**Introduction to Software Engineering (CS550)**

**Homework #5**

**Name: Hailu Belay Kahsay                                                         ID: 20155624**

**Q1.**

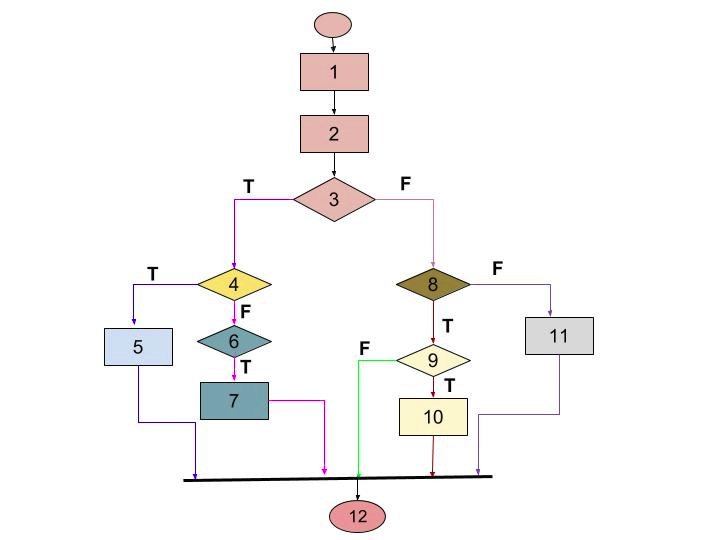
**Solution:**

|  |  |
| --- | --- |
| **ID** | TC-1 |
| **Title** | Equilateral Triangle |
| **Pre-Condition** | Read integer value for each side of the triangle(let **a, b** and **c** be the sides of the triangle) |
| **Test Steps** | 1. No side may have a length of Zero 2. Each side must be shorter than the sum of all sides divided by 2. i.e.  S=(a+b+c)/2            Then **S>a, S>b, and S>c** must hold   1. Each side of the triangle are equal. i.e. **a=b=c** |
| **Expected Result** | The program prints ***“The triangle is Equilateral Triangle”*** message to the user |

|  |  |
| --- | --- |
| **ID** | TC-2 |
| **Title** | Isosceles Triangle |
| **Pre-Condition** | Read integer value for each side of the triangle(let **a, b** and **c** be the sides of the triangle) |
| **Test Steps** | 1. No side may have a length of Zero 2. Each side must be shorter than the sum of all sides divided by 2. i.e.  S=(a+b+c)/2            Then **S>a, S>b, and S>c** must hold   1. Any two sides of the triangle are equal. i.e. **(a=b and  ac) or (a=c and  ab) or (b=c and  ab)** |
| **Expected Result** | The program prints ***“The triangle is Isosceles Triangle”*** message to the user |
| **ID** | TC-3 |
| **Title** | Scalene Triangle |
| **Pre-Condition** | Read integer value for each side of the triangle(let **a,b** and **c** be the sides of the triangle) |
| **Test Steps** | 1. No side may have a length of Zero 2. Each side must be shorter than the sum of all sides divided by 2. i.e.  S=(a+b+c)/2            Then **S>a, S>b, and S>c** must hold   1. Each side of the triangle should be equal. i.e. **abc** |
| **Expected Result** | The program prints ***“The triangle is Scalene Triangle”*** message to the user |

**Q2.**

**2.1 Solution:**

****

As shown in the above flowchart, branch coverage testing technique can test all of the six statements, following the paths listed below, without detecting the outright error, i.e. ***if… else if*** statement without ***else*** statement next to line #9 as decision coverage testing uses fault-tolerant error-handling technique.

Path 1: 1-2-3-4-5-12

Path 2: 1-2-3-4-6-7-12

Path 3: 1-2-3-8-9-12

Path 4: 1-2-3-8-9-10-12

Path 5: 1-2-3-8-11-12

**2.2 Solution:**

As branch coverage testing can’t detect the error mentioned above, condition testing technique can be used in order to reveal the error.

**Q3.**

1. **Solution:**

A test set T for program P with requirement set R is considered adequate if for each requirement r in R there is at least one test case in T that tests the correctness of P with respect to r. When a decision is composed of a compound condition, decision coverage does not imply that each simple condition within a compound condition has taken both values true and false. Hence, we need to construct a test set T with test cases that test the program when all values x, y and z is true and false:

The above test set T is not multiple condition adequate and does not reveal the error.

1. **Solution:**

For the two simple conditions. **x< y** and **x< z**, the four possible combinations of the outcomes of these two simple conditions need to be tested. Hence, Test set T can be enhanced, as shown below, so that it is multiple-condition adequate and does reveal the error:

1. **Solution:**

Yes, it does

**Q4.**

1. **Solution:**

**If one test case is designed for each combination of all the parameters, then there will be: = 3 X 3 X 3 X 5 X 2 = 270** Valid test cases combinations can be designed.

1. **Solution:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TCs | Call Type | Billing Type | Phone Type | Discount Type | Call Purpose |
| T1 | Local | Caller | Mobile (M) | A | Official (O) |
| T2 | Long Distance | Collect | Internet (I) | B | Private (P) |
| T3 | International | Clover | Public (P) | C | Private (P) |
| T4 | Local | Clover | Public (P) | D | Official (O) |
| T5 | Long Distance | Caller | Mobile (M) | E | Private (P) |

1. **Solution:**

**Step #1: Arranging variables and values involved.**

Order the variables so that the one with the most number of values is first and the one with the least is last.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Discount Type | Call Type | Billing Type | Phone Type | Call Purpose |
| A | Local | Caller | Mobile (M) | Official (O) |
| B | Long Distance | Collect | Internet (I) | Private (P) |
| C | International | Clover | Public (P) |  |
| D |  |  |  |  |
| E |  |  |  |  |

**Step #2: Arrange variables to create test suite**

Let’s start filling in the table column by column. Initially table should look something like this. The five values of ***Discount Type*** (variable having highest number of values) should be written three times each (three is the number of values of next highest variable i.e. ***Call Type, Billing Type and Phone Type***).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Discount Type | Call Type | Billing Type | Phone Type | Call Purpose |
| A |  |  |  |  |
| A |  |  |  |  |
| A |  |  |  |  |
| B |  |  |  |  |
| B |  |  |  |  |
| B |  |  |  |  |
| C |  |  |  |  |
| C |  |  |  |  |
| C |  |  |  |  |
| D |  |  |  |  |
| D |  |  |  |  |
| D |  |  |  |  |
| E |  |  |  |  |
| E |  |  |  |  |
| E |  |  |  |  |

The Call Type column has three values. That’s how many times we need to insert the values of the first column, Discount Type.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Discount Type | Call Type | Billing Type | Phone Type | Call Purpose |
| A | Local |  |  |  |
| A | Long Distance |  |  |  |
| A | International |  |  |  |
| B | Local |  |  |  |
| B | Long Distance |  |  |  |
| B | International |  |  |  |
| C | Local |  |  |  |
| C | Long Distance |  |  |  |
| C | International |  |  |  |
| D | Local |  |  |  |
| D | Long Distance |  |  |  |
| D | International |  |  |  |
| E | Local |  |  |  |
| E | Long Distance |  |  |  |
| E | International |  |  |  |

For each set of values in Discount Type column, we put all values of Call Type column. Repeat the same for Billing Type column.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Discount Type | Call Type | Billing Type | Phone Type | Call Purpose |
| A | Local | Caller |  |  |
| A | Long Distance | Collect |  |  |
| A | International | Clover |  |  |
| B | Local | Caller  **Swap** |  |  |
| B | Long Distance | Collect |  |  |
| B | International | Clover |  |  |
| C | Local | Caller |  |  |
| C | Long Distance | Collect |  |  |
| C | International | Clover |  |  |
| D | Local | Caller |  |  |
| D | Long Distance | Collect |  |  |
| D | International | Clover |  |  |
| E | Local | Caller |  |  |
| E | Long Distance | Collect |  |  |
| E | International | Clover |  |  |

We have a Local and Caller, but there’s no Local and Collect, and Local and Clover. The same is true for the other call types; hence, we can swap around the values in the second set in the third column.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Discount Type | Call Type | Billing Type | Phone Type | Call Purpose |
| A | Local | Caller |  |  |
| A | Long Distance | Collect |  |  |
| A | International | Clover |  |  |
| B | Local | Collect |  |  |
| B | Long Distance | Clover |  |  |
| B | International | Caller |  |  |
| C | Local | Clover |  |  |
| C | Long Distance | Caller |  |  |
| C | International | Collect |  |  |
| D | Local | Caller |  |  |
| D | Long Distance | Collect |  |  |
| D | International | Clover |  |  |
| E | Local | Collect |  |  |
| E | Long Distance | Clover |  |  |
| E | International | Caller |  |  |

This looks much better!

We will repeat the same steps for Billing Type and Phone Type and columns

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Discount Type | Call Type | Billing Type | Phone Type | Call Purpose |
| A | Local | Caller | Mobile (M) |  |
| A | Long Distance | Collect | Internet (I) |  |
| A | International | Clover | Public (P) |  |
| B | Local | Collect | Mobile (M) |  |
| B | Long Distance | Clover | Internet (I) |  |
| B | International | Caller | Public (P) |  |
| C | Local | Clover | Mobile (M) |  |
| C | Long Distance | Caller | Internet (I) |  |
| C | International | Collect | Public (P) |  |
| D | Local | Caller | Mobile (M) | **Swap** |
| D | Long Distance | Collect | Internet (I) |  |
| D | International | Clover | Public (P) |  |
| E | Local | Collect | Mobile (M) |  |
| E | Long Distance | Clover | Internet (I) |  |
| E | International | Caller | Public (P) |  |

Swap around the values in the fourth set in the fourth column

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Discount Type | Call Type | Billing Type | Phone Type | Call Purpose |
| A | Local | Caller | Mobile (M) |  |
| A | Long Distance | Collect | Internet (I) |  |
| A | International | Clover | Public (P) |  |
| B | Local | Collect | Mobile (M) |  |
| B | Long Distance | Clover | Internet (I) |  |
| B | International | Caller | Public (P) |  |
| C | Local | Clover | Mobile (M) |  |
| C | Long Distance | Caller | Internet (I) |  |
| C | International | Collect | Public (P) |  |
| D | Local | Caller | Internet (I) |  |
| D | Long Distance | Collect | Public (P) |  |
| D | International | Clover | Mobile (M) |  |
| E | Local | Collect | Public (P) |  |
| E | Long Distance | Clover | Mobile (M) |  |
| E | International | Caller | Internet (I) |  |

At last, fill the last column (Call Purpose) rows as shown in the table below. Hence, the test case for the pairwise testing is shown in the table below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TCs | Discount Type | Call Type | Billing Type | Phone Type | Call Purpose |
| T1 | A | Local | Caller | Mobile (M) | Official (O) |
| T2 | A | Long Distance | Collect | Internet (I) | Private (P) |
| T3 | A | International | Clover | Public (P) | Official (O) |
| T4 | B | Local | Collect | Mobile (M) | Private (P) |
| T5 | B | Long Distance | Clover | Internet (I) | Official (O) |
| T6 | B | International | Caller | Public (P) | Private (P) |
| T7 | C | Local | Clover | Mobile (M) | Official (O) |
| T8 | C | Long Distance | Caller | Internet (I) | Private (P) |
| T9 | C | International | Collect | Public (P) | Official (O) |
| T10 | D | Local | Caller | Internet (I) | Private (P) |
| T11 | D | Long Distance | Collect | Public (P) | Official (O) |
| T12 | D | International | Clover | Mobile (M) | Private (P) |
| T13 | E | Local | Collect | Public (P) | Official (O) |
| T14 | E | Long Distance | Clover | Mobile (M) | Private (P) |
| T15 | E | International | Caller | Internet (I) | Official (O) |

**Step 3: Re-order the variables and their values**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TCs | Call Type | Billing Type | Phone Type | Discount Type | Call Purpose |
| T1 | Local | Caller | Mobile (M) | A | Official (O) |
| T2 | Long Distance | Collect | Internet (I) | A | Private (P) |
| T3 | International | Clover | Public (P) | A | Official (O) |
| T4 | Local | Collect | Mobile (M) | B | Private (P) |
| T5 | Long Distance | Clover | Internet (I) | B | Official (O) |
| T6 | International | Caller | Public (P) | B | Private (P) |
| T7 | Local | Clover | Mobile (M) | C | Official (O) |
| T8 | Long Distance | Caller | Internet (I) | C | Private (P) |
| T9 | International | Collect | Public (P) | C | Official (O) |
| T10 | Local | Caller | Internet (I) | D | Private (P) |
| T11 | Long Distance | Collect | Public (P) | D | Official (O) |
| T12 | International | Clover | Mobile (M) | D | Private (P) |
| T13 | Local | Collect | Public (P) | E | Official (O) |
| T14 | Long Distance | Clover | Mobile (M) | E | Private (P) |
| T15 | International | Caller | Internet (I) | E | Official (O) |

**Q5.**

**Solution:**

As the source code has looping and conditional statements, ***loop testing*** strategy will effectively test the program. It is the testing of resource(s) multiple times under program control. The most important aspect of this test is to ensure that the control loop is executed multiple times and exited when a particular condition is satisfied. The goal of loop testing is to test while-do, repeat-until, (or do-while) and any other loops in a program thoroughly - by trying to ensure that each is executed at minimal, typical, and (if this is defined) maximal values - and to try to ``break'' the program, by trying to have a loop executed with a fewer than minimum, as well as a larger than maximal, number of iterations.

**Q6.**

1. **Solution:** It is a [Integration Testing](http://www.kanoah.com/blog/integration-testing/) approach where top level units are tested first and lower level units are tested step by step after that. i.e. testing takes place from top to down following the control flow of the software system using stubs.

**A**

**C**

**B**

**Step 1:** Test A individually (use stubs for B and C)

**Step 2:** Test A such that it calls B (stub for C)

If an error occurs, we know that the problem is in B or in the interface between A & B

**Step 3:** Test A such that it calls C (stub for B)

If an error occurs, we know that the problem is in C or in the interface between A & C

\* Stubs are used to simulate the activity of components that are not currently tested; (-) may require many stubs

Hence, when a Top-down testing strategy is used the testing will happen as the diagrams shown below:

**Step 1:** Top Module(C1) is Tested with stubs

C1 is tasted with stubs for C2 and C3

**C1**

**C3**

**C2**

**Step 2:** Stub C2 is replaced(integrated) by the actual component (C2) and is tasted using C4 and C7 stubs, and as C1 is going to be the driver

Driver

**C1**

C1, and C2 are tasted

**C3**

**C2**

**C4**

**C5**

**C7**

**C6**

**Step 3:** Stub C4 is replaced(integrated) by the actual component (C4) and is tasted using C6 stub and C2 is used as the driver (Using Depth-First)

**C1**

**C3**

**C2**

Driver

**C4**

**C5**

C4 is tasted using stub C6 and C2 as a driver

**C7**

**C6**

**Step 4:** Stub C3 is replaced(integrated) by the actual component (C3) and is tasted using stub C5 and component C4, and C1 is as a driver (Using Depth-First)

Driver

**C1**

C3 is tasted using stub C5 and component C4 and driver C1

**C3**

**C2**

**C4**

**C5**

C1, C2, C3 and C4 are tasted

**C7**

**C6**

**Step 5:** Stub C5 is replaced(integrated) by the actual component (C5) and is tasted using C7 stub and C3 is used as the driver (Using Depth-First)

**C1**

**C3**

**C2**

Driver

C5 is tasted using stub C7 and C3 as a driver

**C4**

**C5**

C1, C2, C3, C4 and C5 are tasted

**C7**

**C6**

**Step 6:** Stub C7 is replaced(integrated) by the actual component (C7) and C5 will be the driver. Since C7 is does not have stubs.

**C1**

**C3**

**C2**

**C4**

Driver

**C5**

C1, C2, C3, C4, C5 and C7 are tasted

C5 is tasted using stub C7 and C3 as a driver

**C7**

**C6**

**Step 7:** Stub C6 is replaced(integrated) by the actual component (C6) and C4 will be the driver. C6 uses component (module) C7.

**C1**

**C3**

**C2**

**C4**

**C5**

Driver

C1, C2, C3, C4, C5, C6 and C7 are tasted

**C7**

**C6**

1. **Solution:** Bottom-up integration testing is a [Integration Testing](http://www.kanoah.com/blog/integration-testing/) approach where bottom level units are tested first and upper level units step by step after that. i.e. each module at lower levels is tested with higher modules until all modules are tested.

**A**

**C**

**B**

**Step 1:** Test B, C individually (using drivers)

**Step 2:** Test A such that it calls B

If an error occurs, we know that the problem is in A or in the interface between A & B

**Step 3:** Test A such that it calls C (stub for B)

If an error occurs, we know that the problem is in A or in the interface between A & C

(-) Top level components are the most important yet tested last.

Hence, when a Top-down testing strategy is used the testing will happen as the diagrams shown below:

**Step 1:** Component C7 is tested using C5 as driver.

**C1**

**C3**

**C2**

**C4**

Driver

**C5**

C7 is tasted

**C7**

**C6**

**Step 2:** Component C6 is tested using C4 as driver and component C7. Now both C7 and C7 will be tested

**C1**

**C3**

**C2**

**C4**

Driver

**C5**

Cluster 1 containing Both C6 and C7 is tasted

**C7**

**C6**

Cluster 1

**Step 3:** Component C4 is tested using C2 as driver. And component C5 is also tested using C3 as a driver

**C1**

Driver

**C3**

**C2**

Driver

Cluster 1 containing C4, C6 and C7 is tasted

**C4**

Cluster 2 containing C5 and C7 is tasted

**C5**

Cluster 2

Cluster 1

**C6**

**C7**

**Step 3:** Component C2 is tested using C1 as driver. And component C3 is also tested using C1 as a driver

Driver

**C1**

**C3**

Cluster 2 containing C3, C5 and C7 is tasted

Cluster 1 containing C2, C4, C6 and C7 is tasted

**C2**

**C4**

**C5**

**C7**

Cluster 2

Cluster 1

**C6**

**Step 3:** Finally, the main Component C1 is tested combining the two cluster (cluster 1 and cluster 2)

Driver

**C1**

**C3**

The entire system is tested

Cluster 1 containing C1, C2, C3, C4, C5, C6 and C7 is tasted

**C2**

**C4**

**C5**

**C7**

**C6**

Cluster 1

**Q7.**

**Solution:**

**Low High**

**D B C** **A**

**Big bang Integration Testing**

* Is used to test the entire system once all components are successfully unit tested first. i.e. Once all components or modules are integrated simultaneously; testing the whole system will be done. The philosophy is that we have already tested all of the components so why not just throw them all in together and test the lot?
* Fault localization is difficult; hence **fault isolation capability is low**.

**Top-Bottom Integration Testing**

* It is a [Integration Testing](http://www.kanoah.com/blog/integration-testing/) approach where top level units are tested first and lower level units are tested step by step after that. i.e. testing takes place from top to down following the control flow of the software system using stubs.
* Fault localization is easier; hence **fault isolation capability is high**.

**Bottom-up Integration Testing**

* It is a [Integration Testing](http://www.kanoah.com/blog/integration-testing/) approach where bottom level units are tested first and upper level units step by step after that. i.e. each module at lower levels is tested with higher modules until all modules are tested.
* It takes help of Drivers for testing.
* Fault localization is easier (below average in case of procedural programming approach) but since critical modules (at the top level of software architecture) which control the flow of application are tested last, this may lead be prone to defects.

**Sandwich Integration Testing**

* Is an approach to combine top down testing with bottom up testing
* Fault localization is moderate (Average); it depends on how much of the components has been tested using top-down and/or bottom-up approach.