Desarrollo de interfaces:

8. REALIZACIÓN DE PRUEBAS

Jose Alberto Benítez Andrades
jose@indipro.es
www.indipro.es
@indiproweb
@jabenitez88

REALIZACIÓN DE PRUEBAS – Contenido

- Objetivo, importancia y limitaciones del proceso de prueba. Estrategias.
- Pruebas de integración: ascendentes y descendentes.
- Pruebas de sistema: configuración, recuperación, entre otras.
- Pruebas de regresión.
- Pruebas funcionales.
- Pruebas de capacidad y rendimiento.
- Pruebas de uso de recursos.
- Pruebas de seguridad.
- Pruebas manuales y automáticas. Herramientas software para la realización de pruebas.
- Pruebas de usuario.
- Pruebas de aceptación.
- Versiones alfa y beta.

REALIZACIÓN DE PRUEBAS

- Cuando se desarrolla software se realizan una serie de actividades en la que hay enormes posibilidades de que aparezca el error humano.

- Los errores pueden presentarse desde el primer momento.

- Debido a la imposibilidad humana de trabajar y comunicarse de forma perfecta es necesario acompañar el desarrollo del SW de una actividad que garantice la calidad.

- La pruebas son un elemento crítico para la garantía de la calidad del SW y representan una revisión final de las especificaciones, el diseño y de la codificación.

REALIZACIÓN DE PRUEBAS

- No es raro que una organización de desarrollo de SW emplee entre el 30 y el 40 por ciento del esfuerzo total del proyecto en las pruebas.
- La pruebas de SW para actividades críticas (Control de tráfico aéreo o de reactores nucleares) pueden costar de 3 a 5 veces más que el resto de los pasos de la Ing. De SW juntos.
- En la fases anteriores el Ing. Intenta “construir” el SW.
- Al llegar a la pruebas crea una serie de casos de prueba que intentan “demoler” el SW construido.
- La pruebas son el paso de la Ing. de SW que se puede ver (psicológicamente) como destructivo en lugar de constructivo.

REALIZACIÓN DE PRUEBAS

- Los Ing. de SW son por naturaleza constructivos.
- Las pruebas requieren que se descarten ideas preconcebidas sobre la "corrección" del SW que de acaba de desarrollar y se supere cualquier conflicto de intereses que aparezca cuando se descubran errores.
- Existe un mito que dice que si fuéramos realmente buenos programando, no habría errores que buscar. No habría errores.
- Según el mito existen errores porque somos malos en lo que hacemos y debemos sentirnos culpables por ello.
- Las pruebas serían un reconocimiento de nuestros fallos e implica una buena dosis de culpabilidad.

REALIZACIÓN DE PRUEBAS – Objetivos de las pruebas

- La prueba es el proceso de ejecución de un programa con la intención de descubrir un error.
- Un buen caso de prueba es aquel que tiene una alta probabilidad de mostrar un error no descubierto hasta entonces.
- Una prueba tiene éxito si descubre un error no detectado hasta entonces.
- Hay que quitarse la idea, que normalmente tenemos, de que una prueba tiene éxito si no descubre errores.
- El objetivo es diseñar pruebas que sistemáticamente saquen a la luz diferentes clases de errores, haciéndolo con la menor cantidad de tiempo y esfuerzo.

REALIZACIÓN DE PRUEBAS – Objetivos de las pruebas

- Como ventaja secundaria, la prueba demuestra hasta qué punto la funciones del SW trabajan de acuerdo con las especificaciones.

- Los datos recogidos indican en cierta medida la fiabilidad y calidad del SW como un todo.

- La prueba no puede asegurar la ausencia de defectos; sólo puede demostrar que existen defectos en el SW.

REALIZACIÓN DE PRUEBAS – Principios

- A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente
- Las pruebas deberían planificarse mucho antes de que empiecen.
- El principio de Pareto es aplicable a las pruebas del SW. (80% de los errores en 20% de los módulos)
- Las pruebas deberían empezar por “lo pequeño” y progresar hacia “lo grande”.
- No son posibles las pruebas exhaustivas.
- Para ser más eficaces la pruebas deberían ser realizadas por un equipo independiente.
- Es la facilidad con la que se puede probar un SW.

Caraterísticas que llevan a un SW. Fácil de probar.

- OPERATIVIDAD. Cuanto mejor funcione, más eficientemente se puede probar.
  - Pocos errores
  - Ningún error bloquea las pruebas.

- OBSERVABILIDAD: Lo que ves es lo que pruebas.
  - El código fuente es accesible.
  - Se informa automáticamente de los errores internos.
  - Un resultado incorrecto se identifica fácilmente.
  - Los errores internos se detectan automáticamente a través de mecanismos de autocomprobación.

Caraterísticas que llevan a un SW. Fácil de probar.

- CONTROLABILIDAD. Cuanto mejor podamos controlar el SW , más se puede automatizar y optimizar.
  - El Ing. De pruebas puede controlar directamente los estados y variables del SW y HW.
  - Las pruebas pueden especificarse, automatizarse y reproducirse convenientemente.

- CAPACIDAD DE DESCOMPOSICIÓN. Controlando el ámbito de las pruebas, podemos aislar más rápidamente los problemas y llevar a cabo mejores pruebas.
  - SW. Construido en módulos.
  - Los módulos se pueden probar independientemente.

Caraterísticas que llevan a un SW. Fácil de probar.

- SIMPLICIDAD: Cuanto menos haya que probar, más rápidamente podremos probarlo.
  - Simplicidad funcional (Características mínimas para cumplir los requisitos)
  - Simplicidad Estructural (Arquitectura modularizada)
  - Simplicidad del código.

- ESTABILIDAD. Cuanto menos cambios, menos interrupciones a las pruebas.
  - Cambios infrecuentes.
  - Cambios controlados.
  - El SW se recupera bien de lo fallos.

Caraterísticas que llevan a un SW. Fácil de probar.

- FACULIDAD DE COMPRENSIÓN: Cuanto más información tengamos, más inteligentes serán las pruebas.
  - Entender perfectamente el diseño.
  - Se ha entendido la dependencia entre os componentes.
  - Se conocen los cambios en el diseño.
  - La documentación es instantáneamente accesible, está bien organizada, es específica, detallada y exacta.

ATRIBUTOS DE UNA BUENA PRUEBA

- Tiene una alta probabilidad de encontrar un error.
- No debe ser redundante.
- Debería ser “la mejor de la cosecha”
- No debe ser ni demasiado sencilla ni demasiado compleja.

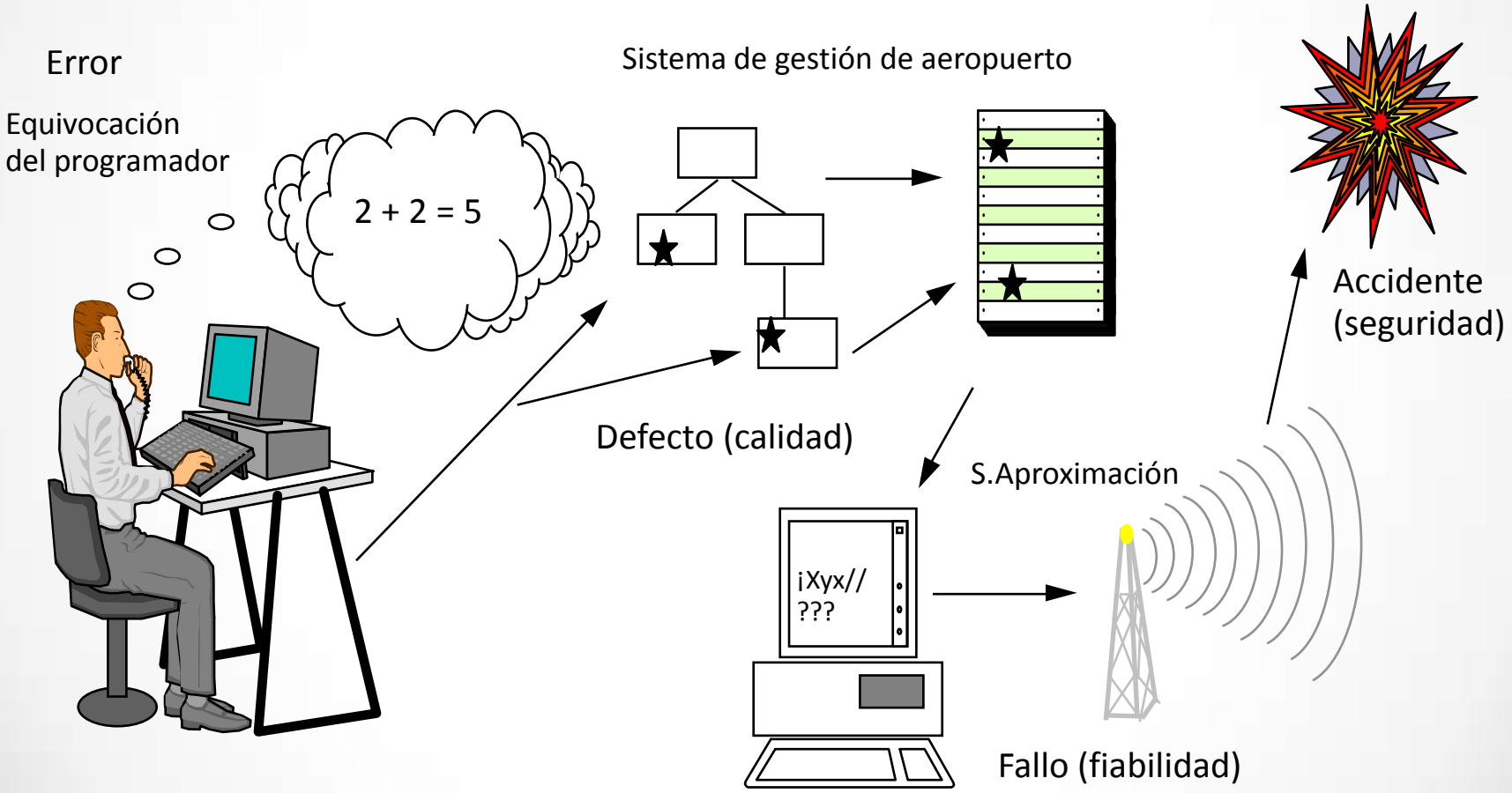
INTRODUCCIÓN

- **Pruebas**: factor crítico para determinar la calidad del software
- La Prueba de Software puede definirse como una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y registran, y se realiza una evaluación de algún aspecto: corrección, robustez, eficiencia, etc.
- El objetivo de una prueba es **descubrir algún error**.
- **Caso de prueba** (test case): «un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular»
- Un caso de prueba es bueno cuando su ejecución conlleva una probabilidad elevada de encontrar un error.
- El éxito de la prueba se mide en función de la capacidad de detectar un error que estaba oculto.

DEFINICIONES

- **Verificación**: ¿estamos construyendo correctamente el producto? ¿el software se ha construido de acuerdo a las especificaciones?
- **Validación:** ¿estamos construyendo el producto correcto? ¿el software hace lo que el usuario realmente requiere?
- **Defecto (bug):** una anomalía en el software como, por ejemplo, un proceso, una definición de datos o un paso de procesamiento incorrectos en un programa
- **Fallo (failure):** cuando el sistema, o alguno de sus componentes, es incapaz de realizar las funciones requeridas dentro de los rendimientos especificados
- **Error (error):** acción humana que conduce a un resultado incorrecto (error de programación)

RELACIÓN ENTRE ERROR, DEFECTO Y FALLO



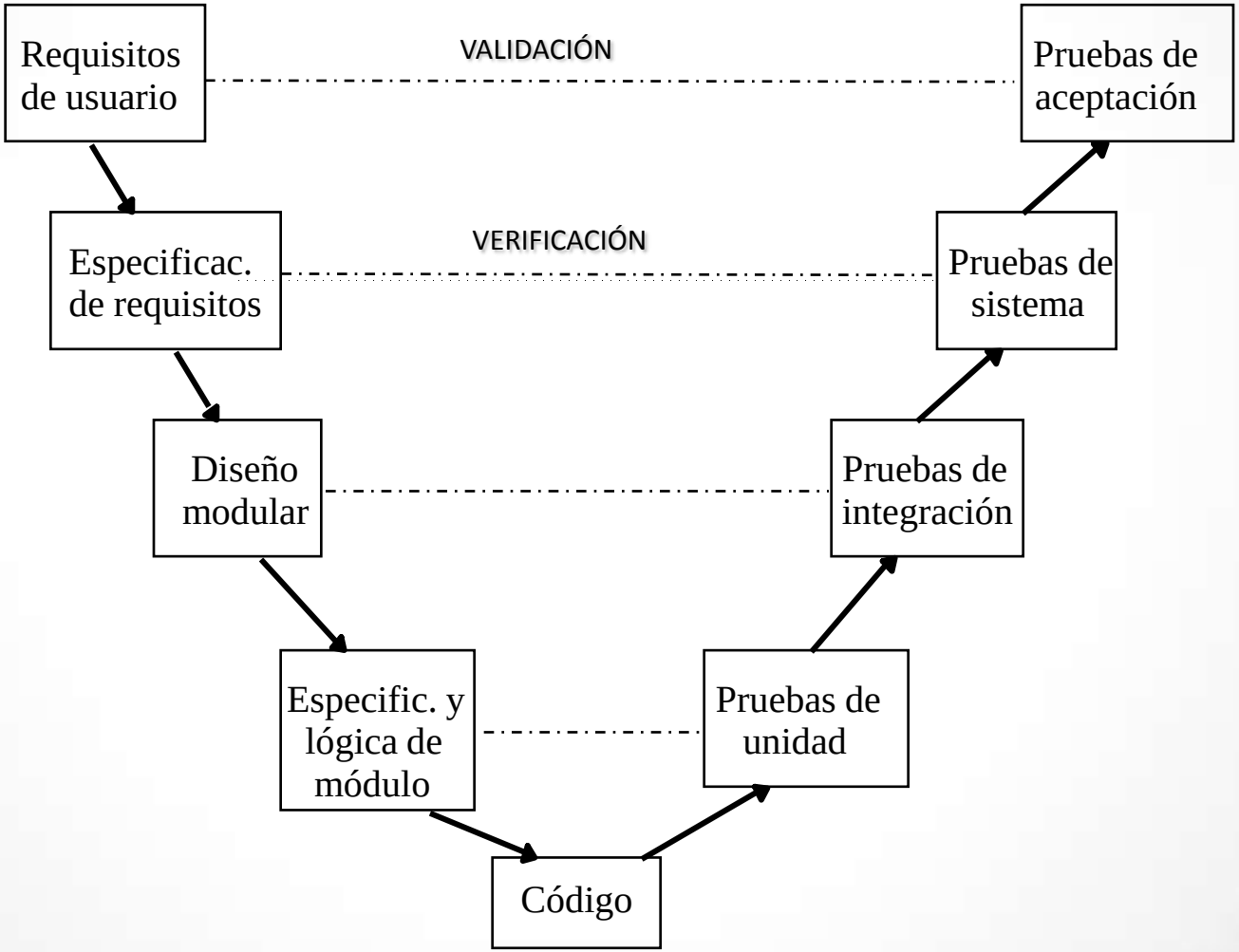
Recomendaciones para la prueba

- Cada caso de prueba debe definir el resultado de salida esperado.
- El programador debe evitar probar sus propios programas, ya que desea (consciente o inconscientemente) demostrar que funcionan sin problemas.
- Se debe inspeccionar a conciencia el resultado de cada prueba, y así, poder descubrir posibles síntomas de defectos.
- Al generar casos de prueba, se deben incluir tanto datos de entrada válidos y esperados como no válidos e inesperados.

Recomendaciones para la prueba

- Las pruebas deben centrarse en dos objetivos (es habitual olvidar el segundo):
  - Probar que el programa hace lo que debe hacer.
  - Probar que el programa no hace lo que no debe hacer.
- Todos los casos de prueba deben documentarse y diseñarse con cuidado.
- No deben hacerse planes de prueba suponiendo que, prácticamente, no hay defectos en los programas y, por lo tanto, dedicando pocos recursos a las pruebas.
- La experiencia indica que donde hay un defecto pueden aparecer otros.

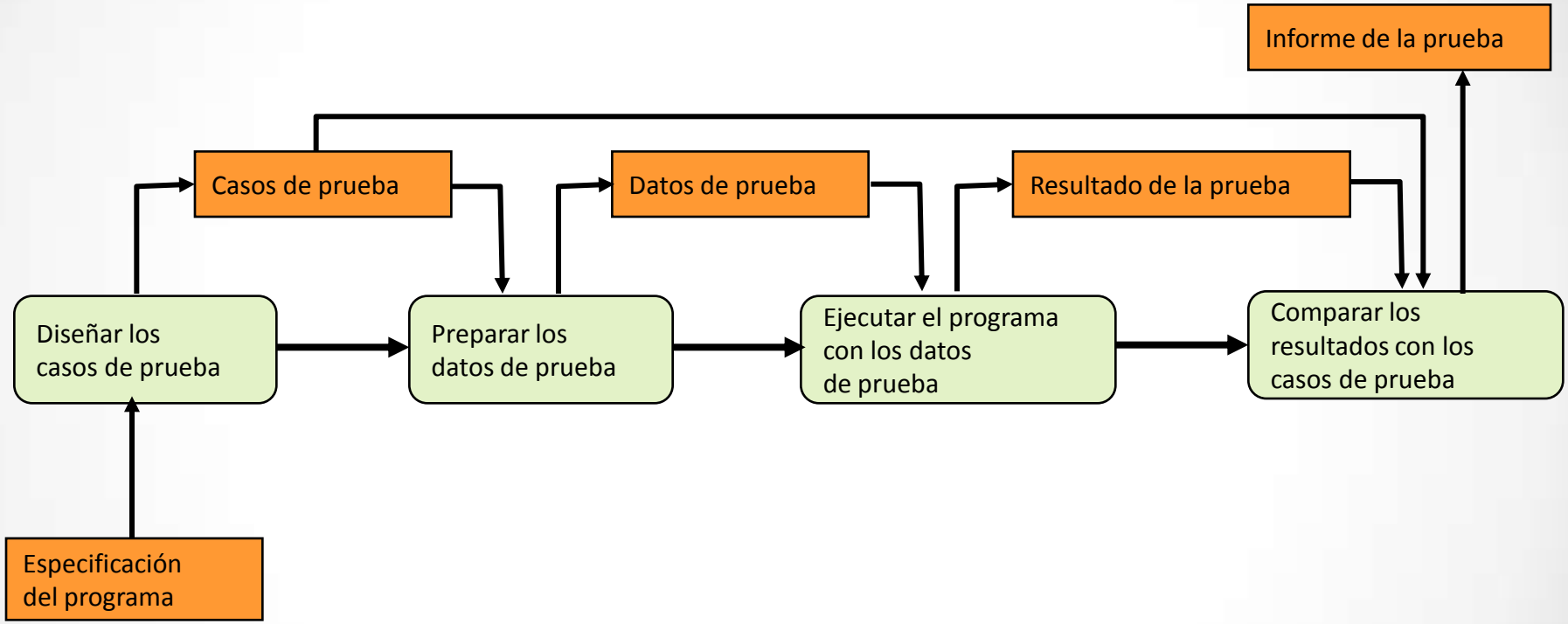
El proceso de prueba



El proceso de prueba

- **Pruebas de unidad**: se prueba cada módulo individualmente.
- **Prueba funcional o de integración**: el software totalmente ensamblado se prueba como un conjunto, para comprobar si cumple o no tanto los requisitos funcionales como los requisitos de rendimiento, seguridad, etc.
- **Prueba del sistema**: el software ya validado se integra con el resto del sistema (por ejemplo, elementos mecánicos, interfaces electrónicas, etc.) para probar su funcionamiento conjunto .
- **Prueba de aceptación**: el producto final se comprueba por el usuario final en su propio entorno de explotación para determinar si lo acepta como está o no.

Flujo de Información de la Prueba



Proceso de depuración

- En cuanto a la localización del error en el código fuente:
  - Analizar la información y pensar.
  - Al llegar a un punto muerto, pasar a otra cosa.
  - Al llegar a un punto muerto, describir el problema a otra persona.
  - Usar herramientas de depuración sólo como recurso secundario.
  - No experimentar cambiando el programa.
  - Se deben atacar los errores individualmente.
  - Se debe fijar la atención también en los datos.

Proceso de depuración

- En cuanto a la corrección del error:
  - Donde hay un defecto, suele haber más.
  - Debe fijarse el defecto, no sus síntomas.
  - La probabilidad de corregir perfectamente un defecto no es del 100%.
  - Cuidado con crear nuevos defectos.
  - La corrección debe situarnos temporalmente en la fase de diseño.

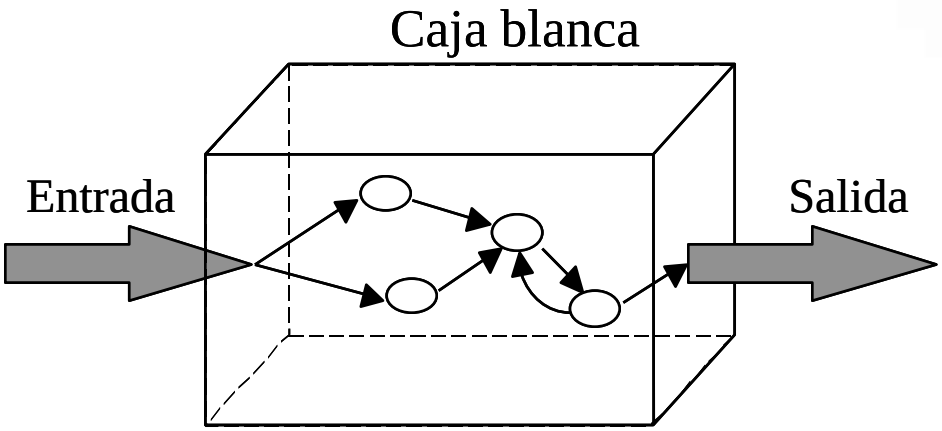
Diseño de casos de Prueba

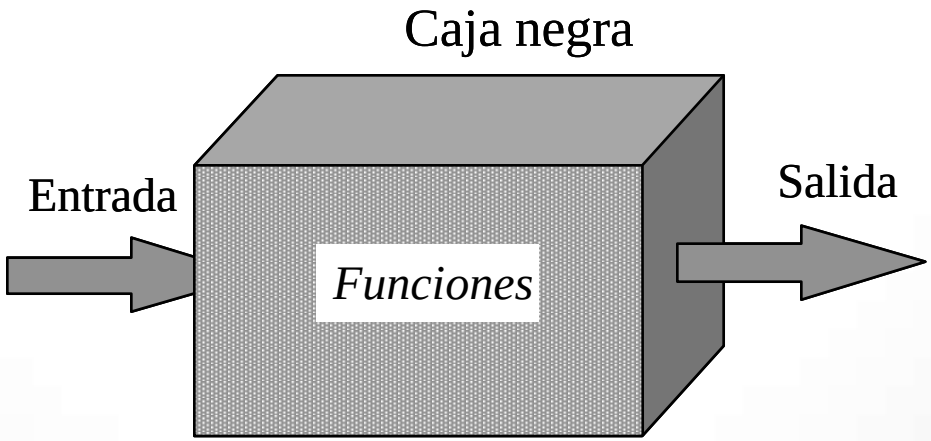
- El diseño de casos de prueba para la verificación del software puede significar un esfuerzo considerable (cerca del 40% del tiempo total de desarrollo)

- Existen fundamentalmente tres enfoques:
  - Prueba de caja blanca
  - Prueba de caja negra
  - Pruebas aleatorias

- Combinar ambos enfoques permite lograr mayor fiabilidad.

Diseño de casos de Prueba





Diseño de casos de Prueba

- La prueba de caja blanca o prueba estructural se basa en el estudio minucioso de toda la operatividad de una parte del sistema, considerando los detalles procedimentales.
  - Se plantean distintos caminos de ejecución alternativos y se llevan a cabo para observar los resultados y contrastarlos con lo esperado.
  - En principio se podría pensar que es viable la verificación mediante la prueba de la caja blanca de la totalidad de caminos de un procedimiento.
  - Esto es prácticamente imposible en la mayoría de los casos reales dada la exponencialidad en el número de combinaciones posibles.

Diseño de casos de Prueba

- La prueba de caja negra o prueba funcional principalmente analiza la compatibilidad en cuanto a las interfaces de cada uno de los componentes software entre sí.
- La prueba aleatoria consiste en utilizar modelos que representen las posibles entradas del programa para crear a partir de ellos los casos de prueba.
  - Modelos estadísticos que simulan las secuencias y frecuencias habituales de los datos de entrada al programa.
  - Se fundamenta en que la probabilidad de descubrir un error es prácticamente la misma si se hacen una serie de pruebas aleatoriamente elegidas, que si se hacen siguiendo las instrucciones dictadas por criterios de cobertura.
  - Sin embargo, pueden permanecer ocultos errores que solamente se descubren ante entradas muy concretas.
  - Este tipo de pruebas puede ser suficiente en programas poco críticos

Pruebas de caja blanca

- La prueba de la caja blanca usa la estructura de control del diseño procedural para derivar los casos de prueba.

- Idea: no es posible probar todos los caminos de ejecución distintos, pero sí confeccionar casos de prueba que garanticen que se verifican todos los caminos de ejecución llamados independientes.

- Verificaciones para cada camino independiente:
  - Probar sus dos facetas desde el punto de vista lógico, es decir, verdadera y falsa.
  - Ejecutar todos los bucles en sus límites operacionales
  - Ejercitar las estructuras internas de datos.

Pruebas de caja blanca

- Consideraciones:
  - Los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa.
  - A menudo creemos que un camino lógico tiene pocas posibilidades de ejecutarse cuando, de hecho, se puede ejecutar de forma regular.
  - Los errores tipográficos son aleatorios.
  - Tal como apuntó Beizer, “los errores se esconden en los rincones y se aglomeran en los límites”.

Tipos de cobertura

- Cobertura de sentencias. cada sentencia o instrucción del programa se ejecute al menos una vez.
- Cobertura de decisiones. cada decisión tenga, por lo menos una vez, un resultado verdadero y, al menos una vez, uno falso.
- Cobertura de condiciones. cada condición de cada decisión adopte el valor verdadero al menos una vez y el falso al menos una vez
- Criterio de decisión/condición. Consiste en exigir el criterio de cobertura de condiciones obligando a que se cumpla también el criterio de decisiones
- Criterio de condición múltiple. Debe garantizar todas las combinaciones posibles de las condiciones dentro de cada decisión.

Tipos de cobertura

EJEMPLO 1:

```
si (a>b)
  entonces a=a-b
  si no b=b-a
fin_si
```

EJEMPLO 2:

```
si (a>b)
  entonces a=a-b
fin_si
```

EJEMPLO 3:

```
i=1
mientras (v[i]<>b)
y(i<>5)
hacer
   i=i+1;
fin_mientras
```

EJEMPLO 4:

```
si (a>b) y (b es
primo)
  entonces a=a-b
fin_si
```

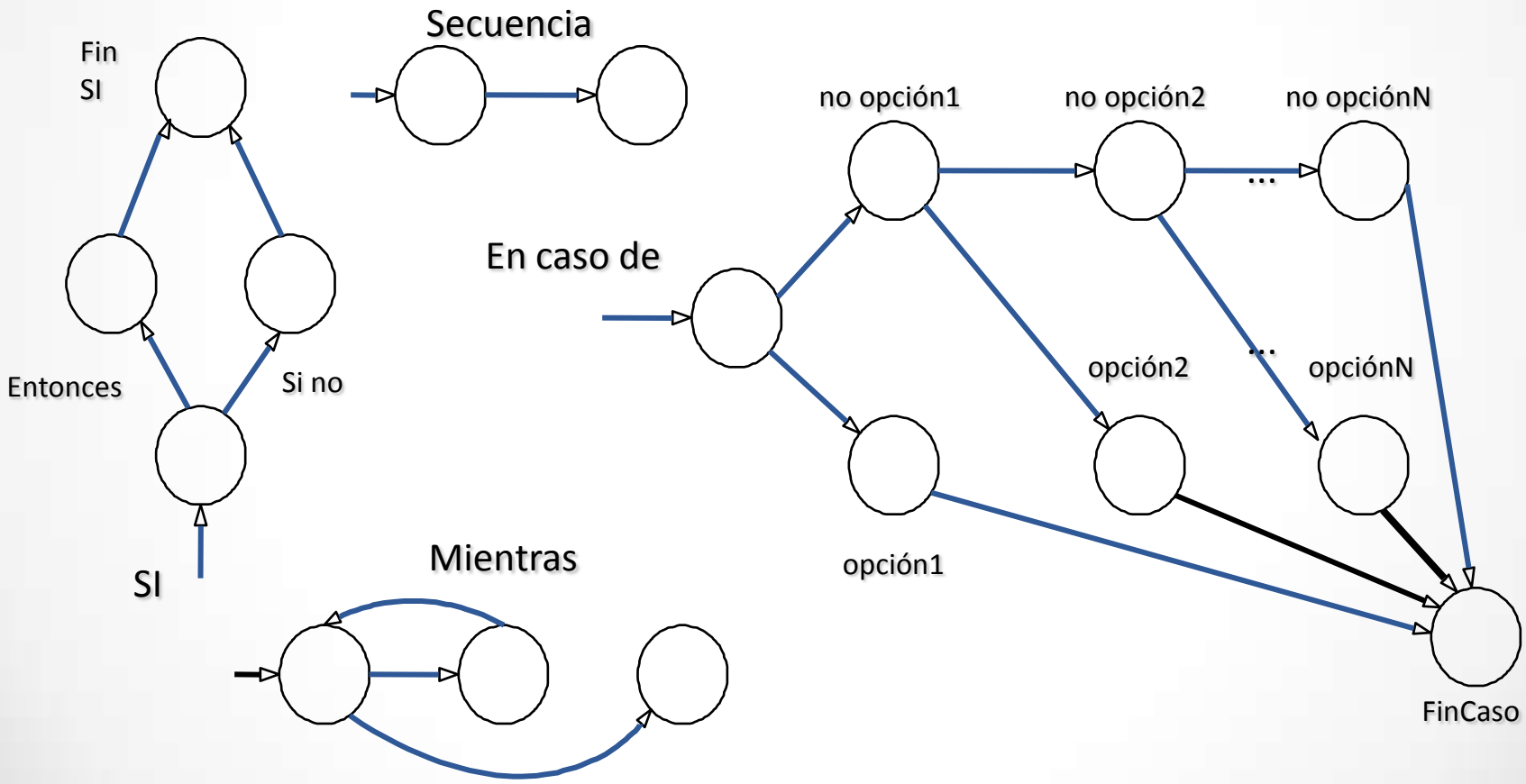
Prueba del camino básico

- La prueba del camino básico es una técnica de prueba de la caja blanca propuesta por Tom McCabe.
- La idea es derivar casos de prueba a partir de un conjunto dado de caminos independientes por los cuales puede circular el flujo de control.
- Camino independiente es aquel que introduce por lo menos una sentencia de procesamiento (o condición) que no estaba considerada en el conjunto de caminos independientes calculados hasta ese momento.
- Para obtener el conjunto un conjunto de caminos independientes se construirá el Grafo de Flujo asociado y se calculará su Complejidad Ciclomática.

Prueba del camino básico: Grafos de Flujo

- El flujo de control de un programa puede representarse por un grafo de flujo.
- Cada nodo del grafo corresponde a una o más sentencias de código fuente
- Cada nodo que representa una condición se denomina nodo predicado.
- Cualquier representación del diseño procedural se puede traducir a un grafo de flujo.
- Un camino independiente en el grafo es el que incluye alguna arista nueva, es decir, que no estaba presente en los caminos definidos previamente.

Prueba del camino básico: Grafos de Flujo



Prueba del camino básico: Grafos de Flujo

- Si se diseñan casos de prueba que cubran los caminos básicos, se garantiza la ejecución de cada sentencia al menos una vez y la prueba de cada condición en sus dos posibilidades lógicas (verdadera y falsa).

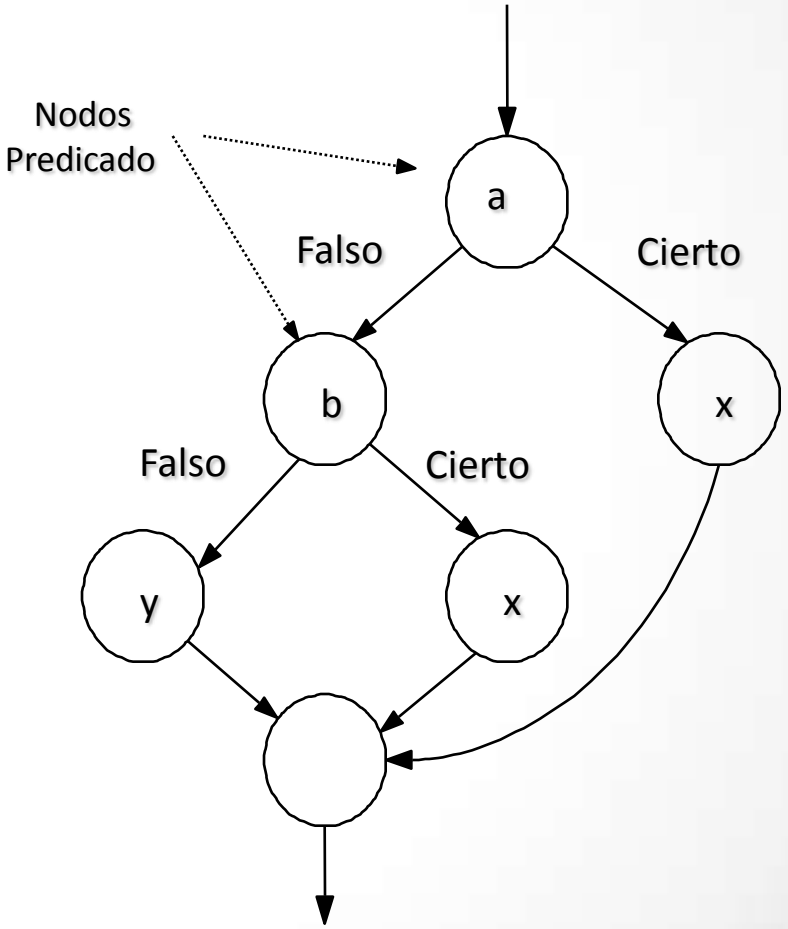
- El conjunto básico para un grafo dado puede no ser único, depende del orden en que se van definiendo los caminos.

- Cuando aparecen condiciones lógicas compuestas la confección del grafo es más compleja.

Prueba del camino básico: Grafos de Flujo

- Ejemplo:

SI a O b
Entonces
hacer x
Si No
hacer y
FinSI

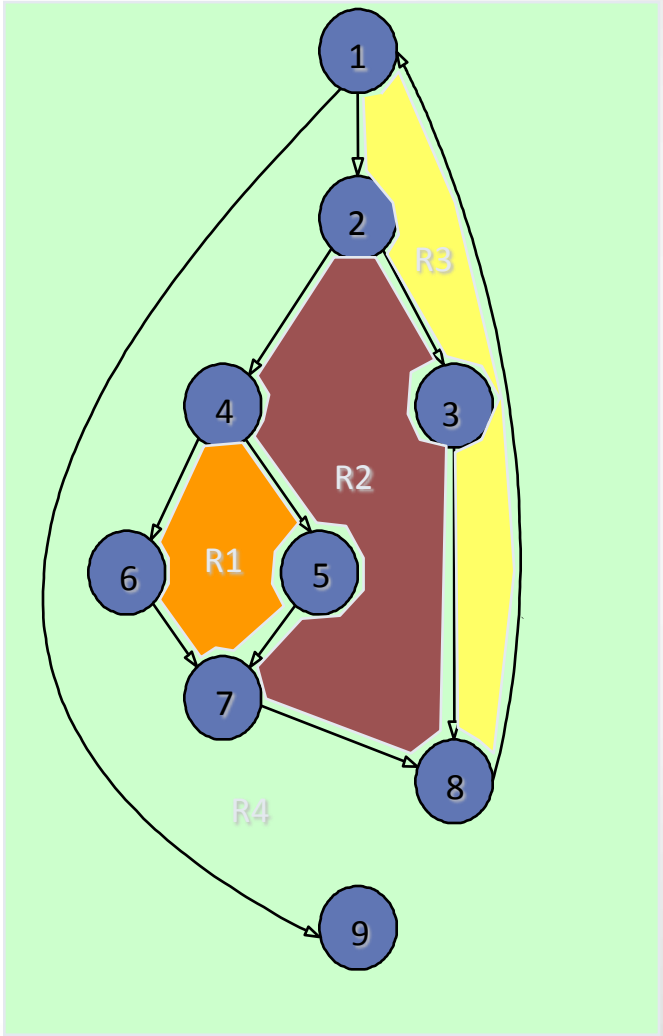


Prueba del camino básico: Complejidad Ciclomática

- Complejidad ciclomática de un grafo de flujo, V(G), indica el número máximo de caminos independientes en el grafo.
- Puede calcularse de tres formas alternativas:
  - **El número de regiones** en que el grafo de flujo divide el plano.
  - **V(G) = A - N + 2**, donde A es el número de aristas y N es el número de nodos.
  - **V(G) = P + 1,** donde P es el número de nodos predicado.

Prueba del camino básico: Complejidad Ciclomática

- V(G) = 4
  - El grafo de la figura delimita cuatro regiones.
  - 11 aristas - 9 nodos + 2 = 4
  - 3 nodos predicado + 1 = 4



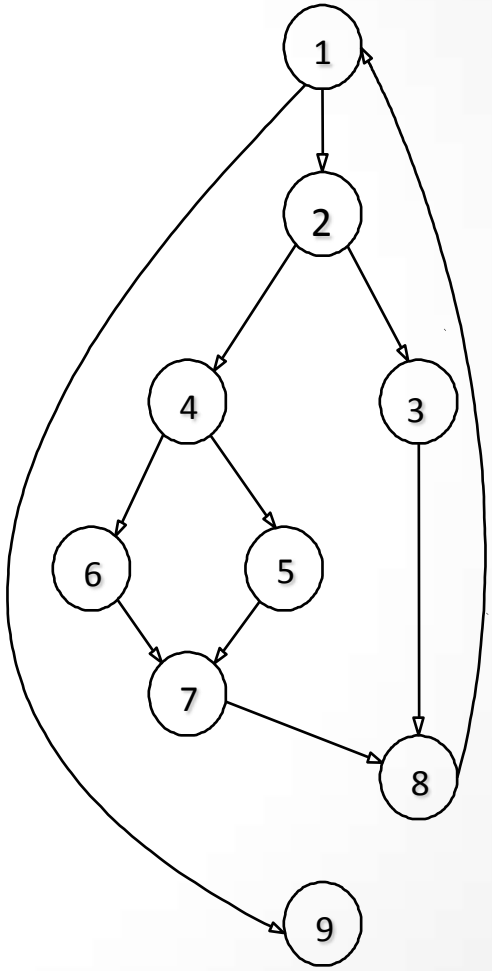
Prueba del camino básico: Complejidad Ciclomática

- El conjunto de caminos independientes del grafo será 4.
  - Camino 1: 1-9
  - Camino 2: 1-2-4-5-7-8-1-9
  - Camino 3: 1-2-4-6-7-8-1-9
  - Camino 4: 1-2-3-8-1-9

- Cualquier otro camino no será un camino independiente, p.e.,

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

  ya que es simplemente una combinación de caminos ya especificados
- Los cuatro caminos anteriores constituyen un conjunto básico para el grafo



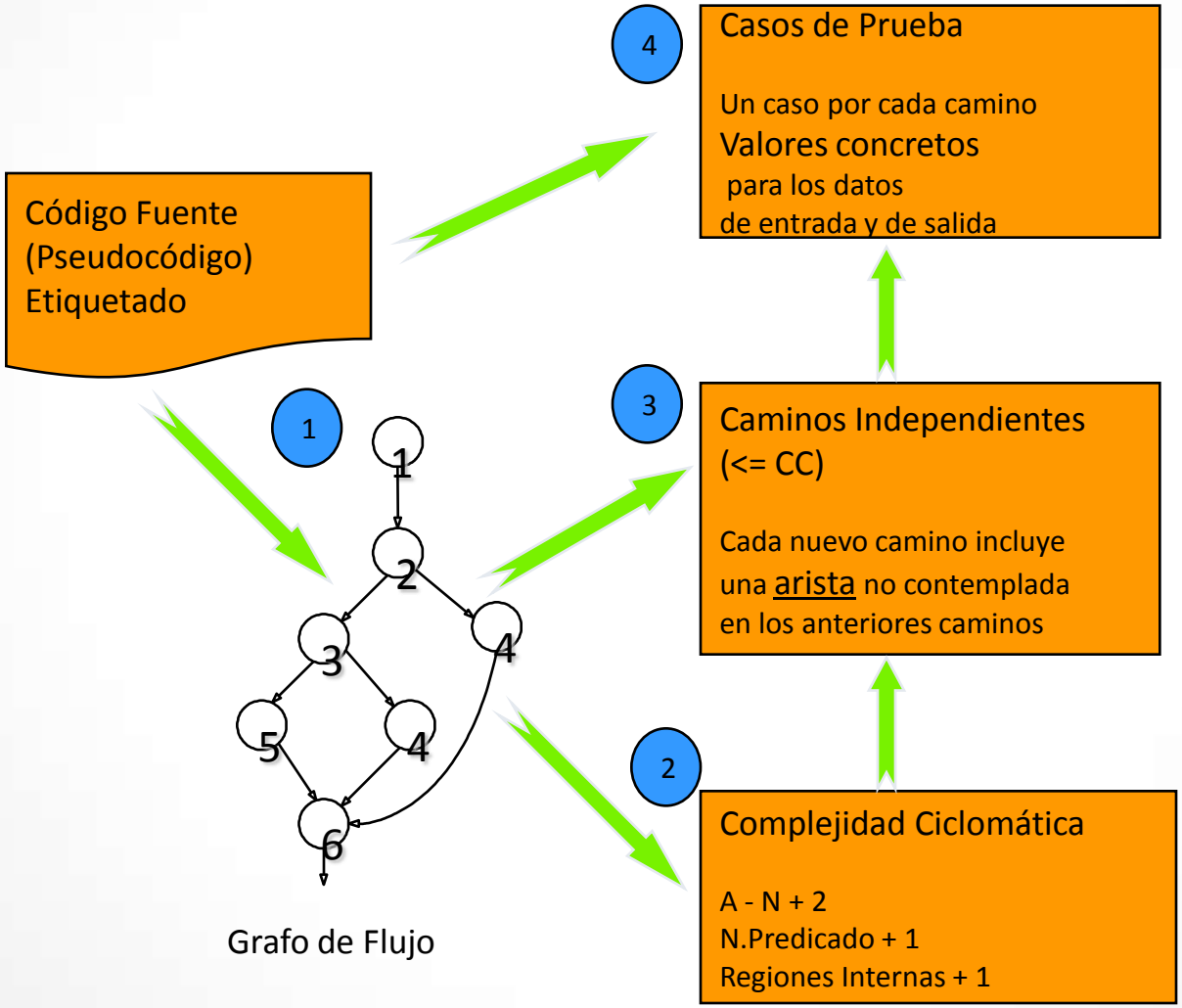
Prueba del camino básico: Derivación de casos de prueba

- El método de prueba del camino básico se puede aplicar a un diseño procedural detallado (pseudocódigo) o al código fuente de la aplicación.

- Pasos para diseñar los casos de prueba:

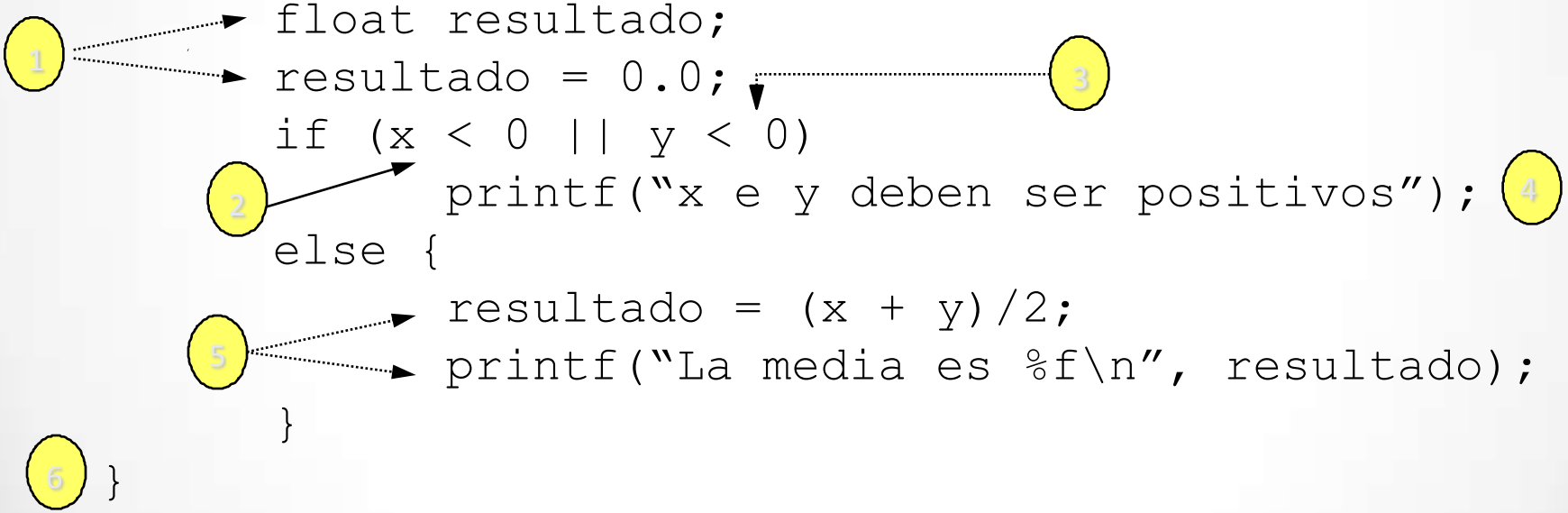
1. Se etiqueta el código fuente, o el pseudocódigo, enumerando cada instrucción y cada condición simple.
2. A partir del código fuente etiquetado, se dibuja el grafo de flujo asociado.
3. Se calcula la complejidad ciclomática del grafo.
4. Se determina un conjunto básico de caminos independientes.
5. Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

Prueba del camino básico: Derivación de casos de prueba

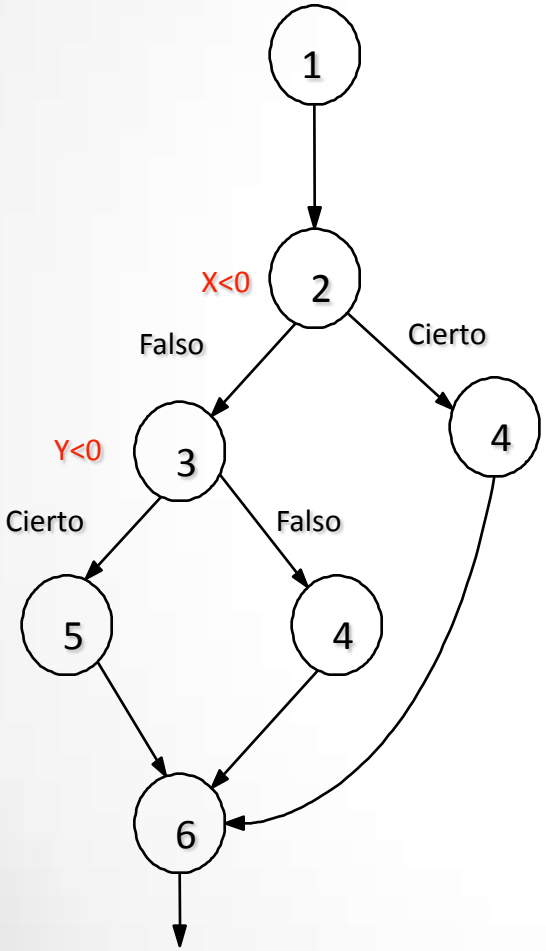


Prueba del camino básico: Derivación de casos de prueba

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



Prueba del camino básico: Ejemplo



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

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

Casos de prueba para cada camino:

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

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

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

Pruebas de estructura de control

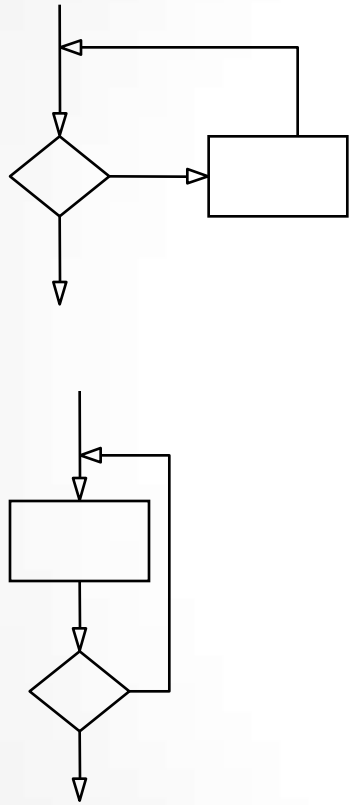
- La técnica de prueba del camino básico del punto anterior es una de las muchas existentes para la pruebas de la estructura de control.
- Aunque sea alta la efectividad de la prueba del camino básico, no nos asegura completamente la corrección del software.
- Veremos a continuación otras variantes de la prueba de la estructura de control. Estas variantes amplían el abanico de pruebas mejorando la fiabilidad de las pruebas de caja blanca.

Pruebas de estructura de control - Prueba de Condiciones

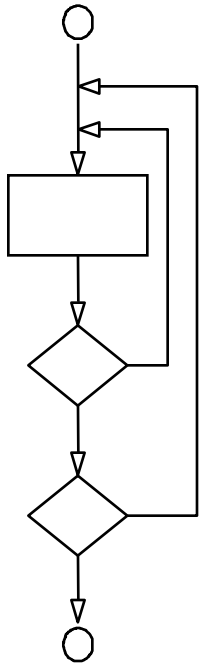
- La prueba de condiciones es un método de diseño de casos de prueba que ejercita las condiciones lógicas contenidas en el módulo de un programa. Los tipos de errores que pueden aparecer en una condición son los siguientes:

  - Existe un error en un operador lógico (que sobra, falta o no es el que corresponde).
  - Existe un error en un paréntesis lógico (cambiando el significado de los operadores involucrados).
  - Existe un error en un operador relacional (operadores de comparación de igualdad, menor o igual, etc.).
  - Existe un error en una expresión aritmética.

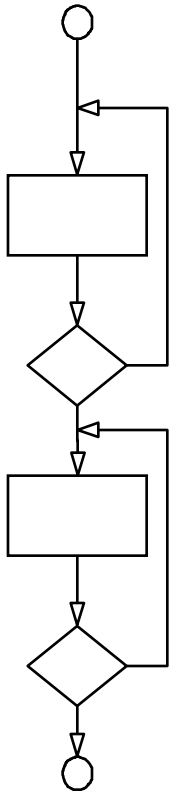
Pruebas de estructura de control - Prueba de Bucles



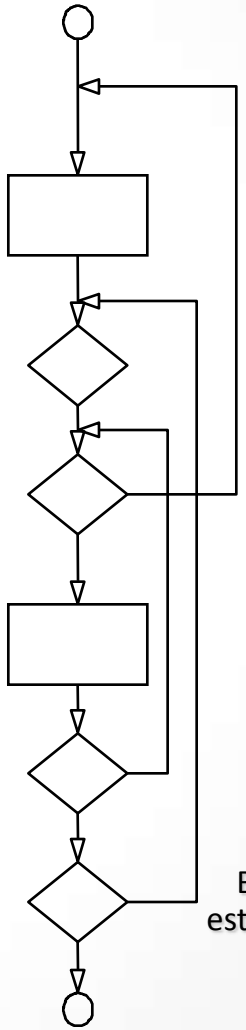
Bucles simples



Bucles anidados



Bucles concatenados



Bucles no estructurados

Pruebas de estructura de control - Prueba de Condiciones

- Pruebas para Bucles simples. (n es el número máximo de iteraciones permitidos por el bucle)
  - Pasar por alto totalmente el bucle
  - Pasar una sola vez por el bucle
  - Pasar dos veces por el bucle
  - Hacer m pasos por el bucle con m < n
  - Hacer n-1, n y n + 1 pasos por el bucle

Pruebas de estructura de control - Prueba de Condiciones

- Pruebas para Bucles anidados
  - Comenzar en el bucle más interior estableciendo los demás bucles en sus valores mínimos.
  - Realizar las pruebas de bucle simple para el más interior manteniendo los demás en sus valores mínimos.
  - Avanzar hacia fuera confeccionando pruebas para el siguiente bucle manteniendo todos los externos en los valores mínimos y los demás bucles anidados en sus valores típicos.
  - Continuar el proceso hasta haber comprobado todos los bucles.
- Pruebas para Bucles concatenados

  - Siempre que los bucles concatenados sean independientes se puede aplicar lo relativo a bucles simples/anidados. En caso de ser dependientes se evaluarán como bucles anidados.

Pruebas de caja negra

- Los métodos de la caja negra se centran sobre los requisitos funcionales del software.

- La prueba de la caja negra intenta encontrar errores de los siguientes tipos:
  - Funciones incorrectas o inexistentes.
  - Errores relativos a las interfaces.
  - Errores en estructuras de datos o en accesos a bases de datos externas.
  - Errores debidos al rendimiento.
  - Errores de inicialización o terminación.

Partición equivalente

- La partición equivalente es un método que divide el campo de entrada de un programa en clases de datos a partir de los cuales se pueden derivar casos de prueba.

- La partición equivalente define casos de prueba para descubrir clases de errores.

- Se define una condición de entrada para cada dato de entrada (valor numérico específico, rango de valores, conjunto de valores relacionados o condición lógica).

Partición equivalente

- Las clases de equivalencia se pueden definir de acuerdo a las siguientes directrices:

  - Si una condición de entrada especifica un rango, se define una clase de equivalencia válida y dos no válidas.
  - Si la condición de entrada es un valor específico, se define una clase de equivalencia válida y dos no válidas.
  - Si la condición de entrada especifica un miembro de un conjunto, se define una clase de equivalencia válida y otra no válida.
  - Si la condición de entrada es lógica, se define una clase válida y otra no válida.

Ejemplo Partición equivalente

- Datos de entrada a una aplicación bancaria
  - Código de área: en blanco o número de 3 dígitos.
  - Prefijo: número de 3 dígitos que no comience por 0 o 1.
  - Sufijo: número de 4 dígitos.
  - Contraseña: en blanco o valor alfanumérico de 6 caracteres.
  - Ordenes: “comprobar”, “depositar”, “pagar factura”, etc.

Ejemplo Partición equivalente

- Condiciones de entrada
  - ❖ Código de área: número de 3 dígitos o no es número
    - ✓ lógica: el código puede ser un número o no.
    - ✓ valor: número de 3 dígitos.
  - ❖ Prefijo: número de 3 dígitos que no comience por 0 o 1.
    - ✓ rango: valores entre 200 y 999.
  - ❖ Sufijo: número de 4 dígitos.
    - ✓ valor: número de 4 dígitos.
  - ❖ Contraseña: en blanco o valor alfanumérico de 6 caracteres.
    - ✓ lógica: la contraseña puede estar presente o no.
    - ✓ valor: cadena de 6 caracteres.
  - ❖ Ordenes: “comprobar”, “depositar”, “pagar factura”, etc.
    - ✓ conjunto: el de todas las ordenes posibles.

Esquema prueba de la partición equivalente

1

Especificar los
datos de entrada:

dato: tipo o descripción

2

Determinar las clases equivalencia

Rango: una válida y dos no válidas.
Valor específico: una válida y dos no válidas.
Conjunto:
- Si cada elmenento se trata igual,
  una válida y otra no válida.
- Si cada elemento se trata diferente,
  una válida por cada elemento y
  otra no válida

Lógica: una válida y otra no válida.

3

DATO ENTRADA	CLASES VÁLIDAS	CLASES NO VÁLIDAS
a	(1)	(2)
b	(3)	(4) (5)
. . .	. . .	. . .

TABLA de CLASES de EQUIVALENCIA

ENUMERADAS

4

Casos de Prueba Válidos:
que cubran clases válidas

Casos de Prueba No Válidos:
que cubran una clase no válida
y el resto válidas

Ejemplo Partición equivalente

- Aplicación bancaria. Datos de entrada:

  - **Código de área:** número de 3 dígitos que no empieza por 0 ni por 1
  - **Nombre de identificación de operación**: 6 caracteres
  - **Órdenes posibles**: “cheque”, “depósito”, “pago factura”, “retirada de fondos”

Ejemplo Partición Equivalente
Determinar las clases de equivalencia

Código de área:

Lógica:

1 clase válida: número

Rango:

1 clase válida: 200 < código < 999

2 clases no válidas: código < 200; código > 999

1 clase no válida: no es número

Nombre de identificación:

Valor específico:

1 clase válida: 6 caracteres

2 clases no válidas: más de 6 caracteres; menos de 6 caracteres

Órdenes posibles:

Conjunto de valores:

1 clase válida: 4 órdenes válidas

1 clase no válida: órden no válida

Ejemplo Partición Equivalente : Tabla de clases de equivalencia

Datos de Entrada	Clases Válidas	Clases No Válidas
Código de área	(1) 200 <= código <= 999	(2) código < 200 (3) código > 999 (4) no es número
Identificación	(5) 6 caracteres	(6) menos de 6 caracteres (7) más de 6 caracteres
Orden	(8) “cheque” (9) “depósito” (10) “pago factura” (11) “retirada de fondos”	(12) ninguna orden válida

Ejemplo Partición Equivalente : Casos de Prueba Válidos

Código	Identificación	Orden	Clases Cubiertas
300	Nómina	“Depósito”	(1)^{C} (5)^{C} (9)^{C}
400	Viajes	“Cheque”	(1) (5) (8)^{C}
500	Coches	“Pago-factura”	(1) (5) (10)^{C}
600	Comida	“Retirada-fondos”	(1) (5) (11)^{C}

Ejemplo Partición Equivalente: Casos de Prueba NO Válidos

Código	Identificación	Orden	Clases Cubiertas
180	Viajes	“Pago-factura”	(2) C (5) (10)
1032	Nómina	“Depósito”	(3) C (5) (9)
XY	Compra	“Retirada-fondos”	(4) C (5) (11)
350	A	“Depósito”	(1) (6) C (9)
450	Regalos	“Cheque”	(1) (7) C (8)
550	Casa	&%4	(1) (5) (12) C

Análisis de valores límite

- La técnica de Análisis de Valores Límites selecciona casos de prueba que ejerciten los valores límite dada la tendencia de la aglomeración de errores en los extremos.

- Complementa la de la partición equivalente. En lugar de realizar la prueba con cualquier elemento de la partición equivalente, se escogen los valores en los bordes de la clase.

- Se derivan tanto casos de prueba a partir de las condiciones de entrada como con las de salida.

Análisis de valores límite

- Directrices:
  - Si una condición de entrada especifica un rango delimitado por los valores a y b, se deben diseñar casos de prueba para los valores a y b y para los valores justo por debajo y justo por encima de ambos.
  - Si la condición de entrada especifica un número de valores, se deben desarrollar casos de prueba que ejerciten los valores máximo y mínimo además de los valores justo encima y debajo de aquéllos.

Análisis de valores límite

- ... Directrices:
  - Aplicar las directrices anteriores a las condiciones de salida. Componer casos de prueba que produzcan salidas en sus valores máximo y mínimo.
  - Si las estructuras de datos definidas internamente tienen límites prefijados (p.e., un array de 10 entradas), se debe asegurar diseñar un caso de prueba que ejercite la estructura de datos en sus límites.

Ejemplo Camino Básico: Código Fuente

```
int contar_letra(char cadena[10], char letra)
{
  int contador, n,  lon;
  n=0; contador=0;
  lon = strlen (cadena);
  if (lon > 0) {
                do {
                  if (cadena[contador] == letra) n++;
                  contador++;
                  lon--;
                } while (lon > 0);
  }
  return n;
}
```

Ejemplo Camino Básico: Código Fuente Etiquetado

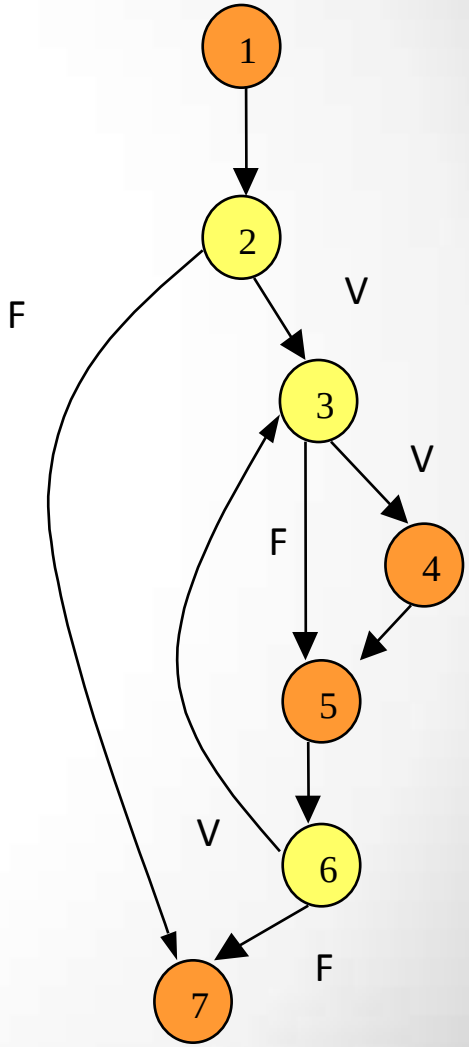
```
int contar_letra(char cadena[10], char letra)
{
  int contador, n,  lon;
  n=0; contador=0;
  lon = strlen (cadena);
  if (lon > 0) {
      do {
        if (cadena[contador] == letra)
              n++;
        contador++;
        lon--;
      } while (lon > 0);
  }
  return n;
}
```

1 2 3 4 5 6 7

Ejemplo Camino Básico: Grafo de Flujo

```
int contar_letra(char cadena[10], char letra)
{
  int contador, n,  lon;
  n=0; contador=0;
  lon = strlen (cadena);
  if (lon > 0) {
     do {
       if (cadena[contador] == letra)
            n++;
       contador++;
       lon--;
     } while (lon > 0);
  }
  return n;
}
```

1 2 3 4 5 6 7



Ejemplo Camino Básico: Caminos Independientes

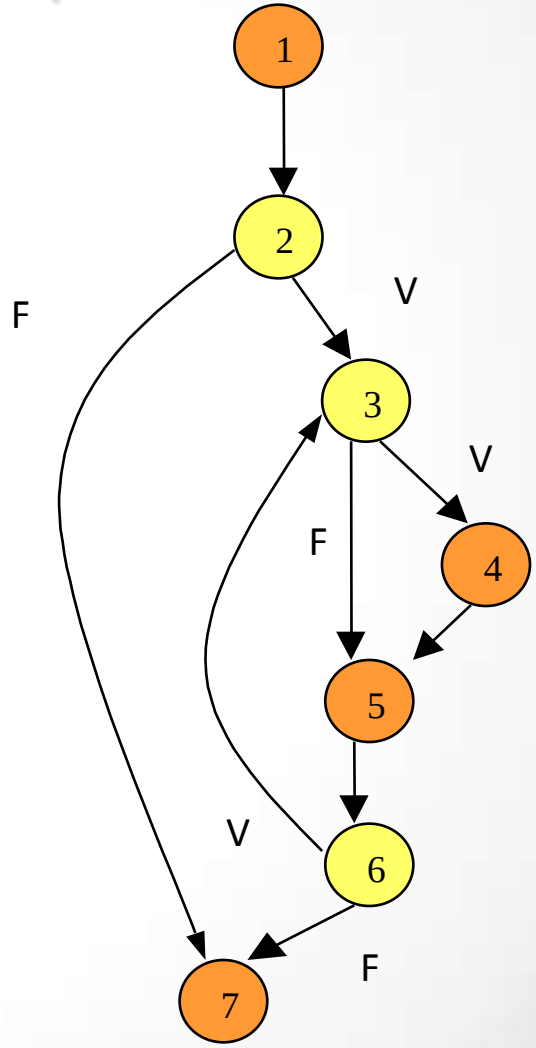
- V(G) = 4;

  Nodos=7; Aristas=9;

  Nodos Predicado=3;

  Regiones = 4

- Conjunto de caminos independientes:
  1. 1-2-7
  2. 1-2-3-4-5-6-7
  3. 1-2-3-5-6-7
  4. 1-2-3-4-5-6-3-5-6-7 (No es el único)



Ejemplo Camino Básico: Casos de Prueba

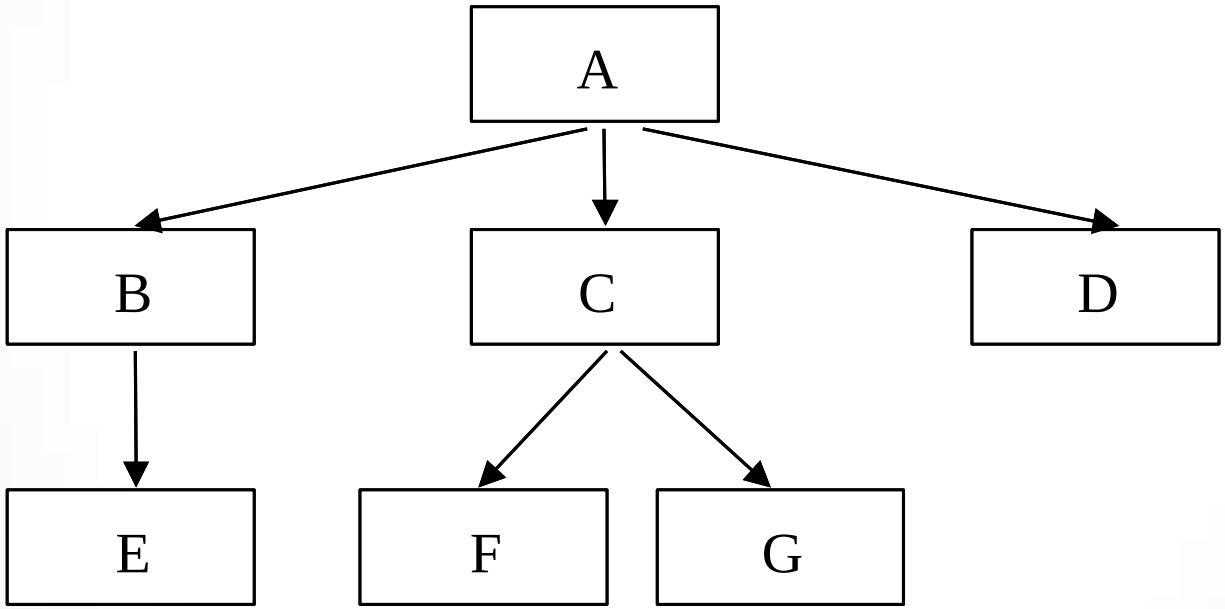
1.	1-2-7	cadena = ””	letra = ‘a’	n = 0;
2.	1-2-3-4-5-6-7	cadena = “a”	letra = ‘a’	n = 1;
3.	1-2-3-5-6-7	cadena = “b”	letra = ‘a’	n = 0;
4.	1-2-3-4-5-6-3-5-6-7	cadena = “ab”	letra = ‘a’	n = 1;

Pruebas de integración

- Se realiza sobre un conjunto de componentes o módulos para formar un sistema o subsistema mayor.
- Mientras las pruebas unitarias las suele llevar a cabo el propio programador, las pruebas de integración suelen realizarse por un equipo independiente.
- Existen dos estrategias para realizar las pruebas de integración
  - Integración descendente (top-down): se comienza con los niveles superiores del sistema y se va integrando hacia niveles inferiores, sustituyendo módulos no probados por módulos ficticios cuando fuera necesario.
  - Integración ascendente (bottom-up): se integran componentes individuales desde los niveles inferiores, y se va ascendiendo hasta completar el sistema.

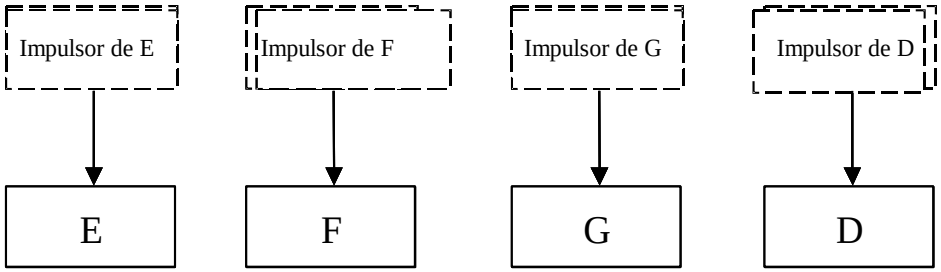
Pruebas de integración

¿ Cómo abordar la prueba de los módulos de la figura ?

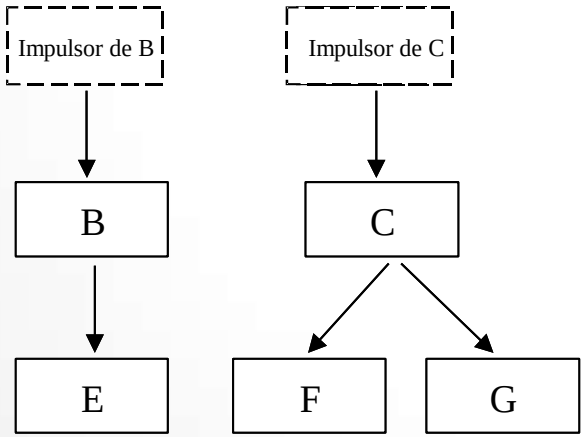


Pruebas de integración: estrategia ascendente

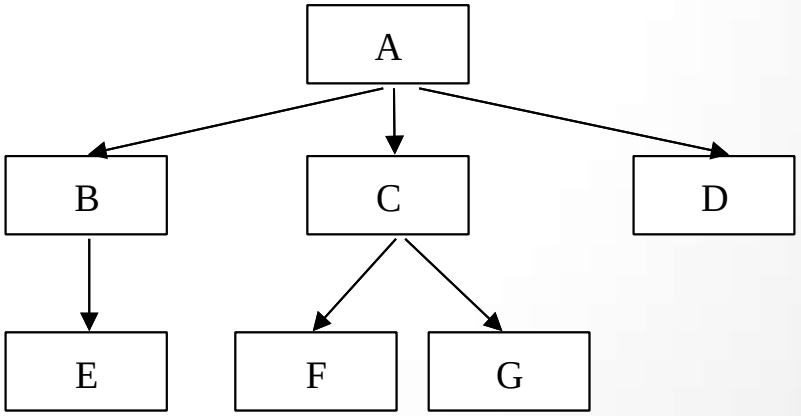
1ª fase



2ª fase



3ª fase



Pruebas de integración: estrategia ascendente

- Ventajas:
  - Se detectan antes los fallos que se produzcan en la parte inferior del sistema.
  - La definición de los casos de prueba es más sencilla, puesto que los módulos inferiores desempeñan funciones más específicas.
  - Es más fácil observar los resultados de la prueba, ya que es en los módulos inferiores donde se elaboran los datos (los módulos superiores suelen ser coordinadores).

- Desventajas:
  - Se requieren módulos impulsores que deben codificarse.
  - El programa o sistema completo no se prueba hasta que se introduce el último módulo.

Pruebas de integración: estrategia descendente

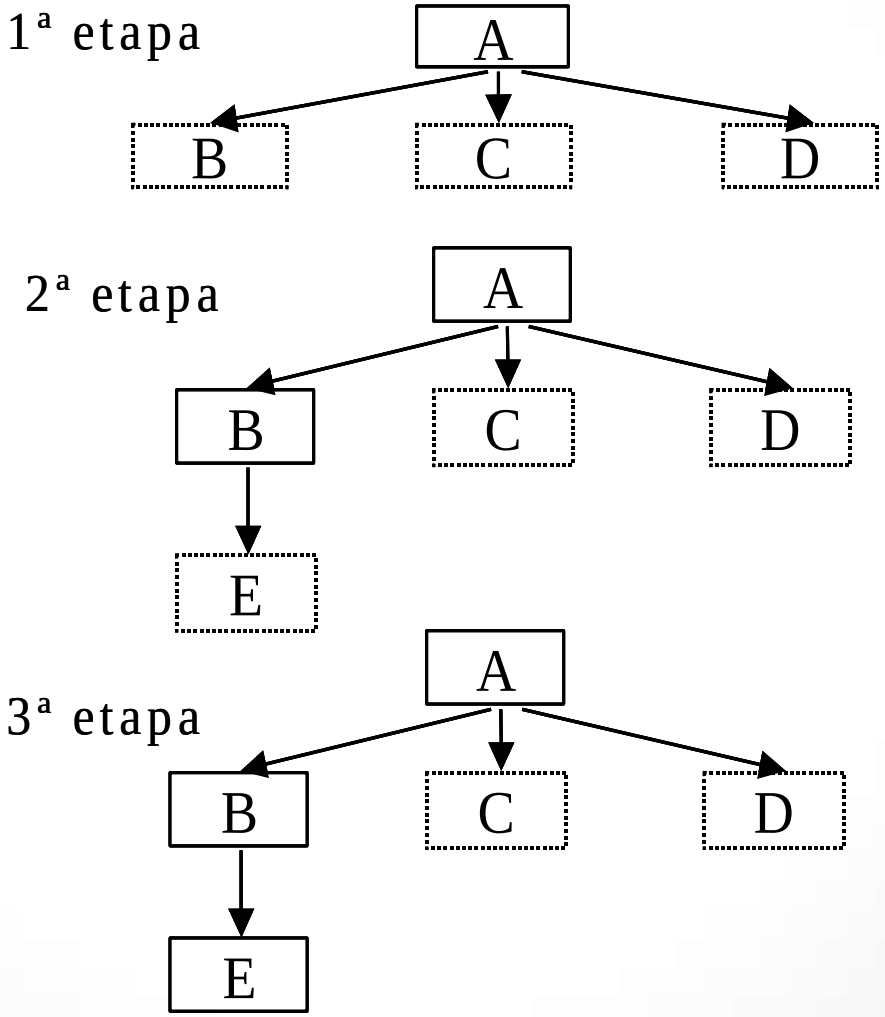
- El módulo raíz o principal se prueba primero. Todos sus subordinados se sustituyen por módulos ficticios.
- Una vez probado el módulo raíz (estando ya libre de defectos) se sustituye uno de sus subordinados ficticios por el módulo implementado correspondiente.
- Cada vez que se incorpora un módulo se efectúan las pruebas correspondientes.
- Al terminar cada prueba, se sustituye un módulo ficticio por su correspondiente real.
- Conviene repetir algunos casos de prueba de ejecuciones anteriores para asegurarse de que no se ha introducido ningún error nuevo.

Pruebas de integración: estrategia descendente

PRIMERO

EN

PROFUNDIDAD

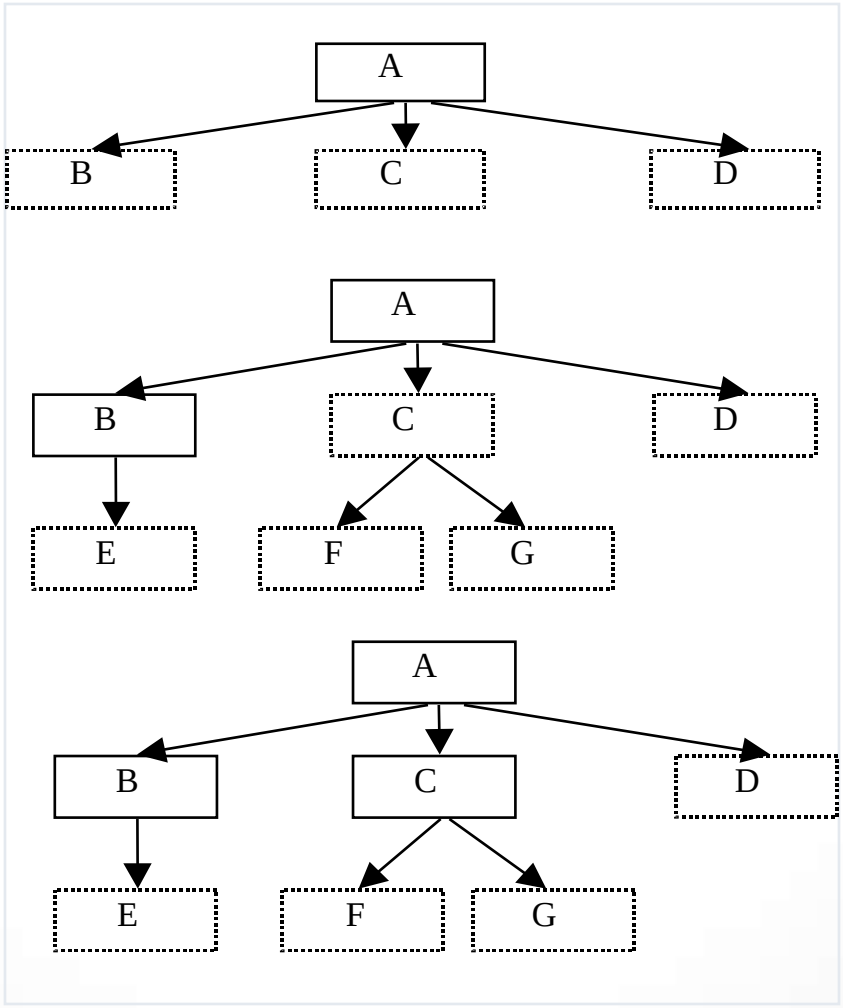


Pruebas de integración: estrategia descendente

PRIMERO

EN

ANCHURA



Pruebas de integración: estrategia descendente

- Ventajas:
  - Los defectos en los niveles superiores del sistema se detectan antes.
  - Una vez incorporadas las funciones que manejan la entrada/salida, es fácil manejar los casos de prueba.
  - Permite ver antes una estructura previa del programa, lo que facilita hacer demostraciones.

- Desventajas:
  - Se requieren módulos ficticios que suelen ser complejos de crear.
  - Antes de incorporar la entrada/salida es complicado manejar los casos de prueba.
  - A veces no se pueden crear casos de prueba, porque los detalles de operación vienen proporcionados por los módulos inferiores.
  - Es más difícil observar la salida, porque los resultados surgen de los módulos inferiores.
  - Pueden inducir a diferir la terminación de la prueba de ciertos módulos, ya que puede parecer que el programa funciona.

Otros tipos de prueba

- Recorridos (“walkthroughs”).
- Pruebas de robustez (“robustness testing”)
- Pruebas de aguante (“stress”)
- Prestaciones (“performance testing”)
- Conformidad u Homologación (“conformance testing”)
- Interoperabilidad (“interoperability tesing”)
- Regresión (“regression testing”)
- Prueba de comparación

Herramientas automáticas de prueba

- **Analizadores estáticos**. Estos sistemas de análisis de programas soportan pruebas de las sentencias que consideran más débiles dentro de un programa.
- **Auditores de código.** Son filtros con el propósito de verificar que el código fuente cumple determinados criterios de calidad (dependerá fuertemente del lenguaje en cuestión).
- **Generadores de archivos de prueba**. Estos programas confeccionan automáticamente ficheros con datos que servirán de entrada a las aplicaciones.
- **Generadores de datos de prueba**. Confeccionan datos de entrada determinados que conlleven un comportamiento concreto del software.
- **Controladores de prueba**. Confeccionan y alimentan los datos de entrada y simulan el comportamiento de otros módulos para restringir el alcance de la prueba.