

3.1. INTRODUCCIÓN

Las pruebas de software consisten en verificar y validar un producto software antes de su puesta en marcha. Constituyen una de las etapas del desarrollo de software, y básicamente consiste en probar la aplicación construida. Se integran dentro de las diferentes fases del ciclo de vida del software dentro de la ingeniería de software.

La ejecución de pruebas de un sistema involucra una serie de etapas que se nombraron en el Capítulo 1: planificación de pruebas, diseño y construcción de los casos de prueba, definición de los procedimientos de prueba, ejecución de las pruebas, registro de resultados obtenidos, registro de errores encontrados, depuración de los errores e informe de los resultados obtenidos.

En este Capítulo se estudian dos enfoques para el diseño de casos de prueba y diferentes técnicas en cada uno para probar el código de los programas.

3.2. TÉCNICAS DE DISEÑO DE CASOS DE PRUEBA

Un **caso de prueba** es un conjunto de entradas, condiciones de ejecución y resultado esperados, desarrollado para conseguir un objetivo particular o condición de prueba. Para llevar a cabo un caso de prueba, es necesario definir las precondiciones y post condiciones, identificar unos valores de entrada y conocer el comportamiento que debería tener el sistema ante dichos valores. Tras realizar ese análisis e introducir dichos datos en el sistema, se observará si su comportamiento es el previsto o no y por qué. De esta forma se determinará si el sistema ha pasado o no la prueba¹.

Para llevar a cabo el diseño de casos de prueba se utilizan dos técnicas o enfoques: **prueba de caja blanca** y **prueba de caja negra** (véase Figura 3.1). Las primeras se centran en validar la estructura interna del programa (necesitan conocer los detalles procedimentales del código) y las segundas se centran en validar los requisitos funcionales sin fijarse en el funcionamiento interno del programa (necesitan saber la funcionalidad que el código ha de proporcionar). Estas pruebas no son excluyentes y se pueden combinar para descubrir diferentes tipos de errores.

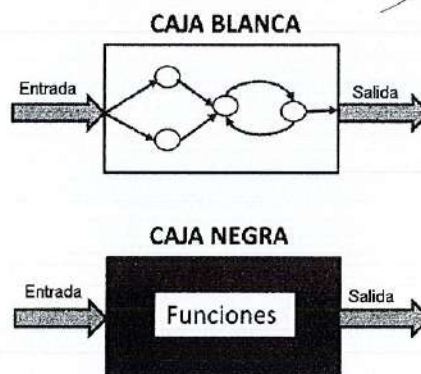


Figura 3.1. Pruebas de caja blanca y negra.

3.2.1. Pruebas de caja blanca

También se las conoce como *pruebas estructurales o de caja de cristal*. Se basan en el minucioso examen de los detalles procedimentales del código de la aplicación. Mediante esta técnica se pueden obtener casos de prueba que:

¹ GUÍA DE VALIDACIÓN Y VERIFICACIÓN. Inteco. Laboratorio Nacional de Calidad del Software.

- Garanticen que se ejecutan al menos una vez todos los caminos independientes de cada módulo.
- Ejecuten todas las sentencias al menos una vez.
- Ejecuten todas las decisiones lógicas en su parte verdadera y en su parte falsa.
- Ejecuten todos los bucles en sus límites.
- Utilicen todas las estructuras de datos internas para asegurar su validez.

Una de las técnicas utilizadas para desarrollar los casos de prueba de caja blanca es la **prueba del camino básico** que se estudiará más adelante.

3.2.2. Pruebas de caja negra

Estas pruebas se llevan a cabo sobre la interfaz del software, no hace falta conocer la estructura interna del programa ni su funcionamiento. Se pretende obtener casos de prueba que demuestren que las funciones del software son operativas, es decir, que las salidas que devuelve la aplicación son las esperadas en función de las entradas que se proporcionen.

A este tipo de pruebas también se les llama **prueba de comportamiento**. El sistema se considera como una caja negra cuyo comportamiento sólo se puede determinar estudiando las entradas y las salidas que devuelve en función de las entradas suministradas.

Con este tipo de pruebas se intenta encontrar errores de las siguientes categorías:

- Funcionalidades incorrectas o ausentes.
- Errores de interfaz.
- Errores en estructuras de datos o en accesos a bases de datos externas.
- Errores de rendimiento.
- Errores de inicialización y finalización.

Existen diferentes técnicas para confeccionar los casos de prueba de caja negra, algunos son: **clases de equivalencia**, **análisis de valores límite**, métodos basados en grafos, pruebas de comparación, etc. En este capítulo se estudiarán algunas de estas técnicas.

3.3. ESTRATEGIAS DE PRUEBAS DEL SOFTWARE

La estrategia de prueba del software se puede ver en el contexto de una espiral (véase Figura 3.2):

- En el vértice de la espiral comienza la **prueba de unidad**. Se centra en la unidad más pequeña de software, el módulo tal como está implementado en código fuente.
- La prueba avanza para llegar a la **prueba de integración**. Se toman los módulos probados mediante la prueba de unidad y se construye una estructura de programa que esté de acuerdo con lo que dicta el diseño. El foco de atención es el diseño.

Algunas de las herramientas que se utilizan para pruebas unitarias son: JUnit, CPPUnit, PHPUnit, etc.

3.3.2. Prueba de integración

En este tipo de prueba se observa como interaccionan los distintos módulos. Se podría pensar que esta prueba no es necesaria, ya que, si todos los módulos funcionan por separado, también deberían funcionar juntos. Realmente el problema está aquí, en comprobar si funcionan juntos.

Existen dos enfoques fundamentales para llevar a cabo las pruebas:

- **Integración no incremental o *big bang*.** Se prueba cada módulo por separado y luego se combinan todos de una vez y se prueba todo el programa completo. En este enfoque se encuentran gran cantidad de errores y la corrección se hace difícil.
- **Integración incremental.** El programa completo se va construyendo y probando en pequeños segmentos, en este caso los errores son más fáciles de localizar. Se dan dos estrategias *Ascendente* y *Descendente*. En la integración *Ascendente* la construcción y prueba del programa empieza desde los módulos de los niveles más bajos de la estructura del programa. En la *Descendente* la integración comienza en el módulo principal (programa principal) moviéndose hacia abajo por la jerarquía de control.

La Figura 3.4 representa varios módulos y la interconexión entre ellos. El módulo principal es el que está en la raíz, M1. La figura muestra una estrategia de integración *Ascendente*. Se empieza probando los módulos de más bajo nivel en la jerarquía modular del sistema y se procede a probar la integración de abajo hacia arriba hasta llegar al programa principal M1.

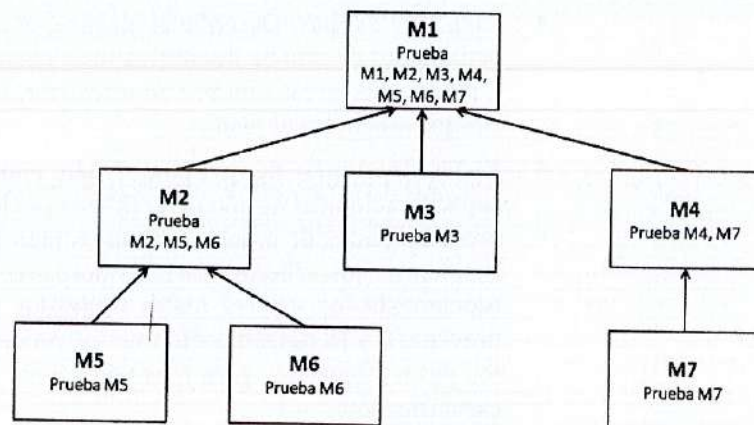


Figura 3.4. Prueba de integración *Ascendente*.

3.3.3. Prueba de validación

La validación se consigue cuando el software funciona de acuerdo con las expectativas razonables del cliente definidas en el documento de especificación de requisitos del software o ERS. Se llevan a cabo una serie de pruebas de caja negra que demuestran la conformidad con los requisitos. Las técnicas que utilizaremos son:

- **Prueba Alfa.** Se lleva a cabo por el cliente o usuario en el lugar de desarrollo. El cliente utiliza el software de forma natural bajo la observación del desarrollador que irá registrando los errores y problemas de uso.

- **Prueba Beta.** Se lleva a cabo por los usuarios finales del software en su lugar de trabajo. El desarrollador no está presente. El usuario registra todos los problemas que encuentra, reales y/o imaginarios, e informa al desarrollador en los intervalos definidos en el plan de prueba. Como resultado de los problemas informados el desarrollador lleva a cabo las modificaciones y prepara una nueva versión del producto.

3.3.4. Prueba del sistema

La prueba del sistema está formada por un conjunto de pruebas cuya misión es ejercitar profundamente el software. Son las siguientes:

- **Prueba de recuperación.** En este tipo de prueba se fuerza el fallo del software y se verifica que la recuperación se lleva a cabo apropiadamente.
- **Prueba de seguridad.** Esta prueba intenta verificar que el sistema está protegido contra accesos ilegales.
- **Prueba de resistencia (Stress).** Trata de enfrentar el sistema con situaciones que demandan gran cantidad de recursos, por ejemplo, diseñando casos de prueba que requieran el máximo de memoria, incrementando la frecuencia de datos de entrada, que den problemas en un sistema operativo virtual, etc.

3.4. DOCUMENTACIÓN PARA LAS PRUEBAS

El estándar IEEE 829-1998 describe el conjunto de documentos que pueden producirse durante el proceso de prueba. Son los siguientes:

- **Plan de Pruebas.** Describe el alcance, el enfoque, los recursos y el calendario de las actividades de prueba. Identifica los elementos a probar, las características que se van a probar, las tareas que se van a realizar, el personal responsable de cada tarea y los riesgos asociados al plan.
- **Especificaciones de prueba.** Están cubiertas por tres tipos de documentos: la especificación del diseño de la prueba (se identifican los requisitos, casos de prueba y procedimientos de prueba necesarios para llevar a cabo las pruebas y se especifica la función de los criterios de pasa no-pasa), la especificación de los casos de prueba (documenta los valores reales utilizados para la entrada, junto con los resultados previstos), y la especificación de los procedimientos de prueba (donde se identifican los pasos necesarios para hacer funcionar el sistema y ejecutar los casos de prueba especificados).
- **Informes de pruebas.** Se definen cuatro tipos de documentos: un informe que identifica los elementos que están siendo probados, un registro de las pruebas (donde se registra lo que ocurre durante la ejecución de la prueba), un informe de incidentes de prueba (describe cualquier evento que se produce durante la ejecución de la prueba que requiere mayor investigación) y un informe resumen de las actividades de prueba.

3.5. PRUEBAS DE CÓDIGO

La prueba del código consiste en la ejecución del programa (o parte de él) con el objetivo de encontrar errores. Se parte para su ejecución de un conjunto de entradas y una serie de condiciones de ejecución; se observan y registran los resultados y se comparan con los resultados esperados. Se observará si el comportamiento del programa es el previsto o no y por qué.

Para las pruebas de código se van a mostrar diferentes técnicas que dependerán del tipo de enfoque utilizado: de caja blanca, se centran en la estructura interna del programa; o de caja negra, más centrado en las funciones, entradas y salidas del programa.

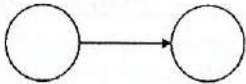
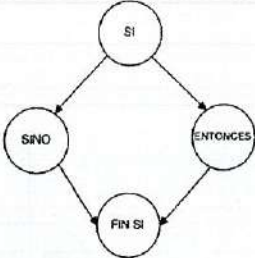
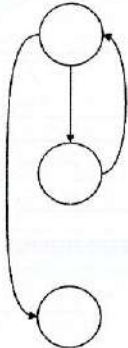

3.5.1. Prueba del camino básico

La prueba del camino básico es una técnica de prueba de caja blanca que permite al diseñador de casos de prueba obtener una medida de la complejidad lógica de un diseño procedimental y usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución. Los casos de prueba obtenidos del conjunto básico garantizan que durante la prueba se ejecuta por lo menos una vez cada sentencia del programa².

Para la obtención de la **medida de la complejidad lógica** (o **complejidad ciclomática**) emplearemos una representación del flujo de control denominada *grafo de flujo* o *grafo del programa*.

NOTACIÓN DE GRAFO DE FLUJO

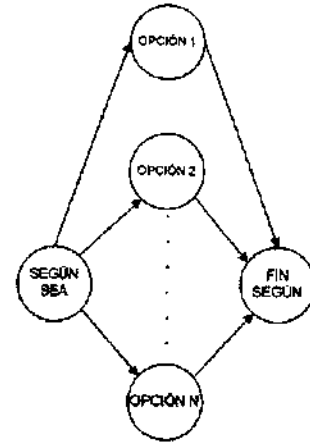
El grafo de flujo de las estructuras de control se representa de la siguiente forma:

ESTRUCTURA	GRAFO DE FLUJO
SECUENCIAL Instrucción 1 Instrucción 2 Instrucción n	
CONDICIONAL Si <condición> Entonces <Instrucciones> Si no <Instrucciones> Fin si	
HACER MIENTRAS Mientras <condición> Hacer <instrucciones> Fin mientras	
REPETIR HASTA Repetir <instrucciones> Hasta que <condición>	

² Ingeniería del software. Un enfoque práctico. Roger S. Pressman.

CONDICIONAL MÚLTIPLE

Según sea <variable> **Hacer**
Caso opción 1:
 <Instrucciones>
Caso opción 2:
 <Instrucciones>
Caso opción 3:
 <Instrucciones>
Otro caso:
 <Instrucciones>
Fin según



Donde cada círculo representa una o más sentencias, sin bifurcaciones, en pseudocódigo o código fuente. A continuación, construimos un grafo de flujo a partir de un diagrama de flujo (Figura 3.5).

Ejemplo 1: se muestra el diagrama de flujo y el grafo de flujo para un programa que lee 10 números de teclado y muestra cuantos de los números leídos son pares y cuántos son impares. Para comprobar si el número es par o impar utilizamos el operador % de Java (devuelve el resto de la división) que devuelve 0 si es par. La estructura principal corresponde a un MIENTRAS (o WHILE) y dentro hay una estructura SI (o IF). Véase Figura 3.5.

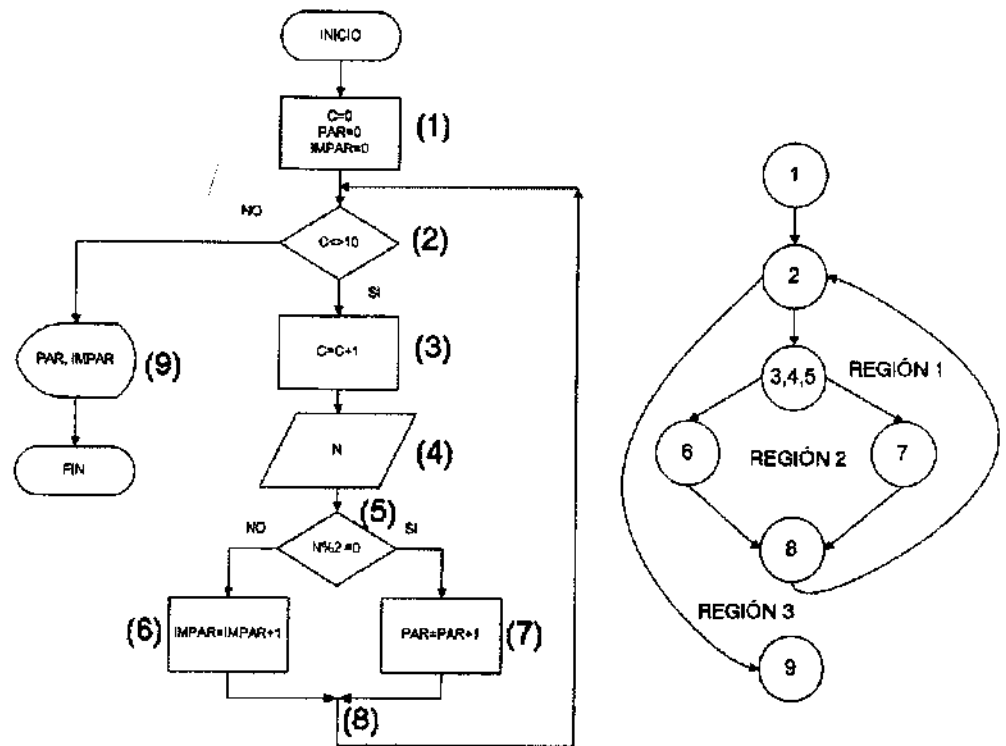


Figura 3.5. Diagrama de flujo y grafo de flujo.

Se numeran en el diagrama de flujo cada uno de los símbolos, y los finales de las estructuras de control (por ejemplo, el (9) es el final del WHILE) aunque no tengan ningún símbolo (por ejemplo, el número (8) es el final de una estructura condicional).

Cada círculo del grafo de flujo se llama **nodo**. Representa una o más sentencias procedimentales. Un solo nodo se puede corresponder con una secuencia de símbolos del proceso y un rombo de decisión. Un ejemplo es el nodo numerado como 3, 4, 5.

Las flechas del grafo de flujo se denominan **aristas o enlaces** y representan el flujo de control, como en el diagrama de flujo. Una arista termina en un nodo, aunque el nodo no tenga ninguna sentencia procedimental; es el caso del nodo numerado como 8.

Las áreas delimitadas por aristas y nodos se llaman **regiones**, el area exterior del grafo es otra región más. En el ejemplo se muestran 3 regiones, 8 aristas y 7 nodos.

El nodo que contiene una condición se llama **nodo predicado** y se caracteriza porque de él salen dos o más aristas. En el ejemplo anterior se muestran 2 nodos predicado, el representado por el número 2 y el representado por 3, 4, 5. Únicamente de estos nodos pueden salir dos aristas.

Cuando en un diseño procedimental se encuentran condiciones compuestas, es decir cuando en una condición aparecen uno o más operadores (por ejemplo, que la edad sea < de 20 y el curso primero de DAM) se complica la generación del grafo de flujo. En este caso se crea un nodo aparte para cada una de las condiciones, véase la Figura 3.6. El grafo de la derecha muestra la situación: Nodo A edad < 20, nodo B curso primero de DAM:

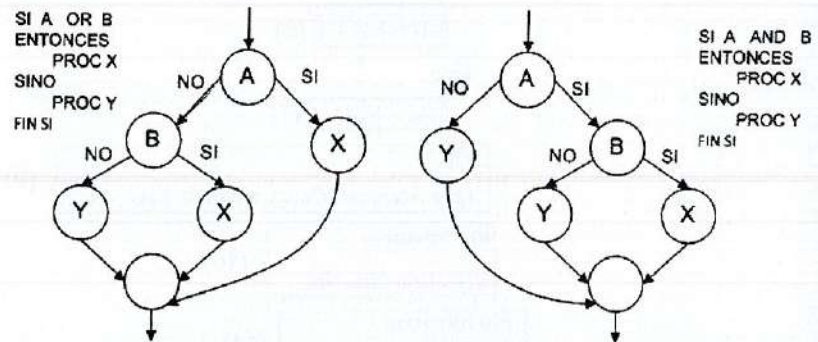


Figura 3.6. Lógica compuesta.

La Figura 3.7 muestra un pseudocódigo y el grafo de flujo correspondiente usando la lógica compuesta:

Inicio

Leer edad (1)
Leer curso

Si edad < 20 y curso = "1DAM" Entonces (2) (3)

Mostrar "Aceptado" (4)

Sino

Mostrar "No Aceptado" (5)

Finsi (6)

Fin

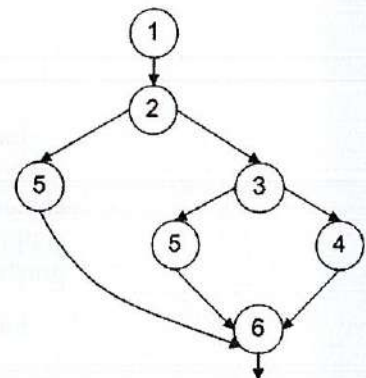


Figura 3.7. Ejemplo de lógica compuesta.

Ejemplo 2: se dispone de un fichero de *Alumnos* con la siguiente estructura de registro: *Curso*, *Nombre*, *Sexo* (puede ser H o M) y *Nota*. El fichero está ordenado ascendente por curso. Vamos a realizar un proceso que lea los registros del fichero y muestre por cada curso el número de hombres y el número de mujeres. Se construirá el pseudocódigo y el grafo de flujo véase Figura 3.8. En el pseudocódigo se muestran en un cuadro las sentencias que representan en un nodo; al lado hay un número entre paréntesis que se corresponde con su posición en el grafo de flujo (recuerda que cada nodo puede corresponder a una o más sentencias de un proceso y una condición). La estructura principal corresponde a un MIENTRAS (mientras haya registros en el fichero), dentro hay otro MIENTRAS (para tratar el mismo curso si hay registros) y dentro hay una estructura SI (para contar los hombres y mujeres). Este segundo ejemplo se muestran 5 regiones, 14 aristas y 11 nodos, 4 de ellos son predicados.

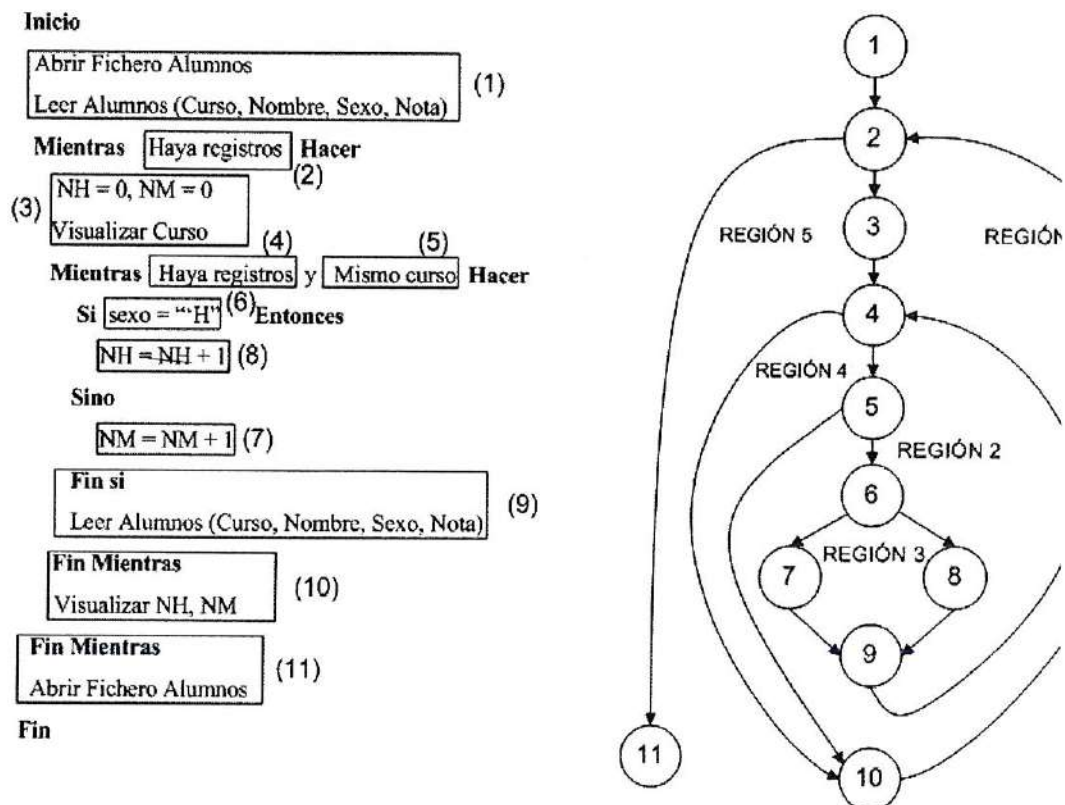


Figura 3.8. Pseudocódigo y grafo de flujo.

COMPLEJIDAD CICLOMÁTICA

La complejidad ciclomática es una métrica del software que proporciona una medida cuantitativa de la complejidad lógica de un programa³. En el contexto del método de prueba de camino básico, la complejidad ciclomática establece el número de caminos independientes en un conjunto básico de caminos de ejecución de un programa, y por lo tanto, el número de casos de prueba que se deben ejecutar para asegurar que cada sentencia se ejecuta al menos una vez.

La complejidad ciclomática $V(G)$ puede calcularse de tres formas:

³ Ingeniería del software. Un enfoque práctico. Roger S. Pressman.

registro:
nente por
a curso el
de flujo,
s que se
n su nodo
encias de
tras haya
siempre y
eres). En
on nodos

1. $V(G) = \text{Número de regiones del grafo.}$
2. $V(G) = \text{Aristas} - \text{Nodos} + 2.$
3. $V(G) = \text{Nodos predicado} + 1$

Para el *Ejemplo 1*, la complejidad ciclomática es 3.

1. $V(G) = \text{Número de regiones del grafo} = 3.$
2. $V(G) = \text{Aristas} - \text{Nodos} + 2 = 8 - 7 + 2 = 3.$
3. $V(G) = \text{Nodos predicado} + 1 = 2 + 1 = 3$

Para el *Ejemplo 2*, la complejidad ciclomática es 5.

1. $V(G) = \text{Número de regiones del grafo} = 5.$
2. $V(G) = \text{Aristas} - \text{Nodos} + 2 = 14 - 11 + 2 = 5$
3. $V(G) = \text{Nodos predicado} + 1 = 4 + 1 = 5$

Se establecen los siguientes valores de referencia de la complejidad ciclomática:

Complejidad ciclomática	Evaluación de riesgo
Entre 1 y 10	Programas o métodos sencillos, sin mucho riesgo.
Entre 11 y 20	Programas o métodos más complejos, riesgo moderado.
Entre 21 y 50	Programas o métodos complejos, alto riesgo.
Mayor que 50	Programas o métodos no testeables, muy alto riesgo.

El valor de $V(G)$ nos da el número de caminos independientes del conjunto básico de un programa. Un camino independiente es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una condición. 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 camino³.

Para el *Ejemplo 1*, un conjunto de caminos independientes será:

- Camino 1: 1 - 2 - 9
- Camino 2: 1 - 2 - 3, 4, 5 - 6 - 8 - 2 - 9
- Camino 3: 1 - 2 - 3, 4, 5 - 7 - 8 - 2 - 9

Para el *Ejemplo 2*, un conjunto de caminos independientes será:

- Camino 1: 1 - 2 - 11
- Camino 2: 1 - 2 - 3 - 4 - 10 - 2 - 11
- Camino 3: 1 - 2 - 3 - 4 - 5 - 10 - 2 - 11
- Camino 4: 1 - 2 - 3 - 4 - 5 - 6 - 8 - 9 - 4 - 10 - 2 - 11
- Camino 5: 1 - 2 - 3 - 4 - 5 - 6 - 7 - 9 - 4 - 10 - 2 - 11

CIÓN 1

medida
ueba del
entes del
casos de
EZ.

OBTENCIÓN DE LOS CASOS DE PRUEBA

El último paso de la prueba del camino básico es construir los casos de prueba que fuerzan ejecución de cada camino. Con el fin de comprobar cada camino, debemos escoger los casos de prueba de forma que las condiciones de los nodos predicado estén adecuadamente establecidas. Una forma de representar el conjunto de casos de prueba es como se muestra en la siguiente tabla. Por ejemplo, representamos los casos de prueba para el **Ejemplo 1**, en la Figura 3.9. muestran los nodos predicado con sus condiciones para que sea más fácil obtener los casos de prueba:

Camino	Caso de prueba	Resultado esperado
1	Escoger algún valor de C tal que NO se cumpla la condición $C < > 10$. $C = 10$	Visualizar el número de pares y el de impares.
2	Escoger algún valor de C tal que SÍ se cumpla la condición $C < > 10$. Escoger algún valor de N tal que NO se cumpla la condición $N \% 2 = 0$. $C = 1, N = 5$	Contar números impares.
3	Escoger algún valor de C tal que SÍ se cumpla la condición $C < > 10$. Escoger algún valor de N tal que SÍ se cumpla la condición $N \% 2 = 0$. $C = 2, N = 4$	Contar números pares.

Camino 1: 1-2-8

Camino 2: 1-2-3, 4, 5-6-8-2-9

Camino 3: 1-2-3, 4, 5-7-8-2-9

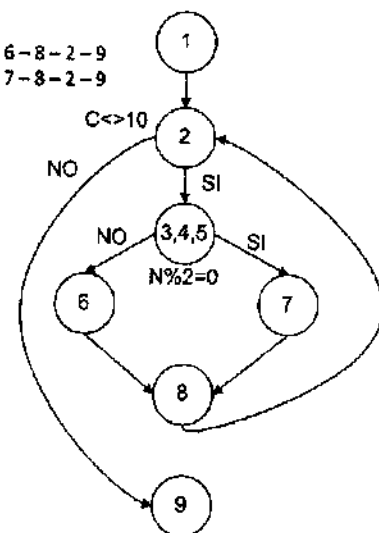


Figura 3.9. Nodos predicado del Ejemplo 1 con sus condiciones.

El camino 1 no puede ser probado por sí solo, debe ser probado como parte de las pruebas de los caminos 2 y 3.

Ejemplo 3: en la Figura 3.10 se muestra una función Java y el correspondiente grafo de flujo. Calculamos la complejidad ciclomática, calculamos los caminos independientes y elaboramos los casos de prueba. El grafo de flujo tiene 8 aristas, 7 nodos, 2 nodos predicados y 3 regiones. El valor de $V(G)$ es 3, por tanto tenemos tres caminos independientes que probar:


```

static void visualizarMedia(float x, float y) {
    float resultado = 0; (1)
    if (x < 0 || y < 0) (3)
    (2) System.out.println("X e Y deben ser positivos"); (4)
    else {
        resultado = (x + y) / 2;
        System.out.println("La media es: " + resultado); (5)
    }
    (6)
}

```

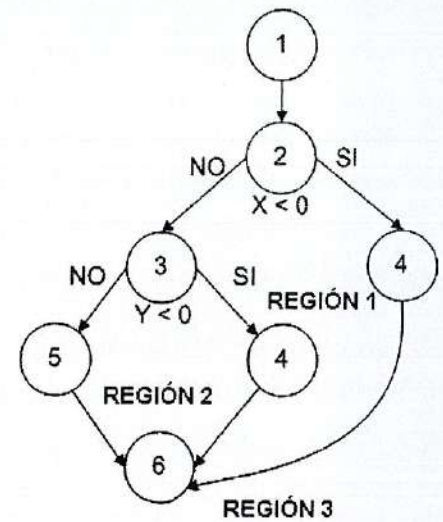


Figura 3.10. Ejemplo 3.

Los caminos independientes y los casos de prueba para cada camino se muestran en la siguiente tabla:

Camino	Caso de prueba	Resultado esperado
Camino 1: 1-2-3-5-6	Escoger algún X e Y tal que NO se cumpla la condición $X < 0 \vee Y < 0$. $X=4, Y=5$ <code>visualizarMedia(4,5)</code>	Visualiza: La media es: 4.5
Camino 2: 1-2-4-6	Escoger algún X tal que SÍ se cumpla la condición $X < 0$ (Y puede ser cualquier valor) $X=-4, Y=5$ <code>visualizarMedia(-4,5)</code>	Visualiza: X e Y deben ser positivos
Camino 3: 1-2-3-4-6	Escoger algún X tal que NO cumpla la condición $X < 0$ y escoger algún Y que SÍ cumpla la condición $Y < 0$ $X=4, Y=-5$ <code>visualizarMedia(4,-5)</code>	Visualiza: X e Y deben ser positivos

Ejemplo 4: En la Figura 3.11 se muestra un pseudocódigo y el correspondiente grafo de flujo. Calculamos la complejidad ciclomática, los caminos independientes y elaboramos los casos de prueba. El grafo de flujo tiene 8 aristas, 7 nodos, 2 nodos predicho y 3 regiones. El valor de $V(G)$ es 3, por tanto tenemos tres caminos independientes que probar:

Camino	Caso de prueba	Resultado esperado
Camino 1: 1-2-4-5-7	$\text{cant} = 200, \text{pvp} = 125$	Descuento del 5% Visualizar el importeTotal 23750
Camino 2: 1-2-4-6-7	$\text{cant} = 5, \text{pvp} = 125$	Descuento 0. Visualizar el importeTotal 625
Camino 3: 1-2-3-7	$\text{cant} = 2000, \text{pvp} = 125$	Descuento 10%. Visualizar el importeTotal 225000

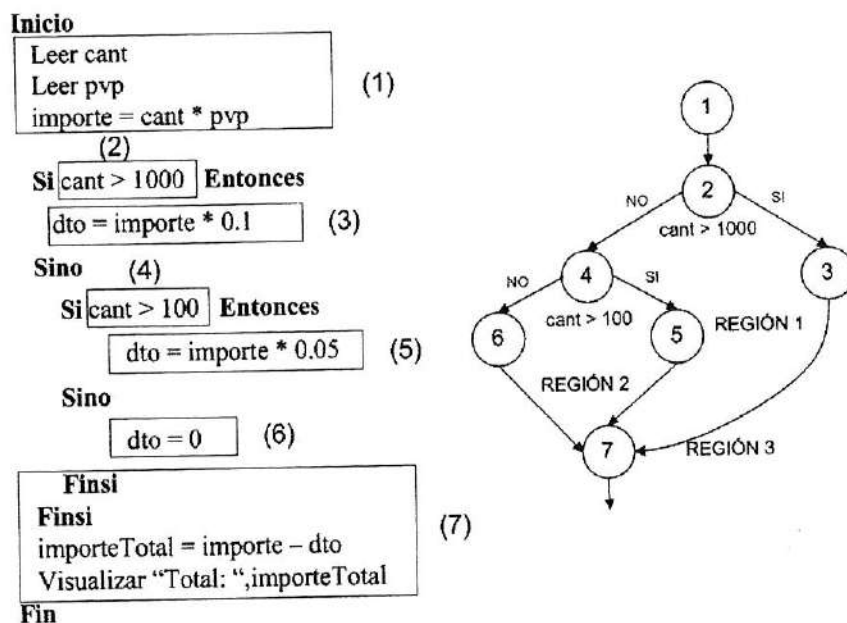


Figura 3.11. Ejemplo 4.

ACTIVIDAD 3.1

Realiza los Ejercicios 4 y 5.

3.5.2. Partición o clases de equivalencia

La partición equivalente es un método de prueba de caja negra que divide los valores de campos de entrada de un programa en clases de equivalencia. Por ejemplo, supongamos un campo de entrada llamado *número de empleado*, definido con una serie de condiciones: número de 3 dígitos y el primero no puede ser 0. Entonces se puede definir una clase de equivalencia válida: *número de empleado < 100*; y otra válida: *número de empleado comprendido entre 999*.

Para identificar las clases de equivalencia se examina cada condición de entrada (son del dominio de valores de la entrada y normalmente son una frase en la especificación) divide en dos o más grupos. Se definen dos tipos de clases de equivalencia:

- **Clases válidas:** son los valores de entrada válidos.
- **Clases no válidas:** son los valores de entrada no válidos.

Las clases de equivalencia se definen según una serie de directrices:

1. Si una condición de entrada especifica un **rango**, se define una clase de equivalencia válida y dos no válidas. Ejemplo de rango: la nota debe tener un valor entre 1 y 10.
2. Si una condición de entrada requiere un **valor específico**, se define una clase de equivalencia válida y dos no válidas. Ejemplo de valor específico: el número de departamento puede ser blanco o tener 2 dígitos.
3. Si una condición de entrada especifica un **miembro de un conjunto**, se define una clase de equivalencia válida y una no válida. Ejemplo: el curso puede tener los siguientes valores: "1DAM", "2DAM", "1SMR" y "2SMR".

4. Si una condición de entrada es **lógica**, se define una clase de equivalencia válida y una no válida. Ejemplo de entrada lógica: el salario debe ser > 0 .

La siguiente tabla resume el número de clases de equivalencia válidas y no válidas que hay que definir para cada tipo de condición de entrada:

Condiciones de entrada	Nº de Clases de equivalencia válidas	Nº de Clases de equivalencia no válidas
1. Rango	1 CLASE VÁLIDA Contempla los valores del rango	2 CLASES NO VÁLIDAS Un valor por encima del rango Un valor por debajo del rango
2. Valor específico	1 CLASE VÁLIDA Contempla dicho valor	2 CLASES NO VÁLIDAS Un valor por encima Un valor por debajo
3. Miembro de un conjunto	1 CLASE VÁLIDA Una clase por cada uno de los miembros del conjunto	1 CLASE NO VÁLIDA Un valor que no pertenece al conjunto
4. Lógica	1 CLASE VÁLIDA Una clase que cumpla la condición	1 CLASE NO VÁLIDA Una clase que no cumpla la condición

Ejemplo 5: se va a realizar una entrada de datos de un empleado por pantalla gráfica, se definen 3 campos de entrada y una lista para elegir el oficio. La aplicación acepta los datos de esta manera:

- *Empleado*: número de tres dígitos que no empieza por 0.
- *Departamento*: en blanco o número de dos dígitos.
- *Oficio*: Analista, Diseñador, Programador o Elige oficio.

Si la entrada es correcta el programa asigna un salario (que se muestra en pantalla) a cada empleado según estas normas:

- S1 si el *Oficio* es Analista se asigna 2500.
- S2 si el *Oficio* es Diseñador se asigna 1500.
- S3 si el *Oficio* es Programador se asigna 2000.

Si la entrada no es correcta el programa muestra un mensaje indicando la entrada incorrecta:

- ER1 si el *Empleado* no es correcto.
- ER2 si el *Departamento* no es correcto
- ER3 si no se ha elegido *Oficio*.

Para representar las clases de equivalencia para cada condición de entrada se puede usar una tabla. En cada fila se definen las clases de equivalencia para la condición de entrada, se añade un código a cada clase definida (válida y no válida) para usarlo en la definición de los casos de prueba. Las clases de equivalencia para el *Departamento* para indicar que es un número de dos dígitos se pueden expresar como valor específico o como rango, cualquier opción es válida. Por ello esta última se ha codificado como V3', NV4' y NV5':

Condición de entrada	Clases de equivalencia	Clases Válidas	COD	Clases no Válidas	C
Empleado	Rango	100 >= Empleado <= 999	V1	Empleado < 100 Empleado > 999	N N
Departamento	Lógica (puede estar o no)	En blanco.	V2	No es un número.	N
	Valor	Cualquier número de dos dígitos.	V3	Número de más de 2 dígitos. Número de menos de 2 dígitos	N N
	Rango	10 >= Departamento <= 99	V3'	Dep > 99 Dep < 10	N N
Oficio	Miembro de un conjunto	Oficio = "Programador"	V4	Oficio = "Elige oficio"	N
		Oficio = "Analista"	V5		
		Oficio = "Diseñador"	V6		

A partir de esta tabla se generan los casos de prueba. Utilizamos las condiciones de entrada y las clases de equivalencia (a las que se asignó un código en la columna COD, también se le ha asignado un número a cada clase). Los representamos en otra tabla donde cada fila representa un caso de prueba con los códigos de las clases de equivalencia que se aplican a los valores asignados a las condiciones de entrada y el resultado esperado según el enunciado del problema:

CASO DE PRUEBA	Clases de equivalencia	CONDICIONES DE ENTRADA			Resultado esperado
		Empleado	Departamento	Oficio	
CP1	V1, V3, V4	200	20	Programador	S3
CP2	V1, V2, V5	250		Analista	S1
CP3	V1, V3, V6	450	30	Diseñador	S2
CP4	V1, V2, V4	220		Programador	S3
CP5	NV1, V3, V6	90	35	Diseñador	ER1
CP6	V1, NV3, V5	100	AD	Analista	ER2
CP7	V1, V2, NV8	300		Elige oficio	ER3
CP8	V1, NV4, V6	345	123	Diseñador	ER2
....					

Al rellenar la tabla de casos de prueba se han tenido en cuenta estas dos reglas:

- los casos de prueba válidos (CP1, CP2, CP3 y CP4) cubren tantas clases de equivalencia válidas como sea posible y
- los casos de prueba no válidos (CP5, CP6, CP7 y CP8) cubren una sola clase de equivalencia no válida (si se prueban múltiples clases de equivalencia no válidas en el mismo caso de prueba, puede ocurrir que algunas de estas pruebas nunca se ejecuten porque la primera enmascara a las otras o termina la ejecución del caso de prueba).

Los casos de prueba se van añadiendo a la tabla hasta que todas las clases de equivalencia válidas y no válidas hayan sido cubiertas. Por ejemplo, a la tabla anterior le faltan clases de equivalencia válidas: (V1, V2, V6) y (V1, V3, V5); y no válidas (NV2, V2, V4) y (V1, V6).

Clases	COD
	NV1
	NV2
	NV3
le 2	NV4
s de	NV5
	NV4'
	NV5'
io"	NV8

es de entrada y
también se podría
onde cada fila
se aplican, los
enunciado del

Resultado esperado
S3
S1
S2
S3
ER1
ER2
ER3
ER2

tas clases de

sóla clase no
nismo caso de
en porque la

equivalencia
tan clases de
y (V1, NV5,

Ejemplo 6: tenemos una función Java que recibe un número entero y devuelve una cadena con el texto "Par" si el número recibido es par, o "Impar" si el número es impar.

```
public String parImpar(int numero) {
    String cad="";
    if(numero % 2 == 0)
        cad="Par";
    else
        cad="Impar";
    return cad;
}
```

En este ejemplo tenemos una condición de entrada que requiere un valor específico, un número entero, entonces según la segunda directriz se define una clase de equivalencia válida y dos no válidas. Como en este caso los números son tratados de forma diferente podemos crear una clase de equivalencia para cada entrada válida.

Condición de entrada	Clases de equivalencia	Clases Válidas	COD	Clases no Válidas	COD
numero	Valor Par	Cualquier número entero par	V7	Número impar Cadena	NV9 NV10
	Valor Impar	Cualquier número entero impar	V8	Número par Cadena	NV11 NV12

Los casos de prueba serían los siguientes:

CASO DE PRUEBA	Clases de equivalencia	CONDICIONES DE ENTRADA numero	Resultado esperado
CP1	V7	20	Par
CP2	V8	25	Impar
CP3	NV9	45	Error, número impar
CP4	NV10	"we"	Error, es una cadena
CP5	NV11	10	Error, número par
CP6	NV12	"ad"	Error, es una cadena"

ACTIVIDAD 3.2

Realiza el Ejercicio 2.

3.5.3. Análisis de valores límite

El análisis de valores límite se basa en que los errores tienden a producirse con más probabilidad (por razones que no están del todo claras) en los límites o extremos de los campos de entrada.

Esta técnica complementa a la anterior y los casos de prueba elegidos ejercitan los valores justo por encima y por debajo de los márgenes de la clase de equivalencia. Además, no solo se centra en las condiciones de entrada, sino que también se exploran las condiciones de salida definiendo las clases de equivalencia de salida.

Las reglas son las siguientes:

1. Si una condición de entrada especifica un **rango de valores**, se deben diseñar caso de prueba para los límites del rango y para los valores justo por encima y por debajo del rango. Por ejemplo, si una entrada requiere un rango de valores entero comprendidos entre 1 y 10, hay que escribir casos de prueba para el valor 1, 10, 0 y 11.
2. Si una condición de entrada especifica un **número de valores**, se deben diseñar caso de prueba que ejerciten los valores, máximo, mínimo, un valor justo por encima de máximo y un valor justo por debajo del mínimo. Por ejemplo, si el programa requiere de dos a diez datos de entradas, hay que escribir casos de prueba para 2, 10, 1 y 11 datos de entrada.
3. Aplicar la regla 1 para la condición de salida. Por ejemplo, si se debe aplicar sobre un campo de salida un descuento de entre un 10% mínimo y un 50% máximo (dependiendo del tipo de cliente); se generarán casos de prueba para 9,99%, 10%, 50% y 50,01%.
4. Usar la regla 2 para la condición de salida. Por ejemplo, si la salida de un programa es una tabla de temperaturas de 1 a 10 elementos, se deben diseñar casos de prueba para que la salida del programa produzca 0, 1, 10 y 11 elementos. Tanto en esta regla como en la anterior, hay que tener en cuenta que no siempre se podrán generar resultados fuera del rango de salida.
5. Si las estructuras de datos internas tienen **límites preestablecidos** (por ejemplo, un array de 100 elementos), hay que asegurarse de diseñar casos de prueba que ejerciten la estructura de datos en sus límites, primer y último elemento.

Ejemplo 7: determinar los casos de prueba para los siguientes elementos según las condiciones de entrada y de salida:

	Condiciones de entrada y salida	Casos de prueba
Código	Entero de 1 a 100	Valores: 0, 1, 100, 101
Puesto	Alfanumérico de hasta 4 caracteres	Longitud de caracteres: 0, 1, 4, 5
Antigüedad	De 0 a 25 años (Real)	Valores: 0, 25, -0.1, 25.1
Horas semanales	De 0 a 60	Valores: 0, 60, -1, 61
Fichero de entrada	Tiene de 1 a 100 registros	Para leer 0, 1, 100 y 101 registros
Fichero de salida	Podrá tener de 0 a 10 registros	Para generar 0, 10 y 11 registros (no se puede generar -1 registro)
Array interno	De 20 cadenas de caracteres	Para el primer y último elemento.

Ejemplo 8: partimos del *Empleado* (que tiene que ser un número de tres dígitos que empiece por 0) del **Ejemplo 5** del epígrafe anterior. Utilizando esta técnica, para la clase equivalencia V1 que representa un rango de valores ($100 \leq \text{Empleado} \leq 999$) se deben generar dos casos de prueba con el límite inferior y el superior del rango (para identificar estos casos de prueba utilizamos *V1a* para el límite inferior 100, y *V1b* para el superior 999):

CASO DE PRUEBA	Clases de equivalencia	CONDICIONES DE ENTRADA			Resultado esperado
		Empleado	Departamento	Oficio	
CP11	V1a, V3, V4	100	20	"Programador"	S3
CP12	V1b, V2, V5	999		"Analista"	S1
CP13	NV1, V3, V6	99	30	"Diseñador"	ER1
CP14	NV2, V2, V4	1000		"Programador"	ER1