

Tema 3

Diseño y realización de pruebas

Desarrollo de aplicaciones multiplataforma

Carlos Alberto Cortijo Bon



Esta obra está bajo una [Licencia Creative Commons Atribución-NoComercial-SinDerivadas 4.0 Internacional](https://creativecommons.org/licenses/by-nc-nd/4.0/).

Índice

1. Desarrollo del software	1
2. Pruebas unitarias: objetivos y tipos	3
3. Métodos de caja blanca	4
3.1. Método de los caminos básicos	4
3.1.1. Grafos de flujo	6
3.1.1.1. Sentencias condicionales	7
3.1.1.2. Sentencias de selección switch	8
3.1.1.3. Bucles	9
3.1.1.4. Condiciones compuestas	10
3.1.2. Conjunto mínimo de caminos básicos, complejidad ciclomática	12
3.1.3. Obtención de un conjunto de condiciones de prueba	13
3.2. Pruebas de bucles	23
4. Métodos de caja negra, método de las clases de equivalencia	24
4.1. Identificación de condiciones de entrada y sus correspondientes clases de equivalencia	25
4.2. Elaboración del conjunto de casos de prueba	28
4.3. Análisis de valores límites	30

1. Desarrollo del software

El proceso de desarrollo de software es un proceso complejo que incluye distintas etapas. A grandes rasgos se puede distinguir análisis, diseño y programación.

Según el modelo en V, en cada etapa se pasa a un nivel más bajo de abstracción, y en ella se genera:

- Documentación de análisis o diseño, que especifica cómo debe ser el sistema desarrollado. Esta se utiliza en la siguiente fase de desarrollo.
- Documentación de pruebas, que especifica las pruebas que deben realizarse para verificar que el sistema, una vez desarrollado, es conforme a lo establecido en el documento anterior. Si no fuera así, esto obligaría a revisar y, en su caso, hacer cambios en el sistema desarrollado.

Es decir, que cada fase de desarrollo lleva aparejada una fase de pruebas.

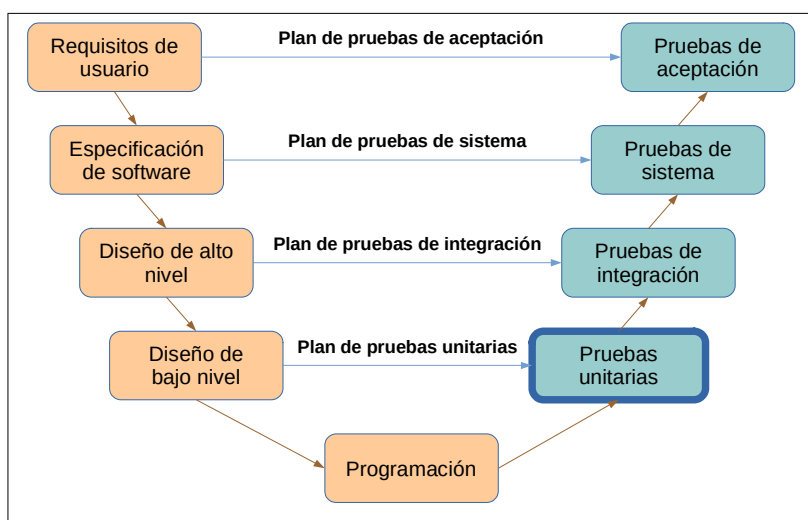


Ilustración 3.1: Modelo en V para desarrollo y prueba de software

Este tema se centrará en las pruebas unitarias, que se hacen sobre un componente individual, una vez terminado su desarrollo.

Las fases de desarrollo y de prueba son las siguientes.

Requisitos de usuario	<p>Llevada a cabo por un consultor o analista funcional. Es una persona que puede tener una base técnica, pero que, sobre todo, tiene conocimiento de los procesos de negocios del cliente.</p> <p>El resultado es un documento de requisitos del usuario, que recoge el ámbito y propósito del sistema a implementar, qué incluye y qué no, y la funcionalidad requerida.</p> <p>Otro resultado es el documento de pruebas de aceptación o <i>sign-off</i>. Este detalla las pruebas que llevará a cabo el cliente, de manera para que, si se realizan correctamente, este dé su conformidad al sistema.</p>
------------------------------	---

Especificación del software	<p>Llevada a cabo por un analista funcional a partir del documento de requisitos de usuario. Es una persona que tiene una base técnica, además de un conocimiento de los procesos de negocios del cliente. El resultado es uno o varios documentos de análisis funcional que especifican la funcionalidad que debe tener el sistema que hay que desarrollar, para que satisfaga los requisitos del usuario.</p> <p>Otro resultado es un plan de pruebas del sistema. Estas pruebas las llevará a cabo la empresa de desarrollo de software para verificar que el sistema completo funciona correctamente, antes de realizar las pruebas de aceptación con el cliente.</p> <p>No solo se verifica el software, sino también la configuración de todo el sistema, incluyendo todos los servicios que utiliza la aplicación, como por ejemplo bases de datos o servicios diversos prestados por diferentes tipo de servidores. Un tipo de pruebas que se realiza habitualmente son las pruebas de carga. En estas se somete el sistema a un alto volumen de trabajo, con muchos usuarios utilizando el sistema a la vez, o con la ejecución de procesos que implican grandes volúmenes de información leída o generada.</p>
Diseño de alto nivel	<p>Llevada a cabo por un analista, para una parte del sistema cuya funcionalidad se especifica en un análisis funcional. En este documento se detalla, sin entrar en excesivos detalles técnicos, cómo se va a implementar una parte del sistema.</p> <p>El resultado es uno o varios documentos de diseño de alto nivel.</p> <p>Otro resultado es un plan de pruebas de integración. Con estas pruebas se verifica que una parte del sistema, que consta de varios módulos, componentes o programas, funciona correctamente. Y en particular que los diversos módulos funcionan correctamente en conjunto.</p>
Diseño de bajo nivel	<p>Llevada a cabo por un analista o analista-programador. En esta fase se especifica en detalle cómo se implementa un módulo, componente, o programa en particular. Esto queda reflejado en un documento de diseño de bajo nivel o diseño técnico.</p> <p>El conjunto de pruebas a desarrollar queda reflejado en un plan de pruebas unitarias, que es un documento muy detallado, que especifica las condiciones de prueba a ejecutar. Cada condición de prueba incluye la especificación de las condiciones iniciales, que se deben cumplir antes de ejecutar la prueba. Por ejemplo, que se hayan creado determinados datos. Por ejemplo, si se quiere probar un módulo que permite crear pedidos de clientes, normalmente será necesario que existan previamente clientes y artículos con determinadas características. También una descripción detallada de las acciones a realizar y de los resultados esperados. Si las pruebas unitarias las ejecuta una persona, normalmente esta debe incluir evidencias de los resultados obtenidos (por ejemplo, capturas de pantalla, o resultado de determinadas consultas sobre una base de datos). Pero también estas pruebas se pueden realizar de manera automática, utilizando distintas herramientas.</p> <p>En este tema se aprenderán diversas técnicas para crear planes de pruebas unitarios lo más completos, pero a la vez sencillos, que sea posible.</p> <p>También se aprenderá a utilizar herramientas que permiten realizar las pruebas unitarias de manera automática.</p>
Programación	<p>Llevada a cabo por un programador o analista-programador, de acuerdo a lo especificado en un diseño de bajo nivel.</p> <p>El resultado es un módulo, componente o programa.</p>

La última fase es la de codificación o programación, en la que se escriben un conjunto de módulos o unidades utilizando un lenguaje de programación. Cada módulo o unidad es el resultado de implementar un diseño de bajo nivel, y debe probarse por separado y de manera sistemática, de acuerdo a un plan de pruebas unitarias creado para el diseño de bajo nivel.

Una vez concluidas las pruebas unitarias de un módulo o unidad, este es susceptible de ser utilizado para la siguiente fase de prueba, las pruebas de integración, en la que se prueban varios módulos en conjunto. Después se realizarían las pruebas de sistema y, por último, las de aceptación.

Si las pruebas unitarias fallan, esto debe quedar reflejado en la documentación de pruebas. Puede ser necesario revisar el plan de pruebas unitarias, pero lo más habitual es que sea necesario revisar y hacer correcciones en el módulo, componente o programa, antes de volver a realizar las pruebas unitarias. Si fallan las pruebas de etapas posteriores, será necesario documentar estos fallos, proporcionando información lo más detallada posible, y después realizar los ajustes o correcciones apropiados y volver a ejecutar las pruebas.

2. Pruebas unitarias: objetivos y tipos

El objetivo de las pruebas es descubrir errores. Se puede demostrar la existencia de un error pero, en la práctica, no se puede demostrar la ausencia de errores.

Las pruebas se desarrollan ejecutando un plan de pruebas previamente redactado. Y en el caso de pruebas unitarias, un plan de pruebas unitarias (*unit test plan*, en inglés).

Existen metodologías para el diseño de pruebas unitarias que ayudan a crear planes de pruebas unitarias más exhaustivos y que, por tanto, permiten detectar una mayor cantidad de fallos. Es decir, que permiten tener, si no la seguridad absoluta, una mayor confianza en la calidad del *software*, una vez que se ha ejecutado el plan de pruebas sin encontrar errores.

En este tema se hablará solo de pruebas unitarias (*unit tests*). Se explicarán distintos tipos de técnicas para la creación de planes de pruebas unitarias. Se aplicarán para programas particulares y se utilizarán herramientas de software para automatizar su realización.

Un plan de prueba unitario incluye, entre otras cosas, un conjunto de condiciones de prueba. Cada condición de prueba específica:

- Unas condiciones previas que se tienen que cumplir antes de realizar la prueba.
- Unas acciones a realizar sobre el programa. Normalmente consisten en valores determinados a introducir, o en acciones a realizar sobre la interfaz gráfica de usuario del programa.
- Los resultados esperados. Normalmente consiste en una información que muestra el programa, o en un estado observable de la interfaz gráfica de usuario, o en determinados contenidos de determinados ficheros.

Las condiciones de prueba se especifican de manera muy precisa, de manera que si diferentes personas realizaran el plan de pruebas, obtendrían los mismos resultados. Por tanto, la ejecución de un plan de pruebas unitarias es susceptible de ser automatizada. Y como se verá más adelante, existen herramientas que permiten automatizar la ejecución de los planes de prueba. Para automatizar la ejecución de un plan de prueba unitaria hay que crear un guión o script (*unit test script*). Este es ejecutado por una herramienta, que muestra los resultados de la ejecución y, en particular, los errores encontrados. Esta herramienta puede, además, recopilar información adicional acerca de la ejecución del plan de pruebas. Por ejemplo, el tiempo que ha llevado ejecutar cada condición de prueba, o qué partes del código fuente se han ejecutado o no durante la ejecución del plan de pruebas.

Existen dos tipos de pruebas unitarias que se pueden realizar. Un plan de pruebas unitario puede incluir pruebas de ambos tipos.

- **Pruebas de caja negra.** Se crean teniendo en cuenta únicamente la funcionalidad especificada para la unidad, y no asume ningún conocimiento acerca de cómo se ha implementado.
- **Pruebas de caja blanca.** Se crean teniendo en cuenta la manera en que se ha implementado la unidad. En particular, el código fuente.

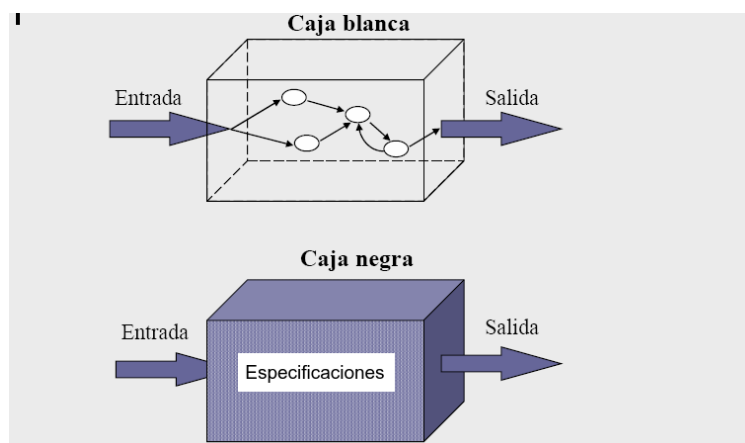
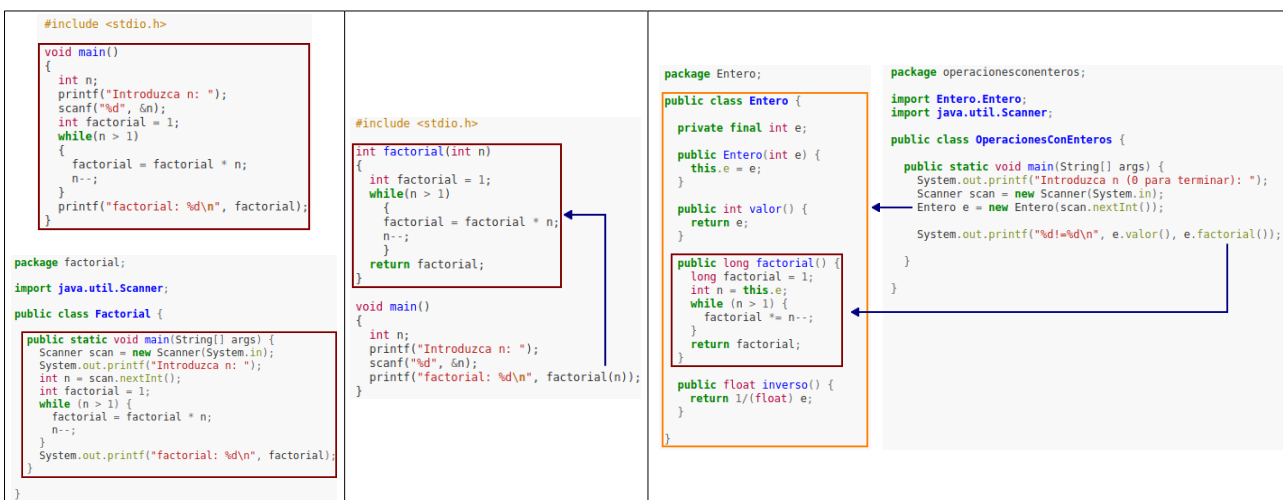


Ilustración 3.2: Pruebas de caja negra y de caja blanca

3. Métodos de caja blanca

Las pruebas de caja blanca se crean teniendo en cuenta el código fuente de la unidad que se quiere probar. Puede ser un programa relativamente sencillo, una función o procedimiento (en lenguajes no orientados a objetos) o un método de una clase (en lenguajes orientados a objetos).



Programa (lenguajes C y Java)

Función (lenguaje C)

Método de clase (lenguaje Java)

Ilustración 3.3: Unidades de prueba. Programa, función y método de clase

3.1. Método de los caminos básicos

Para el método de los caminos básicos, se hará abstracción del lenguaje de programación, y se expresarán los programas a probar en forma de diagramas de flujo, como los que se vieron en el primer tema.

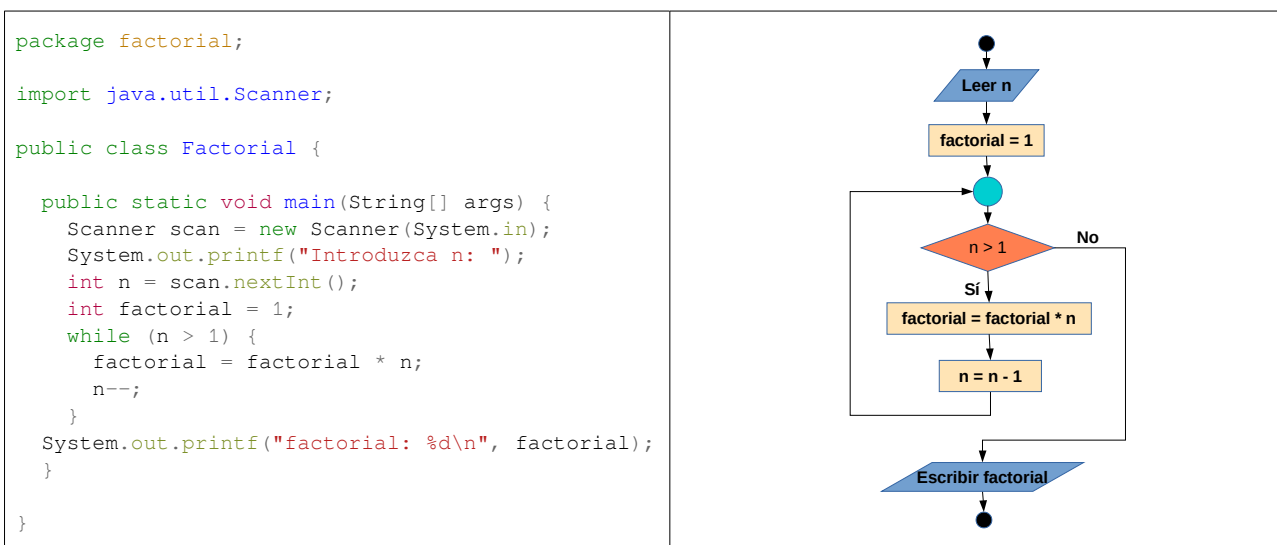
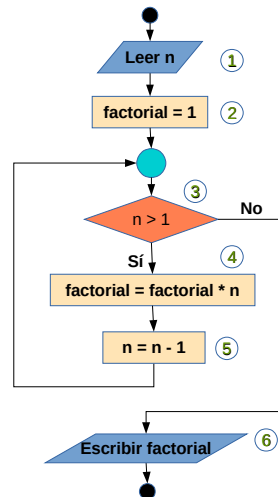


Ilustración 3.4: Diagrama de flujo para programa en Java que calcula el factorial de un número

Los nodos de decisión contienen una condición y dos nodos sucesores, uno para cuando se cumple la condición y otro para cuando no se cumple. Se representan con un rombo, dentro del cual se representa la condición.

Cada ejecución del programa se corresponde con un camino o recorrido en el diagrama de flujo desde el nodo inicial hasta el nodo final.

Con el programa anterior, el recorrido depende del valor de n que se introduzca. Cada condición de prueba corresponde a un valor de n , que determina un camino o recorrido distinto desde el nodo inicial (1) al nodo final (6). En este camino se puede pasar más de una vez por el mismo nodo.



Condición de prueba para $n=1$	
1	Se lee valor 1
2	factorial = 1
3	$n=1$, por lo que no se cumple la condición $n>1$
6	Se escribe 1
Camino: [1, 2, 3, 6]	

Condición de prueba para $n=2$	
1	Se lee valor 2
2	factorial = 1
3	$n=2$, por lo que se cumple la condición $n>1$
4	factorial = $1 * 2 = 2$
5	$n = 1$
3	$n=1$, por lo que no se cumple la condición $n>1$
6	se escribe 2
Camino: [1, 2, 3, 4, 5, 3, 6]	

Condición de prueba para $n=3$	
1	Se lee valor 3
2	factorial = 1
3	$n=3$, por lo que se cumple la condición $n>1$
4	factorial = $1 * 3 = 3$
5	$n = 2$
3	$n=2$, por lo que se cumple la condición $n>1$
4	factorial = $3 * 2 = 6$
5	$n = 1$
3	$n=1$, por lo que no se cumple la condición $n>1$
6	se escribe 6
Camino: [1, 2, 3, 4, 5, 3, 4, 5, 3, 6]	

Ilustración 3.5: Condiciones de prueba y sus correspondientes caminos

Se puede ver que bastaría con las condiciones de prueba para $n=1$ y $n=2$ para que se ejecutaran al menos una vez todas las sentencias del programa o, planteado de otra forma, para que se pasara por todos los nodos del diagrama de flujo. También bastaría con las condiciones de prueba para $n=1$ y $n=3$. Ambos conjuntos son no redundantes, porque no se puede eliminar ninguna sin que dejen de ejecutarse todas las sentencias del programa. El conjunto $\{n=1, n=2, n=3\}$ sería redundante, porque se puede eliminar cualquiera de ellas, y con el conjunto resultante se siguen ejecutando todas las sentencias del programa.

Un **camino básico** es uno que no incluye más de dos veces un nodo de decisión correspondiente a un mismo bucle. Esto significa que no se entra más de una vez en el bucle. Si aparece una sola vez, no se entra. Si aparece dos veces, se

entra una vez tras la primera, y no se entra tras la segunda. Si apareciera tres veces, se entraría dos veces, y entonces no sería un camino básico.

En el grafo de flujo anterior, los caminos para $n=1$ [1, 2, 3, 6] y $n=2$ [1, 2, 3, 4, 5, 3, 6] son caminos básicos. El camino para $n=3$ [1, 2, 3, 4, 5, 3, 4, 5, 3, 6] no es un camino básico, porque aparece tres veces el nodo de decisión 3, que está al principio del bucle.

3.1.1. Grafos de flujo

Si de lo que se trata es de encontrar un conjunto de condiciones de prueba que corresponda a un conjunto de caminos que incluya todos los nodos del diagrama de flujo, se pueden sustituir las secuencias que no son nodos de decisión por un único nodo, y con ello se obtiene un **grafo de flujo** equivalente.

El diagrama de flujo anterior contiene las secuencias de nodos no de decisión [1, 2] y [4, 5]. Cada una de ellas se puede sustituir por un único nodo. En el grafo de flujo resultante, Los nuevos nodos se numeran de nuevo, y la nueva numeración no tiene ninguna relación con la antigua. Los antiguos números de nodo se muestran junto a los nuevos, para hacer más clara la correspondencia entre ellos.

Los nodos de decisión se muestran con un color de fondo distinto, y la condición se muestra en el nodo.

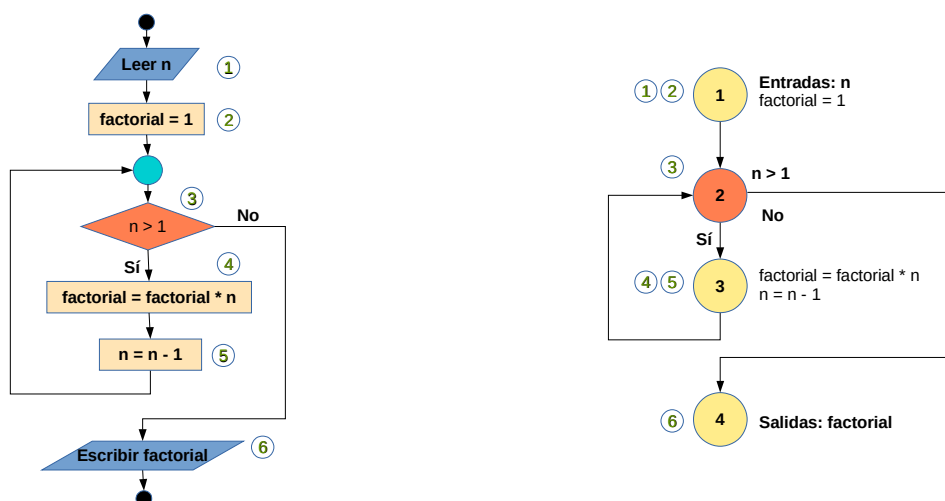
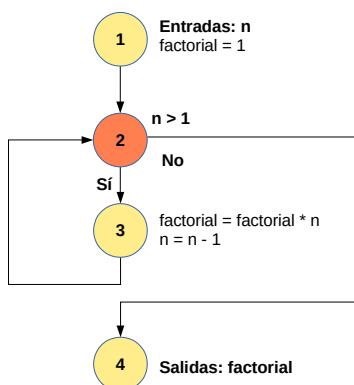


Ilustración 3.6: Diagrama de flujo y grafo de flujo correspondiente

Se pueden expresar los caminos para cada condición de prueba de acuerdo a los nodos del grafo de flujo.



Condición de prueba para $n=1$

Camino: {1, 2, 4}

Condición de prueba para $n=2$

Camino: {1, 2, 3, 2, 4}

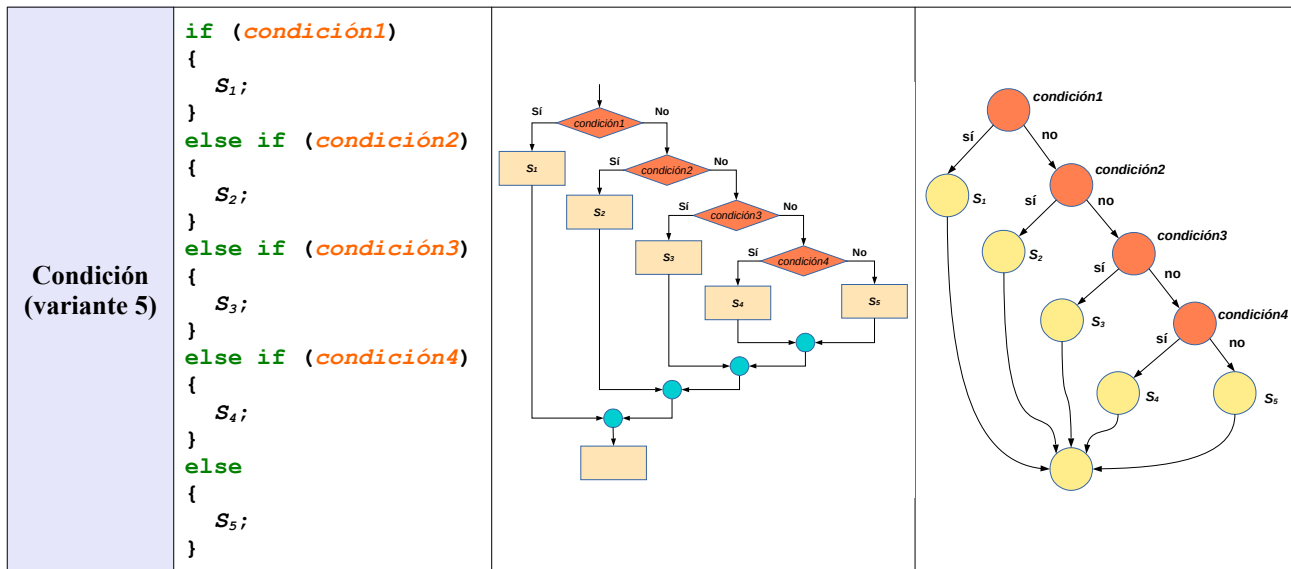
Condición de prueba para $n=3$

Camino: {1, 2, 3, 2, 3, 2, 4}

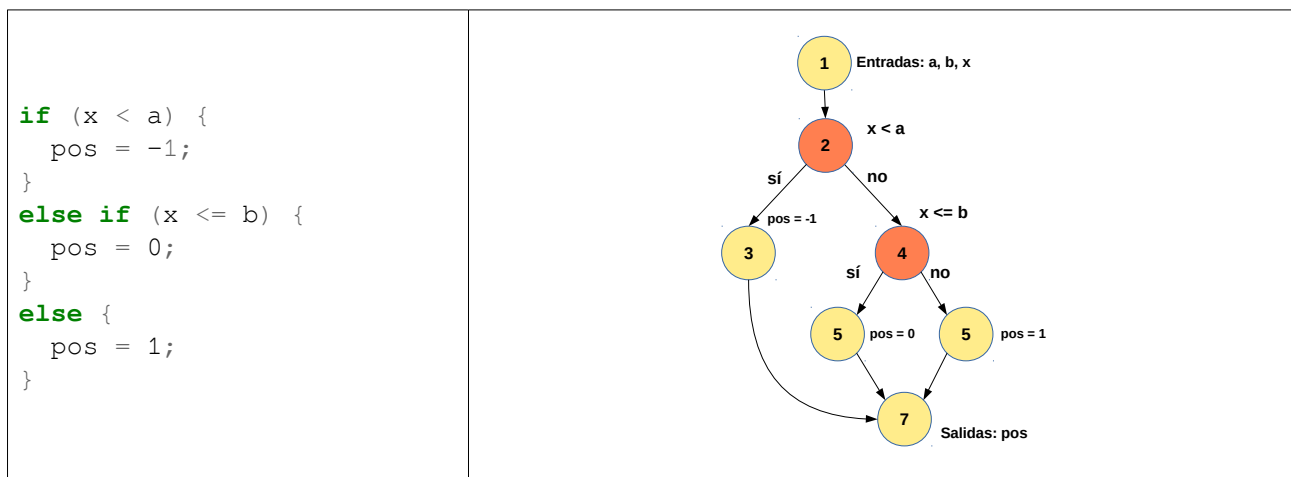
A continuación se muestra la equivalencia entre diversas estructuras que se pueden expresar con lenguajes estructurados tales como C++, Java o C#, diagramas de flujo y grafos de flujo.

3.1.1.1. Sentencias condicionales

	Código en Java	Diagrama de flujo	Grafo de flujo
Condicional (variante 1)	<pre> if (condición) { S; } </pre>		
Condicional (variante 2)	<pre> if (condición) { S1; } else { S2; } </pre>		
Condicional compuesta (variante 3)	<pre> if (condición1) { S1; } else if (condición2) { S2; } </pre>		
Condición (variante 4)	<pre> if (condición1) { S1; } else if (condición2) { S2; } else { S3; } </pre>		



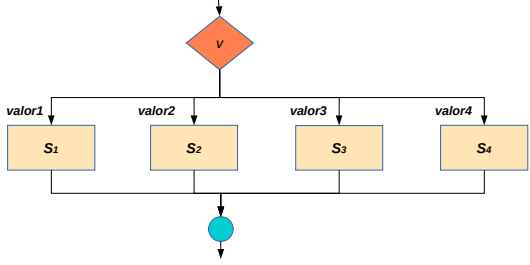
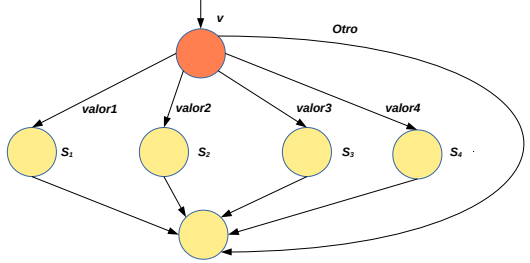
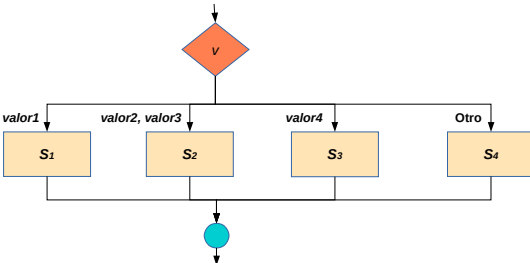
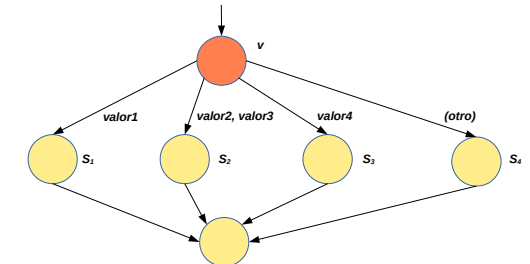
Por ejemplo, la siguiente sentencia determina si un número x está antes, dentro, o después de un intervalo $[a, b]$. En estos tres casos, asigna a la variable pos , respectivamente, los valores -1 , 0 y 1 . Al lado se muestra el grafo de flujo correspondiente.



3.1.1.2. Sentencias de selección switch

La sentencia `switch` es equivalente a una sentencia `if ... else if ... else if ... else`, en la que las condiciones consisten en comparar el valor de una variable o expresión s con valores literales dados.

Código en Java	Código equivalente	Diagrama de flujo y grafo de flujo
----------------	--------------------	------------------------------------

<p>Sentencia de selección (variante 1)</p>	<pre> switch (v) { case valor1: S₁; break; case valor2: S₂; break; case valor3: S₃; break; case valor4: S₄; break; } </pre>	<pre> if (v == valor1) { S₁; } else if (v == valor2) { S₂; } else if (v == valor3) { S₃; } else if (v == valor4) { S₄; } </pre>	  <p>Nota: al igual que en toda sentencia condicional <code>if</code> hay una rama alternativa, que puede estar vacía, en toda sentencia de selección <code>switch</code> hay una rama alternativa, que puede estar vacía. Se sigue cuando <code>v</code> no tiene ninguno de los valores de las otras ramas.</p>
<p>Sentencia de selección (variante 2)</p>	<pre> switch (v) { case valor1: S₁; break; case valor2: case valor3: S₂; break; case valor4: S₃; break; default: S₄; } </pre>	<pre> if (v == valor1) { S₁; } else if (v == valor2 v == valor3) { S₂; } else if (v == valor4) { S₃; } else { S₄; } </pre>	  <p>Nota: en esta sentencia de selección la rama alternativa no está vacía.</p>

3.1.1.3. Bucles

Existen distintos tipos de bucles, pero el único realmente necesario es el bucle `while`. Todos los demás se pueden expresar utilizando bucles `while`.

	Código en Java	Diagrama de flujo	Grafo de flujo
Bucle while	<pre>while (condición) { S; }</pre>		
Bucle do ... while	<pre>do { S; } while (condición)</pre>		
Bucle for	<pre>for(Inic; cond; FinIter) { S; }</pre>		

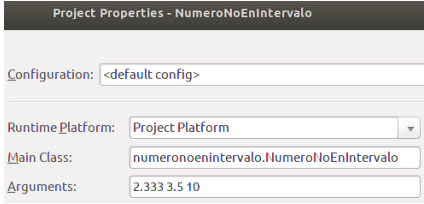
3.1.1.4. Condiciones compuestas

Con los operadores lógicos && (operador y) y || (operador o), se pueden formar condiciones compuestas. Valgan como ejemplo las siguientes condiciones, que determinan si un valor x está entre otros dos valores a y b. Esta sintaxis es común al lenguaje C, Java, y muchos otros.

$x \geq a \text{ y } x \leq b$	$x \geq a \ \&\& \ x \leq b$
$x < a \text{ o } x > b$	$x < a \ \ x > b$

A continuación se muestran dos programas que determinan si un valor dado x está entre otros dos dados a y b. Cada uno utiliza una forma diferente de obtener las entradas para el programa: hacer que el usuario las introduzca, o bien obtenerlas desde parámetros de línea de comandos. En el tema anterior se explicó cómo introducir los valores para estos últimos desde el entorno de desarrollo. Se puede utilizar cualquiera de estos planteamientos para obtener los valores de entrada para los programas que se desarrollen.

Se muestran en colores distintos los bloques para obtener los valores de entrada, para mostrar los valores de salida, y para el algoritmo en sí. El primer programa ilustra el uso del operador lógico &&, y el segundo el del operador ||.

<pre>package numeroenintervalo; import java.util.Scanner; public class NumeroEnIntervalo { public static void main(String[] args) { Scanner scan = new Scanner(System.in); System.out.printf("Introduzca inicio de intervalo [a,b]: a="); float a = scan.nextFloat(); System.out.printf("Introduzca final de intervalo [a,b]: b="); float b = scan.nextFloat(); System.out.printf("Introduzca x para saber si x en [a, b]: x="); float x = scan.nextFloat(); boolean enIntervalo; if (x >= a && x <= b) { enIntervalo = true; } else { enIntervalo = false; } System.out.printf("%f en intervalo [%f, %f]: %b\n", x, a, b, enIntervalo); } }</pre>	<pre>package numeroenintervalo; public class NumeroNoEnIntervalo { public static void main(String[] args) { float a = Float.parseFloat(args[0]); float b = Float.parseFloat(args[1]); float x = Float.parseFloat(args[2]); boolean enIntervalo; if (x < a x > b) { enIntervalo = false; } else { enIntervalo = true; } System.out.printf("%f en intervalo [%f, %f]: %b\n", x, a, b, enIntervalo); } }</pre>
<p>Output - NumeroEnIntervalo (run) ×</p> <pre>run: Introduzca inicio de intervalo [a,b]: a=2,333000 Introduzca final de intervalo [a,b]: b=3,5 Introduzca x para saber si x en [a, b]: x=10 10,000000 en intervalo [2,333000, 3,500000]: false</pre>	 <p>Project Properties - NumeroNoEnIntervalo</p> <p>Configuration: <default config></p> <p>Runtime Platform: Project Platform</p> <p>Main Class: numeroenintervalo.NumeroNoEnIntervalo</p> <p>Arguments: 2.333 3.5 10</p> <p>Output - NumeroNoEnIntervalo (run) ×</p> <pre>run: 10,000000 en intervalo [2,333000, 3,500000]: false</pre>

No siempre que se evalúa una expresión lógica en la que se utiliza uno de estos operadores para combinar dos expresiones lógicas, se evalúan ambas:

- Con el operador `&&`, si la primera expresión no se cumple, entonces la expresión completa no se cumple, y la segunda no se comprueba.
- Con el operador `||`, si la primera expresión se cumple, entonces la expresión completa se cumple, y la segunda no se comprueba.

Esto significa que hay que desdoblarse los nodos de decisión en los que aparecen expresiones compuestas, que utilizan estos operadores. Los grafos de flujo para los programas anteriores, con los correspondientes nodos de decisión desdoblados, son los siguientes. En el nodo inicial se indican las entradas, y en el nodo final se indican las salidas.

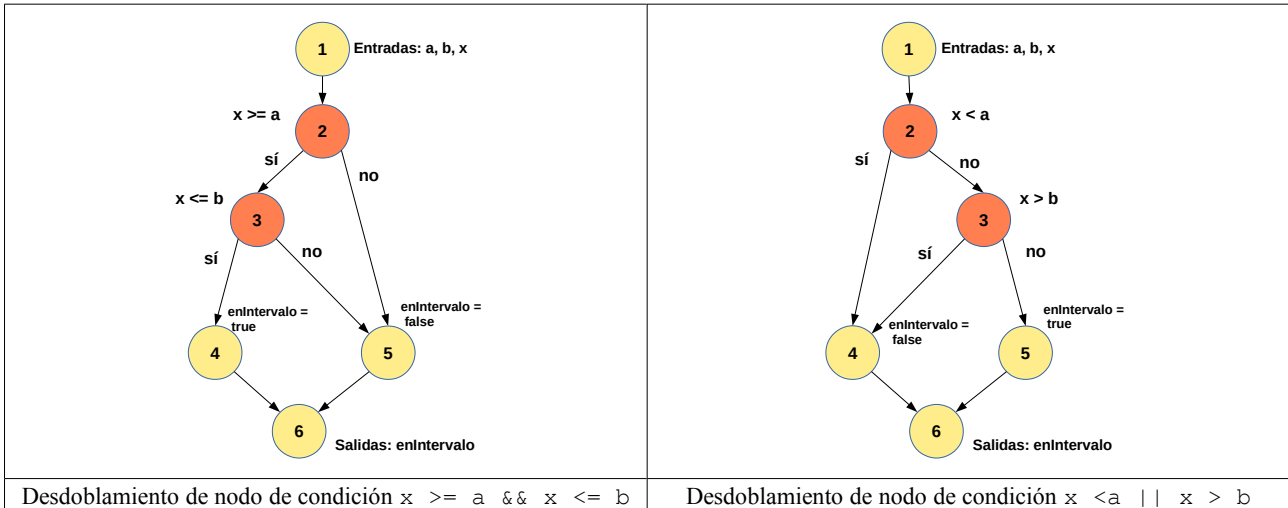


Ilustración 3.7: Desdoblamiento de condiciones compuestas con operadores lógicos && (y), || (o)

Se pueden componer más de dos condiciones con estos operadores, en cuyo caso habría que desdoblar más de una vez.

3.1.2. Conjunto mínimo de caminos básicos, complejidad ciclomática

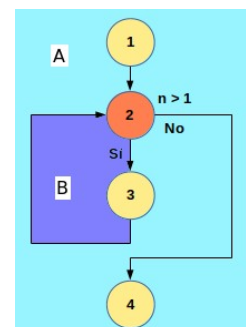
Un conjunto mínimo de caminos básicos cumple las siguientes condiciones.

1. Solo incluye caminos básicos. Con un camino básico, como ya se ha visto, no se entra más de una vez en ningún bucle.
2. Incluye todos los nodos del grafo de flujo. Es decir, toda sentencia se ejecuta en al menos un camino básico.
3. Incluye todos los arcos del grafo de flujo. Es decir, toda rama de una sentencia condicional se recorre en al menos un camino básico.
4. Para cualquier camino básico hay otro con respecto al cual hay como máximo una diferencia, entendiendo como tal un camino distinto tomado a partir de una sentencia condicional.
5. Es no redundante. Es decir, que no se puede eliminar ningún camino básico sin que dejen de cumplirse las anteriores condiciones.

Todos los conjuntos mínimos de condiciones de prueba para este grafo de flujo (es decir, cuyos caminos incluyen todos los nodos y arcos del diagrama/graf de flujo, y en los que no se puede eliminar ninguna condición de prueba sin que esto deje de cumplirse), constan de dos condiciones de prueba. Esto no es casualidad, y tiene relación con propiedades estructurales del grafo de flujo.

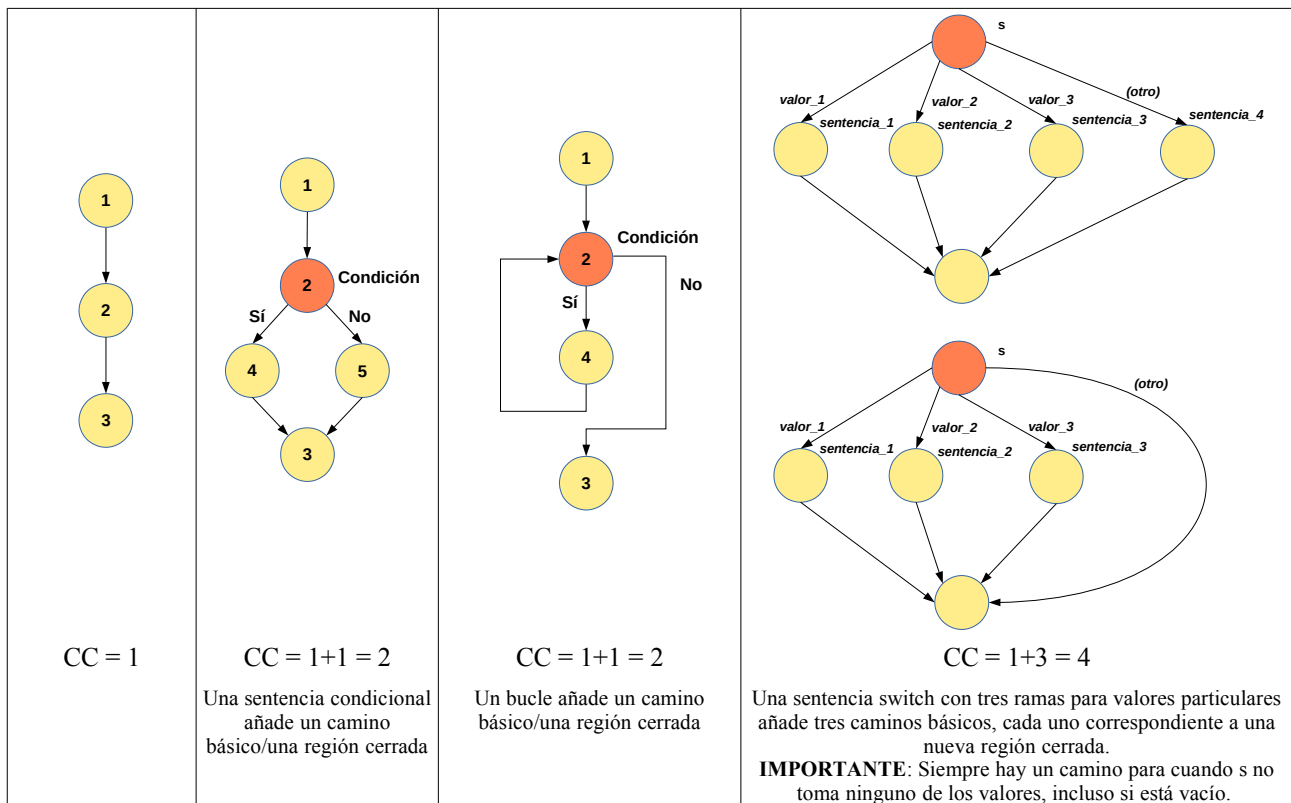
El número de elementos de cualquier conjunto de caminos básicos no redundante en el que aparecen todos los nodos del grafo de flujo es igual a la **complejidad ciclomática** de un grafo de flujo, y esta es igual a:

- Número de regiones, o bien número de regiones cerradas más 1. A menos que no haya ningún nodo de decisión, hay siempre una región abierta. En este ejemplo hay dos regiones: la abierta A y una cerrada B.
- Número de nodos de decisión más uno. Cada nodo de decisión añade una región cerrada al grafo. En este caso hay un nodo de decisión, y se tendría $1+1=2$.
- Aristas+2-Nodos. En este caso, hay 4 aristas y cuatro nodos, y se tendría $4+2-4=2$.



Otra manera de verlo es la siguiente:

- Un grafo de flujo lineal tiene 0 nodos de decisión y 1 camino básico, y solo una región (abierta).
- Por cada nodo de decisión, se añade un camino básico, y una región cerrada.



3.1.3. Obtención de un conjunto de condiciones de prueba

Para encontrar un conjunto mínimo de condiciones de prueba para un programa, se siguen los siguientes pasos:

1. Calcular la complejidad ciclomática del grafo de flujo para el programa. Es aconsejable calcularla por más de uno de los criterios anteriores, y verificar que sale el mismo número.
2. Encontrar un conjunto de caminos básicos que entre todos incluyan todos los nodos del grafo de flujo. Para ello, se puede proceder como sigue:
 - a) Obtener el primer camino básico no entrando en ningún bucle y tomando uno cualquiera de los posibles caminos para cada nodo de decisión.
 - b) Para cada uno de los nodos de decisión que no sean de entrada a un bucle incluidos en el camino anterior, obtener variantes que se diferencian del camino anterior solo en la rama seguida a partir del nodo de decisión, pero no en ninguna otra rama tomada en ningún otro nodo de decisión.
 - c) Para cada bucle obtenido en el camino anterior, obtener una variante del camino anterior, entrando una sola vez en el bucle.
 - d) Repetir el proceso con cada uno de los nuevos caminos básicos así obtenidos.

Obtener todas las variantes puede no ser tan sencillo, porque en el camino alternativo para un nodo de decisión (y esto incluye como caso particular uno que lleve al interior de un bucle), se puede encontrar una nueva sentencia condicional o un nuevo bucle. Conviene hacerlo con sistema y siguiendo un orden. Una buena estrategia es obtener las variaciones de abajo a arriba. Una vez que se han recorrido todas las ramas existentes a partir de un nodo de decisión situado más abajo, da igual cuál de ellas se recorra cuando se recorren nuevas ramas de un nodo de decisión situado más arriba.

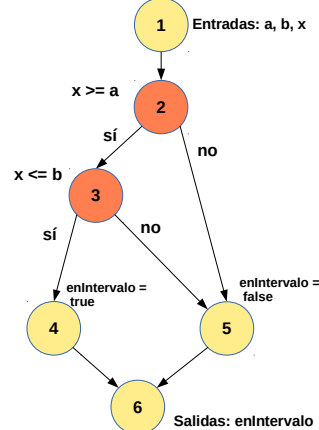
3. Encontrar unas condiciones previas para que se recorra cada uno de los caminos básicos anteriores. Las condiciones previas consistirán en una o varias asignaciones particulares de valores a variables del programa. Esto es sencillo para programas sencillos, pero puede ser bastante más complicado para programas no tan sencillos. La ejecución de este conjunto de condiciones de pruebas garantizará que se recorren todos los caminos básicos y, por tanto, que se ejecutan todas las sentencias del programa. Muchas veces se podrá elegir más de un valor. Por ejemplo, si para que se siga un camino básico la condición es $n > 2$, se puede especificar como condición previa $n = 4$, pero se podría haber elegido cualquier otro valor mayor que 2.
4. Determinar los resultados esperados para cada condición de prueba. Esto no debe hacerse a partir del propio programa, porque entonces no se estaría probando el programa, sino analizando su comportamiento, sino que debe hacerse a partir de la funcionalidad esperada del programa, es decir, de sus especificaciones. Por ejemplo, para el programa anterior, para el cálculo del factorial de un número, el resultado esperado para la condición de prueba $\{n=5\}$, sería 120, es decir, 120.

Atención: un conjunto mínimo de condiciones de prueba puede no ser suficiente

Un conjunto mínimo de condiciones de prueba, como el obtenido con el anterior método, no es necesariamente suficiente. De hecho, es muy conveniente realizar pruebas adicionales, para valores más grandes de n . Es frecuente que los bucles hayan fallos que no se manifiestan con una única ejecución del bucle, pero sí con varias. Pero sí es un buen punto de partida, porque al menos se garantiza que se ejecutan todas las sentencias del programa. Se pueden añadir condiciones de prueba adicionales para probar casos particulares de especial interés, y también para probar varias ejecuciones de cada bucle.

A continuación se aplica este método con los ejemplos vistos hasta ahora.

Ejemplo 1: Determinar si un número dado x está en un intervalo dado $[a, b]$

Programa	Grafo de flujo
<pre>// Entradas: a, b, x if (x >= a && x <= b) { enIntervalo = true; } else { enIntervalo = false; } // Salidas: enIntervalo</pre>	
<p>Complejidad ciclomática: 3 3 regiones, 2 cerradas + 1 abierta 2 nodos de decisión: $2+1 = 3$</p>	<p>Caminos básicos (son 3) C_1: [1, 2, 3, 4, 6] C_2: [1, 2, 3, 5, 6] C_3: [1, 2, 5, 6]</p>
<p>Condiciones previas generales: a, b y x son números reales, $a \leq b$.</p> <p>Condiciones previas para cada camino básico Para C_1: Para que se haga este camino, debe cumplirse $x \geq a$ y $x \leq b$. Por ejemplo: $a=3.5, b=5.47, x=4.32$ Para C_2: Para que se haga este camino, debe cumplirse $x \geq a$ y $x > b$. Por ejemplo: $a=3, b=4, x=4.22$ Para C_3: Para que se haga este camino, debe cumplirse $x < a$. Por ejemplo: $a=2.56, b=34.4, x=1.678$</p>	

Condiciones de prueba

Condiciones previas generales: a y b son números reales, $a \leq b$

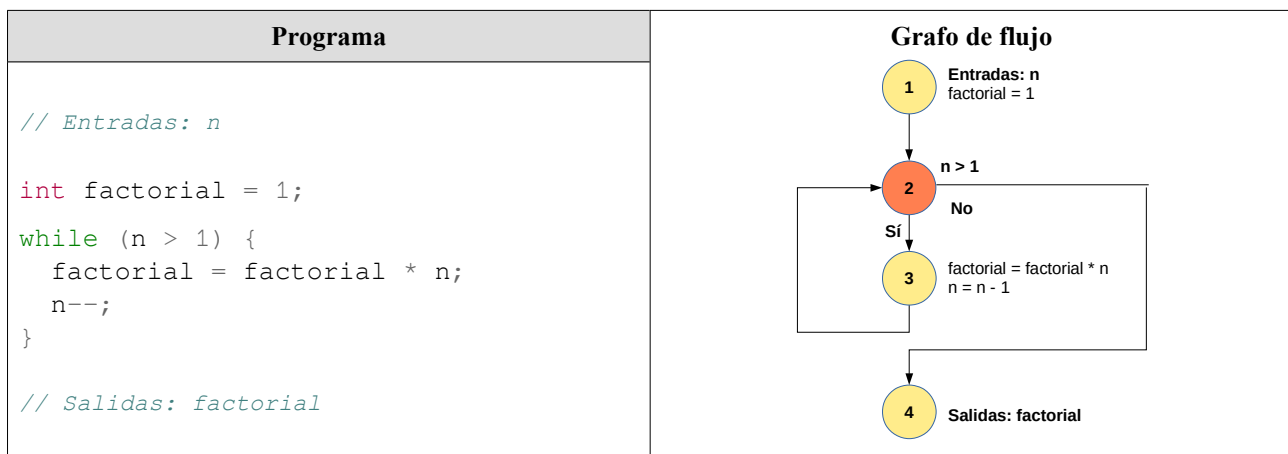
Camino básico	Condiciones previas	Resultados esperados
[1, 2, 3, 4, 6]	$a=3.5$, $b=5.47$, $x=4.32$	$\text{enIntervalo} = \text{true}$
[1, 2, 3, 5, 6]	$a=3$, $b=4.22$, $x=5.041$	$\text{enIntervalo} = \text{false}$
[1, 2, 5, 6]	$a=2.56$, $b=34.4$, $x=1.678$	$\text{enIntervalo} = \text{false}$

Atención: condiciones previas generales

En ocasiones interesa establecer unas condiciones previas generales. Estas son, normalmente:

- **Especificaciones de tipos.** Estas son relevantes para el establecimiento de las condiciones previas, que incluyen seleccionar un valor para cada variable de entrada. Por ejemplo: si los valores pueden ser reales, como en este caso, conviene seleccionar valores con decimales.
Interesa hacer abstracción de los tipos existentes en cada posible lenguaje de programación. Por lo tanto, se indicarán tipos generales como:
 - Entero. Posibles valores serían, por ejemplo: 1, 0, -3, 10, -8.
 - Cadena de caracteres. Posibles valores serían, por ejemplo: "hola", "aa90xfa", _b0.
 - Número real. Posibles valores serían, por ejemplo: 4.3453453, 4, -432.342423. En general se escribirán con punto decimal, pero según el entorno y el lenguaje de programación, puede tener que utilizarse la coma y no el punto como separador de decimales.
- **Restricciones sobre los valores de entrada.** Estas se especifican de manera general, antes de las condiciones previas para cada camino básico. Y con ello se puede asumir que se cumplen siempre. Por ejemplo, en el ejemplo anterior, en que se tienen como entradas los extremos de un intervalo $[a, b]$, debe cumplirse que $a \leq b$. Si no, no sería un intervalo válido. Otras restricciones típicas son, por ejemplo, que una variable no pueda tomar valores negativos, o que tenga que tomar valores en un rango determinado, o un valor de un conjunto cerrado de valores. El motivo para especificar estas restricciones aparte y como condiciones previas es no complicar innecesariamente el programa. De igual manera que se hace abstracción de cómo se obtienen los valores (si los introduce el usuario, o si se obtienen desde argumentos de línea de comandos), se hace abstracción de las verificaciones y validaciones previas, que no tienen ningún interés en lo que respecta a las pruebas (*testing*) de caja blanca.

Ejemplo 2: Calcular el factorial de un número.



Complejidad ciclomática: 2 2 regiones, 1 cerradas + 1 abierta 1 nodos de decisión: 1+1 = 2	Caminos básicos (son 2) C ₁ : [1, 2, 4] C ₂ : [1, 2, 3, 2, 4]	
Condiciones previas generales: n es un número entero, n >= 1. Condiciones previas para cada camino básico Para C ₁ : Para que se haga este camino, debe cumplirse n<=1, y como se tiene que n>=1, entonces debe ser n=1. Para C ₂ : Para que se entre en el bucle, debe cumplirse n>1, y para que se entre exactamente una vez, debe ser n=2.		
Condiciones de prueba		
Camino básico	Condiciones previas	Resultados esperados
[1, 2, 4]	n=1	factorial = 1
[1, 2, 3, 2, 4]	n=2	factorial = 2

Si bien este conjunto de caminos mínimos hace que se ejecute al menos una vez cada sentencia del programa, no parece que un plan de pruebas solo con estas dos condiciones de prueba sea suficiente. Como mínimo deberían añadirse varios valores más de n. Al menos $n=3$, para que el bucle se ejecute dos veces, y algún valor algo más alto, como por ejemplo $n=10$, para verificar que el programa funciona correctamente con algunas iteraciones más del bucle.

Si se prueba con valores más altos, se encontrará, al menos con Java, y normalmente con cualquier lenguaje de programación, que el resultado no se calcula correctamente, porque $n!$ crece muy rápidamente conforme crece n, y normalmente desborda rápidamente la capacidad de cualquier tipo existente en cualquier lenguaje de programación. Esta es una limitación intrínseca de los lenguajes de programación con la que hay que contar. La manera en que este problema se manifiesta y las posibles soluciones alternativas dependen mucho del lenguaje en cuestión.

Actividad 1

Dibuja el grafo de flujo para un algoritmo que, dados dos números enteros a y b, da como resultado: 1 si $a > b$. -1 si $a < b$ 0 si $a = b$ Se da el código del algoritmo en Java. Obtén la complejidad ciclomática, un conjunto de caminos básicos, y un conjunto de condiciones de pruebas basado en el conjunto de caminos básicos. Nota: Para tener un programa funcional, faltaría añadir al principio la lectura y validación de las entradas, y al final la escritura del valor de la salida. Cuando se leen las entradas, si cualquier valor introducido es incorrecto, se mostraría un mensaje de error apropiado y se terminaría inmediatamente la ejecución del programa.	<pre>// Entradas int a; int b; // Salidas int comparacion; // Algoritmo if (a > b) { comparacion = 1; } else if (a < b) { comparacion = -1; } else { comparacion = 0; }</pre>
--	---

Actividad 2

Dibuja el grafo de flujo para un algoritmo que calcule el salario de un trabajador según el número de horas trabajadas y el tipo de salario. Se da el código del algoritmo en Java. Si el salario es de tipo A, se pagan 6€ por hora, hasta 40 horas, y a partir de ellas a 9€ por hora.	<pre>// Entradas String tipo; // A, B o C int horas; // Salidas double salarioSemanal;</pre>
--	---

<p>Si el salario es de tipo B, se pagan 7€ por hora, hasta 40 horas, y a partir de ellas a 9,5€ por hora.</p> <p>Si el salario es de tipo C, se pagan 8€ por hora, hasta 40 horas, y a partir de ellas a 10€ por hora.</p> <p>Obtén la complejidad ciclomática, un conjunto de caminos básicos, y un conjunto de condiciones de pruebas basado en el conjunto de caminos básicos.</p> <p>Crea el correspondiente programa en Java, que tome las entradas de parámetros de línea de comandos, y ejecuta y verifica las condiciones de prueba.</p> <p>Este programa debe obtener el valor para la entrada (tipo de salario) de un parámetro de línea de comandos. Antes de ejecutar el algoritmo en sí, debe validarse el valor introducido y, si no es correcto, debe mostrarse un mensaje de error apropiado y terminar la ejecución del programa.</p>	<pre>// Algoritmo double salHNormal = 0; double salHEExtra = 0; switch (tipo) { case "A": salHNormal = 6; salHEExtra = 9; break; case "B": salHNormal = 7; salHEExtra = 9.5; break; case "C": salHNormal = 8; salHEExtra = 10; break; } if(horas <= 40) { salarioSemanal = horas * salHNormal; } else { salarioSemanal = 40 * salHNormal + (horas-40) * salHEExtra; }</pre>
--	---

Actividad 3

<p>Dibuja el grafo de flujo para un algoritmo que calcula, si existen, las soluciones x1 y x2 para una ecuación de segundo grado $ax^2+bx+c=0$.</p> <p>a, b y c son las entradas, y x1 y x2 las salidas.</p> <p>Se da el código del algoritmo en Java.</p> <p>Obtén la complejidad ciclomática, un conjunto de caminos básicos, y un conjunto de condiciones de pruebas basado en el conjunto de caminos básicos.</p> <p>La fórmula general para las soluciones es</p> $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \text{ es decir:}$ $x1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \text{ y } x2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$ <p>Pero hay varios casos particulares que hay que considerar antes.</p>	<pre>// Entradas double a; double b; double c; // Salidas double x1; double x2; // Algoritmo if (a == 0) { if (b == 0) { x1 = x2 = Double.NaN; } else { x1 = -c / b; x2 = Double.NaN; } } else { double disc = b * b - 4 * a * c;</pre>
---	--

<p>Si $a=b=0$, entonces se tiene $c=0$, que no es una ecuación, porque su validez no depende del valor de x, sino del valor de c. En ese caso, el valor para ambas salidas $x1$ y $x2$ es NaN.</p> <p>Nota: NaN (que significa <i>Not a Number</i>) es un valor especial para los tipos <code>float</code> y <code>double</code> en el lenguaje de programación Java.</p> <p>En otro caso, hay que considerar como un caso particular $a=0$ y, en este caso, no aplicar esta fórmula, que provocaría una división por cero, sino devolver la solución de la ecuación de primer grado resultante $bx+c=0$, es decir, $x = -\frac{c}{b}$. En este caso, solo habría una solución, y los valores para las salidas serían $x1 = -\frac{c}{b}$ y $x2 = \text{NaN}$.</p> <p>Si la ecuación no tiene raíces reales ($b^2-4ac < 0$), el valor para ambas salidas $x1$ y $x2$ debe ser NaN.</p> <p>Si $b^2-4ac = 0$, solo hay una raíz real que se obtiene en $x1$, y $x2$ es NaN.</p>	<pre>double denom = 2 * a; if(disc > 0) { x1 = (-b + Math.sqrt(disc)) / denom; x2 = (-b - Math.sqrt(disc)) / denom; } else if (disc == 0) { x1 = -b / denom; x2 = Double.NaN; } else { x1 = x2 = Double.NaN; }</pre>
---	--

Actividad 4

<p>Dibuja el grafo de flujo para un algoritmo que, dados tres números enteros, determina si alguno de ellos es la suma de los otros dos. Los números ($n1$, $n2$ y $n3$), pueden estar desordenados. Es decir, no necesariamente $n1 \leq n2 \leq n3$.</p> <p>Se da el código del algoritmo en Java. El programa solo debe devolver sí o no, cierto o verdadero. Lo primero que se hace es ordenar los números. Una vez ordenados, solo hay que verificar si el último es la suma de los anteriores.</p> <p>Obtén la complejidad ciclomática, un conjunto de caminos básicos, y un conjunto de condiciones de pruebas basado en el conjunto de caminos básicos.</p>	<pre>// Entradas int n1; int n2; int n3; // Salidas boolean esSuma; // Algoritmo // Primero se ordenan los números if(n2 < n1) { int aux = n1; n1 = n2; n2 = aux; } // n1 y n2 ya ordenados if(n3 < n1) { int aux = n1; n1 = n3; n3 = n2; n2 = aux; } else if (n3 < n2) { int aux = n2; n2 = n3; n3 = aux; } if(n1 + n2 == n3) { esSuma = true; } else { esSuma = false; }</pre>
---	---

La entrada de un programa puede ser una **secuencia de valores**. Por ejemplo: un programa puede pedir repetidamente que se introduzca un número, o bien se le pueden pasar por línea de comandos una secuencia de números. Conceptualmente se puede entender que ambas cosas son lo mismo. Para estos casos se puede representar la entrada como una matriz o *array* con índices enteros a partir de 0. Esta no deja de ser una representación arbitraria, pero es la habitual en los lenguajes de programación. Si pues, una secuencia de números (o en general, de datos del tipo que sea) se representaría como a , siendo $a[0]$, $a[1]$, $a[2]$, ..., $a[\text{long}(a)-1]$ sus elementos, siendo $\text{long}(a)$ la longitud de la secuencia o *array*. Por ejemplo: si la secuencia es $a=\{4, -3, 0, 3\}$, entonces $\text{long}(a)=4$, $a[0]=4$, $a[1]=-3$, $a[2]=0$ y $a[3]=3$. Si la secuencia a está vacía, se representa como $\{\}$, y su longitud es 0, es decir, $\text{long}(\{\})=0$.

Ejemplo 3: Programa que calcula la suma de todos los números positivos de una secuencia de números enteros, y su grafo de flujo correspondiente, podrían ser los siguientes:

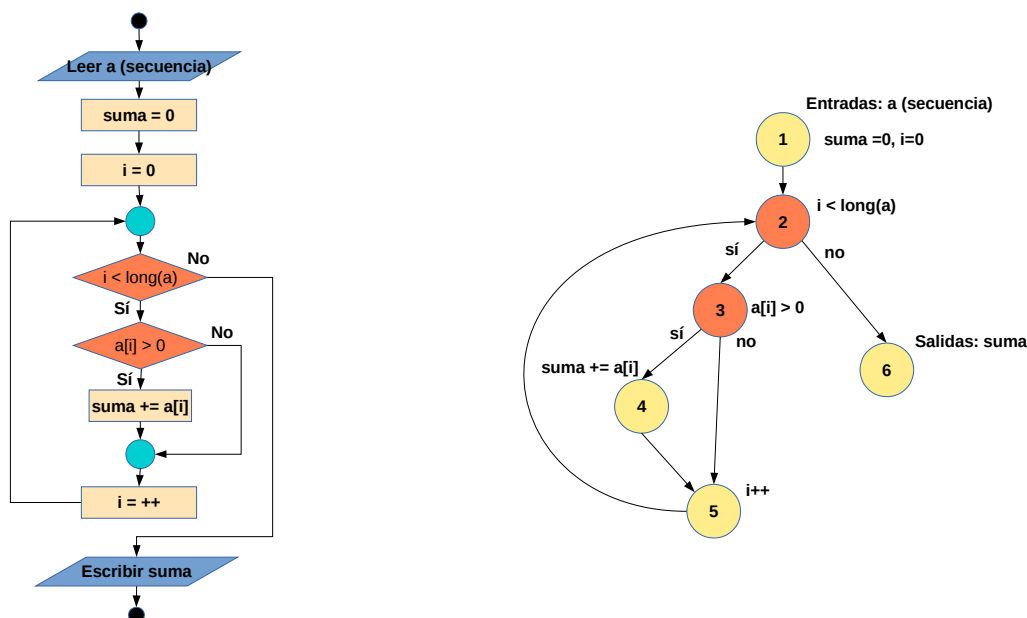


Ilustración 3.8: Diagrama de flujo y grafo de flujo para cálculo de suma de números positivos de una secuencia

<p>Complejidad ciclomática: 3</p> <p>3 regiones, 2 cerradas + 1 abierta 2 nodos de decisión: $2+1 = 3$</p>	<p>Camino básicos (son 3)</p> <p>C_1: [1, 2, 6] C_2: [1, 2, 3, 5, 2, 6] C_3: [1, 2, 3, 4, 5, 2, 6]</p>
<p>Condiciones previas generales: a es una secuencia de números enteros.</p> <p>Condiciones previas para cada camino básico</p> <p>Se puede primero deducir el número de elementos que debe contener la secuencia. Se entra en el bucle una vez por cada elemento de la secuencia, y se pasa por el nodo 2 ese número de veces más uno, porque la última vez no se entra. Es decir, la secuencia debe tener 0 elementos para que se haga C_1, y uno para que se haga C_2 y C_3.</p> <ul style="list-style-type: none"> Para C_1: La secuencia tiene 0 elementos: $a=[]$. El resultado debe ser $\text{suma}=0$. Para C_2: La secuencia tiene un elemento, se entra una vez en el bucle con $i=0$, y $a[i]=a[0]\leq 0$. Por ejemplo: $a=[-3]$. El resultado debe ser $\text{suma}=0$. Para C_3: La secuencia tiene un elemento, se entra una vez en el bucle con $i=0$, y $a[i]=a[0]>0$. Por ejemplo, $a=[5]$. El resultado debe ser $\text{suma}=3$. 	

Condiciones de prueba		
Camino básico	Condiciones previas	Resultados esperados
[1, 2, 6]	a={}	suma=0
[1, 2, 3, 5, 2, 6]	a={-3}	suma=0
[1, 2, 3, 4, 5, 2, 6]	a={5}	suma=5

Por supuesto que este conjunto de condiciones de prueba no es necesariamente suficiente. Prueba los casos en que la secuencia no tiene ningún elemento, en que tiene un elemento positivo, y en que tiene un elemento negativo. Pero se deberían añadir condiciones de prueba para probar algunas secuencias con más de un elemento, y con una combinación de números positivos y negativos.

Actividad 5

<p>Se da el código en Java de un algoritmo que, dada una secuencia de números reales, obtiene la suma de los números positivos y la suma de los números negativos.</p> <p>Obtén la complejidad ciclomática, un conjunto de caminos básicos, y un conjunto de condiciones de pruebas basado en el conjunto de caminos básicos.</p>	<pre>// Entradas double[] sec; // Salidas double sumaPos; double sumaNeg; // Algoritmo sumaPos = sumaNeg = 0; for(int i = 0; i < sec.length; i++) { if(sec[i] > 0) { sumaPos += sec[i]; } else if(sec[i] < 0) { sumaNeg += sec[i]; } }</pre>
---	---

Actividad 6

<p>Se da el código en Java de un algoritmo que, dada una secuencia de caracteres, muestra el número de veces que aparece cada una de las cinco vocales.</p> <p>Obtén la complejidad ciclomática, un conjunto de caminos básicos, y un conjunto de condiciones de pruebas basado en el conjunto de caminos básicos.</p>	<pre>// Entradas char[] sec; // Salidas int sumaA; int sumaE; int sumaI; int sumaO; int sumaU; // Algoritmo sumaA = sumaE = sumaI = sumaO = sumaU = 0; for(int i = 0; i < sec.length; i++) { switch(sec[i]) { case 'a': sumaA++; break; case 'e': sumaE++; } }</pre>
--	---

	<pre> break; case 'i': sumaI++; break; case 'o': sumaO++; break; case 'u': sumaU++; break; } }</pre>
--	--

Actividad 7

<p>Se da el código en Java de un algoritmo que, dada una posición inicial (x, y) y una secuencia de movimientos, calcula la posición final.</p> <p>x e y son enteros, y la secuencia de movimientos es una secuencia cuyos valores solo pueden ser uno de los siguientes:</p> <ul style="list-style-type: none"> • “<” que significa hacia la izquierda (decrementar x). • “^” que significa hacia arriba (incrementar y). • “>” que significa hacia la derecha (incrementar x). • “v” que significa hacia abajo (decrementar y). <p>Tanto x como y, como también las coordenadas de las posiciones intermedias y también de la posición final pueden tener coordenadas negativas.</p> <p>Obtén la complejidad ciclomática, un conjunto de caminos básicos, y un conjunto de condiciones de pruebas basado en el conjunto de caminos básicos.</p>	<pre> // Entradas int xInic; int yInic; char[] mov; ; // Salidas int x; int y; // Algoritmo int xMax = 9; int yMax = 9; x = xInic; y = yInic; for(int i = 0; i < mov.length; i++) { switch(mov[i]) { case '<': x--; break; case '^': y++; break; case '>': x++; break; case 'v': y--; break; } }</pre>
---	--

Actividad 8

<p>Lo mismo que para la actividad anterior, pero restringido a una cuadrícula de 10 x 10, donde las coordenadas tanto para x como para y solo pueden tomar valores de 0 a 9. Esto es válido para la posición inicial, para todas las posiciones intermedias, y para la posición final. Si un movimiento no se puede realizar porque supondría salir de la cuadrícula, entonces no altera la posición.</p>	<pre> // Algoritmo x = xInic; y = yInic; for(int i = 0; i < mov.length; i++) { switch(mov[i]) { case '<': if(x > 0) {</pre>
---	--

<p>Por ejemplo: si la coordenada x es 0, el movimiento “<” (hacia la izquierda, decrementar x), no cambiaría la posición.</p> <p>Obtén la complejidad ciclomática, un conjunto de caminos básicos, y un conjunto de condiciones de pruebas basado en el conjunto de caminos básicos.</p>	<pre> x--; } break; case '^': if(y < yMax) { y++; } break; case '>': if(x < xMax) { x++; } break; case 'v': if(y > 0) { y--; } break; } } </pre>
---	--

Actividad 9

<p>Se da el código en Java de un algoritmo que, dado un número positivo y menor que 10, obtiene su valor en números romanos.</p> <p>Obtén la complejidad ciclomática, un conjunto de caminos básicos, y un conjunto de condiciones de pruebas basado en el conjunto de caminos básicos.</p>	<pre> // Entradas int n; // Salidas String rom; // Algoritmo rom = ""; if(n == 9) { rom += "IX"; n -= 9; } else if(n == 4) { rom += "IV"; n -= 4; } else if(n >= 5) { n -= 5; rom += "V"; } while(n > 0) { n--; rom += "I"; } </pre>
---	---

Actividad 10

<p>Completa el ejercicio anterior para que funcione para números hasta 19. Cambia el algoritmo y refleja los cambios realizados en el diagrama de flujo, y a partir de ahí haz lo mismo que para ejercicios anteriores. Se puede hacer aumentando la complejidad ciclomática solo en uno.</p>

Actividad 11

Se da el código en Java de un algoritmo que ordena una secuencia de números.

Dibuja el diagrama de flujo. Obtén la complejidad ciclométrica, un conjunto de caminos básicos, y un conjunto de condiciones de prueba basado en el conjunto de caminos básicos.

```
// Entradas
int[] sec;

// Salidas
// sec

for (int i = 0; i < sec.length; i++) {
    for(int j = i+1; j < sec.length; j++) {
        if(sec[i] > sec[j]) {
            int aux = sec[i];
            sec[i] = sec[j];
            sec[j] = aux;
        }
    }
}
```

3.2. Pruebas de bucles

El conjunto de condiciones de prueba obtenidas con el método de los caminos mínimos es como mínimo un buen punto de partida, porque con ellas se ejecutan todas las sentencias del programa.

Pero como ya se ha visto, deja algo que desear en lo que respecta a la prueba de los bucles, porque para cada bucle hay un camino mínimo en el que se ejecuta una única vez. Es conveniente incluir condiciones de prueba adicionales para cada bucle. Porque además los bucles son estructuras de una importancia fundamental para la mayoría de programas.

Cabe distinguir distintos tipos de escenarios.

a) **Bucles simples.**

Los bucles simples pueden tener un número máximo de iteraciones fijo (por ejemplo: de 1 a 20). Pero habitualmente no tienen un número máximo. Es el caso del programa de ejemplo anterior para el cálculo del factorial de un número. En este programa, el número de iteraciones depende de n , siendo n el número cuyo factorial $n!$ se quiere calcular.

Para un bucle simple, deben incluirse condiciones de prueba para:

- 1) La ejecución del bucle 0 veces, es decir, que no se entre en el bucle.
- 2) La ejecución del bucle una vez.
- 3) La ejecución del bucle un número típico de veces. El significado de “típico” depende del programa a probar y del uso que se le vaya a dar.

Además, si hay un máximo número de veces m que el bucle se puede ejecutar, deben incluirse condiciones de prueba para:

- 4) La ejecución del bucle el $m-1$ veces.
- 5) La ejecución del bucle m veces.

Ya se ha visto que con el método de los caminos mínimos solo se incluyen casos de prueba para 1) y 2)

b) **Bucles anidados.** Se sugiere una estrategia para ir probando los bucles de dentro a afuera, ejecutando los más externos al que se está probado una sola vez. Esto puede no ser siempre sencillo o viable.

- 1) Comenzar con el bucle más interno, ejecutando los demás una sola vez.
- 2) Realizar las pruebas de bucles simples para el bucle más interno, ejecutando los más externos una sola vez.

- 3) Progresar hacia afuera en el siguiente bucle más externo, ejecutando los más externos a este una sola vez.
- 4) Continuar hasta que se hayan probado todos los bucles.

4. Métodos de caja negra, método de las clases de equivalencia

Los métodos de caja negra se basan en las especificaciones del programa para elaborar los casos de prueba. El programa se ve como una caja negra cuyos detalles de funcionamiento interno se desconocen, pero que debe funcionar de acuerdo a las especificaciones. El objetivo del conjunto de pruebas unitarias es verificar que funciona de acuerdo a ellas.

Lo único que se tiene para juzgar el correcto funcionamiento de un programa es su conducta observable, y más concretamente, el valor de sus salidas para cada posible valor de sus entradas. Normalmente no se puede estudiar el funcionamiento del programa para todos los posibles valores para sus entradas, por lo que hay que seleccionar un subconjunto de ellos, lo más representativos que sea posible.

Por supuesto, nunca se podrá tener la total seguridad de que el programa está libre de errores. Especialmente cuando, como es habitual, el conjunto de posibles combinaciones de valores para las entradas es infinito o enormemente grande.

Hay distintos métodos de caja negra. Aquí se va a incidir en el método de las clases de equivalencia. Y se complementará este método con el análisis de valores límites.

Con el método de las clases de equivalencia, se dividen los posibles valores de las entradas de un programa en conjuntos de valores para los que el programa tendrá una conducta equivalente. Estas son **clases de equivalencia**.

Hay clases de equivalencia para valores válidos, o **clases de equivalencia válidas**. Se supondrá que las entradas al programa se proporcionan como texto, en forma de cadenas de caracteres. Entonces, si por ejemplo una entrada puede tomar cualquier valor numérico, se pueden introducir valores erróneos, que no representan un número. Y el programa debe hacer lo correcto cuando eso sucede. Si el programa se comporta igual para todos ellos, dando el mismo mensaje de error (valor no numérico), entonces existe una única clase de equivalencia que incluye todos los valores no válidos. Las clases de equivalencia para valores no válidos se denominan **clases de error**.

Ejemplo

Se plantea un problema de ejemplo para entender el planteamiento general del método. Se tiene un programa con una entrada n que puede tomar valores numéricos, es decir, cualquier valor positivo o negativo, y con decimales. Tiene una única salida cuyo valor es -1 si el valor es negativo, 0 si el valor es cero, o 1 si el valor es positivo. Su conducta es, por tanto, equivalente para todos los valores positivos (la salida toma el valor 1), y para todos los valores negativos (la salida toma el valor -1), y para 0 (la salida toma el valor 0). También hay que considerar el caso en que la entrada tiene un valor no numérico. Para cualquier valor no numérico, el programa debe dar un mismo mensaje de error. Por lo tanto, el conjunto de posibles valores válidos para la entrada se ha dividido en tres clases de equivalencia válidas y en una clase de error. El programa se comporta igual para todos los valores de la misma clase de equivalencia. El plan de prueba unitaria debe incluir condiciones de prueba para valores particulares de cada una de las clases de equivalencia. Por ejemplo:

- Verificar que, con la entrada “hola”, el programa muestra un mensaje de error que indica que el valor de la entrada no es numérico.
- Verificar que, con la entrada -5 , la salida del programa es -1 .
- Verificar que, con la entrada 5.6 , la salida del programa es 1 .
- Verificar que, con la entrada 0 , la salida del programa es 0 .

Por lo tanto, el diseño de casos de prueba, según esta técnica, consta de los siguientes pasos:

1. Identificar las condiciones de entrada.

2. Identificar las clases de equivalencia para cada condición de entrada, distinguiendo entre clases válidas y clases de error.
3. Identificar los casos de prueba, de manera que:
 - a) Se pruebe un valor particular de cada clase de error, en una condición de prueba que no cubra más de una clase de error.
 - b) Se pruebe un valor particular de cada clase válida. Las condiciones de prueba para clases válidas pueden cubrir más de una clase válida. De hecho, deben cubrir todas las combinaciones de clases válidas que tengan sentido, como se explica a continuación.

Cada uno de los pasos anteriores se ve en un apartado a continuación.

4.1. Identificación de condiciones de entrada y sus correspondientes clases de equivalencia

Como ya se ha comentado, hay dos tipos de clases de equivalencia:

- **Clases de equivalencia válidas.** Representan entradas válidas al programa.
- **Clases de equivalencia de error.** Representan entradas no válidas al programa, para las cuales el programa mostrará una notificación de error, y no tendrá ninguna salida.

Lo primero que hay que hacer es identificar las condiciones de entrada. Y para cada una de ellas identificar las clases de equivalencia. En el caso más simple, una condición de entrada es una variable individual.

Ejemplo 1. Un programa tiene una entrada n que puede tomar valores numéricos, es decir, cualquier valor positivo o negativo, y con decimales. Tiene una única salida cuyo valor es -1 si el valor es negativo, 0 si el valor es cero, o 1 si el valor es positivo. Si el valor no es numérico, debe mostrarse un mensaje de error. Primero hay que tener en cuenta que el programa se comportará igual para todos los valores no numéricos. Estos son valores no válidos, y para ellos se mostrará un mismo mensaje de error. Por tanto, todos ellos deben pertenecer a una misma clase de error. Si el valor es numérico, se comporta igual para todos los valores positivos (la salida toma el valor 1), para todos los valores negativos (la salida toma el valor -1), y para 0 (la salida toma el valor 0).

La siguiente tabla muestra todas estas clases de equivalencia, diferenciando entre las válidas y las de error. A cada una se le asigna un número para identificarla.

Condición de entrada	Clases de equivalencia válidas	Clases de equivalencia de error
n	$n < 0$ 1 $n = 0$ 2 $n > 0$ 3	n no numérico 4

Para un planteamiento más sistemático, muchas veces se hace una primera distinción en una clase de valores válidos y otra de valores no válidos, y después una posterior en distintas clases de valores válidos.

Condición de entrada	Clases de equivalencia válidas	Clases de equivalencia de error
n	n numérico 1	n no numérico 2
n	$n < 0$ 3 $n = 0$ 4 $n > 0$ 5	

Hasta ahora, todas las clases de equivalencia se han establecido en base a condiciones de entrada que dependían de una única entrada. Pero no siempre es así, como se ve en los siguientes ejemplos.

Ejemplo 2: Programa que tiene dos entradas de tipo entero horas y minutos, y como salida una cadena de caracteres *hh:mm*, donde *hh* representa la hora con dos dígitos, y *mm* representa los minutos con dos dígitos.

La tabla para las clases de equivalencia válidas y de error para cada condición de entrada sería la siguiente.

Condición de entrada	Clases de equivalencia válidas	Clases de equivalencia de error
horas	Número entero 1	No número entero 2
horas	$0 \leq \text{horas} \leq 23$ 3	horas < 0 4 horas > 23 5
minutos	Número entero 6	No entero 7
minutos	$0 \leq \text{minutos} \leq 59$ 8	minutos < 0 9 minutos > 59 10

Ejemplo 3. Un programa tiene una entrada que representa el número de horas trabajadas en una semana, y que es un número entero no negativo. Tiene también otra entrada, numérica, que representa el sueldo por cada hora trabajada, que por supuesto no puede tomar valor 0 ni valores negativos, y que puede tener decimales. Cada hora extra trabajada (a partir de 40 horas) se paga a un 50% más. Este programa tiene una única salida, el sueldo por la semana de trabajo.

Ahora mismo no interesa el cálculo del salario mensual, sino establecer las clases de equivalencia. Hay que tener siempre cuidado con los valores en los límites del rango de valores válidos. En este caso, serían 0 para las horas trabajadas (que es un valor válido) y 0 para el salario por hora (que no es un valor válido, pero sí lo sería cualquier valor positivo, por cercano a cero que fuera).

Condición de entrada	Clases de equivalencia válidas	Clases de equivalencia de error
horas	Número entero 1	No número entero 2
horas	Número no negativo 3	Número negativo 4
horas	Entero entre 0 y 40 (inclusivos) 5 Entero mayor que 40 6	
sueldo_hora	Numérico 7	No numérico 8
sueldo_hora	sueldo_hora > 0 9	sueldo_hora <= 0 10

Ejemplo 4. Un programa determina si un valor x está dentro de un intervalo $[a, b]$, siendo a , b y x entradas del programa. Para cualquier intervalo $[a, b]$, existen tres posibles casos dependiendo de x , que corresponden a tres clases de equivalencia válidas: $x < a$ (con salida del programa *false*, que significa que x no está en $[a, b]$), $x \leq a \leq b$ (con salida del programa *true*), y $x > b$ (con salida del programa *false*). Estas son todas las clases válidas que dependen de las tres variables a , b y x , que habría que considerar además de las clases que dependen solo de los valores de a , b y x , respectivamente. También hay clases de equivalencia que dependen de a y b , porque el límite inferior del intervalo (a) no puede ser mayor que el superior (b).

Condición de entrada	Clases de equivalencia válidas	Clases de equivalencia de error
a	a numérico 1	a no numérico 2
b	b numérico 3	b no numérico 4
x	x numérico 5	x no numérico 6
a, b	$a \leq b$ 7	$a > b$ 8

a, b, x	$x < a$ 9 $x \leq a \leq b$ 10 $x > b$ 11	
---------	---	--

Ejemplo 5. las soluciones de una ecuación de segundo grado $ax^2+bx+c=0$ vienen dadas por $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. La ecuación tiene solución (o mejor dicho, tiene soluciones que son números reales) cuando

$b^2 - 4ac \geq 0$. Pero hay que distinguir entre el caso en que esta desigualdad es estricta, en el que hay dos soluciones, y el caso en que es igualdad, en el que hay una única solución. Es decir, en base a las entradas a , b y c se establece una clase de error y dos clases válidas. Si $a=0$, se tendría una división por cero con la anterior expresión. Pero en ese caso,

$ax^2+bx+c=bx+c=0$, y la solución de la ecuación $bx+c=0$ es $x = -\frac{c}{b}$. Pero de nuevo se puede tener una

división por cero si $b=0$. Pero esto corresponde al caso en que $a=b=0$. En ese caso hay que mostrar un mensaje de error, porque no se tendría una ecuación, sino una expresión que sería cierta o falsa no según el valor de x , sino según el valor de c .

Por supuesto, todo esto es complementario a las condiciones de entrada para los valores de a , de b y de c para las que se obtiene una partición de los valores que incluye clases válidas y clases de error

Condición de entrada	Clases de equivalencia válidas	Clases de equivalencia de error
a	a numérico 1	a no numérico 2
b	b numérico 3	b no numérico 4
c	c numérico 5	c no numérico 6
a, b	$a \neq 0$ 7 $a=0$ y $b \neq 0$ 8	$a=0$ y $b=0$ 9
a, b, c	$b^2 - 4ac > 0$ 10 $b^2 - 4ac = 0$ 11	$b^2 - 4ac < 0$ 12

Nota: Las clases válidas 8 y 9 incluyen los casos degenerados en los que $a=0$ y, por tanto, no se trata de una ecuación de segundo grado. En esos casos, no se podría aplicar la fórmula para una ecuación de segundo grado, porque se dividiría por cero. Dentro del caso $a=0$ Hay que distinguir dos casos:

- Si $b \neq 0$, se tiene una ecuación de primer grado $bx+c=0$, con una única solución $x = -c/b$.
- Si $b=0$, no se trata de una ecuación, porque la expresión $ax^2+bx+c=0$ queda reducida a $c=0$. En este caso, debe mostrarse un mensaje de error.

La clase 12 es una clase de error. Al aplicar la fórmula se tendría la raíz cuadrada de un número negativo, que no es un número real.

Se pueden dar reglas generales o criterios para obtener las clases de equivalencia válidas y no válidas para cada tipo de condición de entrada.

- **Rango o intervalo de valores para una entrada.** Considerar una clase de equivalencia válida (valores dentro del rango o intervalo) y dos clases de error (una con valores por debajo y otra con valores por encima). Por ejemplo, si una cantidad entera representa las horas del día, el rango es de 0 a 23, y las clases de equivalencia son: una clase válida para entre 0 y 23, y dos clases de error, una por debajo de cero, y otra por encima de 23. O más formalmente: $\{x \text{ entero con } 0 \leq x \leq 23\}$, $\{x \text{ entero con } x < 0\}$, y $\{x \text{ entero con } x > 23\}$.
- **Conjunto de valores de entrada para una entrada.** Considerar una clase de equivalencia válida y una de error. Pero si distintos valores pueden ser tratados de distinta forma, considerar una clase no válida y tantas

clases válidas como tenga sentido. Por ejemplo: si el tipo de jugador de baloncesto puede ser base, escolta, alero y pívot, pero el tratamiento es uno para base y escolta, otro para alero, y otro para pívot, considerar las clases de equivalencia válidas {base, escolta}, {alero} y {pívot}, y una clase de error que incluye el resto de valores. Si el salario por hora normal y por hora extra varían de acuerdo a un tipo de empleado que puede ser A, B o C, entonces debe haber una clase de equivalencia diferente para cada uno de estos valores.

- **Condición expresada por una expresión lógica.** Es una condición que se cumple o no. Por ejemplo, que una cadena de caracteres tenga una longitud mayor que 10, para que se considere una contraseña segura. O para una ecuación de segundo grado $ax^2+bx+c=0$, que $b^2-4ac \geq 0$, para que tenga soluciones reales. En estos ejemplos, se tendría una clase válida si se cumple la condición, y una de error si no. Pero la expresión puede establecer una partición en clases válidas. Por ejemplo, para la ecuación de segundo grado, las soluciones se calcularán de forma distinta dependiendo de si $a=0$ o no, pero las dos clases de equivalencia para cuando esta condición se cumple y para cuando no son ambas clases válidas.

4.2. Elaboración del conjunto de casos de prueba

Para elaborar un conjunto de casos de prueba, una vez que se tienen las condiciones de entrada y las clases de equivalencia, hay que seguir los siguientes pasos:

1. Asignar a cada clase de equivalencia un identificador único. En los ejemplos anteriores se ha asignado un número.
2. Incluir un caso de prueba para cada clase de error, y no incluir más de una clase de error para ningún caso de prueba. De esa forma, solo puede darse un error con cada caso de prueba, y se prueba cada uno por separado.
3. Incluir un caso de prueba para cada posible combinación de distintas clases de equivalencia válidas correspondientes a distintas condiciones de entrada.
4. Para cada caso de prueba, dar valores apropiados a todas las entradas (en la columna de condiciones previas), de manera que se tenga la combinación de clases de equivalencia cubierta por la condición de prueba. Conviene evitar los valores extremos de los intervalos de valores que puedan formar una clase de equivalencia. Estos merecen un tratamiento aparte y complementario, el análisis de valores límites, que se verá más adelante.
5. Para cada caso de prueba, especificar los valores que deben obtenerse para todas las variables de salida (en la columna de resultados esperados).

Los casos de prueba representan en una tabla con las siguientes columnas. Las clases cubiertas se anotan para facilitar la elaboración y verificar que todas las clases de equivalencia. Se puede ver que tiene dos columnas equivalentes a otras las que se obtenían con el método de los caminos mínimos, a saber: Entradas y Salidas (esta última es equivalente a resultados esperados).

Clases cubiertas	Condiciones previas	Resultados esperados

Por ejemplo, para el programa de ejemplo anterior, con dos entradas de tipo entero horas y minutos, que tiene como salida una cadena de caracteres *hh:mm*, donde *hh* representa la hora con dos dígitos, y *mm* representa los minutos con dos dígitos, se tenían las siguientes clases de equivalencia.

Condición de entrada	Clases de equivalencia válidas	Clases de equivalencia de error
horas	Número entero 1	No número entero 2
horas	$0 \leq \text{horas} \leq 23$ 3	horas < 0 4 horas > 23 5
minutos	Número entero 6	No entero 7

minutos	$0 \leq \text{minutos} \leq 59$ 8	minutos < 0 9 minutos > 59 10
---------	---	--

La tabla de condiciones de prueba quedaría como sigue. Las primeras condiciones de prueba corresponden a clases de error. Solo la última corresponde una combinación de clases válidas, la única que existe.

Clases Cubiertas	Condiciones previas	Resultados esperados
2	horas=a minutos=43	ERROR: horas no es un número entero
4	horas = -5 minutos = 20	ERROR: horas es un número negativo
5	horas=26 minutos=34	ERROR: horas mayor de 23
7	horas = 4 minutos = 6.76	ERROR: minutos no es un número entero
9	horas=12 minutos=-12	ERROR: minutos es un número negativo
10	horas=9 minutos=87	ERROR: minutos mayor de 59
1 , 3 , 6 , 8	horas=5 minutos=28	05:28

Actividad 12

Un programa calcula el sueldo semanal de un empleado según el salario base por hora y el número de horas trabajadas. El salario base por horas es un número real (puede tener decimales), y las horas trabajadas son un número entero (no puede tener decimales). A partir de 40 horas, se paga un 50% más por cada hora. Obtener un conjunto de condiciones de prueba por el método de las clases de equivalencia.

Actividad 13

Obtén un conjunto de condiciones de prueba por el método de las clases de equivalencia para el siguiente programa, ya conocido, que calcula el sueldo de un empleado según su tipo y el número de horas trabajadas. El número de horas trabajadas es un número real (puede tener decimales).

- Tipo A: hasta 40horas, a 6€ por hora, y a partir de ese número de horas a 9€ por hora.
- Tipo B: hasta 40horas, a 7€ por hora, y a partir de ese número de horas a 9,5€ por hora
- Tipo C: hasta 40horas, a 8€ por hora, y a partir de ese número de horas a 10€ por hora.

Actividad 14

Un programa con tres entradas a, b, y x, de tipo número real, tiene como salida true si x está dentro del intervalo [a, b], es decir, si $a \leq x \leq b$, y false en caso contrario. Obtener un conjunto de condiciones de prueba por el método de las clases de equivalencia.

Actividad 15

Un programa calcula las soluciones x_1 y x_2 a una ecuación de segundo grado $ax^2+bx+c=0$, que son
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Hay que considerar varios casos especiales:

- Cuando $a=b=0$, se tiene $c=0$, lo que no depende de x , sino que es cierto o falso dependiendo del valor de c . En este caso, debe dar un mensaje de error, indicando que con $a=0$ y $b=0$ no se trata de una ecuación válida.
- En otro caso, cuando $a=0$, se tiene $ax+b=0$, que es una ecuación de primer grado con una única solución. Entonces deben dar como salida $x1 = -\frac{c}{b}$ y $x2 = \text{NaN}$. Nota: NaN es un valor especial (Not a Number).
- En otro caso, si la ecuación no tiene raíces reales ($b^2 - 4ac < 0$), el programa debe dar un mensaje de error indicando que la ecuación no tiene raíces reales
- En otro caso, si $b^2 - 4ac = 0$, la ecuación solo tiene una solución, y debe devolver $x1 = -\frac{b}{2a}$ y $x2 = \text{NaN}$.
- En otro caso, la ecuación tiene dos soluciones, $x1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$ y $x2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$.

Obtén un conjunto de condiciones de prueba por el método de las clases de equivalencia. En un ejemplo anterior ya se han obtenido las condiciones de entrada.

Por supuesto, por muchas pruebas que se hagan, no se podrá demostrar la corrección del programa. Se trata de maximizar la probabilidad de encontrar fallos, si es que los hay, con un número reducido de pruebas. La gran ventaja de los métodos de caja negra es que no es necesario ningún conocimiento de cómo se ha implementado un programa, solo de sus especificaciones, es decir, de una descripción de lo que debe hacer, y no de cómo lo hace.

Validaciones, y uso complementario de los métodos de caja blanca y de caja negra

Esta metodología incide en un aspecto que se ha pasado por alto con la metodología de caja blanca: las validaciones. Con el método de caja blanca se asumía que los valores de entrada eran correctos. Con esta metodología se consideran valores de entrada incorrectos y se verifica que el programa hace lo que se espera de él cuando se introducen como entrada. Con el método de los caminos básicos podrían haberse tenido en cuenta las validaciones. Pero estas pertenecen a una fase inicial del programa que complica innecesariamente el análisis con el método de los caminos básicos, aumentando la complejidad ciclomática, al añadir algunos nuevos con los que se muestran mensajes de error, y aumentando la longitud de los ya existentes, al añadir nuevos nodos al principio para las validaciones.

En definitiva, el método de las clases de equivalencia se presta de manera más natural al diseño de condiciones de prueba para validaciones. Cada una se corresponde con una clase de error para la que se crea una condición de prueba específica e independiente del resto. Los métodos de caja blanca y de caja negra pueden utilizarse de manera complementaria. Al conjunto de condiciones de prueba obtenido con el método de los caminos básicos se pueden añadir:

1. las condiciones de prueba para valores incorrectos.
2. Las condiciones de prueba para valores válidos que prueben situaciones distintas, y no redunden en situaciones ya probadas.

A continuación se ve un método de caja negra complementario al de las clases de equivalencia y que se centra en valores particulares en los que suelen fallar los programas: los valores límites.

4.3. Análisis de valores límites

Este es un complemento al método de las clases de equivalencia, y se centra en casos particulares en los que los programas tienden a fallar más, a saber: los valores límites, o extremos, de cada intervalo o rango de valores válidos. Con este método se añaden casos de prueba para estos valores extremos, y para valores inferiores y superiores cercanos a ellos.

El análisis de valores límites permite detectar errores de programación que se producen por usar una desigualdad en lugar de una estricta o viceversa, es decir: \leq en lugar de $<$ o viceversa, o \geq en lugar de $>$, o viceversa.

También puede sacar a relucir una deficiente especificación de la funcionalidad del programa para estos valores extremos. Una especificación, por ejemplo, podría indicar que un descuento se debe aplicar a partir de un importe de 5000€. Pero con ese “a partir de” no queda del todo claro si para esa cantidad exacta debe aplicarse el descuento. Un programador podría obviar esta indefinición y hacer que se aplique el descuento con cantidades estrictamente mayores, o bien que lo aplique también para esta cantidad. El desarrollo de un plan de pruebas unitarias es una oportunidad más para clarificar qué debe hacer el programa en ese caso, para revisar las especificaciones y el programa, y para probarlo en ese caso. También podría especificarse que una cantidad se debe dividir por otra, pero no quedar claro si esta última puede ser cero, y qué debería hacerse en ese caso. El programador podría hacer que el programa realice la división sin más, y esto podría provocar un error en tiempo de ejecución que termine abruptamente con la ejecución del programa. O bien podría, en lugar de eso, mostrar un mensaje de error. Lo primero no sería del agrado del cliente o usuario final. En cuanto a lo segundo, es posible que este piense que este caso es válido, y que debería haberse tratado como un caso especial. Lo importante es que, aplicando el método de las clases de equivalencia e incluyendo el análisis de los valores límites, estos casos se incluyen en el plan de prueba y se prueban.

Deben probarse tanto los valores extremos como valores inferiores y superiores cercanos a ellos. Y esto último significa cosas distintas cuando los datos son de tipo entero y cuando son de tipo real (con decimales).

Por ejemplo, si v es una variable de entrada de tipo entero, y debe estar en un rango de 10 a 20, habría que incluir una condición de prueba para:

- Ambos valores extremos: 10 y 20. Estos pertenecerían a una clase válida.
- Valores inmediatamente por encima y por debajo de estos valores extremos. Estos serían por una parte 9 (valor no válido) y 11 (valor válido), y por otra 19 (valor válido) y 21 (valor no válido).

Si en cambio v es una variable de entrada de tipo real (número con decimales) y debe estar en un rango de 0 a 10 ambos incluidos, es decir, que $0 \leq v \leq 10$. Habría que incluir una condición de prueba para:

- Ambos valores extremos: 0 y 10. Estos pertenecerían a una clase válida.
- Valores por debajo, pero cercanos, a los valores extremos. Estos podrían ser: por una parte -0.001 (valor no válido) y 0.02 (valor válido), y por otra parte 9.998 (valor válido) y 10.01 (valor no válido).

Con variables que toman valores numéricos enteros, los valores extremos siempre son válidos. Con variables que toman valores numéricos reales (con decimales), los valores extremos pueden ser o no válidos, dependiendo de las especificaciones del programa. Se podría tener un caso como el anterior, pero con el valor 0 no válido, es decir, con $0 < v \leq 10$. Entonces el valor extremo 0 no sería válido, pero sí cualquiera mayor que él y muy cercano, por cercano que fuera a 0.

Estos casos adicionales se pueden añadir en una nueva tabla. En cada fila habría una condición de prueba para un valor límite. Para las variables que no intervienen en este valor límite debería haber un valor que no sea valor límite.

La tabla de condiciones de prueba para valores límites podría ser la siguiente.

Clase de equivalencia	Condiciones previas	Resultados esperados
3	horas = 0, minutos = 5	00:05
3	horas = -1, minutos = 17	ERROR: ...
3	horas = 1, minutos = 11	01:11
3	horas = 23, minutos = 8	23:08
3	horas = 22, minutos = 14	22:14
3	horas = 24, minutos = 20	ERROR: ...
8	minutos = 0, horas = 2	02:00
8	minutos = 59, horas = 5	05:59

8	minutos = -1, horas = 3	ERROR: ...
8	minutos = 1, horas = 8	08:01
8	minutos = 58, horas = 11	11:58
8	minutos = 60, horas = 6	ERROR: ...

Actividad 16

Se ha propuesto una actividad para aplicar el método de las clases de equivalencia a un programa que calcula el sueldo semanal de un empleado según el salario base por hora y el número de horas trabajadas. A partir de 40 horas, se paga un 50% más por cada hora. Completar el conjunto de condiciones de prueba con condiciones resultantes de aplicar el análisis de valores límites.

Actividad 17

Se ha propuesto una actividad para aplicar el método de las clases de equivalencia a un programa que calcula el sueldo de un empleado según su tipo y el número de horas trabajadas. Las horas pueden darse con decimales.

- Tipo A: hasta 40 horas, a 6€ por hora, y a partir de ese número de horas a 9€ por hora.
- Tipo B: hasta 40 horas, a 7€ por hora, y a partir de ese número de horas a 9,5€ por hora
- Tipo C: hasta 40 horas, a 8€ por hora, y a partir de ese número de horas a 10€ por hora.

Completar el conjunto de condiciones de prueba con condiciones resultantes de aplicar el análisis de valores límites.

Uso complementario de los métodos de caja blanca y de caja negra

En el ejercicio anterior deben salir 6 condiciones de prueba para combinaciones de clases válidas: por una parte, A, B o C para el tipo, y por otra hasta 40 y más de 40 para las horas trabajadas. Un programa con la funcionalidad descrita en la actividad anterior se propuso para el método de caja blanca de los caminos mínimos, y con él se obtenía un conjunto de condiciones de prueba más conciso: 4 condiciones de prueba, una para cada uno de los cuatro caminos básicos. Esta reducción en el número de condiciones de prueba es posible debido al conocimiento del código del programa. Podría haberse implementado el mismo algoritmo de otra manera, con la que tuviera complejidad ciclomática 6.

Con los métodos de caja blanca se puede aprovechar el conocimiento del código de programa para obtener un conjunto de condiciones de prueba más escueto. Pero que, en cualquier caso, garantiza la cobertura del código. Es decir, que se pase al menos una vez cada nodo del grafo de flujo y que se recorran todos sus arcos. No solo eso, sino que, para cada nodo condicional, hay dos caminos mínimos que solo se diferencian en el arco que se sigue a partir de él.

El método de caja negra de las clases de equivalencia, junto con el análisis de valores límites, son complementarios. Añaden condiciones de prueba para valores no válidos para las entradas, y condiciones de prueba para los valores límites. Ambos son aspectos importantes. Si las validaciones no se realizan correctamente, se puede poner a los programas en situaciones indeseables que suelen provocar fallos en tiempo de ejecución. Y son frecuentes los fallos con valores extremos para las entradas, debidos a errores de programación, y también a unas especificaciones deficientes.

Una buena estrategia para desarrollar un plan de pruebas unitarias puede ser:

1. Aplicar el método de caja blanca de los caminos básicos. Incluir todas las condiciones de prueba obtenidas (una para cada camino básico).
2. Aplicar el método de caja negra de las clases de equivalencia.
3. Añadir las condiciones de prueba para valores incorrectos.
4. Añadir las condiciones de prueba para valores correctos que no incidan en situaciones ya probadas con alguna condición de prueba ya existente. No hay una regla exacta para ello: a veces estará claro que una condición de prueba es redundante con otra ya existente, otras veces no lo estará tanto.

5. Realizar el análisis de valores límites. Añadir las condiciones de prueba que no sean redundantes con otras ya existentes. En cualquier caso, cada valor límite deberá probarse en al menos una condición de prueba.

Las condiciones de prueba para valores válidos que prueben situaciones distintas, y no redunden en situaciones ya probadas.

Actividad 18

Se ha propuesto una actividad para aplicar el método de las clases de equivalencia a un programa con tres entradas a , b , y x , de tipo número real, que tiene como salida true si x está dentro del intervalo $[a, b]$, es decir, si $a \leq x \leq b$, y false en caso contrario. Completar el conjunto de condiciones de prueba con condiciones resultantes de aplicar el análisis de valores límites.

Actividad 19

Se ha propuesto una actividad para aplicar el método de las clases de equivalencia a un programa que calcula las soluciones x_1 y x_2 de una ecuación de segundo grado $ax^2 + bx + c = 0$.

Completar el conjunto de condiciones de prueba con condiciones resultantes de aplicar el análisis de valores límites. Tener en cuenta que algunos valores límites dependen de más de una variable. Por ejemplo: la ecuación solo tendrá soluciones reales si $b^2 - 4ac \geq 0$. Por tanto, una condición previa será esta o, formulada de manera equivalente, $b^2 \geq 4ac$. Los valores para a , b y c para los que se cumple esta desigualdad están en una clase válida, y los valores para los que no ($b^2 < 4ac$), en una clase no válida. Los valores para los que $b^2 = 4ac$ son un caso extremo y además válido. Ahora hay que asignar valores para a , b y c para tres condiciones de prueba adicionales: una en la que b^2 sea igual a $4ac$ (caso válido), otra en la que sea un poco menor (caso no válido), y otra en la que sea un poco mayor (caso válido). Si se fija, por ejemplo, $b=6$, el primer caso se podría tener con $a=4$ y $c=9$, el segundo con $a=3.99$ y $c=9$, y el tercero con $a=4.01$ y $c=9$.

También dependen de más de una variable las condiciones de prueba para $a=0, b=0$ y para $a=0, b \neq 0$.

Actividad 20

Crear un conjunto de condiciones de prueba según el método de las clases de equivalencia, y completarlo con el análisis de valores límites, para un programa que determina la intersección de dos intervalos $[a_1, a_2]$ y $[b_1, b_2]$, si esta existe. O en otras palabras, si se solapan, incluso aunque sea en un solo punto. Las salidas del programa serán i_1 , i_2 , que son los límites del intervalo $[i_1, i_2]$ intersección de los dos anteriores, si existe (los intervalos se solapan), o bien ambas iguales a NaN si no existe (es decir, que los intervalos se solapan).

Ayuda: dos intervalos $[a_1, a_2]$ y $[b_1, b_2]$ NO se solapan en cualquiera de los siguientes dos casos:

- 1) El segundo intervalo queda completamente por debajo del primero, es decir, $b_2 < a_1$ 2) El segundo intervalo queda completamente por encima del primero, es decir, $a_1 < b_1$



Es decir, que se solapan en caso contrario, esto es, si no se cumple que $b_2 < a_1$ y tampoco que $a_2 < b_1$, es decir, si se cumple que $b_2 \geq a_1$ y que $a_2 \geq b_1$.

Es decir, que con respecto a la relación entre b_2 y a_1 , hay dos clases de equivalencia, ambas válidas. Y análogamente, con respecto a la relación entre a_2 y b_1 , hay dos clases de equivalencia, ambas válidas. El comportamiento del programa será distinto para cada combinación de una de las primeras dos clases anteriores con una de las segundas.

Los valores límites corresponden a los casos en que estas desigualdades no estrictas son igualdades, es decir, a los casos en que $b_2 = a_1$ y que $b_1 = a_2$.