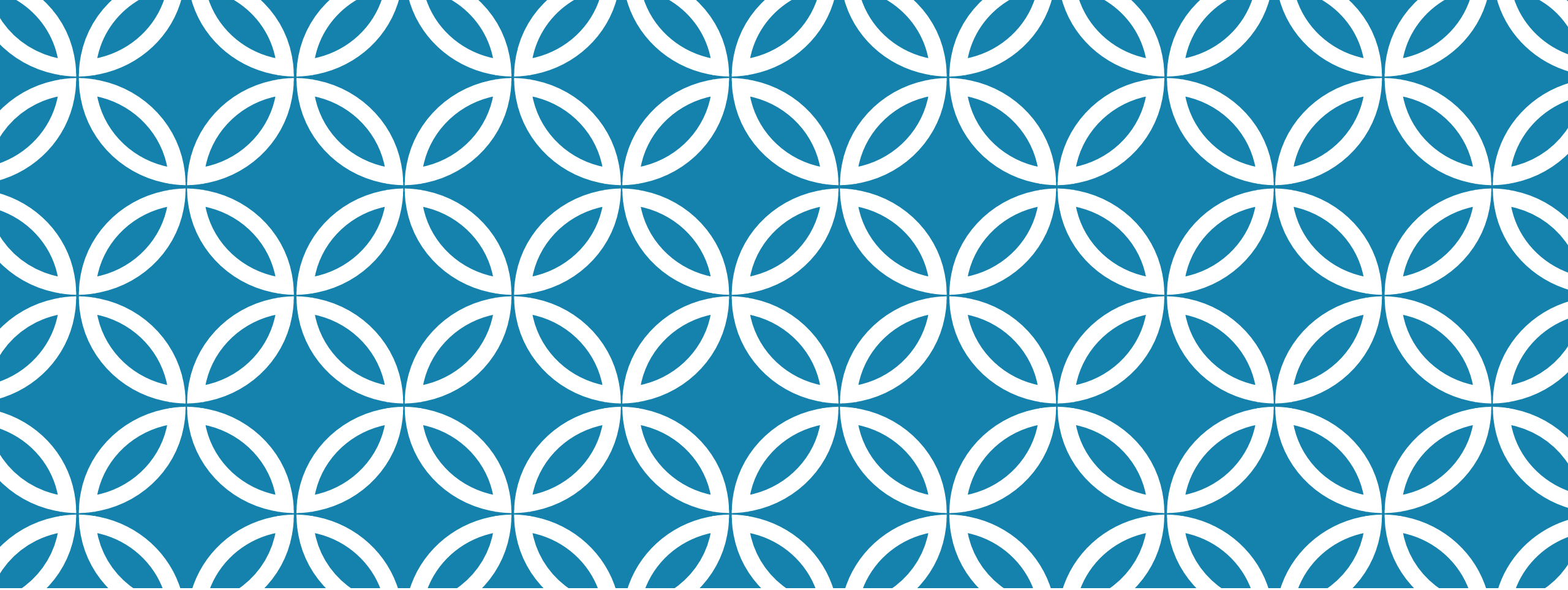


MÓDULO PROFESIONAL ENTORNOS DE DESARROLLO

UD 5 – Diseño y realización de pruebas de caja blanca

CONTENIDO

Introducción

- Prueba
- Casos de prueba

Técnicas de diseño de casos de prueba

- Pruebas de caja blanca o estructurales
- Pruebas de caja negra o funcionales

Técnicas de diseño de casos de prueba de caja blanca

-cobertura lógica

-camino básico

-mutación

PRUEBA. DEFINICIÓN

La prueba (testing) es el proceso de ejecutar un programa con la intención de encontrar errores.

[Glenford J. Myers]

PRUEBA. DEFINICIÓN

IEEE

IEEE es el acrónimo para Institute of Electrical and Electronics Engineers, una asociación profesional sin ánimo de lucro que determina la mayor parte de los estándares en las Ingenierías.

En la literatura existen otras definiciones. Por ejemplo, de acuerdo a la IEEE [IEEE90] el concepto de prueba se define como

"El proceso de operar un sistema o un componente bajo unas condiciones específicas, observando o registrando los resultados obtenidos con el fin de realizar a posteriori una evaluación de ciertos aspectos clave"

CASOS DE PRUEBA. DEFINICIÓN

Con el fin de desarrollar las pruebas de una manera óptima, los **casos de prueba** cobran especial importancia.

La IEEE [IEEE90] define caso de prueba como

"un conjunto de entradas, condiciones de ejecución y resultados esperados que son desarrollados con el fin de cumplir un determinado objetivo"

PRUEBAS

El programa se prueba ejecutando solo unos pocos casos de prueba dado que por lo general es física, económica o técnicamente imposible ejecutarlo para todos los valores de entrada posibles de aquí la frase de Dijkstra.

"Las pruebas del software pueden usarse para demostrar la existencia de errores, nunca su ausencia"

TÉCNICAS DE DISEÑO DE CASOS DE PRUEBAS

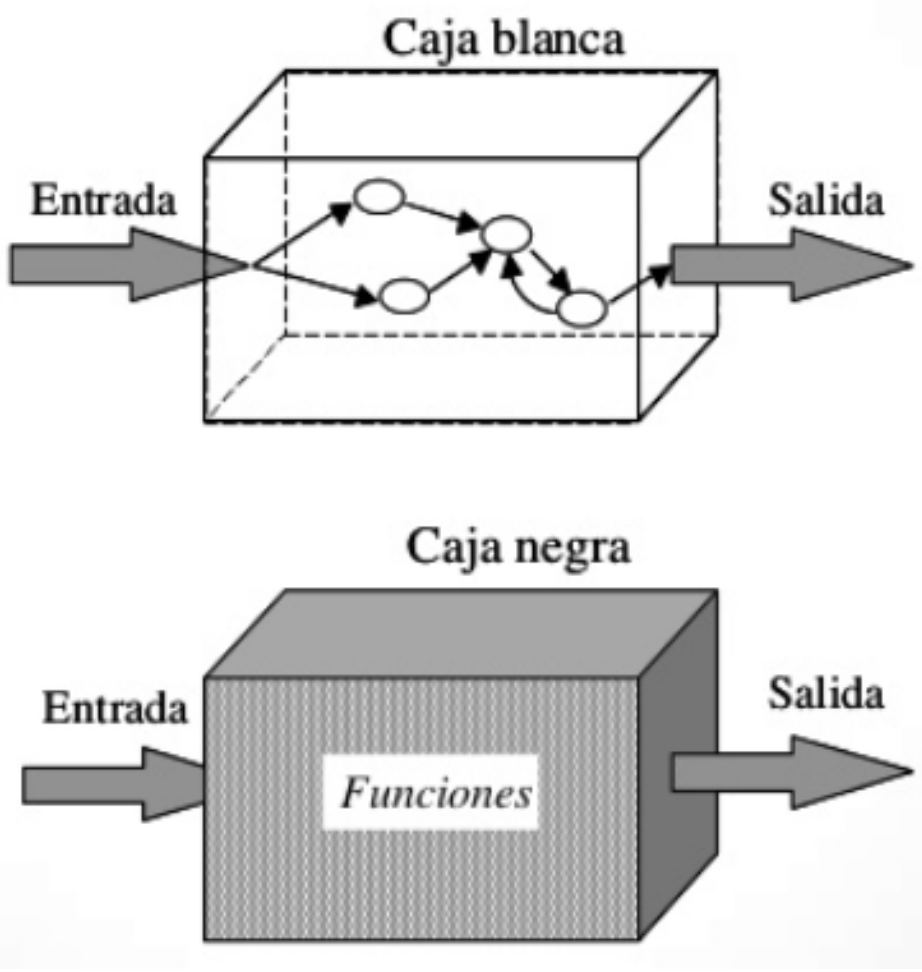
Clasificación clásica de técnicas de prueba:

- **Pruebas estructurales o de caja blanca:** cuando se diseña la prueba a partir del conocimiento de la estructura interna del código. Estas pruebas se centran en la estructura interna del programa, analizando los caminos de ejecución. Problemas: No se prueba la especificación funcional y no se detectan ausencias
- **Pruebas funcionales o de caja negra:** Su objetivo es validar que el código cumple la funcionalidad definida. Problemas: Infinitas posibilidades para las entradas.

TÉCNICAS DE DISEÑO DE CASOS DE PRUEBAS

Las **pruebas de caja blanca** permitirán recorrer todos los posibles caminos del código y ver qué sucede en cada caso posible. Se probará qué ocurre con las condiciones y los bucles que se ejecutan. Las pruebas se llevarán a cabo con datos que garanticen que han tenido lugar todas las combinaciones posibles. Para decidir qué valores deberán tomar estos datos es necesario saber cómo se ha desarrollado el código, buscando que no quede ningún rincón sin revisar.

Las técnicas de **pruebas de caja negra** pretenden encontrar errores en funciones incorrectas o ausentes, errores de interfaz, errores de rendimiento, inicialización y finalización. Se centra en las funciones y en sus entradas y salidas.



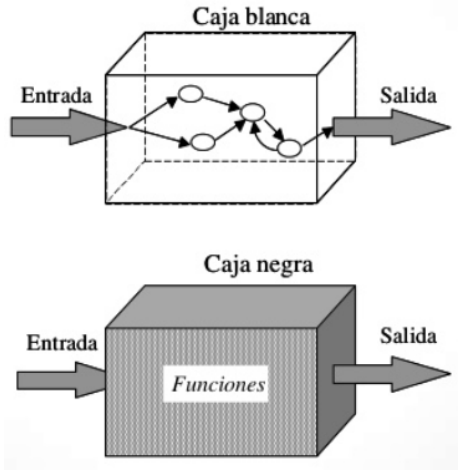
CASOS DE PRUEBA

Un buen caso de prueba es aquel que tiene una probabilidad muy alta de descubrir un nuevo error.

Para **diseñar** los casos de prueba, vamos a estudiar dos métodos:

- Pruebas funcionales o de caja negra: Su objetivo es validar que el código cumple la funcionalidad definida.
- Pruebas estructurales o de caja blanca: Se centran en la implementación de los programas para escoger los casos de prueba ..

PRUEBAS DE CAJA BLANCA



- Las pruebas de caja blanca permitirán recorrer todos los posibles caminos del código y ver qué sucede en cada caso posible.
- Se probará qué ocurre con las condiciones y los bucles que se ejecutan.
- Las pruebas se llevarán a cabo con datos que garanticen que han tenido lugar todas las combinaciones posibles.
- Para decidir qué valores deberán tomar estos datos es necesario saber cómo se ha desarrollado el código, buscando que no quede ningún rincón sin revisar.

PRUEBAS DE CAJA BLANCA

Partiendo de que las pruebas exhaustivas son impracticables, ya que el número de combinaciones es excesivo, se diseñan estrategias que ofrezcan una seguridad aceptable para descubrir errores.

Las principales técnicas de diseño de pruebas de caja blanca son:

- cobertura lógica (o flujo de control)
- técnica del camino básico
- mutación

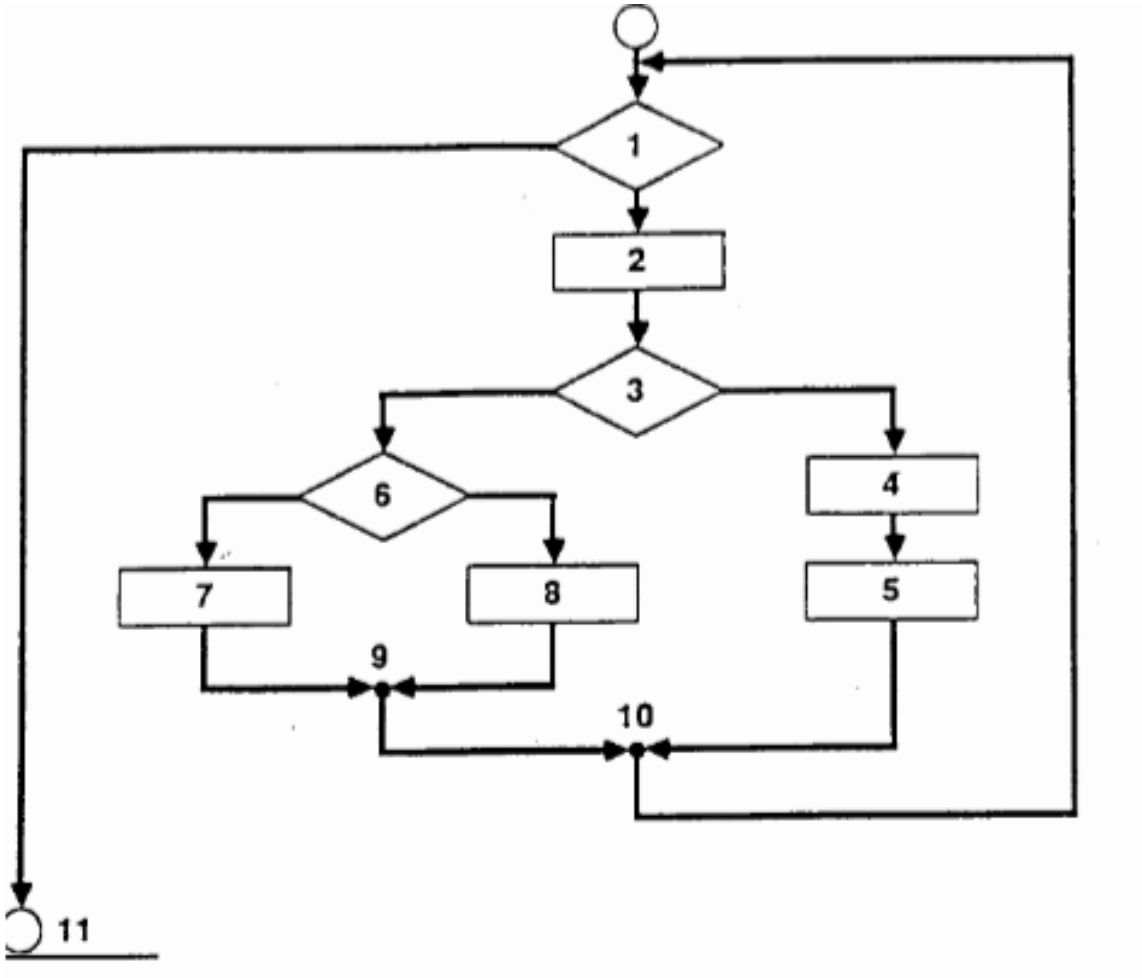
PRUEBAS DE CAJA BLANCA

Cobertura lógica

Como se ha indicado ya, puede ser impracticable realizar una prueba exhaustiva de todos los caminos de un programa. Por ello se han definido distintos **criterios de cobertura lógica**, que permiten decidir qué sentencias o caminos se deben examinar con los casos de prueba. Estos **criterios** son:

- sentencias
- decisiones
- condiciones
- bucles

Los distintos criterios de cobertura lógica, permiten decidir qué sentencias se deben examinar con los casos de prueba.



```java
import java.util.Scanner;
public class Prueba {

    //un número primo es un número natural mayor que 1 que
    //tiene únicamente dos divisores distintos: él mismo y el 1
    public boolean esPrimo(String numero){

        try {
            int num = Integer.parseInt(numero);
            if (num < 1){
                System.out.println("Número no natural o natural no mayor que 1");
                return false;
            }
            else {
                for (int i=2; i<num; i++)
                    if (num%i==0)
                        return false;
                return true;
            }
        } catch (NumberFormatException e){
            System.out.println("Error al convertir a número");
            return false;
        }
    }

    public static void main (String[] args) {
        Prueba prueba = new Prueba();

        System.out.println("Dime un número : ");
        Scanner sc = new Scanner(System.in);
        String n = sc.nextLine();

        System.out.println(n + " es primo: " + prueba.esPrimo(n));
    }
}
```

PRUEBAS DE CAJA BLANCA

Cobertura lógica

Puede ser imposible cubrir el 100% si el programa es muy complejo, pero podemos tener un mínimo de garantías de eficacia si seguimos las sugerencias para diseñar los casos de prueba fijándonos en estos elementos:

Sentencias. se trata de diseñar casos de prueba para que cada sentencia se ejecute al menos una vez.

Decisiones: una decisión es una lista de condiciones conectadas por operadores lógicos. Deben diseñarse suficientes casos de prueba para todas las decisiones sean evaluadas a verdadero/falso al menos una vez

condicion1 **operador** condicion2 **operador** condición3…

Condiciones: Se escriben casos de prueba suficientes para que cada condición en una decisión tenga una vez resultado verdadero y otra falso.

Condiciones múltiples: Se escriben casos de prueba suficientes para que todas las combinaciones posibles de resultados de cada condición se invoquen al menos una vez.

Bucles: se deben diseñar los casos de prueba de manera que se intente ejecutar un bucle en diferentes situaciones límite.

PRUEBAS DE CAJA BLANCA

Cobertura lógica

Cobertura de sentencia

Esta cobertura requiere que se ejecute por lo menos **una vez cada sentencia** del programa.

- Este criterio es necesario pero no suficiente
- Es un criterio débil
- No se comprueban ambas vertientes de las condiciones

PRUEBAS DE CAJA BLANCA

Cobertura lógica

Cobertura de decisiones

Este criterio establece que es necesario escribir un número suficiente de casos de prueba como para que **cada decisión tenga por lo menos un resultado verdadero o falso.**

- Este criterio es más fuerte que el de sentencia pero aún presenta debilidades. En sentencias condicionales compuestas puede quedar enmascarada una de las condiciones

¿Qué ocurre con los segmentos opcionales?

```
if (condicion) {
        ejecuta esto;
}
```

- Hay que probar cuándo la condición se cumple y cuándo falla
- Estos criterios se extienden a las construcciones que suponen elegir uno de entre varios casos. Por ejemplo, el switch.

PRUEBAS DE CAJA BLANCA

Cobertura lógica

Cobertura de condiciones

En este criterio es necesario presentar un número suficiente de casos de prueba de modo que cada **condición en una decisión tenga, al menos una vez, todos los resultados posibles.**

- Este criterio es más fuerte que el anterior.
- Hay que ser cuidadoso con la elección de los casos de prueba por que aunque se garanticen la ejecución de las condiciones puede ocurrir que alguna cláusula de la decisión no sea ejecutada

PRUEBAS DE CAJA BLANCA

Cobertura lógica

Cobertura múltiple

La solución lógica a lo anterior es utilizar un criterio que requiera un número suficiente de casos de prueba tal que **todas las combinaciones posibles de resultados de condición en cada decisión** y todos los puntos de entrada se invoquen al menos una vez.

- Satisface los criterios de cobertura anteriormente citados.

PRUEBAS DE CAJA BLANCA

Cobertura lógica

Cobertura múltiple

¿Qué ocurre cuando la condición es más compleja?

```
if (condicion1 || condicion2) {
        ejecuta esto;
}
```

Solo 2 ramas, pero 4 posibles combinaciones

Prueba 1: Condicion1 = TRUE y Condicion2 = FALSE
Prueba 2: Condicion1 = FALSE y Condicion2 = TRUE
Prueba 3: Condicion1 = FALSE y Condicion2 = FALSE
Prueba 4: Condicion1 = TRUE y Condicion2 = TRUE

PRUEBAS DE CAJA BLANCA

Cobertura lógica

Cobertura de bucles

Los bucles no son más que segmentos de código controlados por decisiones.

- Los bucles son una fuente inagotable de errores, algunos mortales.
- Un bucle se ejecuta un cierto número de veces; pero ese número de veces debe ser muy preciso, y lo más normal es que ejecutarlo una vez de menos o una vez de más tenga consecuencias indeseables.

PRUEBAS DE CAJA BLANCA

Cobertura lógica

Cobertura de bucles

Para bucles WHILE hay que pasar 3 pruebas excepto DO-WHILE que hay que pasar las dos últimas

- 0 ejecuciones
- 1 ejecución
- 2 ejecuciones
- m ejecuciones dónde m<n
- n-1 ejecuciones
- n ejecuciones

(n es el número máximo de iteraciones)

PRUEBAS DE CAJA BLANCA

Cobertura lógica

Cobertura de bucles

- No obstante, conviene no engañarse con los bucles FOR y examinar su contenido. Si dentro del bucle se altera la variable de control, o el valor de alguna variable que se utilice en el cálculo del incremento o del límite de iteración, entonces eso es un bucle FOR con “trampa”.
- También tiene "trampa" si contiene sentencias del tipo EXIT (que algunos lenguajes denominan BREAK) o del tipo RETURN. Todas ellas provocan terminaciones anticipadas del bucle.

PRUEBAS DE CAJA BLANCA

- Las pruebas de caja blanca se pueden hacer con un depurador (debugger)
- Lograr una buena cobertura con pruebas de caja blanca es un objetivo deseable; pero no suficiente a todos los efectos. Un programa puede estar perfecto en todos sus términos, y sin embargo no servir a la función que se pretende.
- Por ejemplo, si escribimos una rutina para ordenar datos por orden ascendente, pero el cliente los necesita en orden decreciente; no hay prueba de caja blanca capaz de detectar la desviación.
- Las pruebas de caja blanca nos convencen de que un programa hace bien lo que hace; pero no de que haga lo que necesitamos.
- Necesitamos comprobar la funcionalidad con las pruebas de caja negra

EJEMPLO 2

```java
import java.util.Scanner;
public class Prueba {

    //un número primo es un número natural mayor que 1 que
    //tiene únicamente dos divisores distintos: él mismo y el 1
    public boolean esPrimo(String numero){

        try {
            int num = Integer.parseInt(numero);
            if (num < 1){
                System.out.println("Número no natural o natural no mayor que 1");
                return false;
            }
            else {
                for (int i=2; i<num; i++)
                    if (num%i==0)
                        return false;
                return true;
            }
        } catch (NumberFormatException e){
            System.out.println("Error al convertir a número");
            return false;
        }
    }

    public static void main (String[] args) {
        Prueba prueba = new Prueba();

        System.out.println("Dime un número : ");
        Scanner sc = new Scanner(System.in);
        String n = sc.nextLine();

        System.out.println(n + " es primo: " + prueba.esPrimo(n));
    }
}
```

Entrada	Resultado esperado	Ejecuciones
-2	Número no natural o natural no mayor que 1	num<1 true
2	es primo	num<1 false ejecuta el bucle 0 veces
3	es primo	num<1 false ejecuta el bucle1 vez
4	no es primo	num<1 false ejecuta el bucle 2 veces num%2==0 true
17	es primo	num<1 false ejecuta el bucle N veces num%2==0 false
"hola"	no es un número	excepción

ACTIVIDAD

2. Realiza pruebas de cobertura para Bin2Dec, AdivinaElNumero

PRUEBAS DE CAJA BLANCA

Técnica del camino básico

Definiciones

Camino: “Secuencia de todas las instrucciones de un programa de principio a fin.

Camino independiente: Es cualquier camino del programa que incluye nuevas instrucciones.

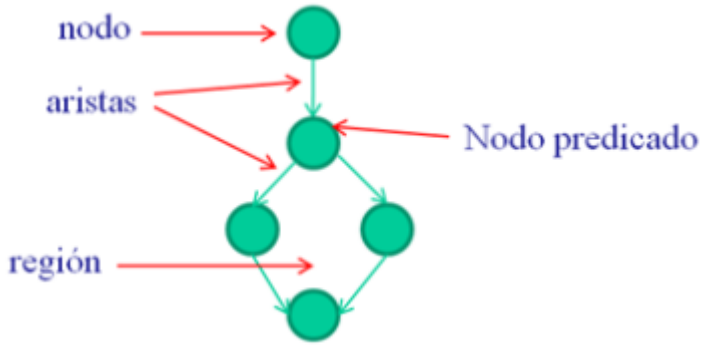
PRUEBAS DE CAJA BLANCA

Técnica del camino básico

Se debe diseñar un caso de prueba para cada camino independiente, de manera que ejecute al menos una vez cada sentencia. Para ello es necesario determinar los posibles **caminos independientes** y preparar suficientes casos de prueba para recorrer todos los caminos.

La técnica del **camino básico** (porpuesto por McCabe) permite obtener una medida de la complejidad de un diseño procedimental y utilizar esa medida como guía para la definición de una serie de caminos básicos de ejecución, diseñando casos de prueba que garanticen que cada camino se ejecuta al menos una vez.

PRUEBAS DE CAJA BLANCA



Técnica del camino básico

La idea es derivar casos de prueba a partir de un conjunto dado de caminos independientes por los cuales puede circular el flujo de ejecución de un programa. Para obtener dicho conjunto de **caminos independientes** se construye el **grafo de flujo** asociado al **código fuente** y se calcula su complejidad ciclomática. Los pasos que se siguen para aplicar esta técnica son:

- A partir del diseño o del código fuente, se dibuja el grafo de flujo asociado.
- Se calcula la complejidad ciclomática del grafo.
- Se determina un conjunto básico de caminos independientes.
- Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

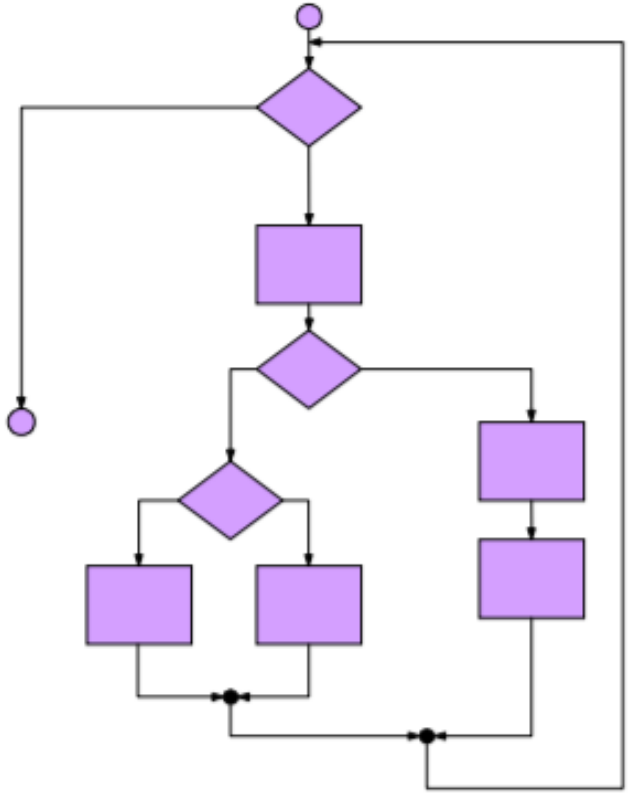
Los casos de prueba derivados del conjunto básico garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa.

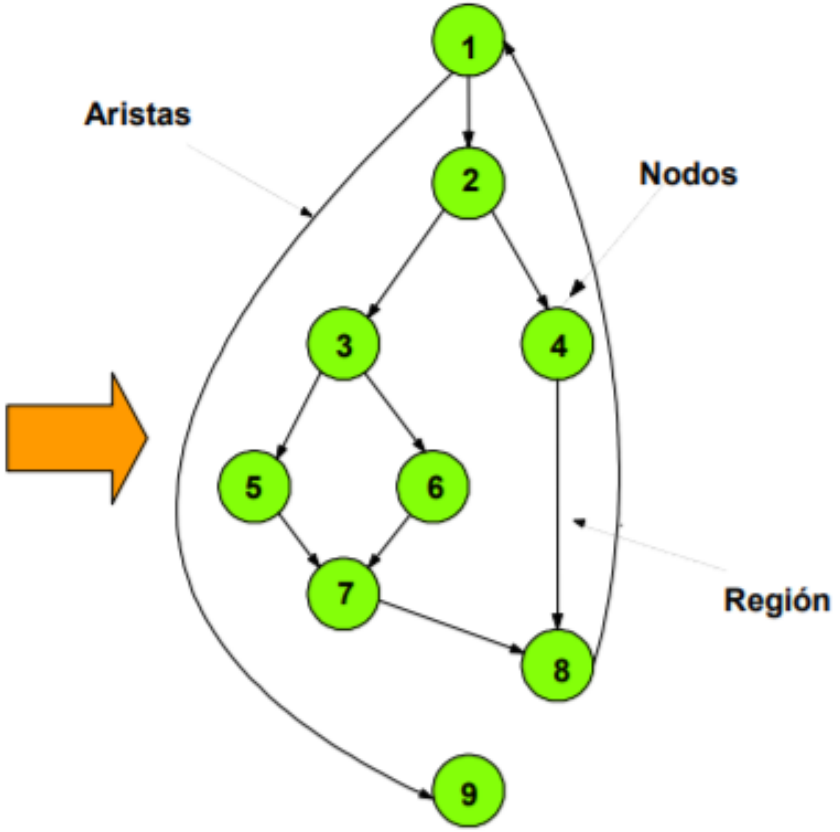
Vamos a ver como se hace esto paso por paso

PRUEBAS DE CAJA BLANCA

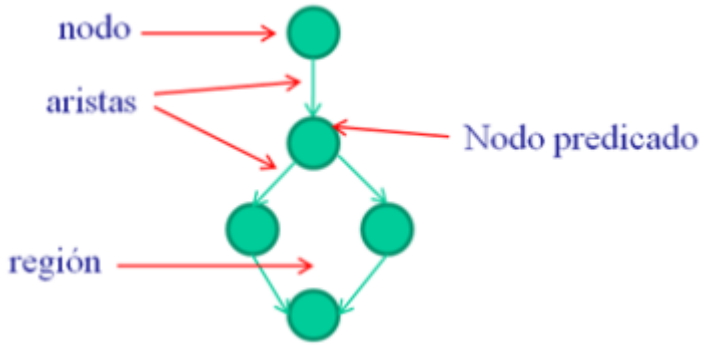
Técnica del camino básico

A continuación se muestra un ejemplo basado en un diagrama de flujo que representa la estructura de control del programa y su correspondiente grafo de flujo





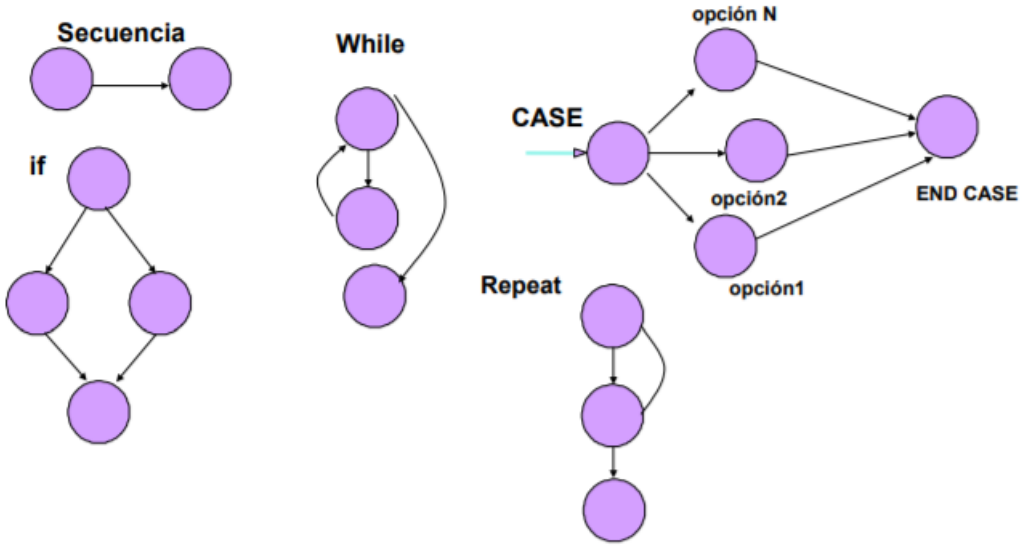
PRUEBAS DE CAJA BLANCA



Técnica del camino básico

El **grafo de flujo** es la representación gráfica de los segmentos del programa. Está compuesto por nodos y aristas.

Se representa el flujo de control con la siguiente notación



PRUEBAS DE CAJA BLANCA

Técnica del camino básico

Esta estrategia requiere diseñar casos de prueba suficientes para recorrer toda la lógica del programa. Se puede saber cuántos casos de prueba hay que crear y ejecutar? Cómo se calcula?

El matemático Thomas J. McCabe llamó complejidad ciclomática (CC) al número de caminos independientes de un grafo de flujo, y propuso la siguiente fórmula para calcularla:

V(G) = aristas-nodos+2

PRUEBAS DE CAJA BLANCA

Técnica del camino básico

A partir del grafo de flujo de la figura, la complejidad ciclomática sería:

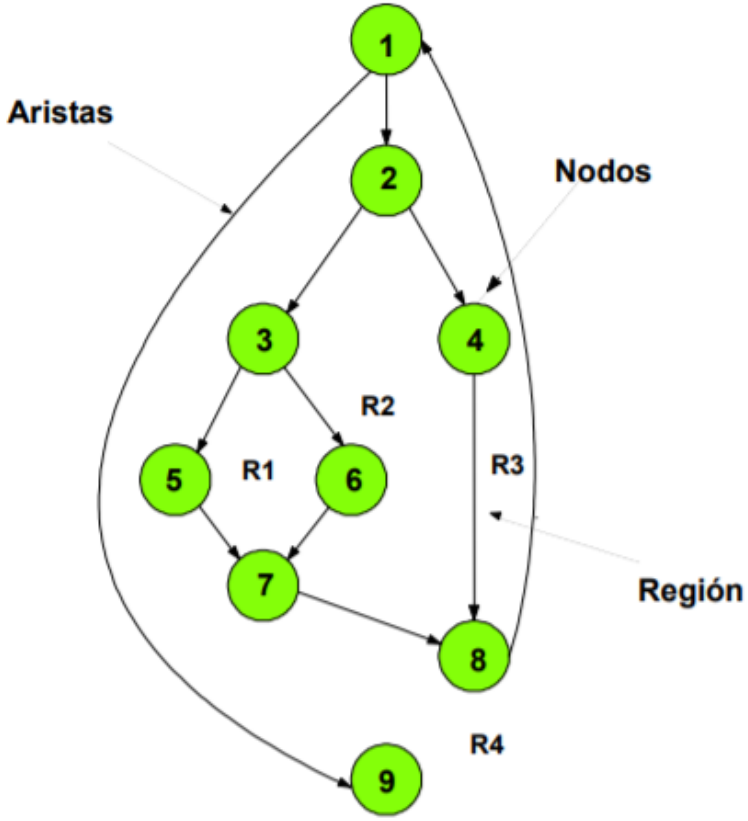
- El grafo tiene 11 aristas y 9 nodos => 11-9+2= 4

A partir del valor de la complejidad ciclomática obtenemos el número de **caminos independientes,** que nos da un valor límite para el número de casos de prueba que tenemos que diseñar.

En el ejemplo, el número de caminos independientes es 4 y son:

- 1-9
- 1-2-4-8-1-9
- 1-2-3-5-7-8-1-9
- 1-2-3-6-7-8-1-9

Esto significa **diseñar 4 casos de prueba**



PRUEBAS DE CAJA BLANCA

Pasos del diseño de pruebas mediante la técnica del camino básico

1. Obtener el **grafo de flujo** a partir del **código fuente**
2. Obtener la **complejidad ciclomática** del grafo de flujo
3. Definir el conjunto de **caminos básicos independientes**
4. Determinar los **casos de prueba** que permitan la ejecución de cada uno de los caminos anteriores.
5. **Ejecutar cada caso de prueba** y comprobar que los resultados son los esperados

VÍDEO: COMO CREAR EL GRAFO DE FLUJO DE UN PROGRAMA [ENLACE]

VÍDEO: COMPLEJIDAD CICLOMÁTICA: COMO CALCULARLA[ENLACE]

VÍDEO: PRUEBA DEL CAMINO BÁSICO [ENLACE]

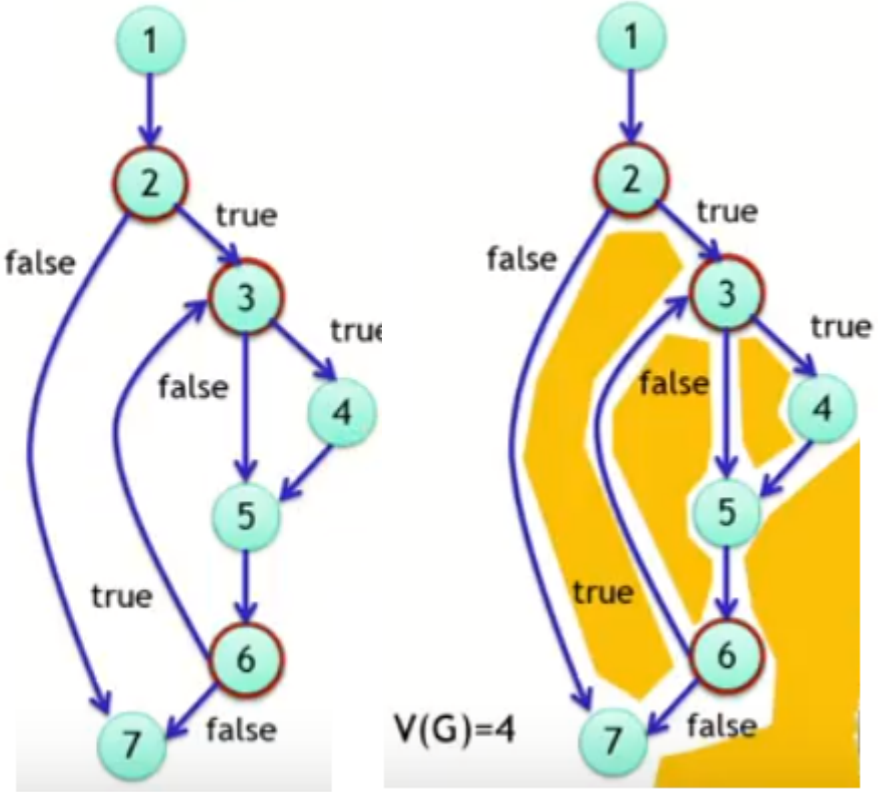
EJEMPLO

```java
public class PruebaGrafoFlujo {
    public static int contarLetras(String cadena, char letra){
        int contador=0, n=0, lon;
        lon = cadena.length();
        if (lon > 0) {
            do {
                if (cadena.charAt(contador)==letra)
                    n++;
                contador++;
                lon--;
            } while(lon>0);
        }
        return n;
    }

    public static void main (String args[]) {
        int cuenta = PruebaGrafoFlujo.contarLetras("Hola  mundo mágico",'o');
        System.out.println(cuenta);
    }
}
```

EJEMPLO

Código fuente, grafo de flujo y complejidad ciclomática

```
public class PruebaGrafoFlujo {
    public static int contarLetras(String cadena, char letra){
        int contador=0, n=0, lon;
        lon = cadena.length();                    1
    2   if (lon > 0) {
            do {
                if (cadena.charAt(contador)==letra)  3
                    n++;  4
                contador++;
                lon--;                            5
        6   } while(lon>0);
        }
        return n;  7
    }

    public static void main (String args[]) {
        int cuenta = PruebaGrafoFlujo.contarLetras("Hola  mundo mágico",'o');
        System.out.println(cuenta);
    }
}
```



La complejidad ciclomática es 4, por lo que habrá un máximo de 4 caminos independientes

EJEMPLO

Construir los caminos. Cada camino tiene que añadir una nueva arista:

1. 1-2-7
2. 1-2-3-4-5-6-7
3. 1-2-3-5-6-7
4. 1-2-3-4-5-6-3-5-6-7 (**no es único**, ya que 1-2-3-5-6-3-4-5-6-7 también añade la arista 6-3)

¡**No hace falta seguir**, pues ya tenemos 4 caminos!

EJEMPLO

Número del Camino	Caso de Prueba	Resultado Esperado

Los que nos queda ahora es construir los casos de prueba, uno para cada camino independiente, asignando valores de entrada y viendo el resultado esperado:

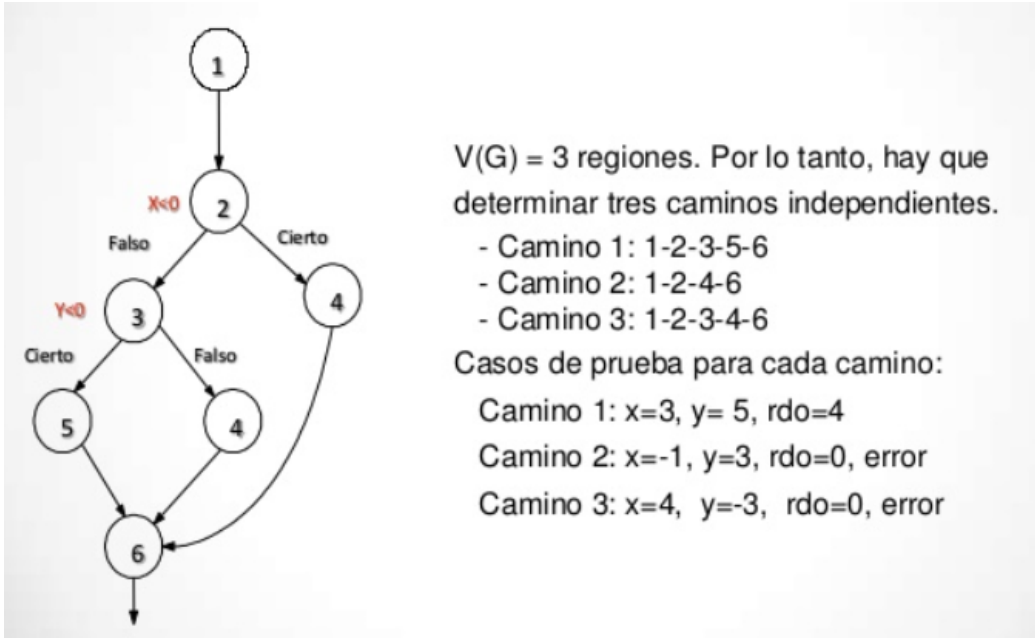
Casos de prueba

Número	Camino independiente	*cadena*	*letra*	*n*
1	1-2-7	""	'a'	0
2	1-2-3-4-5-6-7	"a"	'a'	1
3	1-2-3-5-6-7	"b"	'a'	0
4	1-2-3-4-5-6-3-5-6-7	"ab"	'a'	1

Si al ejecutar estos casos de prueba el valor resultante de *n* (nº de ocurrencias de *letra* en *cadena*) no coincide con este habremos descubierto un error

EJEMPLO

```
void calcula_e_imprime_media(float x, float y)
{
    float resultado;
    resultado = 0.0;
    if (x < 0 || y < 0)
        printf("x e y deben ser positivos");
    else {
        resultado = (x + y)/2;
        printf("La media es %f\n", resultado);
    }
}
```



V(G) = 3 regiones. Por lo tanto, hay que determinar tres caminos independientes.

- Camino 1: 1-2-3-5-6
- Camino 2: 1-2-4-6
- Camino 3: 1-2-3-4-6

Casos de prueba para cada camino:

Camino 1: x=3, y= 5, rdo=4

Camino 2: x=-1, y=3, rdo=0, error

Camino 3: x=4, y=-3, rdo=0, error

ACTIVIDAD

3. Dado el siguiente programa en Java:

A) calcular la complejidad ciclomática de McCabe

B) definir los casos de prueba

```java
import java.util.Scanner;

public class Maximo {
    public static void main (String args[]) {
        Scanner sc = new Scanner(System.in);

        int x,y,z,max;

        System.out.println("Introduce x,y,z: ");
        x =  sc.nextInt();
        y =  sc.nextInt();
        z =  sc.nextInt();

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

VÍDEO: CAMINO BÁSICO. EJEMPLO PRÁCTICO [ENLACE]

COMPLEJIDAD CICLOMÁTICA

La complejidad ciclomática nos da una idea de cómo de complicado/arriesgado es nuestro programa

Complejidad ciclomática	Tipo de programa - evaluación del riesgo
1-10	programa sencillo - riesgo bajo
11-20	programa medio - riesgo moderado
21-50	programa complejo - riesgo alto
>50	programa arriesgado - riesgo muy alto

ACTIVIDAD

4. Realiza a partir del siguiente enunciado y pseudocódigo el grafico de flujo del programa, calcula la complejidad ciclomática, determina los caminos independiente y los casos de prueba. Implementa los mismos en JUnit4

Un negocio vende un producto y realiza envíos por mensajería. El producto tiene un precio base de $125 por unidad, pero si se adquieren más de 100 unidades se ofrece un descuento del 5%; si se adquieren más de 1000 unidades el descuento será de 10%. Sobre el precio se agrega el flete que depende de cuantas cajas se requieran. Cada caja puede almacenar hasta 4 unidades y tiene un costo de $50. El programa lee la cantidad pedida y debe calcular el precio total, el número de cajas y el descuento que se aplicó. Así se tiene que al pedir 150 unidades se requieren 38 cajas y alcanza 5% de descuento, por lo que el costo del producto será $125 X 150 = $18750; el descuento será de $937.50, por lo que deberá pagar $17812.50 más el flete: 38 X $50 = $1,900; El total será $19712.50. Expresado como caso de prueba será:

Entrada	Salida
Cantidad=150	Costo=19712.50; núm. cajas= 38; descuento 937.50

```
1 Leer cant
2 costo1=cant*125
3 numcaj=redondea(0.5+cant/4)
4 flete=numcaj*50
5 Si cant > 1000 entonces
6     desc = costo1*.1
7 de otro modo
8     Si cant>100 entonces
9         desc = costo1*0.05
10    de otro modo
11        desc = 0
12 costoTot =costo1+flete-desc
13 Escribe costoTot, numcaj, desc
```

(a) Código

PRUEBAS DE CAJA BLANCA

Mutación

Se suele utilizar para verificar la bondad de las estrategias de prueba utilizadas.

Se basa en realizar ligeras modificaciones en el programa que darían lugar a un comportamiento anómalo del mismo (resultados distintos) y verificar si la estrategia de prueba utilizada es capaz de detectar estos cambios. (p.e. modificando el operador en una sentencia selectiva o iterativa, eliminando sentencias, etc…).

El aspecto más que importante a tener en cuenta en esta técnica es que hay saber elegir muy bien qué modificar, es decir cómo realizar la mutación, para provocar un comportamiento diferente.

REPASO TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA

Complete los espacios de esta definición de conceptos relacionada con los diferentes tipos de pruebas:

1. las pruebas_____prueban la funcionalidad del programa para el que se diseñan casos de prueba que comprueben las especificaciones del programa.

2. las pruebas_____se centran en la implementación de los programas para escoger los casos de prueba. Lo ideal sería buscar casos de prueba que recorran todos los caminos posibles del flujo de control del programa. Estas se centran en la estructura interna del programa analizando los caminos de ejecución.

REPASO TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA

Complete los espacios de esta definición de conceptos relacionada con los diferentes tipos de pruebas:

3. Una ventaja de las pruebas de_____es que son independientes del lenguaje o paradigma de programación utilizado, por lo que son válidas tanto para programación estructurada como para programación orientada a objetos.

4. El enfoque estructural o las pruebas de_____ , dentro de las pruebas unitarias, sirve para analizar el código en todas sus estructuras, en todos sus caminos del software. Pero existe otro tipo de pruebas que se basan en un enfoque más funcional, llamadas pruebas de_____

Thats All