Práctica VIII – Pruebas de caja blanca

Pasos para la obtención de los casos de prueba:
1. A partir del diseño o del código fuente, dibujar el grafo de flujo asociado (hacer el diseño usando **draw.io**)
2. Se calcula la complejidad ciclomática del grafo.
3. Se determina un conjunto básico de caminos independientes.
4. Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

1 Dibujar el grafo

El grafo de flujo se utiliza para representar flujo de control lógico de un programa. Para ello se utilizan los elementos siguientes:

- **Nodos**: representan cero, una o varias sentencias en secuencia. Cada nodo comprende como máximo una sentencia de decisión (bifurcación).
- **Nodos predicado:** son los nodos que tienen una condición. Son los nodos de los que parten 2 caminos. Se caracterizan porque dos o más nodos emergen de él. Cuando hay una condición compuesta por dos o más operadores lógicos (AND, OR, ...) se crea un nodo distinto por cada una de las condiciones simples.
- **Aristas**: líneas que unen dos nodos.
- **Regiones**: áreas delimitadas por aristas y nodos. Cuando se contabilizan las regiones de un programa debe incluirse el área externa como una región más

2 Calcular la complejidad ciclomática del grafo

La complejidad ciclomática es una métrica del software que proporciona una medida cuantitativa de la complejidad lógica de un programa. En el contexto del método de prueba del camino básico, el valor de la complejidad ciclomática define el número de caminos independientes de dicho programa, y por lo tanto, el número de casos de prueba a realizar.

Posteriormente veremos cómo se identifican esos caminos, pero primero veamos cómo se puede calcular la complejidad ciclomática a partir de un grafo de flujo, para obtener el número de caminos a identificar.

Existen varias formas de calcular la complejidad ciclomática de un programa a partir de un grafo de flujo:

1. A través de la siguiente fórmula: V(G) = Aristas – Nodos + 2
2. A través de esta otra fórmula: V(G) = Nodos Predicado + 1

3. Determinar el conjunto básico de caminos independientes

Un camino independiente es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una condición, respecto a los caminos existentes. En términos del diagrama de flujo, **un camino independiente está constituido por lo menos por una arista que no haya sido recorrida anteriormente a la definición del camino**. No se

considera un camino independiente si es una combinación de caminos ya especificados y no recorre ninguna arista nueva. En la identificación de los distintos caminos de un programa para probar, se debe tener en cuenta que cada nuevo camino debe tener el mínimo número de sentencias nuevas o condiciones nuevas respecto a los que ya existen. De esta manera se intenta que el proceso de depuración sea más sencillo.

4. Preparar casos de prueba que fuercen la ejecución de cada camino básico.

Tendremos que preparar casos de prueba que ejecuten todos los posibles caminos independientes

Ejercicios

De acuerdo con las técnicas de testeo de caja blanca, genera los grafos de algoritmo, calcula los caminos independientes y genera casos de prueba que los cumplan.

1.
```
If (a>1) and (b>5) and (c<2)  then
      x=x+1;
else
      x= x-1;
      end
```

2.
```
If (a>1) or (b>5) or (c<2)  then
       x=x+1;
else
       x= x-1;
end
```

3.
```
Import java.io.*;

Public class Maximo
{
       public static void main (String args[]) throws IOException
       {
       BufferedReader entrada = new BufferedReader (new InputStreamReader (System.in));

       int x,y,z,max;

       System.out.println("Introduce x,y,z: ");
       x =  Integer.parseInt (entrada.readLine());
       y =  Integer.parseInt (entrada.readLine());
       z =  Integer.parseInt (entrada.readLine());

       if (x>y && x>z)
                     max = x;
       else{
              if (z>y)
                     max = z;
              else
                     max = y;
       }
       System.out.println ("El máximo es "+ max);
       } //main
}
```

4.

```
function obtener_media : real ;

var
      n, suma, conta, suma2, total_num : integer ;

begin
      read( n ) ;
      repeat
            if (n >= 20 and n <= 50) then
                  suma := suma + n ;
                  conta := conta + 1
            else
                  suma2 := suma2 + n ;
            total_num := total_num + 1 ;
            read (n) ;
      until n = 0 ;
      obtener_media := suma / conta ;
      write (total_num, suma2) ;
end ;
```



```
public static double ReturnAverage(int value[],
                        int AS, int MIN, int MAX){
  /*
   Function: ReturnAverage  Computes the  average
   of all  those  numbers in the  input array  in
   the  positive  range  [MIN, MAX]. The  maximum
   size  of the array is AS. But, the  array size
   could be smaller than AS in which case the end
   of input is represented by -999.
  */
     int i, ti, tv, sum;
     double av;
     i = 0; ti = 0; tv = 0; sum = 0;
     while (ti < AS && value[i] != -999) {
         ti++;
         if (value[i] >= MIN && value[i] <= MAX) {
            tv++;
            sum = sum + value[i];
         }
         i++;
     }
     if (tv > 0)
        av = (double)sum/tv;
     else
        av = (double) -999;
     return (av);
}
```

Figure 4.6 Function to compute average of selected integers in an array. This program is an adaptation of "Figure 2. A sample program" in ref. 10. (With permission from the Australian Computer Society.)

5.

(Ejercicio sacado del libro Naik, K., & Tripathy, P. (2009). Software testing and quality assurance: theory and practice. Choice Reviews Online, 46(06), 46-3304. https://doi.org/10.5860/choice.46-3304)

Calcula cohesión (int nt1, nt2; String tok1[ ], tok2[ ])

```
1 numAdh = 0
2 Para i de 0 hasta nt1-1
3     Para j de 0 hasta nt2-1
4         Si tok1[i]=tok2[j] entonces
5                 numAdh = numAdh +1
6 total = tok1 + tok2 – numAdh
7 cohesión = numAdh / total
8 regresa cohesión
```

6.