

# Prueba de aplicaciones convencionales

# Una mirada rápida

- ¿Qué es?
- ¿Quién lo hace?
- ¿Por qué es importante?
- ¿Cuáles son los pasos?
- ¿Cuál es el producto final?
- ¿Cómo me aseguro que lo hice bien?

# Fundamentos de las pruebas software

- Comprobabilidad
- Características de la prueba

# Comprobabilidad

- Operatividad
- Observabilidad
- Controlabilidad
- Descomponibilidad
- Simplicidad
- Estabilidad
- Comprensibilidad

# Características de la prueba

- Una buena prueba tiene una alta probabilidad de encontrar un error.
- Una buena prueba no es redundante.
- Una buena prueba debe ser “la mejor de la camada”
- Una buena prueba no debe ser demasiado simple o demasiado compleja.

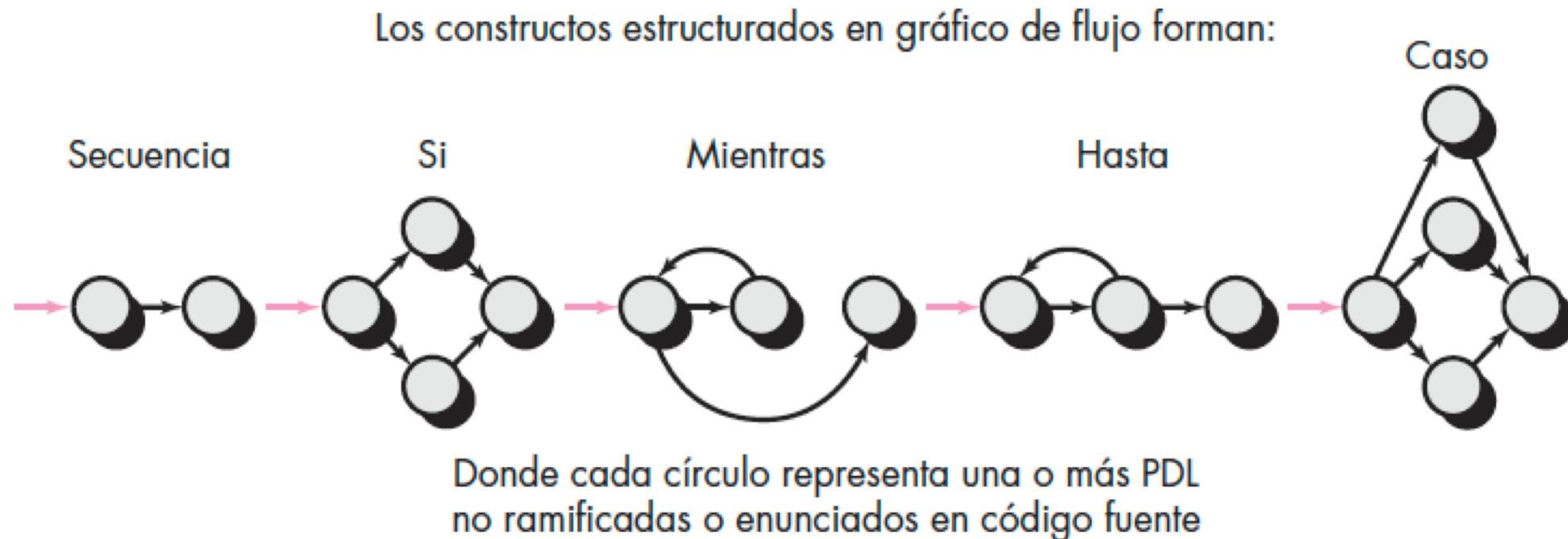
# Prueba de caja blanca

- 1) garanticen que todas las rutas independientes dentro de un módulo se revisaron al menos una vez,
- 2) revisen todas las decisiones lógicas en sus lados verdadero y falso,
- 3) ejecuten todos los bucles en sus fronteras y dentro de sus fronteras operativas y
- 4) revisen estructuras de datos internas para garantizar su validez.

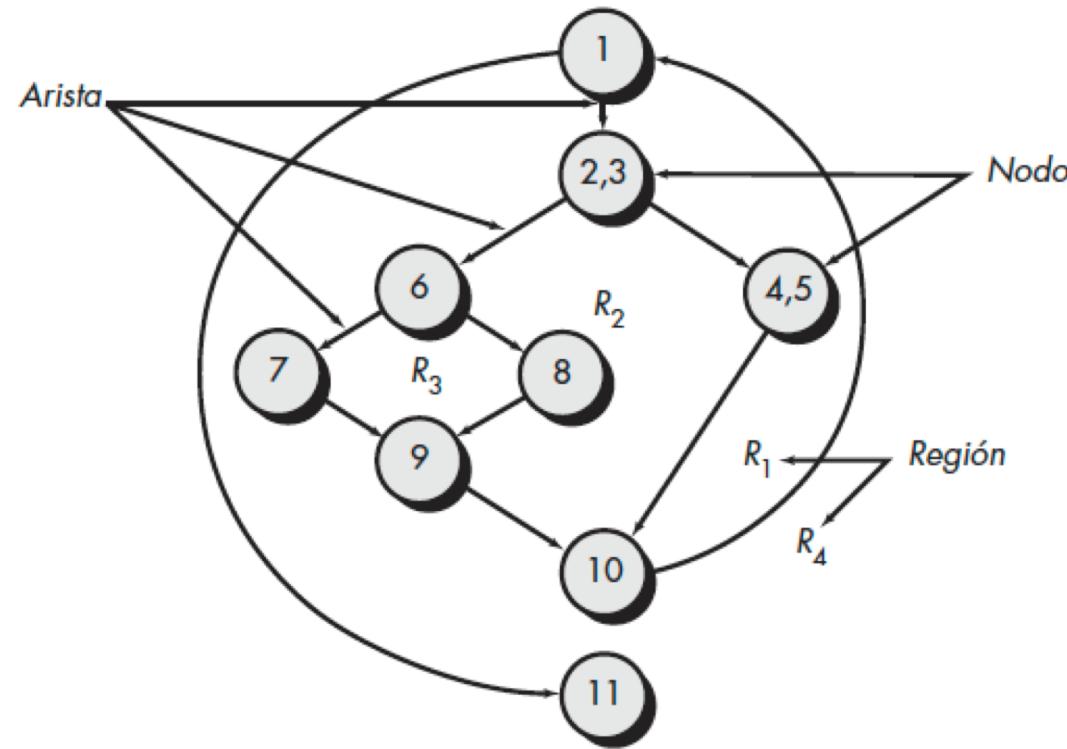
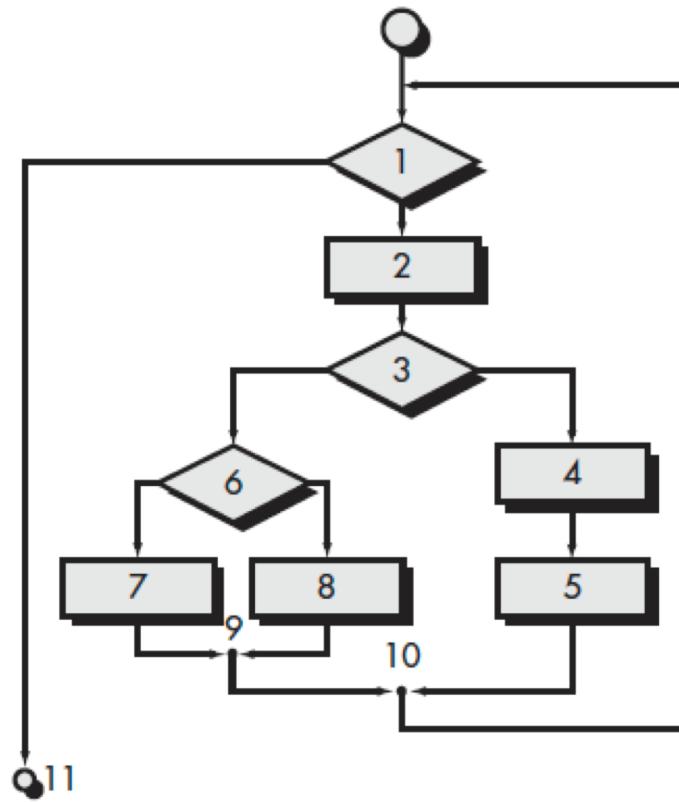
# Prueba de ruta básica

- Notación de gráfico o grafo de flujo
- Rutas de programa independientes
  - Complejidad ciclomática
- Derivación de casos de prueba
- Matrices de grafo

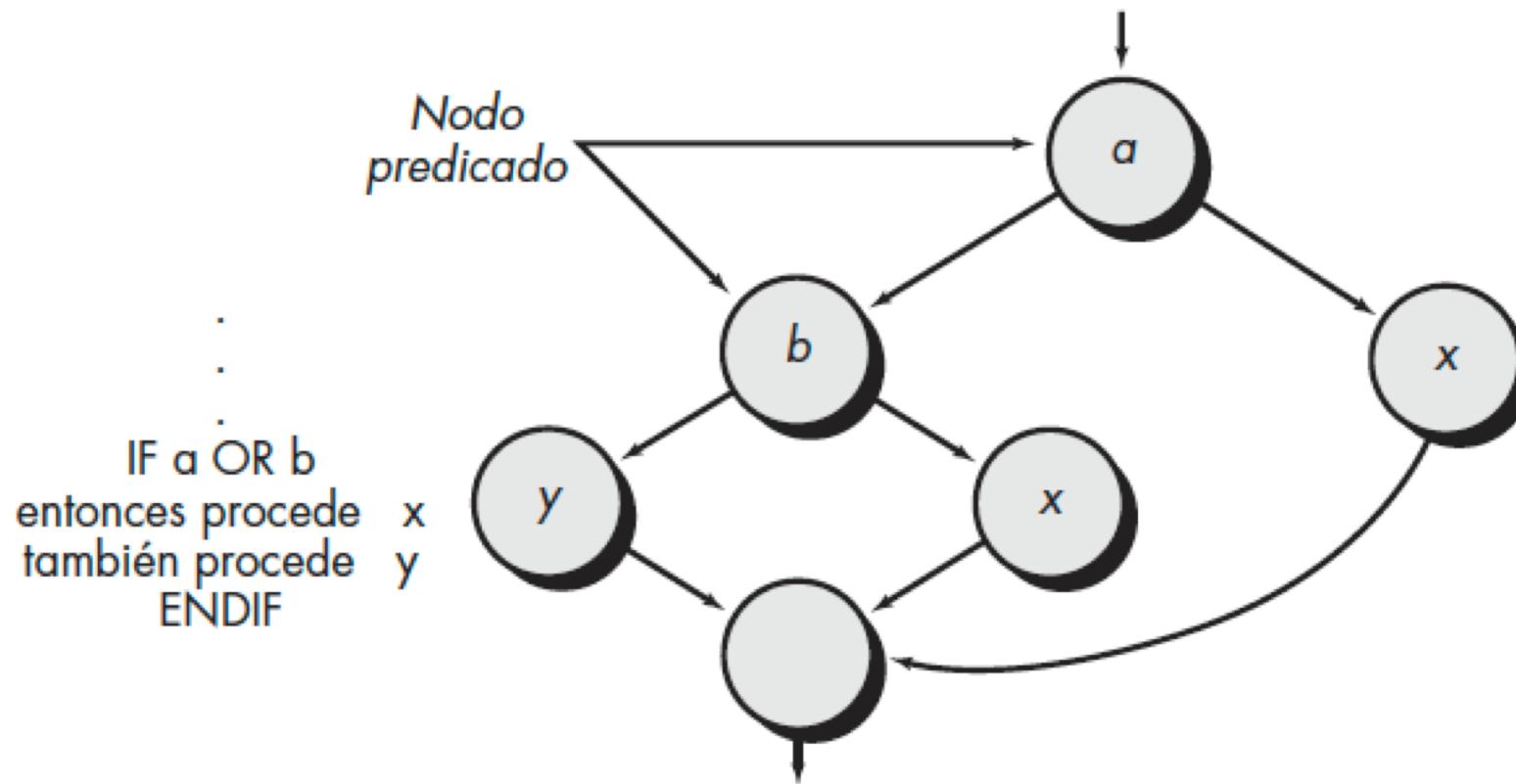
# Notación de gráfico o grafo de flujo



# Notación de gráfico o grafo de flujo



# Notación de gráfico o grafo de flujo



# Rutas de programa independientes

ruta 1: 1-11

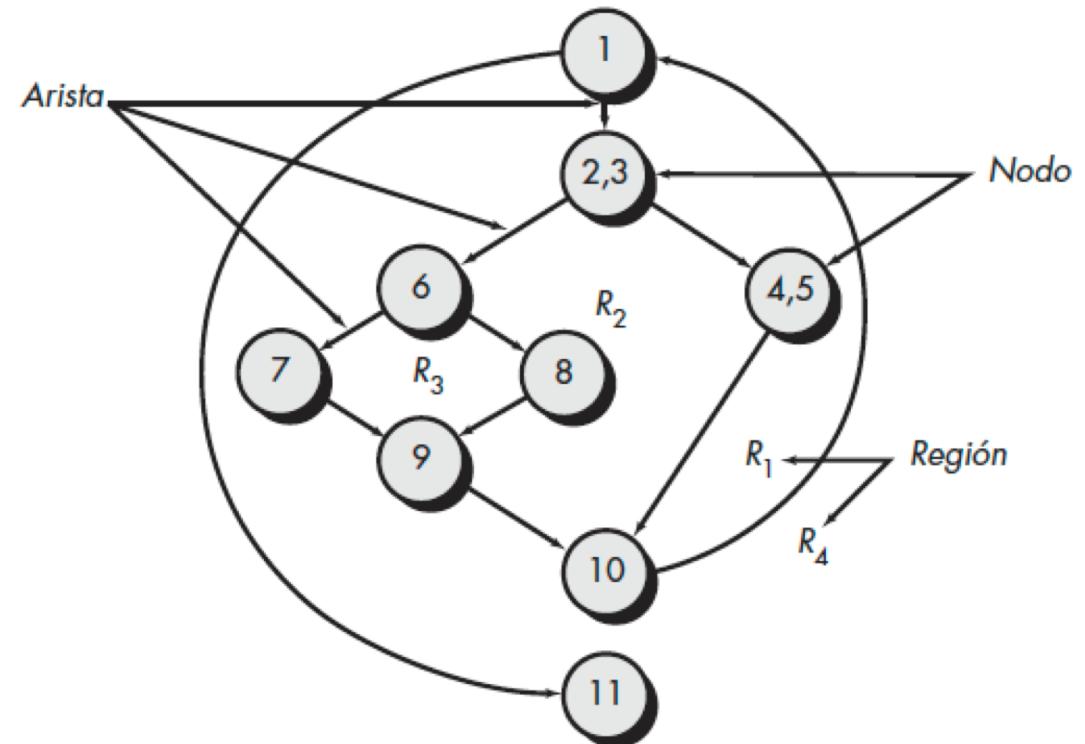
ruta 2: 1-2-3-4-5-10-1-11

ruta 3: 1-2-3-6-8-9-10-1-11

ruta 4: 1-2-3-6-7-9-10-1-11

La siguiente ruta no se considera como independiente

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11



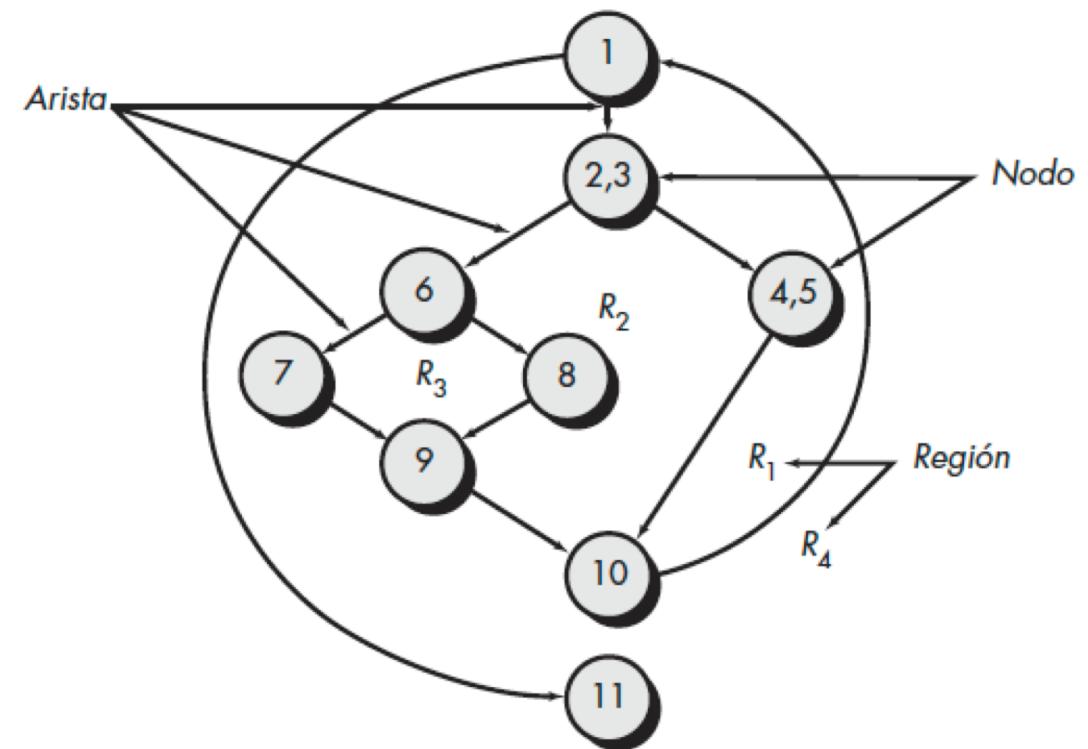
# La complejidad ciclomática

La complejidad se calcula en una de tres formas:

1. El número de regiones del gráfico de flujo corresponde a la complejidad ciclomática.
2. La complejidad ciclomática  $V(G)$  para un gráfico de flujo  $G$  se define como  $V(G) = E - N + 2$  donde  $E$  es el número de aristas del gráfico de flujo y  $N$  el número de nodos del gráfico de flujo.
3. La complejidad ciclomática  $V(G)$  para un gráfico de flujo  $G$  también se define como  $V(G) = P + 1$  donde  $P$  es el número de nodos predicado contenidos en el gráfico de flujo  $G$

# Rutas de programa independientes

1. El gráfico de flujo tiene cuatro regiones.
2.  $V(G) = 11 \text{ aristas} - 9 \text{ nodos} + 2 = 4$ .
3.  $V(G) = 3 \text{ nodos predicado} + 1 = 4$



# Derivación de casos de prueba

PROCEDIMIENTO average;

- \* Este procedimiento calcula el promedio de 100 o menos números que se encuentran entre valores frontera; también calcula la suma y el número total válido.

INTERFACE RETURNS average, total.input, total.valid;

INTERFACE ACCEPTS value, minimum, maximum;

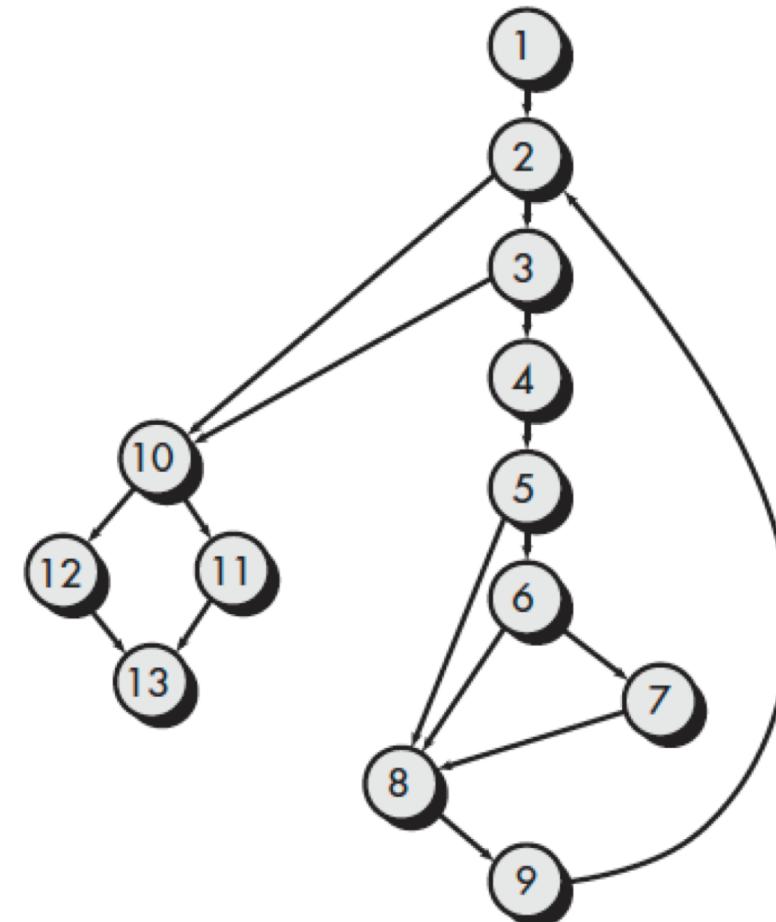
TYPE value[1:100] IS 9SCALAR ARRAY;

TYPE average, total.input, total.valid;

minimum, maximum, sum IS 9SCALAR;

TYPE i IS INTEGER;

```
1 {  
    i = 1;  
    total.input = total.valid = 0;  
    sum = 0;  
    DO WHILE value[i] <> -999 AND total.input < 100  3  
        4 increment total.input by 1;  
        IF value[i] >= minimum AND value[i] <= maximum  6  
            5 {  
                THEN increment total.valid by 1;  
                sum = sum + value[i]  
            }  
            ELSE skip  
        }  
        ENDIF  
        increment i by 1;  
    }  
    ENDDO  
    IF total.valid > 0  10  
        11 {  
            THEN average = sum / total.valid;  
        }  
        ELSE average = -999;  
    }  
    13 ENDIF  
END average
```

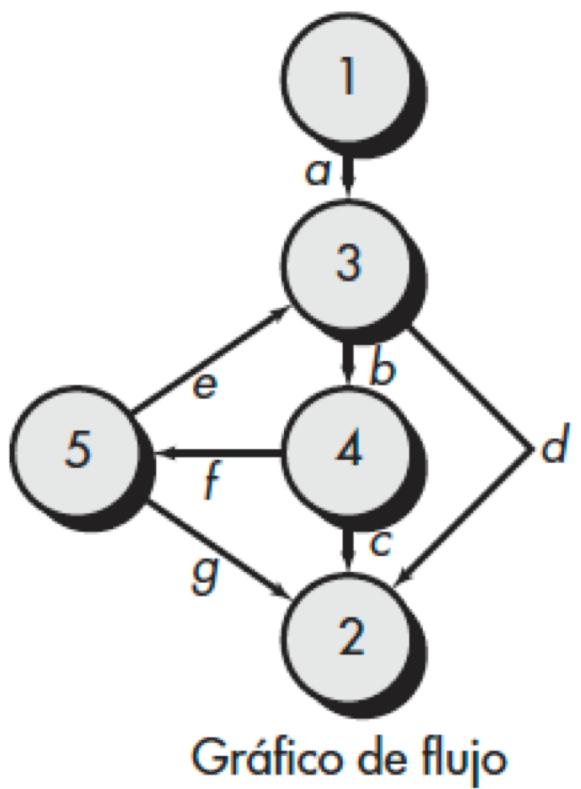


# Derivación de casos de prueba

Es posible aplicar los siguientes pasos para derivar el conjunto básico:

1. Al usar el diseño o el código como cimiento, dibuje el gráfico de flujo correspondiente.
2. Determine la complejidad ciclomática del gráfico de flujo resultante.
3. Determine un conjunto básico de rutas linealmente independientes.
4. Prepare casos de prueba que fuercen la ejecución de cada ruta en el conjunto básico.

# Matrices de grafo



Conectado a nodo

Nodo	1	2	3	4	5
1			a		
2					
3		d		b	
4	c				f
5	g	e			

Matriz de grafo

# Prueba de la estructura de control

- Prueba de condición
- Prueba de bucle

# Prueba de condición

La prueba de condición es un método de diseño de casos de prueba que revisa las condiciones lógicas contenidas en un módulo de programa. Una condición simple es una variable booleana o una expresión relacional, posiblemente precedida de un operador NOT ( $\neg$ ).

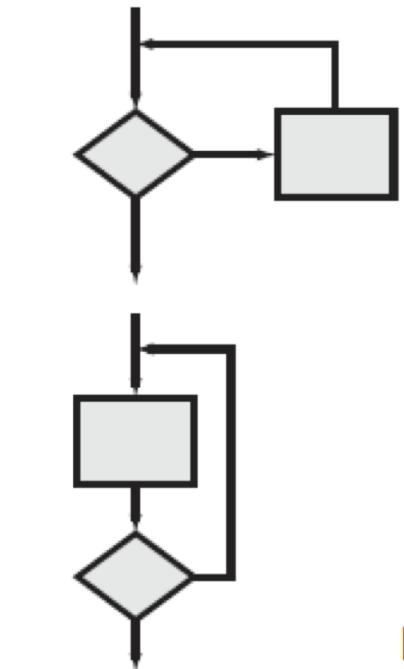
Una expresión relacional toma la forma E1 <operador relacional> E2.

Una condición compuesta se integra con dos o más condiciones simples, operadores booleanos y paréntesis.

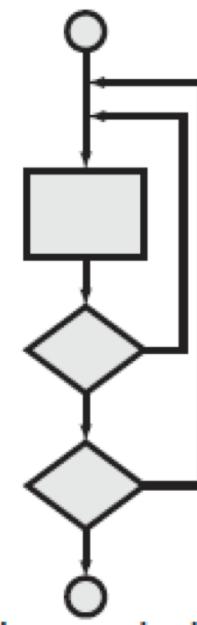
Si una condición es incorrecta, entonces al menos un componente de la condición es incorrecto.

El método de prueba de condición se enfoca en la prueba de cada condición del programa para asegurar que no contiene errores.

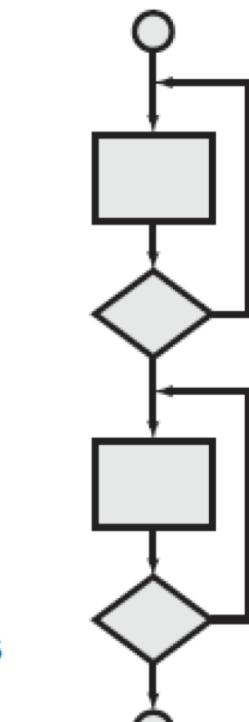
# Prueba de bucle



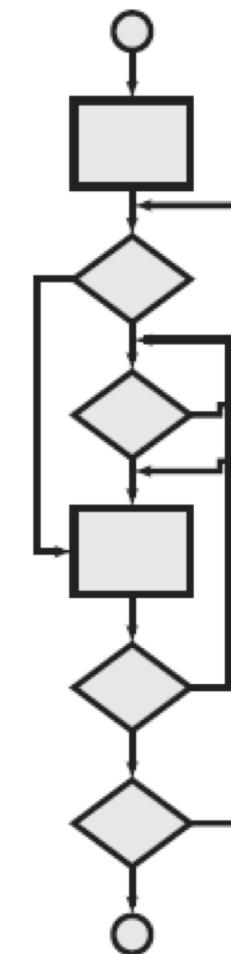
Bucles simples



Bucles anidados



Bucles  
concatenados



Bucles no  
estructurados

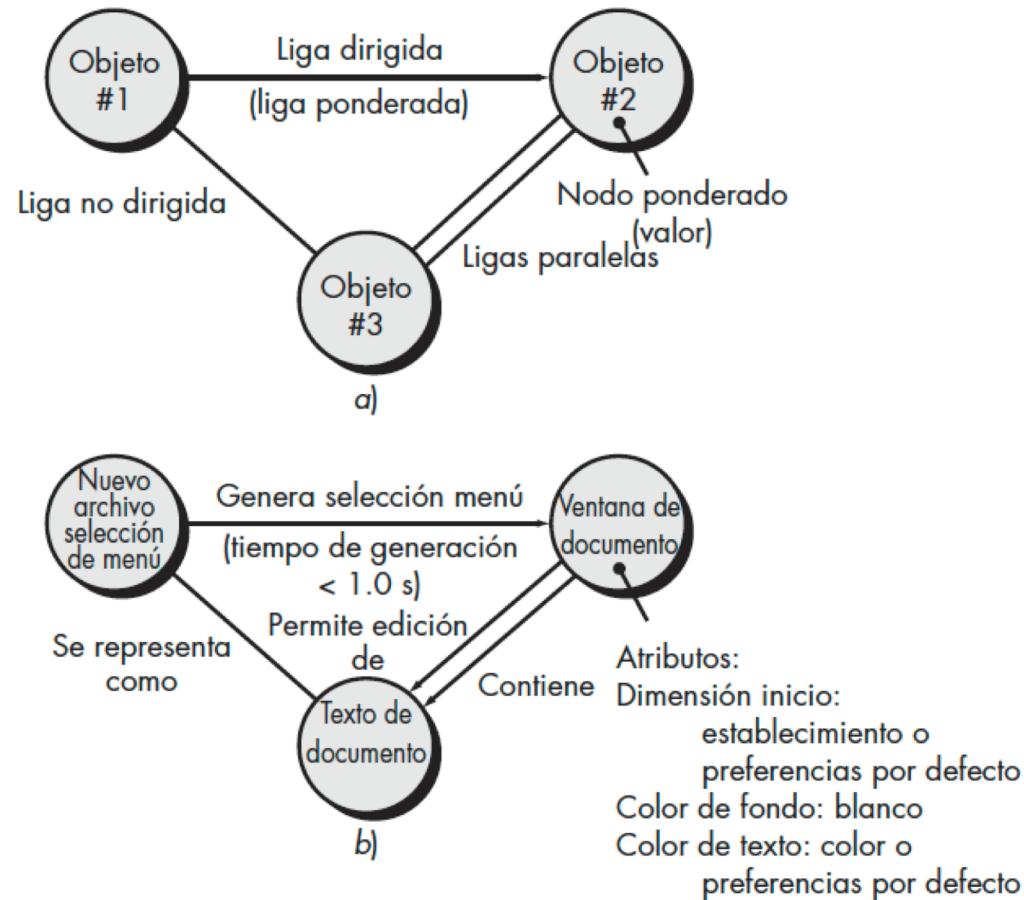
# Prueba de caja negra

Las pruebas de caja negra se enfocan en los requerimientos funcionales del software. No son una alternativa para las técnicas de caja blanca.

Las pruebas de caja negra intentan encontrar errores en las categorías siguientes:

- 1) funciones incorrectas o faltantes,
- 2) errores de interfaz,
- 3) errores en las estructuras de datos o en el acceso a bases de datos externas,
- 4) errores de comportamiento o rendimiento y
- 5) errores de inicialización y terminación.

# Métodos de prueba basados en gráficos



# Partición de equivalencia

La partición de equivalencia es un método de prueba de caja negra que divide el dominio de entrada de un programa en clases de datos de los que pueden derivarse casos de prueba.

El diseño de casos de prueba para la partición de equivalencia se basa en una evaluación de las clases de equivalencia para una condición de entrada.

Una clase de equivalencia representa un conjunto de estados válidos o inválidos para condiciones de entrada.

Por lo general, una condición de entrada es un valor numérico específico, un rango de valores, un conjunto de valores relacionados o una condición booleana. Las clases de equivalencia pueden definirse de acuerdo con los siguientes lineamientos:

1. Si una condición de entrada especifica un rango, se define una clase de equivalencia válida y dos inválidas.
2. Si una condición de entrada requiere un valor específico, se define una clase de equivalencia válida y dos inválidas.
3. Si una condición de entrada especifica un miembro de un conjunto, se define una clase de equivalencia válida y una inválida.
4. Si una condición de entrada es booleana, se define una clase válida y una inválida.

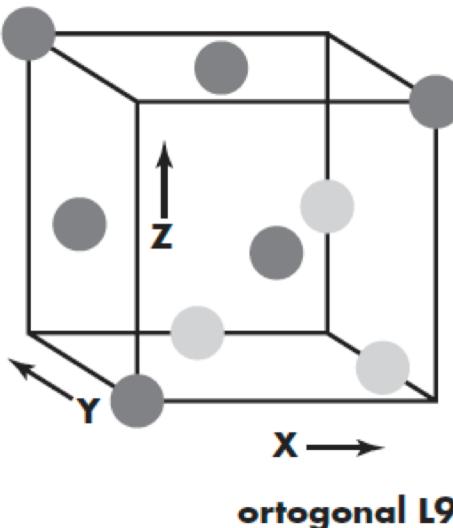
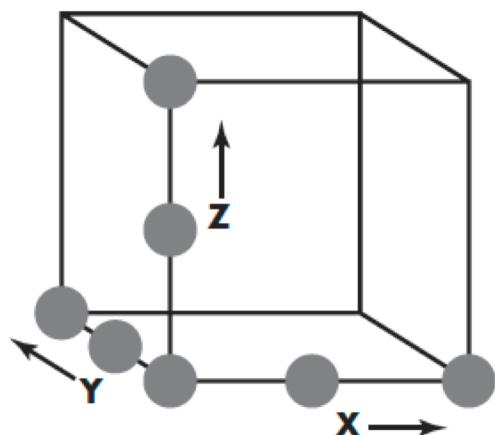
# Análisis de valor de frontera

Un mayor número de errores ocurre en las fronteras del dominio de entrada y no en el “centro”. Por esta razón es que el análisis de valor de frontera (BVA) se desarrolló como una técnica de prueba. El análisis de valor de frontera conduce a una selección de casos de prueba que revisan los valores de frontera.

Los lineamientos para el BVA son similares en muchos aspectos a los proporcionados para la partición de equivalencia:

1. Si una condición de entrada especifica un rango acotado por valores  $a$  y  $b$ , los casos de prueba deben designarse con valores  $a$  y  $b$ , justo arriba y justo abajo de  $a$  y  $b$ .
2. Si una condición de entrada especifica un número de valores, deben desarrollarse casos de prueba que revisen los números mínimo y máximo. También se prueban los valores justo arriba y abajo, mínimo y máximo.
3. Aplicar lineamientos 1 y 2 a condiciones de salida. Deben diseñarse casos de prueba para crear un reporte de salida que produzca el número máximo y mínimo permisible de entradas de tabla.
4. Si las estructuras de datos de programa internos tienen fronteras prescritas, asegúrese de diseñar un caso de prueba para revisar la estructura de datos en su frontera.

# Prueba del array ortogonal



Caso de prueba	Parámetros de prueba			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

# Prueba basada en modelo

La prueba basada en modelo (PBM) es una técnica de prueba de caja negra que usa la información contenida en el modelo de requerimientos como la base para la generación de casos de prueba.

La técnica PBM requiere cinco pasos:

1. Analizar un modelo de comportamiento existente para el software o crear uno.
2. Recorrer el modelo de comportamiento y especificar las entradas que forzarán al software a realizar la transición de estado a estado.
3. Revisar el modelo de comportamiento y observar las salidas esperadas, conforme el software realiza la transición de estado a estado.
4. Ejecutar los casos de prueba.
5. Comparar los resultados reales y esperados y adoptar una acción correctiva según se requiera.

# Prueba para entornos, arquitecturas y aplicaciones especializados

Pruebas de interfaces gráficas de usuario

Prueba de arquitecturas cliente-servidor

Documentación de prueba y centros de ayuda

Prueba para sistemas de tiempo real

# Pruebas de interfaces gráficas de usuario

Las interfaces gráficas para usuario (GUI) presentan interesantes retos de prueba. Puesto que los componentes reutilizables ahora son parte común en los entornos de desarrollo GUI, la creación de la interfaz para el usuario se ha vuelto menos consumidora de tiempo y más precisa. Pero, al mismo tiempo, la complejidad de las GUI ha crecido, lo que conduce a más dificultad en el diseño y ejecución de los casos de prueba.

Debido a que muchas GUI modernas tienen la misma apariencia y ambiente, puede derivarse una serie de pruebas estándar

Como producto del gran número de permutaciones asociadas con las operaciones de la GUI, la prueba de GUI debe abordarse usando herramientas automatizadas. Durante los últimos años apareció una amplia gama de herramientas de prueba GUI.

# Prueba de arquitecturas cliente-servidor

En general, la prueba del software cliente-servidor ocurre en tres niveles diferentes:

- 1) las aplicaciones cliente individuales se prueban en un modo “desconectado”; no se considera la operación del servidor ni la red subyacente.
- 2) El software cliente y las aplicaciones servidor asociadas se prueban en concierto, pero las operaciones de red no se revisan de manera explícita.
- 3) Se prueba la arquitectura cliente-servidor completa, incluidos la operación de red y el rendimiento.

# Prueba de arquitecturas cliente-servidor

Para las aplicaciones cliente-servidor se encuentran comúnmente los siguientes abordajes de prueba:

Pruebas de función de aplicación

Pruebas de servidor

Pruebas de bases de datos

Pruebas de transacción

Pruebas de comunicación de red

# Documentación de prueba y centros de ayuda

El término prueba de software invoca imágenes de gran número de casos de prueba preparados para revisar los programas de cómputo y los datos que manipulan. Es importante notar que las pruebas también deben extenderse al tercer elemento de la configuración del software: la documentación.

Los errores en la documentación pueden ser tan devastadores para la aceptación del programa como los errores en los datos o en el código fuente.

# Pruebas para sistemas en tiempo real

La naturaleza asíncrona agrega un nuevo y potencialmente difícil elemento a la mezcla de pruebas: el tiempo.

Los casos de prueba deben considerar la manipulación de eventos, la temporización de los datos y el paralelismo de las tareas que manejan los datos.

Los métodos amplios de diseño de casos de prueba para sistemas en tiempo real continúan evolucionando. Sin embargo, puede proponerse una estrategia global de cuatro pasos:

- Prueba de tareas
- Prueba de comportamiento
- Prueba intertarea.
- Prueba de sistema

# Patrones para pruebas software

Los patrones de prueba no sólo proporcionan lineamientos útiles conforme comienzan las actividades de prueba; también proporcionan tres beneficios adicionales descritos por Marick:

1. Proporcionan un vocabulario para quienes solucionan problemas. “Oiga, usted sabe, debemos usar un objeto nulo”.
2. Enfocan la atención en las fuerzas que hay detrás de un problema. Esto permite que los diseñadores [de caso de prueba] entiendan mejor cuándo y por qué se aplica una solución.
3. Alientan el pensamiento iterativo. Cada solución crea un nuevo contexto en el que pueden resolverse nuevos problemas.

# Patrones para pruebas software

<b>PairTesting</b>	Patrón orientado a proceso, PairTesting describe una técnica que es análoga a la programación por parejas en la que dos examinadores trabajan en conjunto para diseñar y ejecutar una serie de pruebas que pueden aplicarse a actividades de prueba de unidad, integración o validación.
<b>SeparateTestInterface</b>	Hay necesidad de probar cada clase en un sistema orientado a objetos, incluidas “clases internas” (es decir, clases que no exponen alguna interfaz afuera del componente que los usa). El patrón SeparateTestInterface describe cómo crear “una interfaz de prueba que puede usarse para describir pruebas específicas sobre clases que son visibles solamente de manera interna en un componente”.
<b>ScenarioTesting</b>	Una vez realizadas las pruebas de unidad e integración, hay necesidad de determinar si el software se desempeñará en forma que satisfaga a los usuarios. El patrón ScenarioTesting describe una técnica para revisar el software desde el punto de vista del usuario. Un fallo en este nivel indica que el software fracasó para satisfacer un requisito visible del usuario .

# Resumen

El objetivo principal para el diseño de casos de prueba es derivar un conjunto de pruebas que tienen la mayor probabilidad de descubrir errores en el software. Para lograr este objetivo, se usan dos categorías diferentes de técnicas de diseño de caso de prueba: pruebas de caja blanca y pruebas de caja negra.

Con frecuencia, los desarrolladores de software experimentados dicen: “las pruebas nunca terminan, sólo se transfieren de uno [el ingeniero de software] al cliente. Cada vez que el cliente usa el programa, se realiza una prueba”. Al aplicar el diseño de casos de prueba, pueden lograrse pruebas más completas y, en consecuencia, descubrir y corregir el mayor número de errores antes de comenzar “las pruebas del cliente”.