SENG 275 MID TERM EXAM2 Solutions-O4

Instructor: Dr. Navneet Kaur Popli, Date: 27 Feb 2023, Time: 1:30-2:20 PM PST
Mode: Pen-Paper, Synchronous, Timed, Total Marks: 35

Note: This is an individual activity. Copying or cheating of any kind is not allowed. Supplement with diagrams, tables, graphs, and charts wherever applicable.

Q1) Purchase discount is 0% for up to 500 US $ purchases, 5% is added for each additional 500 US $ up to 2000 US $, and 25% is applied for above 2000 US $. Which test inputs in US $ would be selected for valid equivalence partitions? (1M)

a) 250, 700, 1400, 1800, 4000
b) 250, 1400, 3000
c) -100, 250, 650, 1300, 1700, 2900
d) 200, 720, 1600, 1800, 2100

Q2) Regarding **boundary analysis of the condition** $a \leq 10$, which of the following statements **is true**?

a) There can only be a single on-point which always makes the condition true.
b) There can be multiple on-points for a given condition which may or may not make the condition true.
c) There can only be a single off-point which may or may not make the condition false.
d) There can be multiple off-points for a given condition which always make the condition false.

An on-point is the (single) number on the boundary. It may or may not make the condition true. The off point is the closest number to the boundary that makes the condition to be evaluated to the opposite of the on point. There's only a single off-point.

Q3) Perform boundary analysis on the following equality: x = = 10. What are the on- and off-points? (1M)

A5) The on-point is 10. Here we are dealing with an equality; the value can both go up and down to make the condition false. As such, we can have two off-points: 9 and 11. [if the student answers any one its ok]

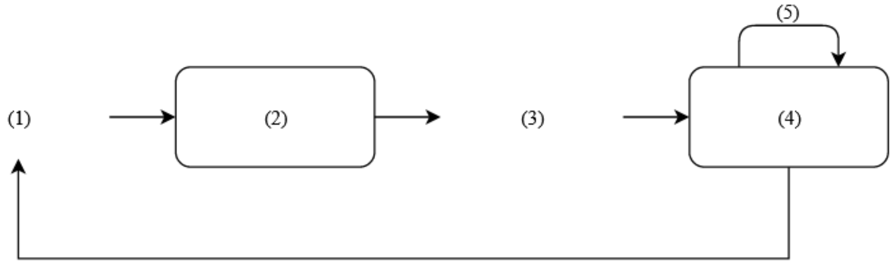
Q4) It is possible for a test to fully cover a line, without achieving full branch coverage of that line. Circle each of the following types of code where this phenomenon could occur: (1M)

- Conditionals
- function calls
-variable declarations

- FOR loops and enhanced FOR loops (foreach loops)
-object calls

A4) conditionals, for loops and foreach loops.

Q5) We have the following skeleton for a diagram illustrating the Test Driven Development cycle. What words/sentences should be at the numbers? (2.5M)



A5)

1. Write failing test
2. Failing test
3. Make test pass
4. Passing test
5. Refactor

Q6) Given below is a feature and a user story from a shopping cart website. Write Given, When, Then, And (And can be use multiple times if required) BDD test scenarios for this feature. Write one valid and one invalid scenario. Note any assumptions that you make. (2+2=4)
Feature: Adding items from wish list to shopping cart.
User Story: As a consumer of a shopping website ABC.com, I should be able to add an item from wish list to shopping cart.

A6)
(The question can have multiple solutions depending upon the understanding of the student. Refer to the TDD/BDD slide deck for more clarification)

Valid Test Scenario:

Given user has logged in as a valid user
And user has selected an item in the wish list to be added to the cart
When the user adds the item to the cart
Then the item should be added to the cart
And the cart item count should be incremented
And item price should be added to the total amount payable

Invalid Test Scenario:

Given user has logged in as a valid user
And user has not selected an item in the wish list to be added to the cart
When the user adds the item to the cart
Then the item should not be added to the cart
And the cart item count should not be incremented
And item price should not be added to the total amount payable

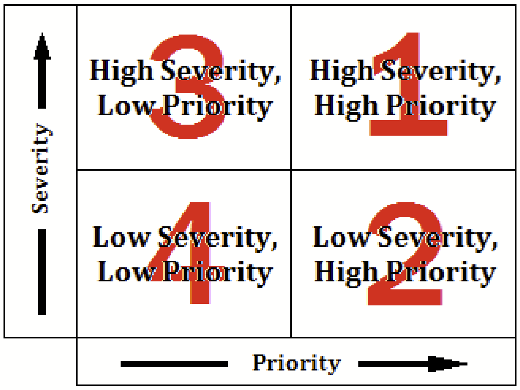
Q7) Give any 3 advantages of following the TDD approach. (3M)
A7)

- **Shared vocabulary** between all stakeholders.
- TDD developers use test cases before they write a single line of code. This approach encourages them to consider **how the software will be used** and what design it needs to have to provide the expected usability.
- Once a test fails, developers understand what needs to be changed and they refactor the code, that is, they rewrite it to improve it without altering its function.
- TDD focuses on the code necessary to pass the test, **reducing it to essentials**.
- It takes the **test unit**, or the smallest bit of functionality, as its basis.
- It's a bit like **building a house brick by brick**—no brick is laid down before it's tested, to make sure it's sound and that it's an integral part of the design.
- In TDD, you achieve **100% coverage test**. Every single line of code is tested, unlike traditional testing.

Q8) Request Traceability Matrix is a document that traces……user requirements………………….. with ……test cases………………. (2M)

Q9) Differentiate between bug priority and bug severity using any example from UVic's Brightspace page. (2M)
A9) Priority: a measure of the significance of the bug from a user-facing perspective - how critical is the bug from the user's point of view.
Priority: the order in which the defect should be fixed.
Priority: the value to the business of fixing the bug
Severity: a technical measure of the impact of the bug on the functionality of the application.

Q10) Explain the Bug resolution matrix. (2M)
A10)


3
High Severity, Low Priority
1
High Severity, High Priority
4
Low Severity, Low Priority
2
Low Severity, High Priority
Severity
Priority

Q11) JUnit randomizes the order of the unit tests it runs, each time the overall suite is executed. Why is this desirable? (2M)
A11) We want to make sure that our tests aren't tightly coupled - one test should have no effect on another. By executing tests in random order, we ensure that any unexpected dependence of one test on another is detected. Specifically, we wish to make sure that no test can be influenced by the state resulting from a previous test - we begin with a 'blank slate' each time.

Q12) Some strategies subsume other strategies. Formally, a strategy X subsumes strategy Y if all elements that Y exercises are also exercised by X. For the following six coverage strategies, fill in the hierarchy below to show the subsumption relationships among these strategies: (3M)

1. Branch + Condition coverage

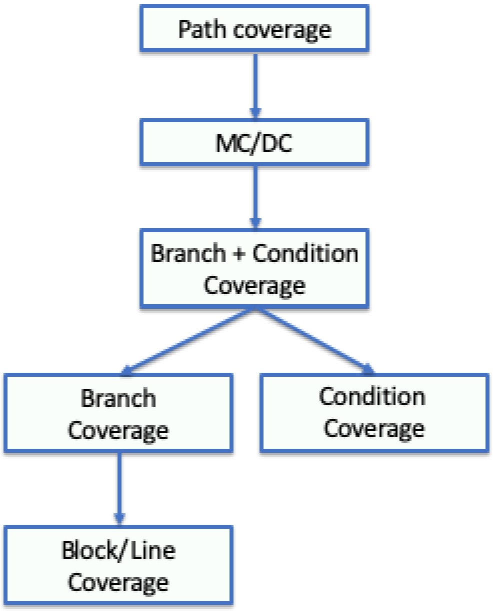
2. MC/DC coverage

3. Path coverage

4. Branch Coverage

5. Block/Line coverage

6. Condition coverage

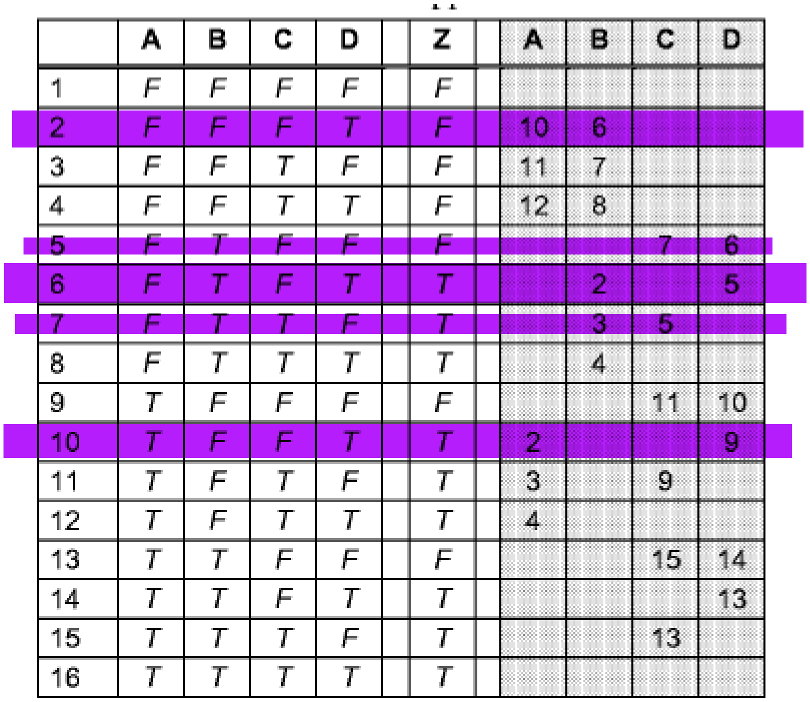


Q13)

```
If ( (( person = = student) || (person = = professor)) && ((department = = ECE) ||
(department = = CS)) )
{
            return ("You are invited to the Software conference");
}
else
{
            return ("You are not invited to the software conference");
}
```

Use the MC/DC approach of software testing to create minimum number of test cases to test this piece of code. Explain the steps you take. (10+2=12M)

A13)



	A	B	C	D		Z		A	B	C	D
1	F	F	F	F		F					
2	F	F	F	T		F		10	6		
3	F	F	T	F		F		11	7		
4	F	F	T	T		F		12	8		
5	F	T	F	F		F				7	6
6	F	T	F	T		T			2		5
7	F	T	T	F		T			3	5	
8	F	T	T	T		T			4		
9	T	F	F	F		F				11	10
10	T	F	F	T		T		2			9
11	T	F	T	F		T		3		9	
12	T	F	T	T		T		4			
13	T	T	F	F		F				15	14
14	T	T	F	T		T					13
15	T	T	T	F		T				13	
16	T	T	T	T		T					

Min MCDC= 10, 2, 6, 7, 5

There can be multiple answers for this question. Check slide deck for explanation.

Q14) Write a parameterized test using @ValueSource or @CSVSource to test the function compassDirection(), which returns a string containing the closest cardinal direction (North, East, South or West) as a string, given an integer number of degrees (with 0 degrees corresponding to due north). Tight boundary testing is not necessary, but you should have at least one test for each possible output and remember to check negative or otherwise potentially problematic integers. The code for the function is shown below: (5M)

```
public static String compassDirection(int degrees) {
    if (degrees < 0) degrees = -degrees;
    degrees %= 360;
    if (degrees > 315 || degrees <= 45) return "North";
    if (degrees <= 135) return "East";
    if (degrees <= 225) return "South";
    return "West";
}
```

A14)

Expected solution would be something along these lines:

```
@ParameterizedTest(name="degrees{0}, direction{1}")
@CsvSource({"-30, North", "40, North", "85, East", "460, East", "205, South",
"310, West"})
void compassDirection(int degrees, String direction) {
    assertEquals(direction, WordUtilities.compassDirection(degrees));
}
```

Q15) Using the TDD approach, write one 'Before' annotation code, and two Tests for a Rectangle class: a Before annotation has 'createObject()' function which creates a new rectangle with length and breadth as integers. The createObject() should have parameters for length and breadth. Also, its return type should be Rectangle. This object can be used by the next two tests. Create code to pass the test of Before annotation. Then create a test for the 'area()' function which calculates area of the rectangle as length*breadth. Create code to pass this test. Then create a test for the 'perimeter()' function which calculates the perimeter of the rectangle as 2*(length+breadth). Create code to pass this test. Do not repeat code in subsequent tests. Write this in a step-by-step form in a way TDD should be performed. Suggest any one way in which the code can be refactored. Write the refactored part of the code. Do not add any new functionality in the refactoring step.(2+2+2+2=8M)

A15)

First write simple tests of area() and perimeter(). These will fail because there is no code to run them.

[If a student does not write all assertions, grade should not be deducted. 2M for creatObject() test and code, 2M for area() test and code, 2M for parameter test and code, 2M for refactoring]

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.Before;
import org.junit.Test;

public class RectangleTestTest {
             Rectangle r;
             @Before
             public void createObject()
             {
                     r=new Rectangle(5,6);

             }
             @Test
             public void testArea()
             {
                    int actual=30;
                    int expected=r.area();
                    assertEquals(actual,expected);
             }
             @Test
             public void testPerimeter()
             {
                    int actual=22;
                    int expected=r.perimeter();
                    assertEquals(actual,expected);

             }

      }
```

Ideally the function createObject() should be written first, then corresponding Rectangle class should be created to pass the test. Then area() should be written which would fail initially. Write code to pass it. In the end perimeter() function should be written which again fails and code should then be written to pass it. These series of steps are essential for the student to show to get grades.

```
public class Rectangle {
        int length, breadth;
        Rectangle(int l, int b)
        {
                length=l;
                breadth=b;
        }
        int area()
        {
                return length*breadth;
        }
        int perimeter()
        {
                return 2*(length+breadth);
        }

}
```

The code can be refactored so that a generalized data type <T> can be used instead of only integer data type. Other refactorings are also possible.

Q16) Is code coverage a good measure of test suite effectiveness? Comment. (1M)
A16) Code coverage does not guarantee that the covered lines or branches have been tested correctly, it just guarantees that they have been executed by a test. There may be missing assertions in a test or low-quality tests. So code coverage is not a good measure of test suite effectiveness.