S3. Métodos de la estrategia de caja blanca

Sitio: Agencia de Habilidades para el Futuro
Curso: Metodología de Pruebas de Sistemas 2° D
Libro: S3. Métodos de la estrategia de caja blanca

Imprimido por: Eduardo Moreno
Día: martes, 26 de agosto de 2025, 00:47

Descripción

Tabla de contenidos

Diagramas de flujo

Antes de introducirnos en la estrategia de caja blanca, veamos una herramienta muy útil a la hora de utilizar esta estrategia: el **diagrama de flujo**.

Un diagrama de flujo es un esquema que describe un proceso, sistema o algoritmo informático. Se usa ampliamente en numerosos campos para documentar, estudiar, planificar, mejorar y comunicar procesos -que suelen ser complejos- mediante diagramas claros y fáciles de comprender.

Los diagramas de flujo emplean rectángulos, óvalos, diamantes y otras numerosas figuras para definir el tipo de paso, junto con flechas conectoras que establecen el flujo y la secuencia. Pueden variar desde diagramas simples y dibujados a mano hasta diagramas exhaustivos creados por computadora que describen múltiples pasos y rutas.

Si tomamos en cuenta todas las diversas figuras, los diagramas de flujo son los más usados por personas con y sin conocimiento técnico en una variedad de campos.

En este primer capítulo, te invitamos a recorrer las principales características de esta herramienta.

Símbolos de los diagramas de flujo

Uso de un diagrama de flujo: la empresa potabilizadora de agua

A lo largo de la materia, y especialmente esta semana, utilizaremos el diagrama de flujo para representar de una manera gráfica el código que deberán construir en la práctica formativa. Si recordamos la semana anterior, para algunos ejemplos de *testing* de caja negra utilizamos el caso de la **empresa potabilizadora de agua**; retomaremos el mismo ejemplo para poner en práctica el uso de diagramas de flujo.

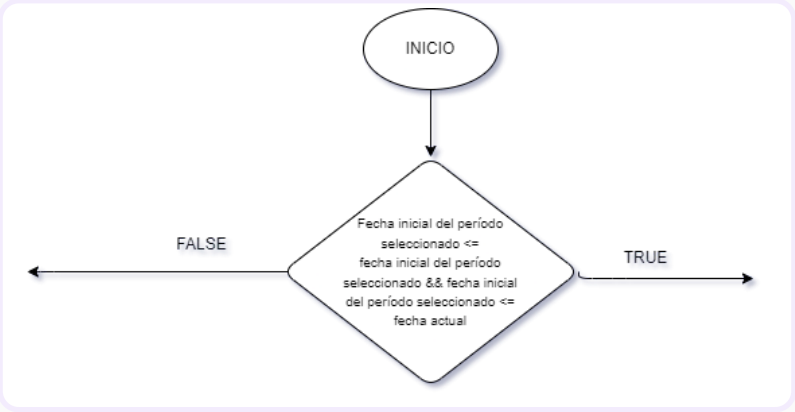
> Podés revisar el documento S2. Historias de usuario: El caso de la Empresa potabilizadora de agua para recordar los requerimientos del dominio y su historia de usuario.

Sabemos que esta Historia de Usuario es una especificación de un requerimiento de código. Se presenta a continuación una parte del pseudocódigo de la misma.

```
If (fecha inicial del período seleccionado <= fecha final del período seleccionado && fecha inicial del período seleccionado <= fecha actual)
    If (Localidad IS NOT NULL && Localidad existe)
        If (Zona IS NOT NULL && Zona existe)
Mostrar reporte estadístico de consumos para los clientes de la localidad y zona seleccionada
        Else
            If (Zona is NULL)
Mostrar reporte estadístico de consumos para TODOS los clientes de la localidad seleccionada
Else
                Mostrar mensaje “La Zona ingresada NO existe”
End if
        End if
    Else
        If (Localidad is NULL)
            Mostrar reporte estadístico de consumos para TODOS los clientes
Else
            Mostrar mensaje “La Localidad ingresada NO existe”
End if
Else
Mostrar mensaje “el período no es válido”
End if
  return go(f, seed, [])
}
```

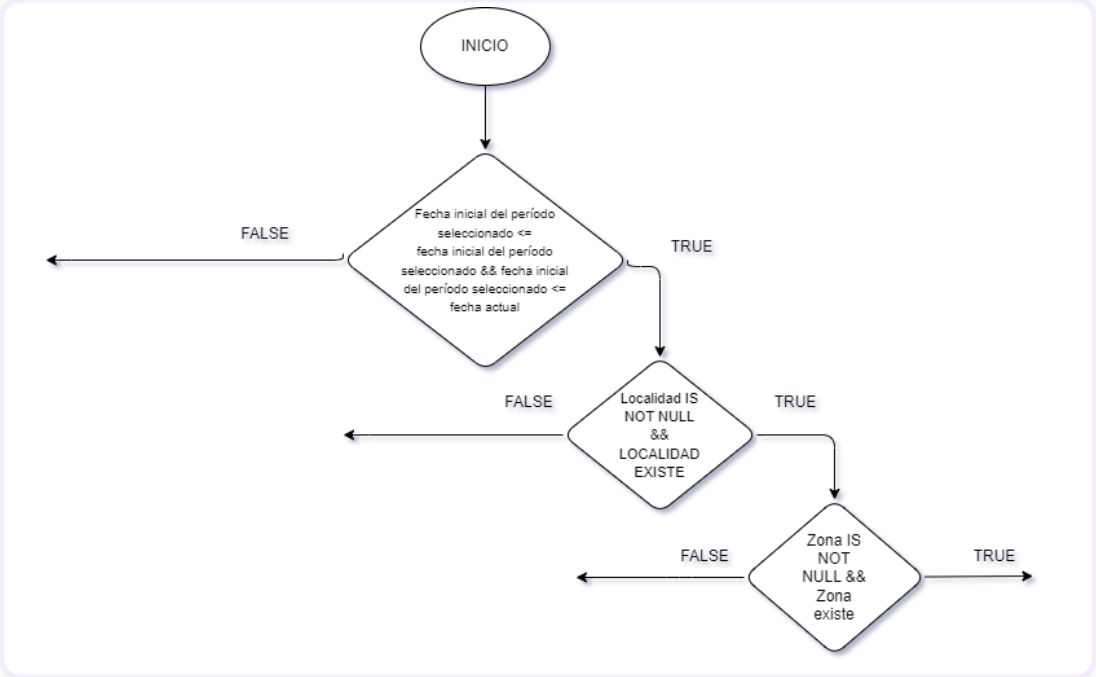
Si quisiéramos armar un diagrama de flujo que representa este pseudocódigo para comprenderlo mejor:

1. comenzaríamos con un símbolo de **inicio**.
2. Luego del símbolo de inicio nos encontramos con una sentencia IF, es decir con una **decisión**. Utilizaremos un símbolo de **decisión** como muestra la imagen a continuación:

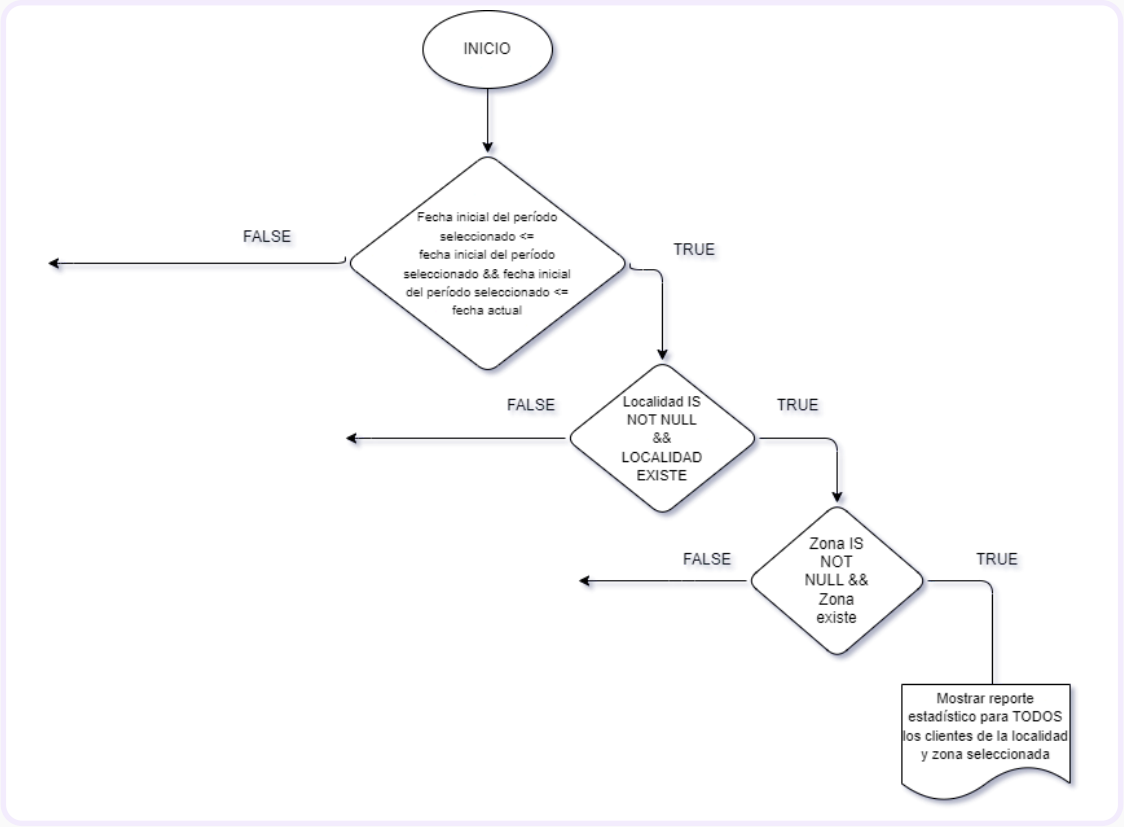


Habiendo graficado la primera decisión, queremos continuar con la siguiente. Para esto:

1. Elegiremos una de las ramas: la verdadera (**True**) o la falsa (**False**), y graficaremos la decisión.
2. En este caso comenzaremos con la rama verdadera. En esta rama tenemos otra decisión, y a la rama verdadera de esa decisión le sigue otra.
3. Si graficamos ambas decisiones nos queda un diagrama como el siguiente:



Luego de graficar las dos decisiones, encontramos una sentencia “mostrar”. Esta es una sentencia del código y utilizamos el símbolo de **proceso** para representarlo. Nos quedaría un diagrama de esta manera:

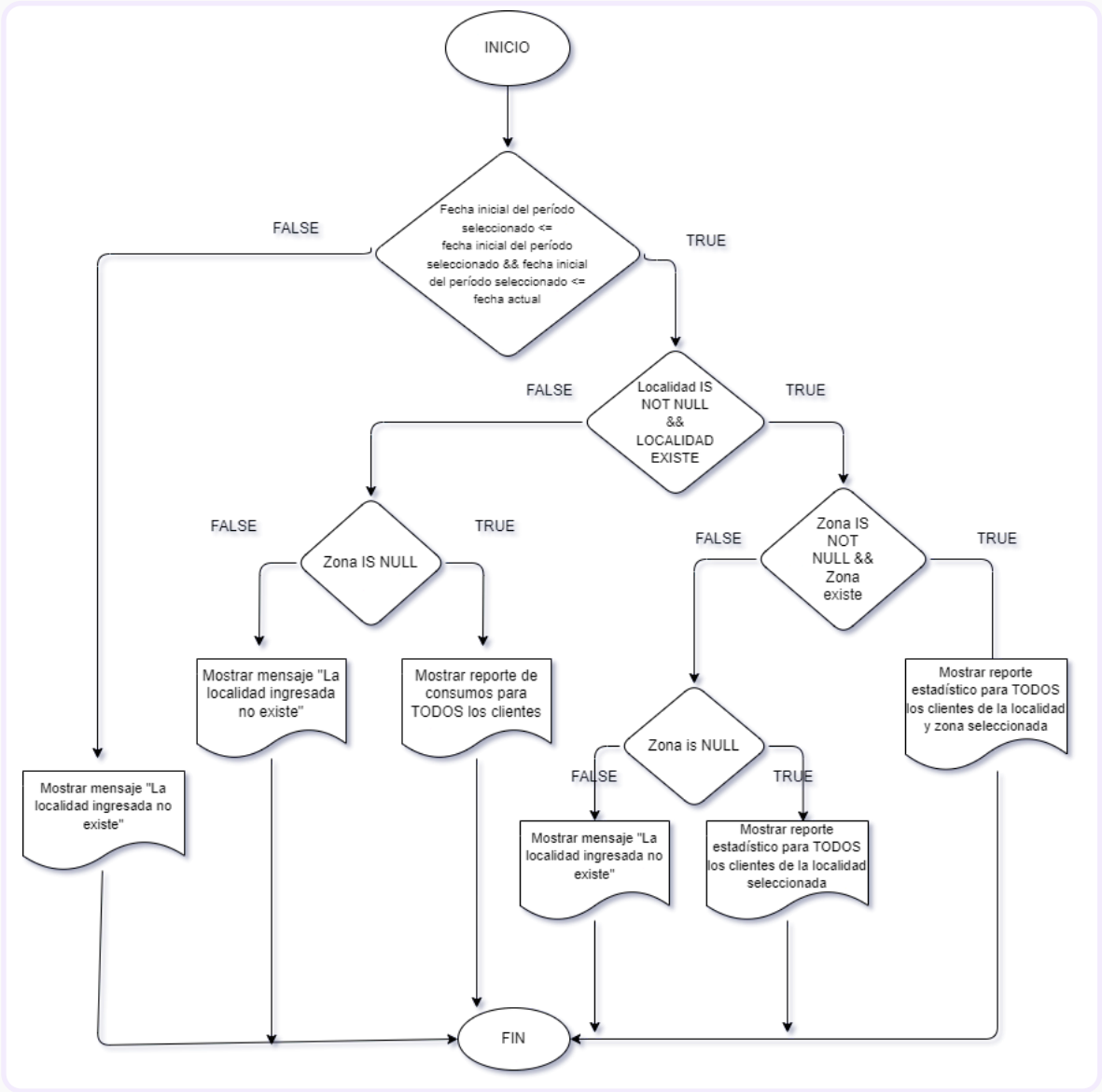


¿Cómo sigue el diagrama? **Veámoslo a continuación**.

Diagrama completo

Para graficar el resto del código, simplemente continuamos utilizando los símbolos correspondientes hasta que hemos graficado todas las decisiones y todas las sentencias normales.

Si terminamos de graficar este código, quedaría como muestra la imagen a continuación:



Ahora sí, sabiendo cómo se construye un diagrama de flujo, **podemos comenzar a estudiar los métodos de prueba de caja blanca**.

Repaso de contenidos previos

Antes de comenzar con los métodos de la estrategia de caja blanca, hagamos un repaso general de lo que vimos hasta el momento.

→ En la semana 1 realizamos una introducción a las estrategias de testing. Recordemos que las estrategias de testing nos sirven para encontrar casos de prueba con el objetivo de economizar tiempo y dinero. Economizar implica utilizar la menor cantidad de casos de prueba posibles para encontrar la mayor cantidad de errores y defectos posibles.

→ Vimos también una introducción a las dos estrategias de *testing* que existen: la estrategia de caja negra y la estrategia de caja blanca.

→ Aprendimos que en la estrategia de caja negra no debería importarnos el comportamiento interno del software ni su estructura. Esta estrategia de testing solo se concentra en encontrar las circunstancias en las cuales el software no se comporta según su especificación.

Estudiamos 4 métodos de pruebas de caja negra:

- Método de partición de clases de equivalencias
- Método de análisis de valores límites
- Método de grafo de causa – efecto
- Método de adivinación de defectos

→ En cuanto a la estrategia de caja blanca, vimos una pequeña introducción. Aprendimos que esta estrategia examina la estructura interna de un programa y que deriva casos de prueba a partir de la lógica del software, ignorando la especificación de requerimientos..

El objetivo de la estrategia de caja blanca es causar que cada declaración en el programa se ejecute al menos una vez, sabiendo que esto será extremadamente complejo e inadecuado. El número de caminos lógicos de un programa puede ser astronómicamente grande, lo que se demostró en la semana 2.

Debemos recordar también que el probar todos los caminos internos del software no necesariamente implica que la prueba está completa y que el software funciona correctamente y cómo debe funcionar.

Recordamos los motivos para esto:

1. Una prueba exhaustiva de caminos no garantiza que el programa cumpla con su especificación. Si un software no cumple con sus especificaciones, la prueba de caminos no encontrara jamás este gran problema.
2. Un programa puede estar erróneo porque faltan programar algunos caminos.
3. La prueba exhaustiva de caminos no descubriría este problema.
4. Un camino podría faltar porque el desarrollador no lo consideró, y una prueba exhaustiva de caminos nunca detectaría la falta de este camino en particular.
5. Una prueba de caminos exhaustiva puede no descubrir errores dados por los datos.

En las siguientes páginas, te invitamos a profundizar en los métodos de la estrategia de caja blanca.

Métodos de caja blanca

La estrategia de caja blanca será el foco de esta semana. Esta estrategia pretende:

- Investigar sobre la estructura interna del código, exceptuando detalles referidos a datos de entrada o salida, para probar la lógica del programa desde el punto de vista algorítmico.
- Realizar un seguimiento del código fuente según se van ejecutando los casos de prueba, determinándose de manera concreta las instrucciones, bloques, etc. que han sido ejecutados por los casos de prueba.
- Desarrollar casos de prueba que produzcan la ejecución de cada posible ruta o camino del programa o módulo, considerándose una ruta como una combinación específica de condiciones manejadas por un programa.

Hay que señalar que no todos los errores de software se pueden descubrir verificando todas las rutas de un programa, hay errores que se descubren al integrar unidades del sistema y pueden existir errores que no tengan relación con el código específicamente, como vimos en la semana 2.

En esta materia veremos como métodos de caja blanca:

1. cobertura de caminos.
2. cobertura de sentencias.
3. cobertura de decisión.
4. cobertura de condición.
5. cobertura de decisión/condición.
6. cobertura múltiple.

Método 1: Cobertura de caminos básicos

Recordemos que la esta estrategia de caja blanca, tiene como objetivo probar la lógica del software y ejecutar todas las líneas de código al menos una vez.

Iniciemos definiendo el primer método de caja blanca.

Definición Cobertura de Caminos Básicos.

Con este método se escriben casos de prueba suficientes para ejecutar todos los caminos de un programa. Entendemos que un camino es una secuencia de sentencias encadenadas desde la entrada de datos hasta la salida del software. Este criterio de cobertura asegura que los casos de prueba diseñados permiten que todas las sentencias del programa sean ejecutadas al menos una vez y que las condiciones sean probadas tanto para su valor verdadero como falso.

Una de las técnicas empleadas para aplicar este criterio de cobertura es la **prueba del camino básico**. Esta técnica se basa en obtener una medida de la complejidad del diseño procedimental de un programa (o de la lógica del programa). Esta medida es la **complejidad ciclomática** de McCabe, y representa un límite superior para el número de casos de prueba que se deben realizar para asegurar que se ejecuta cada camino del programa.

Los pasos a realizar para aplicar esta técnica son:

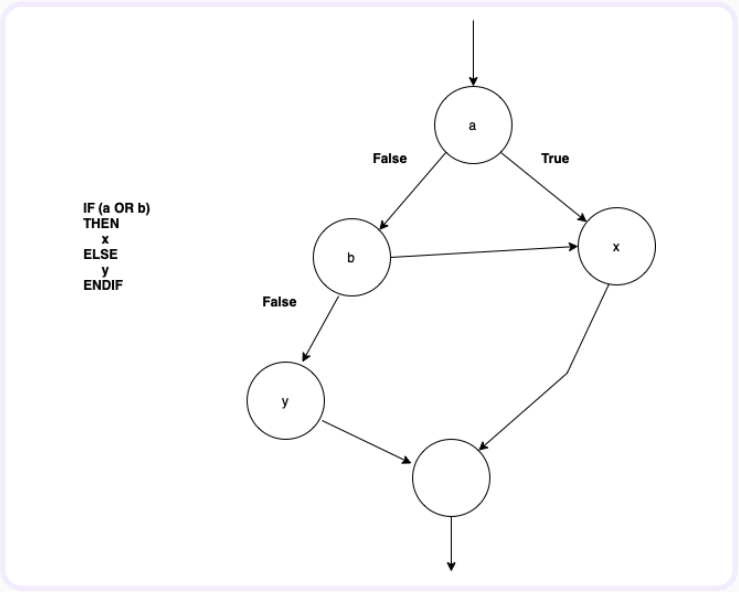
1. Representar el programa en un grafo de flujo (no confundir con un diagrama de flujo).
2. Calcular la complejidad ciclomática.
3. Determinar el conjunto básico de caminos independientes.
4. Derivar los casos de prueba que fuerzan la ejecución de cada camino.

A continuación, se detallan cada uno de los pasos.

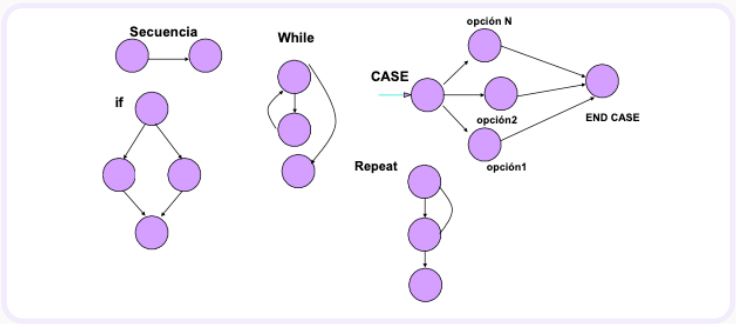
Paso 1: Grafos de flujo

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

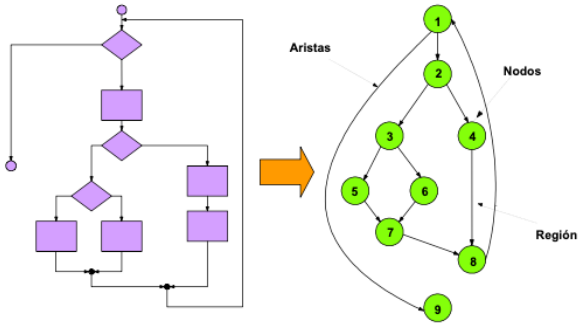
- **Nodos:** representan cero, una o varias sentencias en secuencia. Cada nodo comprende como máximo una sentencia de decisión (bifurcación).
- **Aristas:** líneas que unen dos nodos.
- **Regiones:** áreas delimitadas por aristas y nodos. Cuando se contabilizan las regiones de un programa debe incluirse el área externa como una región más.
- **Nodo predicado:** cuando en una condición aparecen uno o más operadores lógicos (AND, OR, XOR, ...) se crea un nodo distinto por cada una de las condiciones simples. Cada nodo generado de esta forma se denomina nodo predicado. La figura a continuación muestra un ejemplo de condición múltiple.



Así, cada construcción lógica de un programa tiene una representación. La siguiente figura muestra dichas representaciones.



Y la última figura muestra cómo podemos construir un grafo de flujo a partir de un diagrama de flujo. Nótese cómo la estructura principal corresponde a un WHILE, y dentro del bucle se encuentran anidados dos constructores IF.


Aristas
Nodos
Región
1
2
3
4
5
6
7
8
9

Paso 2: complejidad ciclomática

Definición La complejidad ciclomática es una métrica del software que cuantifica la complejidad lógica de un programa. En el contexto del método de prueba del camino básico, la complejidad ciclomática representa el número de rutas independientes dentro del programa, lo que a su vez determina la cantidad de casos de prueba requeridos.

¿Cómo se puede calcular la complejidad ciclomática a partir de un grafo de flujo, para obtener el número de caminos a identificar?

Existen varias formas de calcular esta complejidad:

- El número de regiones del grafo coincide con la complejidad ciclomática, V(G).
- La complejidad ciclomática, V(G), de un grafo de flujo G se define como

  V(G) = Aristas – Nodos + 2
- La complejidad ciclomática, V(G), de un grafo de flujo G se define como

  V(G) = Nodos Predicado + 1

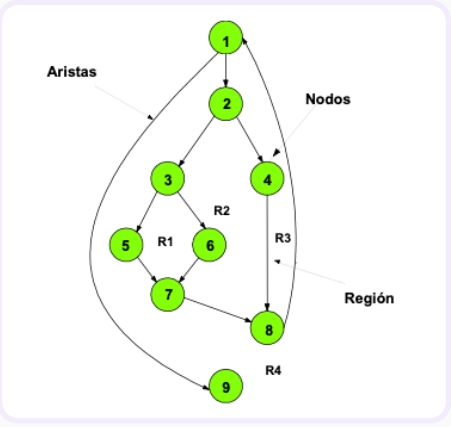
La figura siguiente representa, por ejemplo, las cuatro regiones del grafo de flujo que vimos en la figura del punto anterior, obteniéndose así la complejidad ciclomática de la figura de abajo.

Análogamente se puede calcular el número de aristas y nodos predicados para confirmar la complejidad ciclomática. Así:

V(G) = Número de regiones = 4

V(G) = Aristas – Nodos + 2 = 11-9 + 2 = 4

V(G) = Nodos Predicado + 1 = 3 +1 = 4



Esta complejidad ciclomática determina el número de casos de prueba que deben ejecutarse para garantizar que todas las sentencias de un programa se han ejecutado al menos una vez, y que cada condición se habrá ejecutado en sus vertientes verdadera y falsa.

Paso 3: caminos independientes

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

En la identificación de los distintos caminos de un programa para probar se debe tener en cuenta que cada nuevo camino debe tener el mínimo número de sentencias nuevas o condiciones nuevas respecto a los que ya existen. De esta manera se intenta que el proceso de depuración sea más sencillo.

El conjunto de caminos independientes de un grafo no es único. No obstante, a continuación, se muestran algunas heurísticas para identificar dichos caminos:

- Elegir un **camino principal** que represente una **función válida** que no sea un tratamiento de error. Debe intentar elegirse el camino que atraviese el máximo número de decisiones en el grafo.
- Identificar el segundo camino mediante la localización de la **primera decisión** en el camino de la línea básica alternando su resultado mientras se mantiene el máximo número de decisiones originales del camino inicial.
- Identificar un **tercer camino**, colocando la primera decisión en su valor original a la vez que se altera la **segunda decisión** del camino básico, mientras se intenta mantener el resto de decisiones originales.
- Continuar el proceso hasta haber conseguido **tratar todas las decisiones**, intentando mantener en su origen el resto de ellas.

Este método permite obtener V(G) caminos independientes cubriendo el método de cobertura de decisión y sentencia.

Así por ejemplo, para la el grafo de la figura 4, presentada anteriormente:

los cuatro posibles caminos independientes generados serían:

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

Estos cuatro caminos constituyen el camino básico para el grafo de flujo correspondiente.

Paso 4: casos de prueba

El ultimo paso del método "cobertura de caminos básicos" es construir los casos de prueba que fuerzan la ejecución de cada uno de esos caminos. Ya hemos aprendido a construir casos de prueba la semana 2, pero estos son un poco diferentes.

En la tabla siguiente podemos ver la información necesaria:

Número de Caso de Prueba	Precondiciones	Caso de Prueba	Resultado esperado (Si lo hubiera)

Debido a que estos casos de prueba se centran en el código, no habrá una serie de pasos en el caso de prueba, sino que debemos darle directamente un valor a las variables del mismo.

Método 2: Cobertura de sentencias

Con esta técnica, el objetivo es ejecutar todas las sentencias declarativas al menos una vez, de forma tal que cada línea de código se ejecute al menos una vez.

Esto quiere decir que vamos a diseñar casos de prueba de forma tal que fuercen a pasar por todas las sentencias.

Supongamos que tenemos un programa de cálculo de impuestos que debe calcular el impuesto sobre la renta en función del ingreso anual de una persona. El programa contiene las siguientes sentencias:

1. IngresoAnual = 50,000
2. Impuesto = 0
3. Si IngresoAnual > 30,000, entonces
4. Impuesto = (IngresoAnual - 30,000) * 0.15
5. Fin Si
6. Mostrar "El impuesto a pagar es: " + Impuesto

Para aplicar el método de cobertura de sentencias, necesitamos asegurarnos de que cada sentencia se ejecute al menos una vez durante las pruebas. Veamos cómo podríamos lograrlo:

1. Para cubrir la Sentencia 1, podríamos proporcionar un valor de IngresoAnual diferente de 50,000 en nuestras pruebas, como 40,000.
2. Para cubrir la Sentencia 2, debemos asegurarnos de que la variable Impuesto se inicialice en 0 antes de ejecutar las pruebas.
3. Para cubrir la Sentencia 3, necesitamos ejecutar pruebas con valores de IngresoAnual que sean tanto mayores como menores que 30,000 para evaluar ambas ramas del condicional.
4. Para cubrir la Sentencia 4, debemos ejecutar pruebas donde la condición (IngresoAnual > 30,000) sea verdadera.
5. Para cubrir la Sentencia 5, necesitamos ejecutar pruebas que pasen por el bloque de código dentro del "Si".
6. Para cubrir la Sentencia 6, debemos asegurarnos de que la variable Impuesto se muestre en la salida de las pruebas.

En este ejemplo, la cobertura de sentencias garantiza que todas las sentencias del programa se ejecuten al menos una vez durante las pruebas, lo que ayuda a identificar posibles errores en el código y garantiza una mayor confiabilidad del software.

Método 3: Cobertura de decisión

Con esta técnica, el objetivo es probar tanto la rama falsa como verdadera de cada decisión que tenemos.

Consideramos una decisión al tipo de sentencia que escribimos dentro de un IF, un ciclo FOR o un ciclo WHILE para controlar el flujo del código. Por lo general, una decisión puede tener un valor booleano, aunque en el caso de un SWITCH puede tener más de dos valores.

Podemos decir que las decisiones pueden ser simples o complejas, las últimas incluyen sentencias OR y/o AND.

Es decir, que con esta técnica vamos a forzar a que todas las decisiones de una porción de código se ejecuten tanto por la rama verdadera como por la rama falsa.

Analicemos el siguiente ejemplo

```
Si PuntuaciónDeCliente >= 100, entonces
  ConcederDescuento = Verdadero
Sino
  ConcederDescuento = Falso
Fin Si
```

Para aplicar la cobertura de decisión, necesitamos asegurarnos de que tanto la condición verdadera como la falsa se prueben. Veamos cómo podríamos hacerlo:

1. Caso de Prueba 1 (Condición Verdadera):
    - Establecemos `PuntuaciónDeCliente` en un valor igual o mayor a 100.
    - Verificamos que `ConcederDescuento` sea verdadero.
2. Caso de Prueba 2 (Condición Falsa):
    - Establecemos `PuntuaciónDeCliente` en un valor menor a 100.
    - Verificamos que `ConcederDescuento` sea falso.

En este ejemplo, hemos creado dos casos de prueba para cubrir ambas ramas de la decisión condicional. Esto garantiza que la decisión se haya evaluado en ambas condiciones, lo que es esencial para identificar problemas lógicos en el código y asegurar su correcto funcionamiento.

Método 4: Cobertura de condición

Con esta técnica, el objetivo es probar tanto la rama falsa como verdadera de cada condición de una porción de código. A primera vista puede sonar igual a la anterior, pero ese será sólo el caso de tener decisiones simples.

Definición

Definimos a una condición como cada una de las expresiones atómicas que conforman una decisión y que pueden anidarse utilizando operadores OR y/o AND. Con esta técnica, lo que queremos hacer es probar cada condición individualmente, sin importar el resultado de la decisión.

Es muy importante destacar que en esta técnica **no nos interesa el cortocircuitado de los operadores lógicos** .

En la mayoría de los lenguajes de programación, si observamos una decisión como la siguiente:

```
IF (X > 0 && Y < 15) {
...
}
```

podemos decir que es una sola decisión con 2 condiciones. Estas condiciones están unidas por un operador AND.

En la mayoría de los lenguajes de programación, si yo pruebo la condición "X > 0" como falsa, entonces la segunda condición no se ejecutará, ya que para que esta decisión sea verdadera necesito que ambas condiciones lo sean.

> Esta técnica, a nivel teórico, no contempla este fenómeno (el cortocircuitado de los operadores) ya que es algo que depende enteramente del lenguaje de programación. En la práctica debemos aplicar esta técnica considerando el lenguaje de programación que vayamos a utilizar para respetar el cortocircuitado de los operadores como corresponde.

También es importante destacar que para esta técnica no sólo debemos prestar atención al cortocircuitado, sino a **qué tan posible es ejecutar una prueba**.

Si tomamos el ejemplo del validador de un numero del 1 al 10 que vimos la semana 1,

- Si necesitamos que el usuario ingrese un número entre el 1 y el 10, entonces deberemos probar una entrada válida y esperada (un número del 1 al 10), pero también deberemos probar las entradas inválidas (un número negativo o un número mayor a 10).

entonces podríamos tener un código como el siguiente:

```
IF (X > 1 && x < 10) {
...
}
```

En este ejemplo no podríamos probar con un solo caso de prueba, pero no por el cortocircuitado del operador AND, sino porque es imposible probar las dos ramas falsas a la vez. Como X no puede ser menor a 1 y a la vez mayor a 10, necesitaríamos más casos de prueba para ejecutar todas las pruebas.

Método 5: Cobertura de decisión / condición

Con esta técnica, el objetivo es que cada condición en una decisión tome todas las posibles salidas, al menos una vez, y cada decisión tome todas las posibles salidas, al menos una vez.

Se puede ver como la combinatoria de las dos técnicas anteriores. Esta técnica tiene las mismas consideraciones que las 2 técnicas anteriores.

Supongamos que estamos probando un programa que decide si una persona es elegible para un descuento en una tienda en línea. El programa contiene la siguiente decisión con múltiples condiciones:

```
Si PuntuaciónDeCliente >= 100 y MiembroFrecuente = Verdadero, entonces
  ConcederDescuento = Verdadero
Sino
  ConcederDescuento = Falso
Fin Si
```

Para aplicar la Cobertura de decisión/condición, necesitamos asegurarnos de que todas las condiciones se evalúen tanto en su estado verdadero como falso. Veamos cómo podríamos hacerlo:

1. Caso de Prueba 1 (Todas las condiciones verdaderas):
    - Establecemos PuntuaciónDeCliente en un valor igual o mayor a 100.
    - Establecemos MiembroFrecuente en Verdadero.
    - Verificamos que ConcederDescuento sea verdadero.
2. Caso de Prueba 2 (Todas las condiciones falsas):
    - Establecemos PuntuaciónDeCliente en un valor menor a 100.
    - Establecemos MiembroFrecuente en Falso.
    - Verificamos que ConcederDescuento sea falso.
3. Caso de Prueba 3 (Primera condición verdadera, segunda condición falsa):
    - Establecemos PuntuaciónDeCliente en un valor igual o mayor a 100.
    - Establecemos MiembroFrecuente en Falso.
    - Verificamos que ConcederDescuento sea falso.
4. Caso de Prueba 4 (Primera condición falsa, segunda condición verdadera):
    - Establecemos PuntuaciónDeCliente en un valor menor a 100.
    - Establecemos MiembroFrecuente en Verdadero.
    - Verificamos que ConcederDescuento sea falso.

En este ejemplo, hemos creado cuatro casos de prueba para cubrir todas las combinaciones posibles de condiciones verdaderas y falsas. Esto garantiza que todas las condiciones y combinaciones dentro de la decisión se evalúen exhaustivamente, lo que es esencial para detectar posibles problemas lógicos en el código y asegurar su correcto funcionamiento.

Método 6: Cobertura de condición múltiple

Con esta técnica, el objetivo es probar que todas las combinaciones posibles de resultados de cada condición se invoquen al menos una vez.

Para poder utilizar esta técnica, utilizaremos **tablas de verdad** para ver cómo quedaría la combinatoria de todos los resultados de todas las condiciones de una porción de código.

Si eliminamos las pruebas imposibles, entonces podemos decir que tenemos una cobertura múltiple de las decisiones.

Te invitamos a leer el documento S3-Ejemplos de caja blanca donde analizaremos cómo se aplican los métodos de caja blanca en casos concretos.

Registro y seguimiento de fallas

El seguimiento de errores es el proceso de registrar y monitorear errores o defectos durante las pruebas de software. También se conoce como seguimiento de defectos o seguimiento de problemas. Los sistemas grandes pueden tener cientos o miles de defectos. Cada uno debe ser evaluado, monitoreado y priorizado para la depuración y arreglo. En algunos casos, es posible que sea necesario realizar un seguimiento de los errores durante un período prolongado.

Dice Tutorials Point:

"El seguimiento de defectos es un proceso importante en la ingeniería de software, ya que los sistemas complejos y críticos para el negocio tienen cientos de defectos. Uno de los factores desafiantes es gestionar, evaluar y priorizar estos defectos. El número de defectos se multiplica durante un período de tiempo y, para gestionarlos de forma eficaz, se utiliza un sistema de seguimiento de defectos para facilitar el trabajo".

Los defectos pasaran por varias etapas o estados a lo largo de su vida. Estos estados son los siguientes:

- **Activo**: El error o defecto está siendo investigado por los desarrolladores para ver cuál es su causa y así poder arreglarlo.
- **En prueba**: Está en etapa de pruebas nuevamente donde se verá si se ha corregido o no.
- **Cerrado**: Luego de las nuevas pruebas se puede cerrar el error si es que se ha notado que ya se arregló.
- **Reabierto**: Se volvió a encontrar el mismo error, ya sea por un usuario o un desarrollador.

Los errores se gestionan en función de la prioridad y la gravedad. Los niveles de gravedad ayudan a identificar el impacto relativo de un problema en el lanzamiento de un producto. Estas clasificaciones pueden variar en número, pero generalmente incluyen alguna forma de lo siguiente:

- **Bloqueante**: Inhibe la continuidad de desarrollo o pruebas del programa.
- **Crítico**: Crash de la aplicación, pérdida de datos o fuga de memoria severa.
- **Grave**: Pérdida mayor de funcionalidad, como menús inoperantes, datos de salida extremadamente incorrectos, o dificultades que inhiben parcial o totalmente el uso del programa.
- **Normal**: Una parte menor del componente no es funcional.
- **Leve**: Una pérdida menor de funcionalidad, o un problema que puede ser resuelto de otra manera. Es decir, algo que será un inconveniente para nuestros usuarios, pero podrán seguir utilizando nuestro software.
- **Cosmético**: Un problema estético, como puede ser una falta de ortografía o un texto desalineado.
- **Mejora**: Solicitud de una nueva característica o funcionalidad.

A la hora de registrar o reportar un error es importante que contenga lo siguiente:

- **Nombre**: Un nombre representativo del problema.
- **Caso de prueba que lo encontró**: El Caso de Prueba que estábamos usando para encontrarlo, si es que utilizamos un caso de prueba.
- **Pasos para reproducirlo**: Un listado ordenado de todos los pasos que se necesitaron para llegar a él. Es importante que si reproducimos esos pasos tenemos que llegar al mismo error, ya que un error que no puede reproducirse no es un error.
- **Gravedad**: Alguno de los niveles de gravedad que vimos anteriormente. Necesitamos saber cuál es su gravedad ya que se suele considerar a los errores bloqueantes, críticos y graves como prioritarios para su resolución.

- **Descripción**: Se suele agregar una descripción de qué es lo que ocurre, detallando porqué es un error. En esta descripción suele compararse con lo que debería ser, los resultados esperados de cada caso de prueba. Dentro de la descripción también solemos agregar capturas de pantalla del error, para ayudar a los desarrolladores a resolverlo de la mejor manera

Es importante destacar que el trabajo de los testers es encontrar errores, no resolverlos. Si bien debemos darle un seguimiento a los errores que encontramos para ver que se resuelvan, nosotros no debemos resolverlos ni tampoco es nuestro trabajo sugerir cómo pueden resolverse.