**PROGRAM 1**

Design and develop a program in a language of your choice to solve the triangle problem defined as follows: Accept three integers which are supposed to be the three sides of a triangle and determine if the three values represent an equilateral triangle, isosceles triangle, scalene triangle, or they do not form a triangle at all. Assume that the upper limit for the size of any side is 10. Derive test cases for your program based on boundary-value analysis, execute the test cases and discuss the results.

# REQUIREMENTS:

R1. The system should accept 3 positive integer numbers (a, b, c) which represents 3 sides of the triangle.

R2. Based on the input should determine if a triangle can be formed or not.

R3. If the requirement R2 is satisfied then the system should determine the type of the triangle, which can be

* Equilateral (i.e. all the three sides are equal)
* Isosceles (i.e Two sides are equal)
* Scalene (i.e All the three sides are unequal)

R4. Upper Limit for the size of any side is 10

# DESIGN:

# ALGORITHM:

Step 1: Input a, b & c are three integer values which represent three sides of the triangle.

Step 2: if (a < (b + c)) and (b < (a + c)) and (c < (a + b) then do step 3

Else print not a triangle. do step 6.

Step 3: if (a=b) and (b=c) then

Print triangle formed is equilateral. do step 6.

Step 4: if (a ≠ b) and (a ≠ c) and (b ≠ c) then

Print triangle formed is scalene. do step 6.

Step 5: Print triangle formed is Isosceles.

Step 6: Stop

# PROGRAMCODE:

#include<stdio.h>

int main()

{

int a, b, c;

printf("Enter three sides of the triangle");

scanf("%d%d%d", &a, &b, &c);

if((a > 10) || (b > 10) || (c > 10))

{

printf("Out of range");

exit(0);

}

if((a<b+c)&&(b<a+c)&&(c<a+b))

{

if((a==b)&&(b==c))

{

printf("Equilateral triangle");

}

else if((a!=b)&&(a!=c)&&(b!=c))

{

}

else

}

else

printf("Scalene triangle");

printf("Isosceles triangle");

exit(0);

}

printf("Triangle cannot be formed");

# TESTING:

* 1. Technique used: Boundary value analysis
  2. Test Case design

For BVA problem the test cases generation depends on the output and the constraints of the output. Here we do not take into consideration the constraints on Input domain.

The Triangle problem takes 3 sides as input and checks it for validity, hence n = 3. Since BVA yields (4n + 1) test cases according to single fault assumption theory, hence we can say that the total number of test cases will be (4\*3+1) =12+1=13.

The maximum limit of each side a, b, and c of the triangle is 10 units according to requirement R4. So a, b and c lies between

0≤a≤10

0≤b≤10

0≤c≤10

# Equivalence classes for a:

E1: Values less than 1

E2: Values in the range

E3: Values greater than 10

# Equivalence classes for b:

E4: Values less than 1 E5: Values in the range

E6: Values greater than10

# Equivalence classes for c:

E7: Values less than 1

E8: Values in the range

E9: Values greater than 10

From the above equivalence classes, we can derive the following test cases using boundary value analysis approach.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **TC**  **Id** | **Test Case**  **Description** | **Input Data** | | | **Expected**  **Output** | **Actual**  **Output** | **Status** |
| **a** | **b** | **c** |
| 1 | For A input is not  given | X | 3 | 6 | Not a Triangle |  |  |
| 2 | For B input is not  given | 5 | X | 4 | Not a Triangle |  |  |
| 3 | For C input is not given | 4 | 7 | X | Not a Triangle |  |  |
| 4 | Input of C is in  negative(-) | 5 | 5 | -1 | Not a Triangle |  |  |
| 5 | Two sides are same one side is given different input | 5 | 5 | 1 | Isosceles |  |  |
| 6 | All Sides of inputs are equal | 5 | 5 | 5 | Equilateral |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 7 | Two sides are same one sideis given differentinput | 5 | 5 | 9 | Isosceles |  |  |
| 8 | The input of C is  out of range (i.e., range is <10) | 5 | 5 | 10 | Not a Triangle |  |  |
| 9 | Two sides are same one sideis given differentinput | 5 | 1 | 5 | Isosceles |  |  |
| 10 | Two sides are same one sideis given differentinput | 5 | 2 | 5 | Isosceles |  |  |
| 11 | Two sides are same one sideis given differentinput | 5 | 9 | 5 | Isosceles |  |  |
| 12 | Two sides are same one sideis given differentinput | 5 | 10 | 5 | Not a Triangle |  |  |
| 13 | Two sides are same one sideis given differentinput | 1 | 5 | 5 | Isosceles |  |  |
| 14 | Two sides are same one sideis given differentinput | 2 | 5 | 5 | Isosceles |  |  |
| 15 | Two sides are same one sideis given differentinput | 9 | 5 | 5 | Isosceles |  |  |

# Table-1: Test case for Triangle Problem

**EXECUTION:**

Execute the program and test the test cases in Table-1 against program and complete the table with for Actual output column and Status column

# Test Report:

1. Noof TC‟sExecuted:
2. No of DefectsRaised:
3. Noof TC‟sPass:
4. Noof TC‟sFailed:

# PROGRAM 2

Design, develop, code and run the program in any suitable language to implement the NextDate function. Analyze it from the perspective of boundary value testing, derive different test cases, execute these test cases and discuss the testresults.

# REQUIREMENT SPECIFICATION

Problem Definition: "Next Date" is a function consisting of three variables like: month, date and year. It returns the date of next day as output. It reads current date as input date.

The constraints are

C1: 1 ≤ month ≤ 12

C2: 1 ≤ day ≤ 31

C3: 1812 ≤ year ≤ 2012.

If any one condition out of C1, C2 or C3 fails, then this function produces an output "value of month not in the range 1...12".

Since many combinations of dates can exist, hence we can simply displays one message for this function: "Invalid Input Date".

A very common and popular problem occurs if the year is a leap year. We have taken into consideration that there are 31 days in a month. But what happens if a month has 30 days or even 29 or 28days?

A year is called as a leap year if it is divisible by 4, unless it is a century year. Century years are leap years only if they are multiples of 400. So, 1992, 1996 and 2000 are leap years while 1900 is not a leapyear.

# DESIGN:

# ALGORITHM

Step 1: Input date in format DD.MM.YYYY

Step 2: if MM is 01, 03, 05,07,08,10 do Step3else Step6

Step 3:if DD < 31 then do Step 4 else if DD=31 do Step 5 else output(InvalidDate);

Step 4: tomorrowday=DD+1 goto STEP18

Step 5: tomorrowday=1; tomorrowmonth=month + 1 goto Step 18

Step 6: if MM is 04, 06, 09, 11 do Step 7

Step 7: if DD<30 then do Step 4

else if DD=30 do Step 5

else output(InvalidDate);

Step 8: if MM is 12

Step 9: if DD<31 then Step 4 else Step 10

Step 10: tomorrowday=1, tommorowmonth=1, tommorowyear=YYYY+1; goto Step 18

Step 11: if MM is 2

Step 12: if DD<28 do Step 4

else do Step 13

Step 13: if DD=28 & YYYY is a leap do Step 14 elseStep 15

Step 14: tommorowday=29 goto Step 18

Step 15: tommorowday=1, tomorrowmonth=3, goto Step 18;

Step 16: if DD=29 then do Step 15 else Step 17

Step 17: output(“Cannot have feb”, DD); Step 19

Step 18: output(tomorrowday, tomorrowmonth, tomorrowyear);

Step 19: exit

# PROGRAMCODE:

#include<stdio.h>

main( )

{

int month[12]={31,28,31,30,31,30,31,31,30,31,30,31};

int d,m,y,nd,nm,ny,ndays;

printf("enter the date,month,year");

scanf("%d%d%d",&d,&m,&y);

ndays=month[m-1];

if(y<=1812 && y>2012)

{

printf("Invalid Input Year"); exit(0);

}

if(d<=0 || d>ndays)

{

printf("Invalid Input Day"); exit(0);

}

if(m<1 && m>12)

{

printf("Invalid Input Month"); exit(0);

}

if(m==2)

{

if(y%100==0)

{

if(y%400==0)

ndays=29;

}

else

if(y%4==0)

ndays=29;

}

nd = d + 1;

nm = m;

ny =y;

if(nd>ndays)

{

nd = 1;

nm++;

}

if(nm>12)

{

nm = 1;

ny++;

}

printf("\n Given date is %d:%d:%d",d,m,y);

printf("\n Next day‟s date is %d:%d:%d",nd,nm,ny);

}

# TESTING

Technique used: **Boundary value analysis**

“Boundaryvalueanalysis”testingtechniqueisusedtoidentifyerrorsatboundariesratherthanfinding those exists in center of inputdomain.

Boundary value analysis is a next part of Equivalence partitioning for designing test cases where test cases are selected at the edges of the equivalence classes.

# BVA Procedure

1. Partition the input domain using unidimensional partitioning. This leads to as many partitions as there are input variables. Alternately, a single partition of an input domain can be created using multidimensional partitioning. We will generate several sub-domains in thisstep.
2. Identify the boundaries for each partition. Boundaries may also be identified using special relationships amongst theinputs.
3. Select test data such that each boundary value occurs in at least one testinput.

**BVA: Example:** Create equivalence classes

Assuming that an item code must be in the range 99...999 and quantity in the range 1...100.

# Equivalence classes for code:

E1: Values less than 99 E2: Values in the range

E3: Values greater than 999

# Equivalence classes for qty:

E4: Values less than 1 E5: Values in the range

E6: Values greater than 100

# BVA: Example: Identify boundaries









Equivalence classes and boundaries for findPrice. Boundaries are indicated with an x. Points near the boundary are marked \*.

# TEST CASE DESIGN

The Next Date program takes date as input and checks it for validity. If it is valid, it returns the next date as its output. Here we have three inputs for the program, hence n = 3.

Since BVA yields (4n + 1) test cases according to single fault assumption theory, hence we can say that the total number of test cases will be (4\*3+1) = 12 + 1 = 13.

The boundary value test cases can be generated by using following constraints

C1: 1 ≤MM ≤12

C2: 1 ≤ DD ≤31

C3: 1812 ≤ YYYY ≤ 2012

Here from these constraints we can extract the test cases using the values of MM, DD, and YYYY. The following equivalence classes can be generated for each variable.

# Equivalence classes for MM:

E1: Values less than 1

E2: Values in the range

E3: Values greater than 12

# Equivalence classes for DD:

E4: Values less than 1

E5: Values in the range

E6: Values greater than 31

# Equivalence classes for YYYY:

E7: Values less than 1812

E8: Values in the range

E9: Values greater than 2012

From the above equivalence classes, we can derive the following test cases using boundary value analysis approach.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **TC**  **Id** | **TestCaseDescription** | **Input Data** | | | **Expected Output** | **Actual Output** | **Status** |
| **MM** | **DD** | **YYYY** |
| 1 | Testing for Invalid months with  character is typed | Aa | 15 | 1900 | Invalid Input  Month |  |  |
| 2 | Testing for Invalid Day with character is typed | 06 | Dd | 1901 | Invalid Input  Day |  |  |
| 3 | Testing for Invalid Year with  character is typed | 06 | 15 | 196y | Invalid Input  Year |  |  |
| 4 | Testing for Invalid Day, day with 00 | 03 | 00 | 2000 | Invalid Input  Day |  |  |
| 5 | Testing for Valid input changing the day with in the month. | 03 | 30 | 2000 | 03/31/2000 |  |  |
| 6 | Testing for Valid input changing the day within the month | 03 | 02 | 2000 | 03/03/2000 |  |  |
| 7 | Testing for invalid Day, day with 32 | 03 | 32 | 2000 | Invalid Input  Day |  |  |
| 8 | Testing for invalid Day, month with 00 | 00 | 15 | 2000 | Invalid Input  Month |  |  |
| 9 | Testing for Valid input changing  t he day with in the month, MM=11 DD=15 | 11 | 15 | 2000 | 11/16/2000 |  |  |
| 10 | Testing for Valid Input changing the day with in the month, MM=02  DD=15 | 02 | 15 | 2000 | 02/16/2000 |  |  |
| 11 | Testing for Invalid Month, month with 13 | 13 | 15 | 2000 | Invalid Input  Month |  |  |
| 12 | Testing for Invalid year, year  should >=1812 | 03 | 15 | 1811 | Invalid Input  Year |  |  |
| 13 | Testing for Valid input changing the day within the month MM=03, DD=15 YYYY=2011 | 03 | 15 | 2011 | 03/16/2011 |  |  |
| 14 | Testing for Valid input changing the day with in the month, MM=03, DD=15, YYYY=1813 | 03 | 15 | 1813 | 03/16/1813 |  |  |
| 15 | Testing for invalid year, year should <= 2012 | 03 | 15 | 2013 | Invalid Input Year |  |  |

This is how we can apply BVA technique to create test cases for our NextDate Problem.

# EXECUTION

Execute the program and test the test cases in Table-1 against program and complete the table with for Actual output column and Status column

# Test Report:

* + 1. Noof TC‟sExecuted:
    2. No of DefectsRaised:
    3. Noof TC‟sPass:
    4. Noof TC‟sFailed:

# PROGRAM 3

Design and develop a program in a language of your choice to solve the triangle problem defined as follows: Accept three integers which are supposed to be the three sides of a triangle and determine if the three values represent an equilateral triangle, isosceles triangle, scalene triangle, or they do not form a triangle at all. Assume that the upper limit for the size of any side is 10. Derive test cases for your program based on equivalence class partitioning, execute the test cases and discuss the results.

# REQUIREMENTS

R1. The system should accept 3 positive integer numbers (a, b, c) which represents 3 sides of the triangle.

R2. Based on the input should determine if a triangle can be formed or not.

R3. If the requirement R2 is satisfied then the system should determine the type of the triangle, which can be

* Equilateral (i.e. all the three sides are equal)
* Isosceles (i.e. two sides are equal)
* Scalene (i.e. All the three sides are unequal)

R4. Upper Limit for the size of any side is 10

# DESIGN

Form the given requirements we can draw the following conditions:

C1: a<b+c?

C2: b<a+c?

C3: c<a+b?

C4:a=b?

C5: a=c?

C6:b=c?

According to the property of the triangle, if any one of the three conditions C1, C2 and C3 are not satisfied then triangle cannot be constructed. So only when C1, C2 and C3 are true the triangle can be formed, then depending on conditions C4, C5 and C6 we can decide what type of triangle will be formed. (i.e requirementR3).

# ALGORITHM:

Step 1: Input a, b & c i.e three integer values which represent three sides of the triangle. Step 2: if (a < (b + c)) and (b < (a + c)) and (c < (a + b) then do step 3

Else print not a triangle. do step 6.

Step 3: if (a=b) and (b=c) then

Print triangle formed is equilateral. do step 6.

Step 4: if (a ≠ b) and (a ≠ c) and (b ≠ c) then

Print triangle formed is scalene. do step 6.

Step 5: Print triangle formed is Isosceles.

Step 6: Stop

# PROGRAMCODE

#include<stdio.h>

#include<ctype.h>

#include<process.h>

int main()

{

int a, b, c;

clrscr();

printf("Enter three sides of the triangle");

scanf("%d%d%d", &a, &b, &c);

if((a > 10) || (b > 10) || (c > 10))

{

printf("Out of range");

getch();

exit(0);

}

if((a<b+c)&&(b<a+c)&&(c<a+b))

{

if((a==b)&&(b==c))

{

printf("Equilateral triangle");

}

else if((a!=b)&&(a!=c)&&(b!=c))

{

}

else

{

}

else

printf("Scalene triangle"); printf("Isosceles triangle");

}

getch();

printf("triangle cannot be formed");

return 0;

}

# TESTING

1. Technique used: Equivalence class partitioning
2. Test Case design

Equivalence class partitioning technique focus on the Input domain, we can obtain a richer set of test cases. What are some of the possibilities for the three integers, a, b, and c? They can all be equal, exactly one pair can be equal.

The maximum limit of each side a, b, and c of the triangle is 10 units according to requirement R4. So a, b and c liesbetween

0≤a≤10

0≤b≤10

0≤c≤10

# First Attempt

Weak normal equivalence class: In the problem statement, we note that four possible outputs can occur: Not a Triangle, Scalene, Isosceles and Equilateral. We can use these to identify output (range) equivalence classes asfollows:

R1= {<a,b,c>: the triangle with sides a, b, and c is equilateral}

R2= {<a,b,c>: the triangle with sides a, b, and c is isosceles}

R3= {<a,b,c>: the triangle with sides a, b, and c is scalene}

R4= {<a,b,c>: sides a, b, and c do not form a triangle}

Four weak normal equivalence class test cases, chosen arbitrarily from each class, and invalid values for weak robust equivalence class test cases are as follows.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **TC**  **Id** | **Test Case Description** | **Input Data** | | | **Expected Output** | **Actual Output** | **Status** |
| **a** | **b** | **c** |
| 1 | WN1 | 5 | 5 | 5 | Equilateral |  |  |
| 2 | WN2 | 2 | 2 | 3 | Isosceles |  |  |
| 3 | WN3 | 3 | 4 | 5 | Scalene |  |  |
| 4 | WN4 | 4 | 1 | 2 | Not a Triangle |  |  |
| 5 | WR1 | -1 | 5 | 5 | Value of a is not in the range of permitted values |  |  |
| 6 | WR2 | 5 | -1 | 5 | Value of b is not in the range of permitted values |  |  |
| 7 | WR3 | 5 | 5 | -1 | Value of c is not in the range of permitted values |  |  |
| 8 | WR4 | 11 | 5 | 5 | Value of a is not in the range of permitted values |  |  |
| 9 | WR5 | 5 | 11 | 5 | Value of b is not in the range ofpermitted values |  |  |
| 10 | WR6 | 5 | 5 | 11 | Value of c is not in the range ofpermitted values |  |  |

# Table-1: Weak Normal and Weak Robust Test case for Triangle Problem

**Second attempt**

The strong normal equivalence class test cases can be generated by using following possibilities:

D1 = {<a, b, c>: a=b=c}

D2 = {<a, b, c>: a=b, a≠ c}

D3 = {<a, b, c>: a=c, a≠ b}

D4 = {<a, b, c>: b=c, a≠b}

D5 = {<a, b, c>: a≠ b, a≠ c, b≠ c}

D6 = {<a, b, c>: a≥ b+c}

D7 = {<a, b, c>: b≥ a+ c}

D8 = {<a, b, c>: c≥ a+b}

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **TC**  **Id** | **Test Case**  **Description** | **Input Data** | | | **Expected Output** | **Actual**  **Output** | **Status** |
| **a** | **b** | **c** |
| 1 | SR1 | -1 | 5 | 5 | Value of a is not in the range ofpermitted  values |  |  |
| 2 | SR 2 | 5 | -1 | 5 | Value of b is not in  the range of permitted values |  |  |
| 3 | SR3 | 5 | 5 | -1 | Value of c is not in  the range of permitted values |  |  |
| 4 | SR4 | -1 | -1 | 5 | Value of a, b is not in the range of permittedvalues |  |  |
| 5 | SR5 | 5 | -1 | -1 | Value of b, c is not in the range of permitted values |  |  |
| 6 | SR6 | -1 | 5 | -1 | Value of a, c is not in the range of permitted  values |  |  |
| 7 | SR7 | -1 | -1 | -1 | Value of a, b, c is not  in the range of permitted values |  |  |

# Table-2: Strong Robust Test case for Triangle Problem

**EXECUTION:**

Execute the program and test the test cases in Table-1 and Table-2 against program and complete the table with for Actual output column and Status column

# Test Report:

* + 1. Noof TC‟sExecuted:
    2. No of DefectsRaised:
    3. Noof TC‟sPass:
    4. Noof TC‟sFailed:

# REFERENCES

1. RequirementSpecification
2. Assumptions

# PROGRAM 4

Design,develop,codeandruntheprograminanysuitablelanguagetosolvethecommissionproblem. Analyze it from the perspective of equivalence class testing, derive different test cases, execute these test cases and discuss the testresults.

# REQUIREMENTSPECIFICATION

Problem Definition: The Commission Problem includes a sales person in the former Arizona Territory sold rifle locks, stocks and barrels made by a gunsmith in Missouri. Cost includes

Locks- $45

Stocks- $30

Barrels- $25

The salesperson had to sell at least one complete rifle per month and production limitswere such that the most the salesperson could sell in a month was 70 locks, 80 stocks and 90barrels.

After each town visit, the sales person sent a telegram to the Missouri gunsmith with the number of locks, stocks and barrels sold in the town. At the end of the month, the salesperson sent a very short telegram showing --1 lock sold. The gunsmith then knew the sales for the month were completeandcomputedthesalesperson’s commissionasfollows:

On sales up to(and including) $1000 = 10%

On the sales up to(and includes) $1800 = 15%

On the sales in excess of $1800 = 20%

The commission program produces a monthly sales report that gave the total number of locks, stocks and barrels sold, the salesperson’s total dollar sales and finally the commission

# DESIGN

# ALGORITHM

Step 1: Define lockPrice=45.0, stockPrice=30.0, barrelPrice=25.0

Step 2: Input locks

Step 3: while(locks!=-1) „input device uses -1 to indicate end of data goto Step 12

Step 4:input (stocks, barrels)

Step 5: compute lockSales, stockSales, barrelSales and sales

Step 6: output(“Total sales:” sales)

Step 7: if (sales > 1800.0) goto Step 8 else goto Step 9

Step 8: commission=0.10\*1000.0;

commission=commission+0.15 \* 800.0;

commission = commission + 0.20 \* (sales-1800.0)

Step 9: if (sales > 1000.0) goto Step 10 else goto Step 11

Step 10: commission=0.10\* 1000.0; commission=commission + 0.15 \* (sales-1000.0)

Step 11: Output(“Commission is $”, commission)

Step 12: exit

# PROGRAMCODE:

#include<stdio.h>

int main()

{

int locks, stocks, barrels, t\_sales, flag = 0; float commission;

printf("Enter the total number of locks"); scanf(%d",&locks);

if ((locks <= 0) || (locks > 70)) flag = 1;

printf("Enter the total number of stocks"); scanf("%d",&stocks);

if ((stocks <= 0) || (stocks > 80))

{

flag = 1;

}

printf("Enter the total number of barrelss"); scanf("%d",&barrels);

if ((barrels <= 0) || (barrels > 90))

{

flag = 1;

}

if (flag == 1)

{

printf("invalidinput");

exit(0);

}

t\_sales = (locks \* 45) + (stocks \* 30) + (barrels \* 25); if (t\_sales <= 1000)

{

commission = 0.10 \* t\_sales;

}

else if (t\_sales < 1800)

{

commission = 0.10 \* 1000;

commission = commission + (0.15 \* (t\_sales - 1000));

}

else

{

commission = 0.10 \* 1000;

commission = commission + (0.15 \* 800);

commission = commission + (0.20 \* (t\_sales - 1800));

}

printf("The total sales is %d\n The commission is %f",t\_sales, commission);

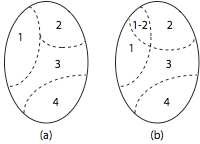
return;

}

# TESTING

Technique used: **Equivalence Class testing**

Test selection using equivalence partitioning allows a tester to subdivide the input domain into a relatively small number of sub-domains, say N>1, as shown.

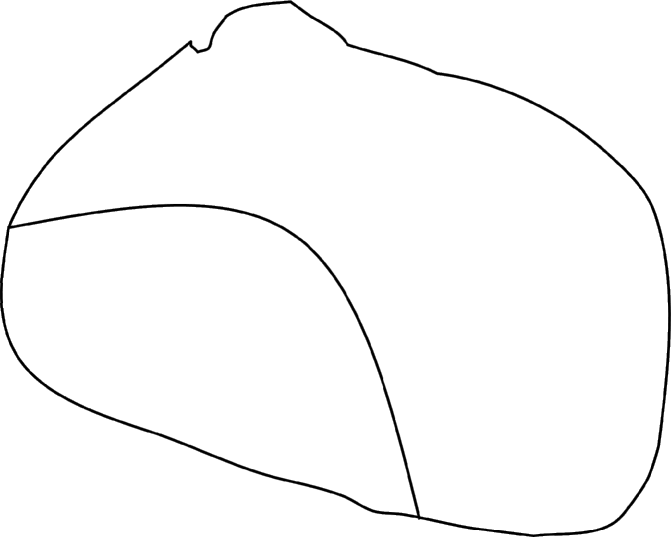


In strict mathematical terms, the sub-domains by definition are disjoint. The four subsets shown in (a) constitute a partition of the input domain while the subsets in (b) are not. Each subset is known as an equivalence class.

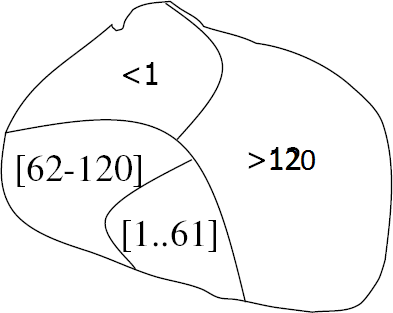
# Example:

Consider an application A that takes an integer denoted by age as input. Let us suppose that the only legal values of age are in the range [1..120]. The set of input values is now divided into a set E containing all integers in the range [1..120] and a set U containing the remaining integers

Further, assume that the application is required to process all values in the range [1..61] in accordance with requirement R1 and those in the range [62..120] according to requirement R2. Thus E is further subdivided into two regions depending on the expectedbehavior.



Similarly, it is expected that all invalid inputs less than or equal to 1 are to be treated in one way while all greater than 120 are to be treated differently. This leads to a subdivision of U into two categories.



Tests selected using the equivalence partitioning technique aim at targeting faults in the application under test with respect to inputs in any of the four regions, i.e. two regions containing expected inputs and two regions containing the unexpectedinputs.

It is expected that any single test selected from the range [1...61] will reveal any fault with respect to R1. Similarly, any test selected from the region [62...120] will reveal any fault with respect to R2. A similar expectation applies to the two regions containing the unexpected inputs.

# TEST CASE DESIGN

The input domain of the commission problem is naturally partitioned by the limits on locks, stocks and barrels. These equivalence classes are exactly those that would also be identified by traditional equivalence class testing. The first class is the valid input; the other two are invalid. The input domain equivalence classes lead to very unsatisfactory sets of test cases. Equivalence classes defined on the output range of the commission function will be animprovement.

The valid classes of the input variables are:

L1 = {locks: 1≤locks≤70}

L2 = {locks = -1} (occurs if locks = -1 is used to control input iteration)

S1 = {stocks:1≤stocks≤80}

B1 = {barrels: 1≤barrels≤90}

The corresponding invalid classes of the input variables are:

L3 = {locks: locks = 0 OR locks < -1}

L4 = {locks: locks > 70}

S2 = {stocks: stocks<1}

S3 = {stocks: stocks>80}

B2 = {barrels: barrels<1}

B3 = {barrels:barrels>90}

One problem occurs, however. The variables lock are also used as a sentinel to indicate no more telegrams. When a value of -1 is given for locks, the while loop terminates, and the values of totallocks, totalstocks and totalbarrels are used to compute sales, and then commission. Expect for the names of the variables and the interval endpoint values, this is identical to our first version of the NextDate function. therefore, we will have exactly one week normal equivalence class test case – and again, it is identical to the strong normal equivalence class test case. Note that the case for locks = -1 just terminates the iteration.

# First attempt

We will have eight weak robust test cases.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **TC**  **Id** | **Test Case Description** | **Input Data** | | | **Sales** | **ExpectedOutput (Commission)** | **Actual Output** | **Status** |
| **Locks** | **Stocks** | **Barrels** |
| 1 | WR1 | 10 | 10 | 10 | $100 | 10 |  |  |
| 2 | WR2 | -1 | 40 | 45 | Program terminates | Program terminates |  |  |
| 3 | WR3 | -2 | 40 | 45 | Values of locks not in therange  1...70 | Values of locks not in the range 1...70 |  |  |
| 4 | WR4 | 71 | 40 | 45 | Values of locks not in the range1...70 | Values of locks not in the range 1...70 |  |  |
| 5 | WR5 | 35 | -1 | 45 | Values of stocks not in therange  1...80 | Values of stocks not in the range 1...80 |  |  |
| 6 | WR6 | 35 | 81 | 45 | Values of stocks not in therange  1...80 | Values of stocks not in the range 1...80 |  |  |
| 7 | WR7 | 10 | 9 | 10 | 970 | 97 |  |  |
| 8 | WR8 | 9 | 10 | 10 | 955 | 95.5 |  |  |

# Second attempt:

Finally, a corner of the cube will be in 3 space of the additional strong robust equivalence class test cases:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **TC**  **Id** | **Test Case Description** | **Input Data** | | | **Sales** | **Expected**  **Output (Commission)** | **Actual Output** | **Status** |
| **Locks** | **Stocks** | **Barrels** |
| 1 | SR1 | -2 | 40 | 45 | Values of locks not in the range  1...70 | Values of locks not in the range 1...70 |  |  |
| 2 | SR2 | 35 | -1 | 45 | Values of stocks not in the range 1...80 | Values of stocks not in the range 1...80 |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 3 | SR3 | 35 | 40 | -2 | Values of barrels not in the range 1...90 | Values of barrels not in the range 1...90 |  |  |
| 4 | SR4 | -2 | -1 | 45 | Values of locks not in the range  1...70  Values of stocks not in the range  1...80 | Values of locks not in the range 1...70 Values of stocks not in the range1...80 |  |  |
| 5 | SR5 | -2 | 40 | -1 | Values of locks not in the range 1...70 Values of barrels not in the range 1...90 | Values of locks not in the range 1...70 Values of barrels not in the range 1...90 |  |  |
| 6 | SR6 | 35 | -1 | -1 | Values of stocks not in the range  1...80  Values of barrels not in the range  1...90 | Values of stocks not in the range 1...80 Values of barrels not in the range 1...90 |  |  |
| 7 | SR7 | -2 | -1 | -1 | Values of locks not in the range 1...70  Values of stocks not in the range  1...80  Values of barrels not in the range 1...90 | Values of locks not in the range 1...70 Values of stocks not in the range 1...80  Values of barrels not in the range 1...90 |  |  |

# EXECUTIONS

Execute the program and test the test cases in Table-1 against program and complete the table with for Actual output column and Status column

# Test Report:

* + 1. Noof TC‟sExecuted:
    2. No of DefectsRaised:
    3. Noof TC‟sPass:
    4. Noof TC‟sFailed:

# REFERENCES

1. RequirementSpecification
2. Assumptions

# PROGRAM 5

Design, develop, code and run the program in any suitable language to solve the commission problem. Analyze it from the perspective of decision table-based testing, derive different test cases, Execute these test cases and discuss the test results.

# REQUIREMENTS:

R1: The system should read the number of Locks, Stocks and Barrels sold in a month.

(i.e 1≤ Locks≤ 70)

(i.e 1 ≤ Stocks ≤ 80)

(i.e 1 ≤ Barrels ≤ 90)

R2: If R1 is satisfied the system should compute the salesperson’s commission depending on the total number of Locks, Stocks & Barrels sold else it should display suitable error message. Following is the percentage of commission for the sales done:

10% on sales up to (and including) $1000

15% on next $800

20% on any sales in excess of $1800

Also, the system should compute the total dollar sales. The system should output salespersons total dollar sales, and his commission.

# DESIGN:

Form the given requirements we can draw the following conditions:

C1:1≤locks≤70? Locks = -1? (occurs if locks = -1 is used to control input iteration)

C2:1≤stocks≤80?

C3: 1≤barrels≤90?

C4: sales>1800?

Here C1 can be expanded as:

C1a: 1≤locks

C1b: locks≤70

C5: sales>1000?

C6: sales≤1000?

# ALGORITHM

Step 1: Input 3 integer numbers which represents number of Locks, Stocks and Barrels sold.

Step 2: compute the total sales = (Number of Locks sold \* 45) +

(Number of Stocks sold \*30) + (Number of Barrels sold \*25)

Step 3: if a totals sale in dollars is less than or equal to $1000 then commission = 0.10\* total Sales do step 6

Step 4: else if total sale is less than $1800 then commission1 = 0.10\* 1000

commission = commission1 + (0.15 \* (total sales–1000)) do step6

Step 5: else commission1 = 0.10\*1000

commission2 = commission1 + (0.15 \* 800))

commission = commission2 + (0.20 \* (total sales – 1800)) do step6

Step 6: Print commission.

Step 7: Stop

# PROGRAMCODE:

#include<stdio.h>

int main()

{

int locks, stocks, barrels, t\_sales, flag = 0; float commission;

printf("Enter the total number of locks"); scanf("%d",&locks);

if ((locks <= 0) || (locks > 70))

{

flag = 1;

}

printf("Enter the total number of stocks"); scanf("%d",&stocks);

if ((stocks <= 0) || (stocks > 80))

{

flag = 1;

}

printf("Enter the total number of barrelss"); scanf("%d",&barrels);

if ((barrels <= 0) || (barrels > 90))

{

flag = 1;

}

if (flag == 1)

{

printf("invalidinput"); getch();

exit(0);

}

t\_sales = (locks \* 45) + (stocks \* 30) + (barrels \* 25); if (t\_sales <= 1000)

{

commission = 0.10 \* t\_sales;

}

else if (t\_sales < 1800)

{

}

else

{

}

commission = 0.10 \* 1000;

commission = commission + (0.15 \* (t\_sales - 1000));

commission = 0.10 \* 1000;

commission = commission + (0.15 \* 800);

commission = commission + (0.20 \* (t\_sales - 1800));

printf("The total sales is %d \n The commission is %f",t\_sales, commission);

return;

}

# TESTING

Technique Used: **Decision Table Approach**

The decision table is given below

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Conditions** | **Condition Entries (Rules)** | | | | | |
| **C1:** 1≤locks≤70? | F | T | T | T | T | T |
| **C2:** 1≤stocks≤80? | -- | F | T | T | T | T |
| **C3:** 1≤barrels≤90? | -- | -- | F | T | T | T |
| **C4: s**ales>1800? | -- | -- | -- | T | F | F |
| **C5: s**ales>1000? | -- | -- | -- | -- | T | F |
| **C6:** sales≤1000? | -- | -- | -- | -- | -- | T |
| **Actions** | **Action Entries** | | | | | |
| a1: com1 = 0.10\*Sales |  |  |  |  |  | X |
| a2:com2= com1+0.15\*(sales-1000) |  |  |  |  | X |  |
| a3:com3 = com2+0.20\*(sales-1800) |  |  |  | X |  |  |
| a4: Out of Range | X | X | X |  |  |  |

Using the decision table we get 6 functional test cases: 3 cases out of range, 1 case each for sales greater than $1800, sales greater than $1000, sales less than or equal to $1000.

# Test Cases Using Decision Table Approach:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **TC ID** | **Test Case Description** | **Locks** | **Stocks** | **Barrels** | **Expected Output** | | **Actual Output** | **Status** |
| 1 | Testing for Requirement 1 Condition 1 (C1) | -2 | 40 | 45 | Out of Range | |  |  |
| 2 | Testing for Requirement 1 Condition 1 (C1) | 90 | 40 | 45 | Out of Range | |  |  |
| 3 | Testing for Requirement 1  Condition 2 (C2) | 35 | -3 | 45 | Out of Range | |  |  |
| 4 | Testing for Requirement 1 Condition 2 (C2) | 35 | 100 | 45 | Out of Range | |  |  |
| 5 | Testing for Requirement 1  Condition 3 (C3) | 35 | 40 | -10 | Out of Range | |  |  |
| 6 | Testing for Requirement 1  Condition 3 (C3) | 35 | 40 | 150 | Out of Range | |  |  |
| 7 | Testing for Requirement 2 | 5 | 5 | 5 | 500 | a1:50 |  |  |
| 8 | Testing for Requirement 2 | 15 | 15 | 15 | 1500 | a2: 175 |  |  |
| 9 | Testing for Requirement 2 | 25 | 25 | 25 | 2500 | a3: 360 |  |  |

**EXECUTION:**

Execute the program against the designed test cases and complete the table for Actual output column and status column.

# TESTREPORT:

* + 1. Noof TC‟sExecuted:
    2. No of DefectsRaised:
    3. Noof TC‟sPass:
    4. Noof TC‟sFailed:

The commission problem is notwell served by a decision table analysis because it has very little decisional. Because the variables in the equivalence classes are truly independent, no impossible rules will occur in a decision table in which condition correspond to the equivalenceclasses.

# REFERENCES:

1. RequirementSpecification
2. Assumptions

# PROGRAM 7

Design and develop a program in a language of your choice to solve the triangle problem defined as follows: Accept three integers which are supposed to be the three sides of a triangle and determine if the three values represent an equilateral triangle, isosceles triangle, scalene triangle, or they do not form a triangle at all. Derive test cases for your program based on decision-table approach, execute the test cases and discuss theresults.

# REQUIREMENTS:

R1. The system should accept 3 positive integer numbers (a,b, c) which represents 3sides of the triangle. Based on the input it should determineif a triangle can be formed or not.

R2. If the requirement R1 is satisfied then the system should determine the type of the triangle, which can be

* Equilateral (i.e. all the three sides areequal)
* Isosceles (i.e Two sides areequal)
* Scalene (i.e All the three sides areunequal)

else suitable error message should be displayed. Here we assume that user gives three positive integer numbers as input.

# DESIGN:

Form the given requirements we can draw the following conditions: C1: a<b+c?

C2: <a+c? C3: <a+b? C4:a=b?

C5: =c? C6:=c?

According to the property of the triangle, if any one of the three conditions C1, C2 and C3 are not satisfied then triangle cannot be constructed. So only when C1, C2 and C3 are true the triangle can be formed, then depending on conditions C4, C5 and C6 we can decide what typeoftrianglewill be formed. (i.e requirementR2).

# ALGORITHM:

Step 1: Input a, b & c i.e three integer values which represent three sides of the triangle.

Step 2: if (a < (b + c)) and (b < (a + c)) and (c < (a + b) then do step 3 else print not a triangle. do step 6.

Step 3: if (a=b) and (b=c) then

Print triangle formed is equilateral. do step 6.

Step 4: if (a ≠ b) and (a ≠ c) and (b ≠ c) then

Print triangle formed is scalene. do step 6.

Step 5: Print triangle formed is Isosceles. Step 6: Stop

# PROGRAM CODE:

#include<stdio.h>

#include<ctype.h>

#include<process.h>

int main()

{

int a, b, c;

clrscr();

printf("Enter three sides of the triangle"); scanf("%d%d%d", &a, &b, &c); if((a<b+c)&&(b<a+c)&&(c<a+b))

{

if((a==b)&&(b==c))

{

printf("Equilateral triangle");

}

else if((a!=b)&&(a!=c)&&(b!=c))

{

}

else

{

}

else

printf("Scalene triangle");

printf(“Isosceles Triangle”);

}

getch();

printf("Triangle cannot be formed");

return 0;

}

# TESTING:

**Technique Used: Decision Table Approach**

Decision Table-Based Testing hasbeenaroundsincetheearly1960‟s; itisusedtodepictcomplex logical relationships between input data. A Decision Table is the method used to build a complete set of test cases without using the internal structure of the program in question. In order to create test cases, we use a table to contain the input and output values of a program.

The decision table is as given below:

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Conditions** | **Condition Entries (Rules)** | | | | | | | | | | |
| R 1 | R 2 | R 3 | R 4 | R 5 | R 6 | R 7 | R 8 | R 9 | R10 | R11 |
| C1: a<b+c? | F | T | T | T | T | T | T | T | T | T | T |
| C2: b<a+c? | -- | F | T | T | T | T | T | T | T | T | T |
| C3: c<a+b? | -- | -- | F | T | T | T | T | T | T | T | T |
| C4: a=b? | -- | -- | -- | F | T | T | T | F | F | F | T |
| C5: a=c? | -- | -- | -- | T | F | T | F | T | F | F | T |
| C6: b=c? | -- | -- | -- | T | T | F | F | F | T | F | T |
| **Actions** | **Action Entries** | | | | | | | | | | |
| a1: Not a Triangle | X | X | X |  |  |  |  |  |  |  |  |
| a2: Scalene |  |  |  |  |  |  |  |  |  | X |  |
| a3: Isosceles |  |  |  |  |  |  | X | X | X |  |  |
| a4: Equilateral |  |  |  |  |  |  |  |  |  |  | X |
| a5: Impossible |  |  |  | X | X | X |  |  |  |  |  |

The“--“ symbol in thetableindicates don’t care values. The tableshows the sixconditionsand5 actions. All the conditions in the decision table are binary; hence, it is called as “Limited Entry decisiontable”.

Each column of the decision table represents a test case. That is, the table is read as follows:

# Action: Not aTriangle

1. When condition C1 isfalsewecan say that with the given “a”“b” and “c”values,it’s Not atriangle.
2. Similarly, condition C2 and C3, if any one of them are false, we can say that with the given “a” “b”and “c”valuesit’sNota triangle.

# Action:Impossible

1. When conditions C1, C2, C3 are true and two conditions among C4, C5, C6 is true, there is no chance of one conditions among C4, C5, C6 failing. So we can neglect these rules. Example: if condition C4: a=b is true and C5: a=c istrue, then it is impossible, that condition C6: b=c will fail, so the action is impossible.

# Action:Isosceles

1. When conditions C1, C2, C3 are true and any one condition among C4, C5 and C6 is true with remaining two conditions false then action is Isoscelestriangle.

Example: If condition C4: a=b is true and C5: a=c and C6: b=c is false; it means two sides are equal. So the action will be Isoscelestriangle.

# Action:Equilateral

1. When conditions C1, C2, C3 are true and also conditions C4, C5 and C6 are true then, the action is Equilateraltriangle.

# Action:Scalene

1. When conditions C1, C2, C3 are true and conditions C4, C5 and C6 are false i.e., sides a, b and c are different, then action is Scalenetriangle.

Number of Test Cases = Number of Rules.

Using the decision table, we obtain 11 functional test cases: 3 impossible cases, 3 ways of failing the triangle property, 1 way to get an equilateral triangle, 1 way to get a scalene triangle, and 3 ways to get an isoscelestriangle.

**Deriving test cases using Decision Table Approach**:

Test Cases:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **TC ID** | **Test Case Description** | **a** | **b** | **c** | **Expected Output** | **Actual Output** | **Status** |
| 1 | Testing for Requirement 1 | 4 | 1 | 2 | Nota Triangle |  |  |
| 2 | Testing for Requirement 1 | 1 | 4 | 2 | Not a Triangle |  |  |
| 3 | Testing for Requirement 1 | 1 | 2 | 4 | Not a Triangle |  |  |
| 4 | Testing for Requirement 2 | 5 | 5 | 5 | Equilateral |  |  |
| 5 | Testing for Requirement 2 | 2 | 2 | 3 | Isosceles |  |  |
| 6 | Testing for Requirement 2 | 2 | 3 | 2 | Isosceles |  |  |
| 7 | Testing for Requirement 2 | 3 | 2 | 2 | Isosceles |  |  |
| 8 | Testing for Requirement 2 | 3 | 4 | 5 | Scalene |  |  |

# EXECUTION:

Execute the program against the designed test cases and complete the table for Actual output column and statuscolumn.

**Test Report**

1. No of TC‟s Executed:08
2. No of DefectsRaised:
3. No of TC‟sPass:
4. No of TC‟sfailed:

The decision table technique is indicated for applications characterized by any of the following:

* Prominent if-then-else logic
* Logical relationships among input variables
* Calculations involving subsets of the input variables
* Cause-and-effect relationship between inputs and outputs

The decision table-based testing works well for triangle problem because a lot of decision making i.e., if-then-else logic takes place.

# REFERENCES:

1. RequirementSpecification
2. Assumptions

# PROGRAM 8

Design,develop,codeandruntheprograminanysuitablelanguagetosolvethecommissionproblem. Analyze it from the perspective of boundary value testing, derive different test cases, execute these test cases and discuss the testresults.

# REQUIREMENTS:

Problem Definition: The Commission Problem includes a salesperson in the formerArizona Territory sold rifle locks, stocks and barrels made by a gunsmith in Missouri. Costincludes

Locks- $45

Stocks- $30

Barrels- $25

The salesperson had to sell at least one complete rifle per month andproduction limits were such that the most the salesperson could sell in a month was 70 locks, 80 stocks and 90barrels.

After each town visit, the sales person sent a telegram to the Missouri gunsmith with the number of locks, stocks and barrels sold in the town. At the end of the month, the salesperson sent a very short telegram showing --1 lock sold. The gunsmith then knew the sales for the month were completeandcomputedthesalesperson’s commissionasfollows:

On sales up to (and including) $1000 = 10%

On the sales up to (and includes) $1800 = 15%

On the sales in excess of $1800 = 20%

The commission program produces a monthly sales report that gave the total number of locks, stocks and barrels sold, the salesperson’s total dollar sales and finally the commission.

# DESIGN

# ALGORITHM

Step 1: Define lockPrice=45.0, stockPrice=30.0, barrelPrice=25.0

Step 2: Input locks

Step 3: while(locks!=-1) „input device uses -1 to indicate end of data goto STEP 12

Step 4: input (stocks, barrels)

Step 5: compute lockSales, stockSales, barrelSales and sales

Step 6: output(“Total sales:” sales)

Step 7: if (sales > 1800.0) goto Step 8 else goto Step 9

Step 8: commission=0.10\*1000.0;

commission=commission+0.15 \* 800.0;

commission = commission + 0.20 \* (sales-1800.0)

Step 9: if (sales > 1000.0) goto Step 10 else goto Step 11

Step 10: commission=0.10\* 1000.0; commission=commission + 0.15 \* (sales-1000.0)

Step 11: Output(“Commission is $”, commission)

Step 12: exit

# PROGRAMCODE:

#include<stdio.h>

int main()

{

int locks, stocks, barrels, t\_sales, flag = 0; float commission;

printf("Enter the total number of locks"); scanf("%d",&locks);

if ((locks <= 0) || (locks > 70)) flag = 1;

printf("Enter the total number of stocks"); scanf("%d",&stocks);

if ((stocks <= 0) || (stocks > 80))

{

flag = 1;

}

printf("Enter the total number of barrels"); scanf("%d",&barrels);

if ((barrels <= 0) || (barrels > 90))

{

flag = 1;

}

if (flag == 1)

{

printf("invalidinput"); getch();

exit(0);

}

t\_sales = (locks \* 45) + (stocks \* 30) + (barrels \* 25); if (t\_sales <= 1000)

{

commission = 0.10 \* t\_sales;

}

else if (t\_sales < 1800)

{

}

else

{

}

commission = 0.10 \* 1000;

commission = commission + (0.15 \* (t\_sales - 1000));

commission = 0.10 \* 1000;

commission = commission + (0.15 \* 800);

commission = commission + (0.20 \* (t\_sales - 1800));

printf("The total sales is %d \n The commission is %f",t\_sales, commission);

return;

}

# TESTING

Technique used: **Boundary value analysis**

“Boundary value analysis” testing technique is used to identify errors at boundaries rather than finding those exist in center of input domain.

Boundary value analysis is a next part ofEquivalence partitioning for designing test cases where test cases are selected at the edges of the equivalenceclasses.

# BVA: Procedure

1. Partition the input domain using unidimensional partitioning. This leads to as many partitions as there are input variables. Alternately, a single partition of an input domain can be created using multidimensional partitioning. We will generate several sub-domains in thisstep.
2. Identify the boundaries for each partition. Boundaries may also be identified using special relationships amongst theinputs.
3. Select test data such that each boundary value occurs in at least one testinput.
4. BVA: Example: Create equivalenceclasses

Assuming that an item code must be in the range 99...999 and quantity in the range 1...100

# Equivalence classes for code:

E1: Values less than 99

E2: Values in the range

E3: Values greater than 999

**Equivalence classes for qty:**

E4: Values less than 1

E5: Values in the range

E6: Values greater than 100

# BVA: Example: Identify boundaries









Equivalence classes and boundaries for findPrice. Boundaries are indicated with an x. Points near the boundary are marked \*.

# Test Case design

The Commission Problem takes locks, stocks and barrels as input and checks it for validity. If it is valid, it returns the commission as its output. Here we have three inputs for the program, hence n = 3.

Since BVA yields (4n + 1) test cases according to single fault assumption theory, hence we can say that the total number of test cases will be (4\*3+1) = 12 + 1 =13.

The boundary value test cases can be generated over output by using following constraints and these constraints are generated overcommission:

C1: Sales up to (and including) $1000 = 10% commission

C2: Sales up to (and includes) $1800 = 15% commission

C3: Sales in excess of $1800 = 20% commission

Here from these constraints we can extract the test cases using the values of Locks, Stocks, and Barrels sold in month. The boundary values for commission are 10%, 15% and20%.

# Equivalence classes for 10% Commission:

E1: Sales less than 1000

E2: Sales equals to1000

**Equivalence classes for 15% Commission:**

E3: Sales greater than 1000 and less than1800

E4: Sales equals to1800

# Equivalence classes for 20% Commission:

E5: Sales greater then 1800

From the above equivalence classes, we can derive the following test cases using boundary value analysis approach.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **TC**  **Id** | **Test Case Description** | **Input Data** | | | **Sales** | **Expected Output**  **(Commission)** | **Actual Output** | **Status** |
| **Locks** | **Stocks** | **Barrels** |
| 1 | Input test cases for Locks=1, Stocks=1, Barrels=1 | 1 | 1 | 1 | 100 | 10 |  |  |
| 2 | Input test cases for Locks=1, Stocks=1, Barrels=2 | 1 | 1 | 2 | 125 | 12.5 |  |  |
| 3 | Input test cases for Locks=1, Stocks=2, Barrels=1 | 1 | 2 | 1 | 130 | 13 |  |  |
| 4 | Input test cases for Locks=2, Stocks=1, Barrels=1 | 2 | 1 | 1 | 145 | 14.5 |  |  |
| 5 | Input test cases for Locks=5, Stocks=5, Barrels=5 | 5 | 5 | 5 | 500 | 50 |  |  |
| 6 | Input test cases for Locks=10, Stocks=10, Barrels = 9 | 10 | 10 | 9 | 975 | 97.5 |  |  |
| 7 | Input test cases for Locks=10, Stocks=9, Barrels=10 | 10 | 9 | 10 | 970 | 97 |  |  |
| 8 | Input test cases for Locks=9, Stocks=10, Barrels=10 | 9 | 10 | 10 | 955 | 95.5 |  |  |
| 9 | Input test cases forLocks=10, Stocks=10, Barrels=10 | 10 | 10 | 10 | 1000 | 100 |  |  |
| 10 | Input test cases for Locks=10, Stocks=10, Barrels=11 | 10 | 10 | 11 | 1025 | 103.75 |  |  |
| 11 | Input test cases for Locks=10, Stocks=11, Barrels=10 | 10 | 11 | 10 | 1030 | 104.5 |  |  |
| 12 | Input test cases for Locks=11, Stocks=10, Barrels=10 | 11 | 10 | 10 | 1045 | 106.75 |  |  |
| 13 | Input test cases for Locks=14, Stocks=14, Barrels=13 | 14 | 14 | 13 | 1400 | 160 |  |  |
| 14 | Input test cases for Locks=18, Stocks=18, Barrels=17 | 18 | 18 | 17 | 1775 | 216.25 |  |  |
| 15 | Input test cases for Locks=18, Stocks=17, Barrels=18 | 18 | 17 | 18 | 1770 | 215.5 |  |  |
| 16 | Input test casesfor Locks=17, Stocks=18, Barrels= 18 | 17 | 18 | 18 | 1755 | 213.25 |  |  |
| 17 | Input test cases for Locks=18, Stocks=18, Barrels=18 | 18 | 18 | 18 | 1800 | 220 |  |  |
| 18 | Input test cases for Locks=18, Stocks=18, Barrels=19 | 18 | 18 | 19 | 1825 | 225 |  |  |
| 19 | Input test cases for Locks=18, Stocks=19, Barrels=18 | 18 | 19 | 18 | 1830 | 226 |  |  |
| 20 | Input test cases for Locks=19, Stocks=18, Barrels=18 | 19 | 18 | 18 | 1845 | 229 |  |  |
| 21 | Input test cases for Locks=48, Stocks=48, Barrels=48 | 48 | 48 | 48 | 4800 | 820 |  |  |

# Table-1: BVB Test case for Commission Problem

This is how we can apply BVA technique to create test cases for our Commission Problem.

# EXECUTIONS

Execute the program and test the test cases in Table-1 against program and complete the table with for Actual output column and Status column

# TEST REPORT:

* + 1. Noof TC‟sExecuted:
    2. No of DefectsRaised:
    3. Noof TC‟sPass:
    4. Noof TC‟sFailed:

# REFERENCES

1. Requirement Specification

2.Assumptions

# PROGRAM 9

Design, develop, code and run the program in any suitable language to implement the NextDate function. Analyze it from the perspective of equivalence class testing, derive different test cases, execute these test cases and discuss the testresults.

# REQUIREMENTSPECIFICATION

Problem Definition: "Next Date" is a function consisting of three variables like: month, date and year. It returns the date of next day as output. It reads current date as input date.

The constraints are

C1: 1 ≤ month ≤ 12

C2: 1 ≤ day ≤ 31

C3: 1812 ≤ year ≤2012

If any one condition out of C1, C2 or C3 fails, then this function produces an output "value of month not in the range1...12".

Since many combinations of dates can exist, hence we can simply display one message for this function: "Invalid Input Date".

A very common and popular problem occurs if the year is a leap year. We have taken into consideration that there are 31 days in a month. But what happensif a month has 30 days or even 29 or 28 days?

A year is called as a leap year if it is divisible by 4, unless it is a century year. Century years are leap years only if they are multiples of 400. So, 1992, 1996 and 2000 are leap years while 1900 is not a leapyear.

Furthermore, in this Next Date problem we find examples of Zipf's law also, which states that "80% of the activity occurs in 20% of the space". Thus in this case also, much of the source-code of Next Date function is devoted to the leap yearconsiderations.

# DESIGN

# ALGORITHM

Step 1: Input date in format DD.MM.YYYY

Step 2: if MM is 01, 03, 05,07,08,10 do Step 3 else Step 6

Step 3:if DD < 31 then do Step 4

else if DD=31 do Step 5 else output(InvalidDate);

Step 4: tomorrowday=DD+1 goto Step 18

Step 5: tomorrowday=1; tomorrowmonth=month + 1 goto Step 18

Step 6: if MM is 04, 06, 09, 11 do Step 7

Step 7: if DD<30 then do Step 4

else if DD=30 do Step 5 else output(InvalidDate);

Step 8: if MM is 12

Step 9: if DD<31 then Step 4 else Step 10

Step 10: tomorrowday=1, tommorowmonth=1, tommorowyear=YYYY+1; goto Step 18

Step 11: if MM is 2

Step 12: if DD<28 do Step 4 else do Step 13

Step 13: if DD=28 & YYYY is a leap do Step 14 else Step 15

Step 14: tommorowday=29 goto Step 18

Step 15: tommorowday=1, tomorrowmonth=3, goto Step 18;

Step 16: if DD=29 then do Step 15 else Step 17

Step 17: output(“Cannot have feb”, DD); Step 19

Step 18: output(tomorrowday, tomorrowmonth, tomorrowyear);

Step 19: exit

# PROGRAMCODE:

#include<stdio.h> main( )

{

int month[12]={31,28,31,30,31,30,31,31,30,31,30,31};

int d,m,y,nd,nm,ny,ndays;

printf("enter the date,month,year"); scanf("%d%d%d",&d,&m,&y); ndays=month[m-1];

if(y<=1812 && y>2012)

{

printf("Invalid Input Year"); exit(0);

}

if(d<=0 || d>ndays)

{

printf("Invalid Input Day"); exit(0);

}

if(m<1 && m>12)

{

printf("Invalid Input Month"); exit(0);

}

if(m==2)

{

if(y%100==0)

{

}

else

}

if(y%400==0)

ndays=29;

if(y%4==0)

ndays=29;

nd = d + 1;

nm = m; ny = y;

if(nd>ndays)

{

nd = 1; nm++;

}

if(nm>12)

{

nm = 1; ny++;

}

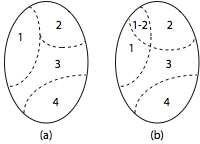
printf("\n Given date is %d:%d:%d",d,m,y); printf("\n Next day‟s date is %d:%d:%d",nd,nm,ny);

}

# TESTING

Technique used: **Equivalence Class testing**

Test selection using equivalence partitioning allows a tester to subdivide the input domain into a relatively small number of sub-domains, say N>1, as shown.

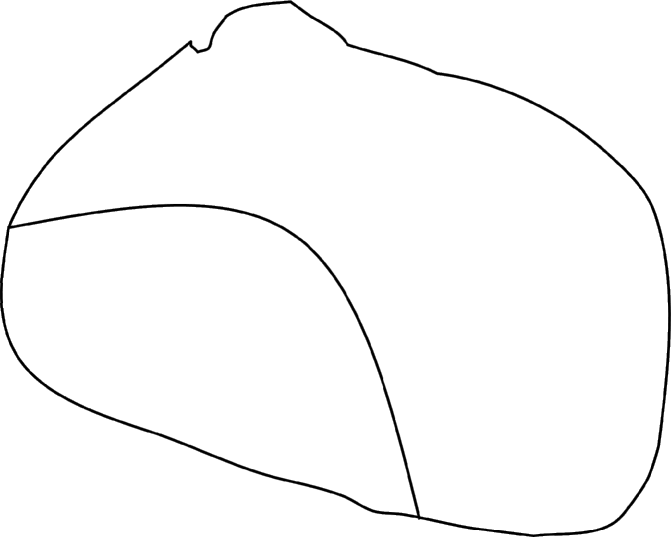


In strict mathematical terms, the sub-domains by definition are disjoint. The four subsets shown in (a) constitute a partition of the input domain while the subsets in (b) are not. Each subset is known as an equivalence class.

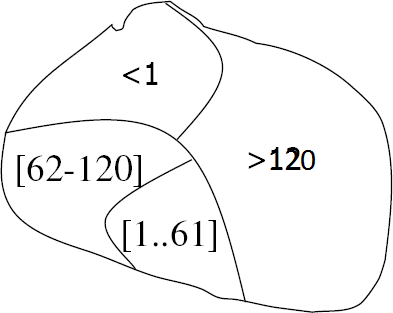
# Example:

Consider an application A that takes an integer denoted by age as input. Let us suppose that the only legal values of age are in the range [1..120]. The set of input values is now divided into a set E containing all integers in the range [1..120] and a set U containing the remaining integers.





Further, assume that the application is required to process all values in the range [1..61] in accordance with requirement R1 and those in the range [62..120] according to requirement R2. Thus E is further subdivided into two regions depending on the expected behavior.

Similarly, it is expected that all invalid inputs less than or equal to 1 are to be treated in one way while all greater than 120 are to be treated differently. This leads to a subdivision of U into two categories.

Tests selected using the equivalence partitioning technique aim at targeting faults in the application under test with respect to inputs in any of the four regions, i.e.,. two regions containing expected inputs and two regions containing the unexpectedinputs.

It is expected that any single test selected from the range [1...61] will reveal any fault with respect to R1. Similarly, any test selected from the region [62...120] will reveal any fault with respect to R2. A similar expectation applies to the two regions containing the unexpected inputs.

# Test Case design

The NextDate function is a function which will take in a date as input and produces as output the next date in the Georgian calendar. It uses three variables (month, day and year) which each have valid and invalid intervals.

# First Attempt

A first attempt at creating an equivalence relation might produce intervals such as these:

# Valid Intervals

M1 = {month: 1 ≤ month ≤ 12}

D1 = {day: 1 ≤day ≤31}

Y1 = {year: 1812 ≤ year ≤2012}

# Invalid Intervals

M2 = {month: month < 1}

M3 = {month: month > 12}

D2 = {day: day < 1}

D3 = {day: day > 31}

Y2 = {year: year < 1812}

Y3 = {year: year > 2012}

At a first glance it seems that everything has been taken into account and our day, month and year intervals have been defined well. Using these intervals we produce test cases using the four different types of Equivalence Classtesting.

Weak and Strong Normal

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **TC**  **Id** | **Test Case Description** | **Input Data** | | | **Expected Output** | **Actual Output** | **Status** |
| **MM** | **DD** | **YYYY** |
| 1 | Testing for Valid input changing the day within the month. | 6 | 15 | 1900 | 6/16/1900 |  |  |

Since the number of variables is equal to the number of valid classes, only one weak normal equivalence class test case occurs, which is the same as the strong normal equivalence class test case (Table 1).

# Weak Robust:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **TC**  **Id** | **Test Case Description** | **Input Data** | | | **Expected**  **Output** | **Actual Output** | **Status** |
| **MM** | **DD** | **YYYY** |
| 1 | Testing for Valid input changing the day within the month | 6 | 15 | 1900 | 6/16/1900 |  |  |
| 2 | Testing for Invalid Day, day with negative number it is not possible | 6 | -1 | 1900 | Day not in range |  |  |
| 3 | Testing for Invalid Day, day with Out of range, i.e., DD=32 | 6 | 32 | 1900 | Day not in range |  |  |
| 4 | Testing for Invalid Month, month with negative number it is not possible | -1 | 15 | 1900 | Month not in Range |  |  |
| 5 | TestingforInvalidmonth, month with out of range i.e., MM=13 it should MM<=12 | 13 | 15 | 1900 | Month not in Range |  |  |
| 6 | Testing forYear,year is out of range YYYY=1899, itshould<=1812 | 6 | 15 | 1899 | Year not in range |  |  |
| 7 | Testing for Year, year is out of range YYYY=2013, itshould<=2012 | 6 | 15 | 2013 | Year not in range |  |  |

**Table 2 : Weak Robust**

(Table 2) we can see that weak robust equivalence class testing will just test the ranges of the input domain once on each class. Since we are testing weak and not normal, there will only be at most one fault per test case (single fault assumption) unlike Strong Robust Equivalence class testing.

# Strong Robust:

This is a table showing one corner of the cube in 3d-space (the three other corners would include a different combination of variables) since the complete table would be too large to show.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **TC ID** | **Test Case Description** | **Input Data** | | | **Expected Output** | **Actual Output** | **Status** |
| **MM** | **DD** | **YYYY** |
| 1 | Testing for Month is not in range MM=-1 i.e., in negative number there is not possible have to be month in negative number | -1 | 15 | 1900 | Month not in range |  |  |
| 2 | Testing for Day is not in range DD=-1 i.e., in negative number there is not possible have to be Day in negative number | 6 | -1 | 1900 | Day not in range |  |  |
| 3 | Testing for Year is not in range YYYY=1899 i.e., Year should  <=1812 | 6 | 15 | 1899 | Year not in range |  |  |
| 4 | Testing for Day and month is not in range MM=-1, DD=-1 i.e., in negative number there is not possible have to be Day and Month in negative number | -1 | -1 | 1900 | 1. Day not in range 2. Month not in range |  |  |
| 5 | 1. Testing for Day is not in range and Year is not in range DD=-1 i.e., in negative number there is not possible have to be Day in negative number,and 2. YYYY=1899, so the range of year is<=1812 | 6 | -1 | 1899 | 1. Day not in range 2. Year not inrange |  |  |
| 6 | 1. Testing for Month is not in range MM=-1 and i.e., in negative number there is not possible have to be Day in negative number,and 2. Year I snot in range YYYY=1899, year should<=1812 | -1 | 15 | 1899 | 1. Month not in range 2. Year not in range |  |  |
| 7 | 1. Testing for Day is no in range DD=-1 i.e., in negative number there is not possible have to be Day in negativenumber 2. Testing for Month is not in range MM=- 1 and i.e., in negative number there is not possible have to be Day in negative number,and 3. Year is not in range YYYY=1899, year should<=1812 | -1 | -1 | 1899 | 1. Day not in range 2. Month not inrange 3. Year not inrange |  |  |

# Second Attempt

As said before the equivalence relation is vital in producing useful test cases and more time must be spent on designing it. If we focus more on the equivalence relation and consider more greatly what must happen to an input date we might produce the following equivalence classes:

M1 = {month: month has 30 days}

M2 = {month: month has 31 days}

M3 = {month: month is February}

Here month has been split up into 30 days (April, June, September and November), 31 days (January, March, April, May, July, August, October and December) and February.

D1 = {day: 1 ≤ day ≤ 28}

D2 = {day: day =29}

D3 = {day: day = 30}

D4 = {day: day =31}

Day has been split up into intervals to allow months to have a different number of days; we also have the special case of a leap year (February 29 days).

Y1 = {year: year = 2000}

Y2 = {year: year is a leap year}

Y3 = {year: year is a common year}

Year has been split up into common years, leap years and the special case the year 2000 so we can determine the date in the month of February.

Here are the test cases for the new equivalence relation using the four types of Equivalence Class testing.

# Weak Normal

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **TC**  **Id** | **Test Case Description** | **Input Data** | | | **Expected**  **Output** | **Actual**  **Output** | **Status** |
| **MM** | **DD** | **YYYY** |
| 1 | Testing for all Valid input changing the day within the month. | 6 | 14 | 2000 | 6/15/2000 |  |  |
| 2 | Testing for Valid input changing the day within the month. | 7 | 29 | 1996 | 7/30/1996 |  |  |
| 3 | Testing for Leafyear,  i.e., MM=2 (Feb) the input DD=30, there is not possible date 30, in leaf year only 28 and 29 will occur. | 2 | 30 | 2002 | Impossible date |  |  |
| 4 | Testing for Impossible  Date, i.e., MM=6 (June) the input DD=31, there is only 30 days in the monthofJune, So,DD=31 is ImpossibleDate. | 6 | 31 | 2000 | Impossible input date |  |  |

**Table 3 Weak normal**

**Strong Normal**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **TC ID** | **Test Case**  **Description** | **Input Data** | | | **Expected**  **Output** | **Actual**  **Output** | **Status** |
| **MM** | **DD** | **YYYY** |
| 1 | SN1 | 6 | 14 | 2000 | 6/15/2000 |  |  |
| 2 | SN2 | 6 | 14 | 1996 | 6/15/1996 |  |  |
| 3 | SN3 | 6 | 14 | 2002 | 6/15/2002 |  |  |
| 4 | SN4 | 6 | 29 | 2000 | 6/30/2000 |  |  |
| 5 | SN5 | 6 | 29 | 1996 | 6/30/1996 |  |  |
| 6 | SN6 | 6 | 29 | 2002 | 6/30/2002 |  |  |
| 7 | SN7 | 6 | 30 | 2000 | Invalid Input Date |  |  |
| 8 | SN8 | 6 | 30 | 1996 | Invalid Input Date |  |  |
| 9 | SN9 | 6 | 30 | 2002 | Invalid Input Date |  |  |
| 10 | SN10 | 6 | 31 | 2000 | Invalid Input Date |  |  |
| 11 | SN11 | 6 | 31 | 1996 | Invalid Input Date |  |  |
| 12 | SN12 | 6 | 31 | 2002 | Invalid Input Date |  |  |
| 13 | SN13 | 7 | 14 | 2000 | 7/15/2000 |  |  |
| 14 | SN14 | 7 | 14 | 1996 | 7/15/1996 |  |  |
| 15 | SN15 | 7 | 14 | 2002 | 7/15/2002 |  |  |
| 16 | SN16 | 7 | 29 | 2000 | 7/30/2000 |  |  |
| 17 | SN17 | 7 | 29 | 1996 | 7/30/1996 |  |  |
| 18 | SN18 | 7 | 29 | 2002 | 7/30/2002 |  |  |
| 19 | SN19 | 7 | 30 | 2000 | 7/31/2000 |  |  |
| 20 | SN20 | 7 | 30 | 1996 | 7/31/1996 |  |  |
| 21 | SN21 | 7 | 30 | 2002 | 7/31/2002 |  |  |
| 22 | SN22 | 7 | 31 | 2000 | 8/1/2000 |  |  |
| 23 | SN23 | 7 | 31 | 1996 | 8/1/1996 |  |  |
| 24 | SN25 | 7 | 31 | 2002 | 8/1/2002 |  |  |
| 25 | SN24 | 2 | 14 | 2000 | 2/15/2000 |  |  |
| 26 | SN26 | 2 | 14 | 1996 | 2/15/1996 |  |  |
| 27 | SN27 | 2 | 14 | 2002 | 2/15/2002 |  |  |
| 28 | SN28 | 2 | 29 | 2000 | Invalid Input Date |  |  |
| 29 | SN29 | 2 | 29 | 1996 | 3/1/1996 |  |  |
| 30 | SN30 | 2 | 29 | 2002 | Invalid Input Date |  |  |
| 31 | SN31 | 2 | 30 | 2000 | Invalid Input Date |  |  |
| 32 | SN32 | 2 | 30 | 1996 | Invalid Input Date |  |  |
| 33 | SN33 | 2 | 30 | 2002 | Invalid Input Date |  |  |
| 34 | SN34 | 2 | 31 | 2000 | Invalid Input Date |  |  |
| 35 | SN35 | 2 | 31 | 1996 | Invalid Input Date |  |  |
| 36 | SN36 | 2 | 31 | 2002 | Invalid Input Date |  |  |

**Table 4: Strong Normal**

**EXECUTION**

Execute the program and test the test cases in Table-1 against program and complete the table with for Actual output column and Status column

# Test Report:

* + 1. Noof TC‟sExecuted:
    2. No of DefectsRaised:
    3. Noof TC‟sPass:
    4. Noof TC‟sFailed:

# REFERENCES:

1. RequirementSpecification
2. Assumptions