materials are to be removed from the room.

AUTHORISED MATERIALS

OPEN BOOK	☒ **YES**	☐ **NO**
CALCULATORS	☐ **YES**	☒ **NO**
SPECIFICALLY PERMITTED ITEMS if yes, items permitted are:	☐ **YES**	☒ **NO**

Candidates must complete this section if required to write answers within this paper

STUDENT ID: __ __ __ __ __ __ __ __   DESK NUMBER: __ __ __ __ __

Answer all questions in the space provided here.

Question	Points	Score
Question 1	15	
Question 2	6	
Question 3	6	
Question 4	8	
Question 5	15	
Total	50	

Part A. Unit Testing

Question 1 .... **15 marks**

Consider a program *FizzPrime* that takes as input two non-negative integers, x and i, both between 0 and 100, both inclusive. The number x is a prime numbers. As output, the program prints the number i itself within the range ($[0, 100]$) when it is not divisible by x. For multiples of x, but not multiples of x^2, the program should print "`Fizz`" instead of the number. For multiples of x^2 but not multiples of x^3, the program should print "`Prime`". Finally, for numbers which are multiples of x^3 the program should print "`FizzPrime`" instead.

(Continued overleaf)

(a) (5 marks) Develop robust equivalence classes for the input variables x and i given the above specification.

(b) (6 marks) Develop test cases using the *robust* (not worst-case) version of the boundary value testing technique.

(c) (4 marks) You have been given the task of performing blackbox testing on an implementation of the above algorithm. Of the main blackbox testing techniques we have discussed: boundary value testing (BVT), special value testing (SVT), equivalence class testing (ECT), and decision table-based testing (DTT), explain why each technique is (or is not) appropriate.

Question 2 .... **6 marks**

The *minimax* algorithm is a way of finding an optimal move in a two-player game for one player, by minimising the possible loss for the worst case scenario (maximum loss). It has been widely used in 2-player zero-sum game plays. The algorithm for the depth limited minimax algorithm is given below.

Procedure 1: The depth-limited minimax algorithm $minimax(node, depth, maximisingPlayer)$

```
Input: node                        ▷ Node where search begins.
Input: depth                       ▷ the maximum depth to search.
Input: maximisingPlayer            ▷ Boolean value representing the player for which
                                     the search is performed.
Output: the best value
if depth = 0 ∨ is_terminal(node) then
    return the heuristic value of node
end
if maximisingPlayer then
    bestValue ← −∞
    foreach child of node do
        val ← minimax(child, depth − 1, false)
        bestValue ← max(bestValue, val)
    end
    return bestValue
else
    bestValue ← +∞
    foreach child of node do
        val ← minimax(child, depth − 1, true)
        bestValue ← min(bestValue, val)
    end
    return bestValue
end
```

(Continued overleaf)

(a) (5 marks) Draw the program graph for the above function.

(b) (1 mark) Calculate the cyclomatic complexity of the program graph in the previous part.

Part B. Integration Testing

Question 3 . . . . **6 marks**

One of the goals of integration testing is to be able to isolate faults when a test case causes a failure. Consider integration testing for a program written in a procedural/object-oriented programming language. Rate the following integration strategies on their abilities of (1) relative fault isolation and (2) testing of co-functionality.

You also need to provide a *rationale* for your answer.

A Big bang

B Decomposition-based top-down integration

C Decomposition-based bottom-up integration

D Decomposition-based sandwich integration

E Call graph-based pairwise integration

F Call graph-based neighbourhood integration (radius 1)

G Call graph-based neighbourhood integration (radius 2)

Show your ratings graphically by placing the letters corresponding to a strategy on a line, as in the example below. Suppose that for the ability of fault isolation, strategies X and Y are about equal and not very effective, and strategy Z is very effective.

Note that this rating is relative and qualitative, so don't agonise over where *exactly* to put a strategy, but focus on their *relative* position.

Fault isolation

Y	
X	Z
Low	High

(Continued overleaf)

Part C. Software Metrics

Question 4 . . . . **8 marks**

In lecture 8 we introduced Weyuker's 9 properties to evaluate software metrics. Some of the properties (for example, properties 1, 3, 4 and 8) are quite simple and intuitive. However, some other properties are a bit more complex and need further analysis.

Weyuker's property 9 states that the complexity of the composition of two programs may be greater than the sum of the complexities of the two taken separately. More formally,

$$\exists A, B \colon \mathit{Program} \bullet M(A) + M(B) < M(A + B)$$

where M represents a given metric and $A + B$ represents the composition of A and B.

The structural metric *depth of nesting* of a program P, denoted $n(P)$, is defined for programs that only contain *structured programming constructs.*

Given a program, the repeated application of the following two operations can be used to decompose it into a unique tree of structured programming constructs.

Sequence: composing two program graphs sequentially by merging one program graph's terminal node with the other program graph's initial node. For example, sequential composition of programs A and B is denoted by $A; B$.

Nesting: replacing one node in one program with the entirety of another program. For example, nesting program B in program A at node x of A is denoted by $A(B, x)$.

The depth of nesting values of programs constructed by the above two operations are defined as below.

Sequence: $n(P_1; P_2; \ldots; P_n) = \max(n(P_1), n(P_2), \ldots, n(P_n))$, and

Nesting: $n(P_1(P_2; \ldots; P_n)) = 1 + \max(n(P_2), \ldots, n(P_n))$, where $P_2, \ldots, P_n$ are sequentially nested inside P_1.

Recall that there are six basic types of structured programming constructs:

Construct	Description	Construct	Description
P_n	sequence ($n = 1, 2, \ldots$)	D_2	while loop
D_0	if-then	D_3	do-while loop
D_1	if-then-else	C_n	case-switch

The depth of nesting value for all the above constructs is 1 except for P_1, which is 0. The depth of nesting value of a program is calculated in a bottom-up fashion.

For Weyuker's property 9 and the metric depth of nesting $n(P)$ of a valid program P, do the following:

(a) State whether the property holds or not.

(b) Prove your claim (informally).

Part D. Mutation Testing

Question 5 . . . . **15 marks**

Mutation testing is a technique to assess the efficacy and quality of a test suite. It works by making *mutants*, syntactic variations of the program under test, and measures how many of the mutants are *killed* by the test suite. The presence of non-equivalent *live* mutants represents inadequacy of the test suite.

The following Java method, `min`, returns the smallest of three integer parameters.

```java
    public int min(int a, int b, int c) {
        int temp = a;
        if (b < a) {
            temp = b;
        }
        if (c < b) {
            temp = c;
        }
        return temp;
    }
```

(a) (3 marks) Come up with an *equivalent* mutant by applying a *first-order* mutation. In your answer, identify:

1. The mutation operator applied,
2. The associated statement to be changed, and
3. What the statement is changed to.

(b) (8 marks) Devise a set of three test cases that achieves 100% statement coverage. Come up with three *non-equivalent* first-order mutants of the original program, making use one of the following mutation operators in each mutant. Determine the *kill rate* of your test suite on the three mutants.

The mutation operators you can use are:

`ror` Relational operator replacement.

`sdl` Statement deletion.

`uoi` Unary operator insertion.

(c) (4 marks) Is there a defect in the program? If so, develop the smallest set of test cases that achieves 100% statement coverage but *does not* reveal the defect. If not, develop the smallest set of test cases that achieves 100% statement coverage.

— Additional page for answers if required. Will be marked. —
— Indicate clearly question number. —

— Additional page for answers if required. Will be marked. —
— Indicate clearly question number. —

—— End of the paper ——