



Validación y verificación: Pruebas de software

Ingeniería del Software

Índice

- ❑ Conceptos previos
- ❑ Prueba del software: objetivo, principios, proceso, diseño...
- ❑ Ciclo de vida de pruebas: tipos
 - ✓ *Pruebas de unidad*
 - Enfoque de caja blanca
 - Enfoque de caja negra
 - Conjetura de errores
 - Enfoque aleatorio
 - ✓ *Pruebas de integración*
 - Descendentes: pruebas de regresión
 - Ascendentes
 - De humo
 - ✓ *Pruebas de sistema*
 - Pruebas de recuperación
 - Pruebas de seguridad
 - Pruebas de resistencia
 - Pruebas de rendimiento
 - ✓ *Pruebas de validación*
 - Revisión de la configuración
 - Pruebas alfa y beta
- ❑ Documentación de la fase de pruebas
- ❑ Depuración

Conceptos previos

- ❑ “As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications”. [Dave Parnas](#)
- ❑ “The software isn't finished until the last user is dead”. [Anonymous](#)
- ❑ “Poor management can increase software costs more rapidly than any other factor”. [Barry Boehm](#)
- ❑ “The best programmers write only easy programs”. [Michael A. Jackson](#)
- ❑ “The trouble with programmers is that you can never tell what a programmer is doing until it's too late”. [Seymour Cray](#)
- ❑ “Fast, good, cheap: pick any two”. [Anonymous](#)

Conceptos previos

❑ ¿Qué es un **defecto** en el software?

- ✓ *Una definición de datos incorrecta*
- ✓ *Un paso de procesamiento incorrecto en el programa*

❑ ¿Qué es un **fallo** en el software?

- ✓ *Es la incapacidad de un sistema para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados*

❑ ¿Qué es un **error** en el software?

- ✓ *Un defecto (¿?)*
- ✓ *Un resultado incorrecto*
- ✓ *Acción humana que lleva a un resultado incorrecto*
- ✓ *La diferencia entre un valor calculado y el verdadero*

Conceptos previos

Un ***error*** (fault) del programador
introduce un ***defecto*** (defect)
y este defecto produce
un ***fallo*** (*failure*)

Conceptos previos

❑ ¿Qué significa **verificar** el software?

- ✓ *Determinar si los productos de una fase dada satisfacen las condiciones impuestas al inicio de la fase.*
- ✓ *¿Estamos construyendo el producto **correctamente**?*

❑ ¿Qué significa **validar** el software?

- ✓ *Evaluar un sistema o uno de sus componentes, durante o al final de su desarrollo, para determinar si satisface los requisitos.*
- ✓ *¿Estamos construyendo el producto **correcto**?*

Conceptos previos

- ❑ La importancia del software y los **costes** asociados a un fallo motivan la creación de pruebas:
 - ✓ *minuciosas y*
 - ✓ *bien planificadas*

- ❑ Las organizaciones de desarrollo de software emplean entre el **30% y el 40%** del esfuerzo total de un proyecto en las pruebas.

- ❑ Las pruebas presentan una interesante **anomalía** para el ingeniero del software
 - ✓ *Durante las fases anteriores de desarrollo, el ingeniero intenta **construir***
 - ✓ *Cuando llegan las pruebas se crea una serie de casos de prueba que intentan **demoler** el software construido.*

- ❑ Las pruebas requieren:
 - ✓ *Descartar **ideas preconcebidas** y*
 - ✓ *Superar cualquier **conflicto de intereses** que aparezcan cuando se descubran errores*

❑ ¿Qué es un **caso de prueba**?

✓ *Un conjunto de:*

- Entradas
- Condiciones de ejecución
- Resultados esperados

✓ *Este conjunto se llama **escenario de prueba***

Pruebas de software

❑ Características:

1. *La prueba es el proceso de ejecución de un programa **con el fin de descubrir un error**, con poca cantidad de tiempo y esfuerzo*
2. *Un buen caso de prueba es aquel que tiene una **alta probabilidad de encontrar un error** no descubierto hasta entonces*
3. *Una prueba tiene éxito si **descubre un error no detectado** con anterioridad*

❑ Consecuencias:

- ✓ *Hay que desechar la idea de que una prueba tiene éxito si no descubre errores*
- ✓ *Las pruebas pueden usarse como una **indicación de la fiabilidad** del software y, de alguna manera, indican su calidad*
- ✓ *Las pruebas **no garantizan la ausencia de defectos***

Pruebas de software

❑ Principios:

1. *A todas las pruebas se les debería poder hacer un **seguimiento** hasta los requisitos del cliente.*
2. *Las pruebas deberían planificarse mucho antes de que empiecen → Planificación anterior a la realización (**Plan de prueba**).*
3. *El **principio de Pareto** (80% resultado/20% esfuerzo) es aplicable:*
 - Al 80% de todos los errores descubiertos durante las pruebas se les puede hacer un seguimiento hasta un 20% de todos los módulos del programa
 - El problema es aislar estos módulos sospechosos y probarlos concienzudamente

Pruebas de software

❑ Principios:

4. *Se deben incluir tanto entradas **correctas** como **incorrectas***
5. *Las pruebas deberían empezar por “lo **pequeño**” y progresar a “lo **grande**”*
6. ***Imposibilidad** de hacer unas pruebas **exhaustivas***
7. *Gran efectividad → realización de pruebas por **equipos independientes***

Pruebas de software

❑ Principios:

8. *Una buena prueba **no debe ser redundante***
9. *Una buena prueba no debería ser **ni demasiado sencilla ni demasiado compleja***
10. *Las pruebas se deben **documentar detalladamente**.*

Pruebas de software

❑ Facilidad de prueba (I):

- ✓ *Es simplemente la facilidad con la que se puede probar un programa*

- ✓ *Características de un software “fácil de probar”:*
 - Operatividad
 - Observabilidad
 - Controlabilidad
 - Capacidad de descomposición
 - Simplicidad
 - Estabilidad
 - Facilidad de comprensión

Pruebas de software

❑ Facilidad de prueba (II):

✓ **Operatividad:** *“Cuanto mejor funcione, más eficientemente se puede probar”*

- El sistema tiene pocos errores
- Ningún error bloquea la ejecución de las pruebas
- El producto evoluciona en fases funcionales

✓ **Observabilidad:** *“Lo que ves es lo que pruebas”*

- Se genera una salida distinta para cada entrada
- Los estados y variables del sistema están visibles
- Todos los factores que afectan a los resultados están visibles
- Un resultado incorrecto se identifica fácilmente
- Se informa automáticamente de los errores internos
- El código fuente es accesible

Pruebas de software

❑ Facilidad de prueba (III):

- ✓ **Controlabilidad:** *“Cuanto mejor podamos controlar el software, más se puede automatizar y optimizar”*
 - El ingeniero de pruebas puede controlar directamente los estados y las variables del hardware y del software
 - Los formatos de las entradas y los resultados son consistentes y estructurados
 - Capacidad de descomposición
- ✓ **Capacidad de descomposición:** *“Controlando el ámbito de las pruebas, podemos aislar más rápidamente los problemas y llevar a cabo mejores pruebas de regresión”*
 - El sistema software está construido con módulos independientes
 - Los módulos del software se pueden probar independientemente (COHESIÓN Y ACOPLAMIENTO)

❑ Facilidad de prueba (IV):

✓ ***Simplicidad:*** “Cuanto menos haya que probar, más rápidamente podremos probarlo”

- Simplicidad funcional
- Simplicidad estructural
- Simplicidad del código

✓ ***Estabilidad:*** “Cuanto menos cambios, menos interrupciones a las pruebas”

- Los cambios del software son infrecuentes
- Los cambios del software están controlados
- Los cambios del software no invalidan las pruebas existentes
- El software se recupera bien de los fallos

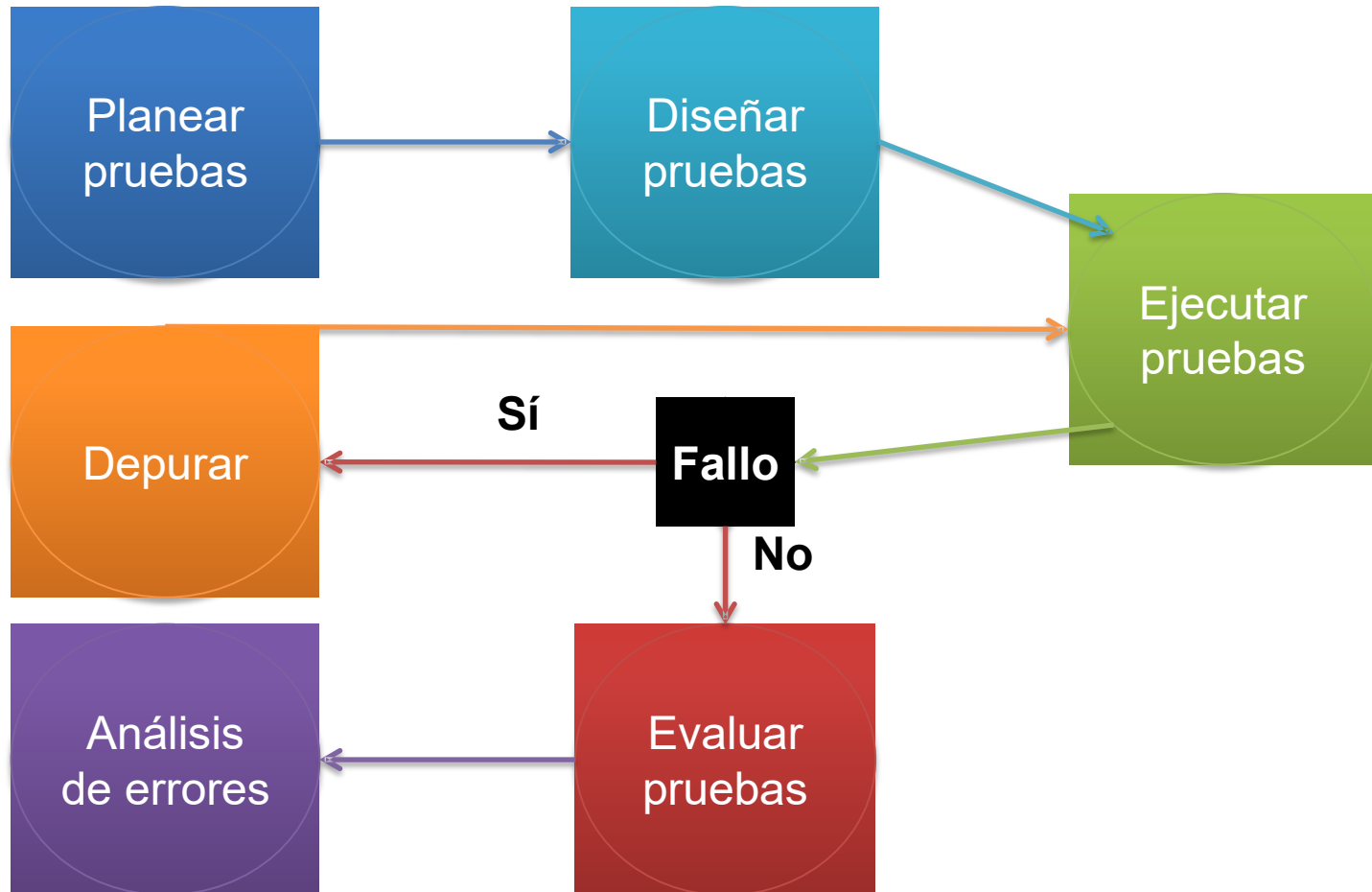
❑ Facilidad de prueba (y V):

✓ ***Facilidad de comprensión:*** “Cuanta más información tengamos, más inteligentes serán las pruebas”

- La documentación técnica está bien organizada
- La documentación técnica es específica y detallada
- La documentación técnica es exacta

Pruebas de software

❑ Proceso de prueba de software:



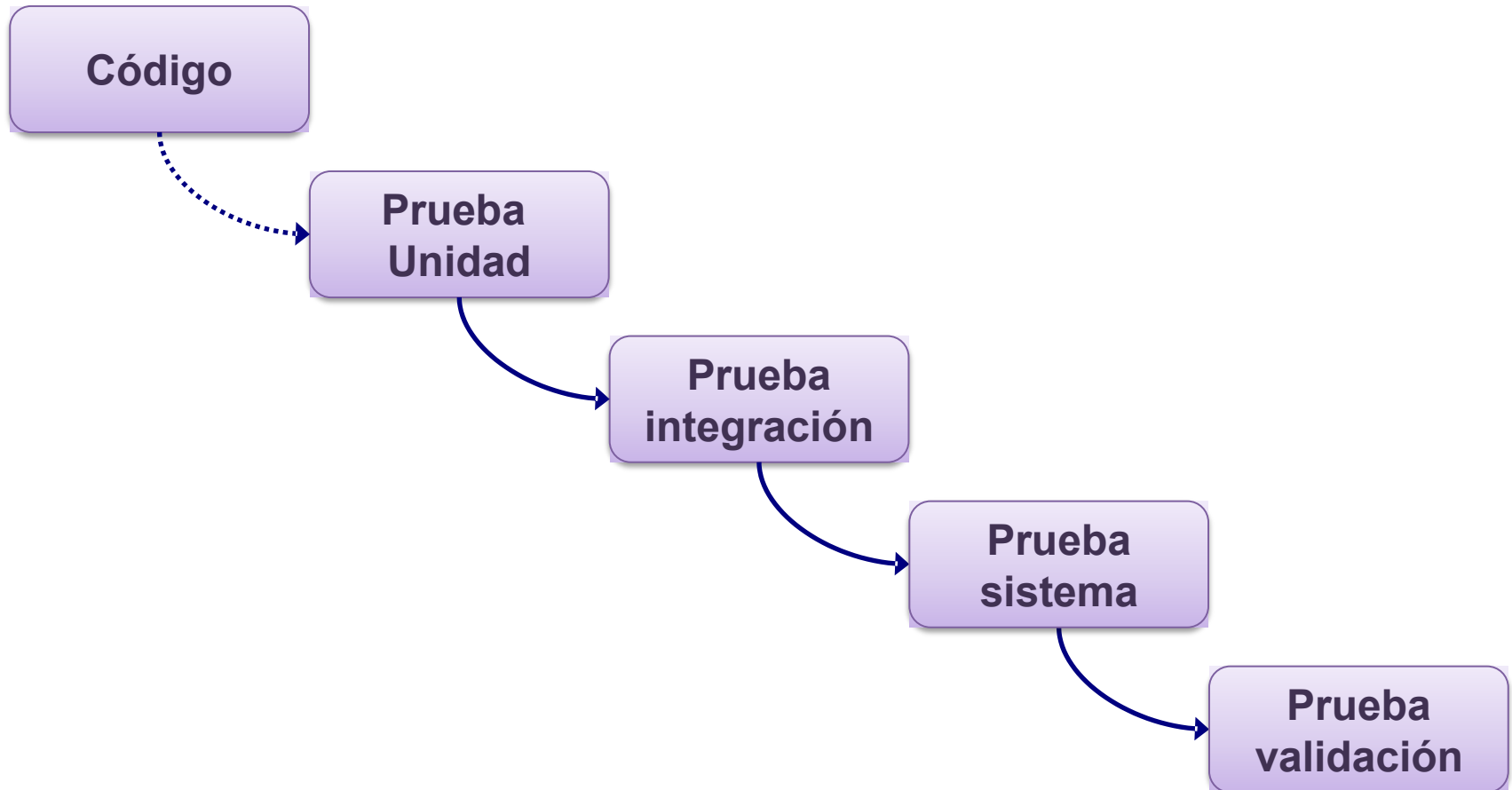
Pruebas de software

❑ Diseño de pruebas:

- ✓ *Probar exhaustivamente un software es **imposible***
- ✓ *Necesitamos técnicas que nos ayuden a confeccionar casos de prueba que me aseguren una **evaluación eficiente** del software*
- ✓ *La mejor técnica es aquella que **con menos casos de prueba** me lo certifique*

Pruebas de software

❑ Ciclo de pruebas:



Pruebas de unidad

- ❑ Enfoques para la construcción de casos de prueba unitarios (aplicable a cualquier producto de ingeniería):
 - ✓ *Caja blanca*
 - Estructural
 - Conociendo el funcionamiento del producto
 - ✓ *Caja negra*
 - Funcional
 - Conociendo la función específica para la que fue diseñado el producto
 - ✓ *Conjetura de errores*
 - ✓ *Enfoque aleatorio*

Pruebas de unidad

Enfoque de *Caja Blanca*

❑ Prueba de caja blanca:

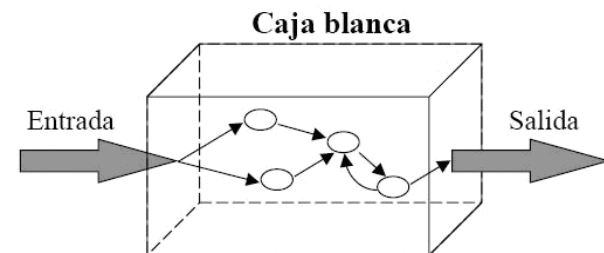
- ✓ Denominada a veces **prueba de caja de cristal**
- ✓ Es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para obtener los casos de prueba

❑ El ingeniero del software puede obtener casos de prueba que:

1. Garanticen que se ejercita **por lo menos una vez** todos los caminos independientes de cada módulo
2. Ejerciten **todas las decisiones lógicas** en sus vertientes verdadera y falsa
3. Ejecuten todos los **bucles en sus límites** y con sus límites operacionales
4. Ejerciten las **estructuras internas** de datos para asegurar su validez

❑ Técnicas de caja blanca:

- ✓ Prueba del camino básico
- ✓ Prueba de la estructura de control



Pruebas de unidad

Enfoque de *Caja Blanca*

❑ Prueba del **camino básico**

- ✓ *Es una técnica de prueba de caja blanca propuesta inicialmente por Tom McCabe (1976)*
- ✓ *Permite obtener una **medida de la complejidad** lógica procedimental*
- ✓ *Para usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución se debe generar un caso de prueba para cada camino de ejecución*
- ✓ *Los casos de prueba garantizan que se ejecuta **por lo menos una vez** cada sentencia del programa*

❑ Elementos:

- ✓ *Notación de grafo de flujo*
- ✓ *Cálculo de la complejidad ciclomática*
- ✓ *Procedimiento de diseño de prueba*

Pruebas de unidad

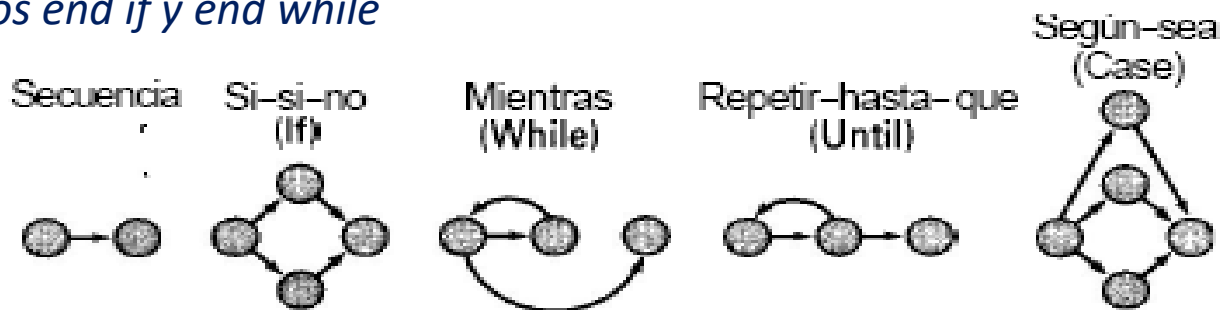
Enfoque de *Caja Blanca*

❑ Prueba del camino básico: Notación de grafo de flujo

- ✓ Representa el flujo de control lógico mediante la notación ilustrada
- ✓ Un solo nodo puede corresponder a una secuencia de cuadros de proceso y a un rombo de decisión
- ✓ Las flechas del grafo de flujo, denominadas **aristas o enlaces**, representan flujo de control y son análogas a las flechas del diagrama de flujo

❑ ¿Cómo construir un diagrama de flujo a partir del código?

- ✓ Señalar todas las decisiones de las sentencias *if-then-else*, *case-of*, *while* y *until*
- ✓ Agrupar resto de sentencias
- ✓ Señalar los *end if* y *end while*

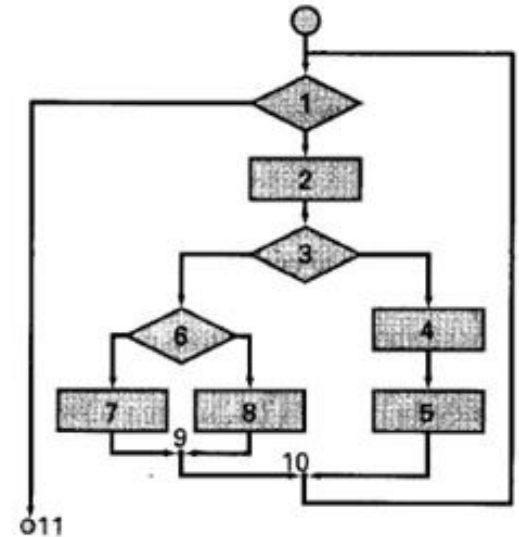
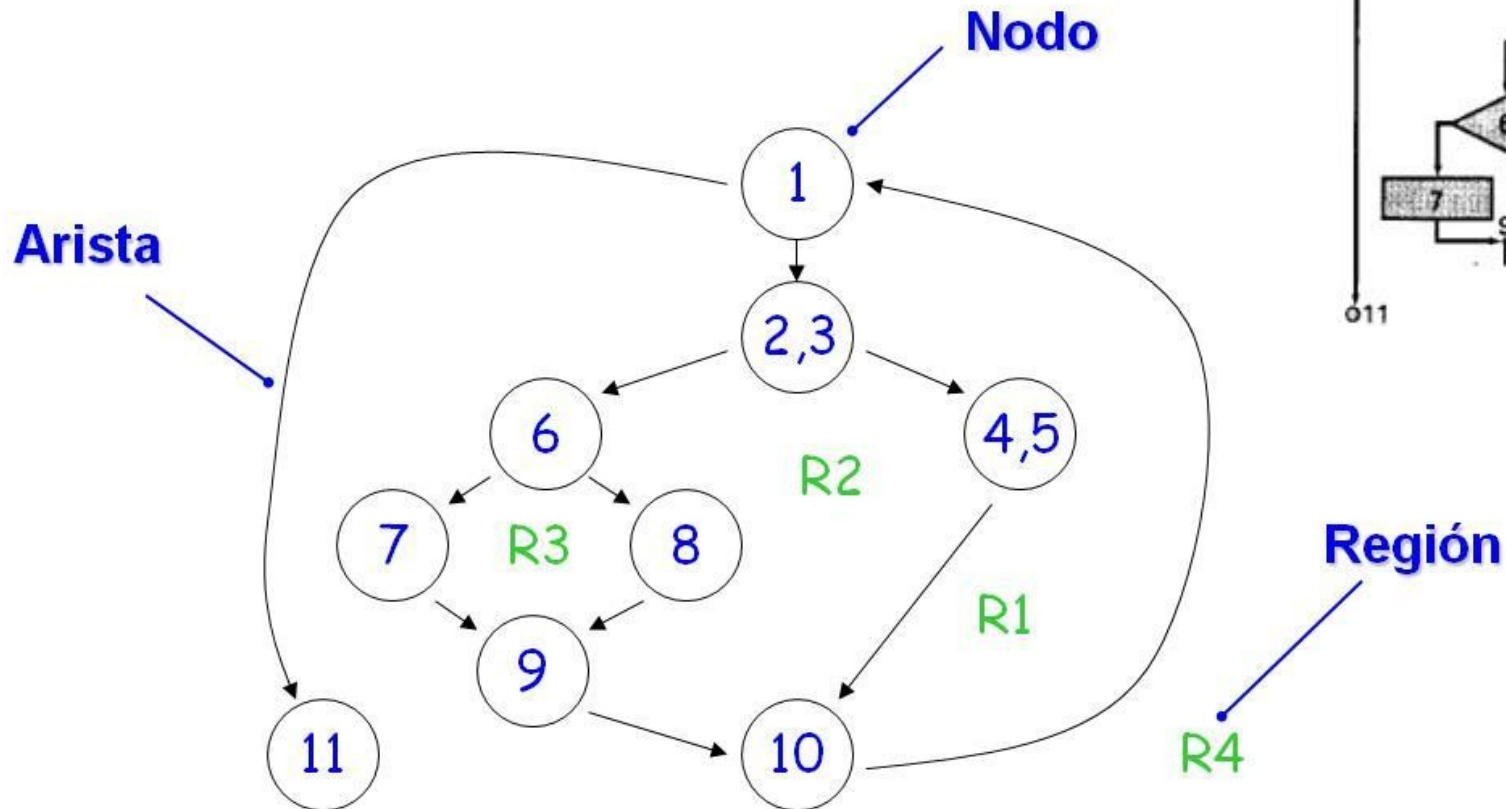


Donde cada círculo representa una o más sentencias, sin bifurcaciones, en LDP o código fuente

Pruebas de unidad

Enfoque de *Caja Blanca*

- ❑ Prueba del camino básico:
Notación de grafo de flujo



Pruebas de unidad

Enfoque de *Caja Blanca*

❑ Prueba del camino básico: **Cálculo de la complejidad ciclomática**

- ✓ *Es una métrica del software que proporciona una **medición cuantitativa** de la complejidad lógica de un programa*
- ✓ *El valor calculado como complejidad ciclomática **define el número de caminos independientes** del conjunto básico de un programa*
- ✓ ***Límite superior** para el número de pruebas que se deben realizar para asegurar que se ejecuta cada sentencia al menos una vez*
- ✓ *Un **camino independiente** es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una nueva condición*
- ✓ *La complejidad ciclomática es una métrica útil para **predecir los módulos** que son más **propensos a error**. Puede ser usada tanto para **planificar pruebas** como para **diseñar casos de prueba***

Pruebas de unidad

Enfoque de *Caja Blanca*

❑ Prueba del camino básico: Cálculo de la complejidad ciclomática

✓ *Ejemplos camino independiente:*

camino 1: 1-11

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

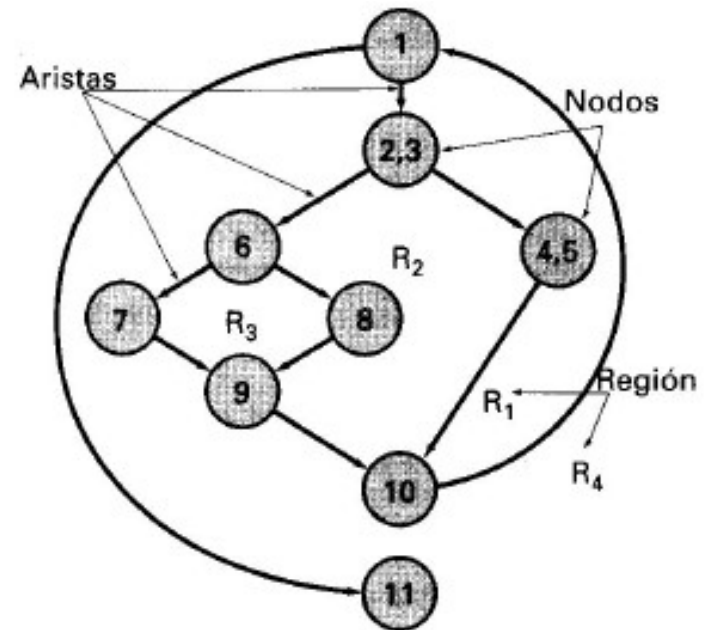
camino 3: 1-2-3-6-8-9-10-1-11

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

✓ *No es un camino independiente:*

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

✓ *Conjunto básico*



b) Grafo de flujo

Pruebas de unidad

Enfoque de *Caja Blanca*

❑ Prueba del camino básico: **Cálculo de la complejidad ciclomática**

✓ *La complejidad ($V(G)$) se puede calcular de 3 formas:*

■ Fórmula de las regiones:

$$V(G) = A - N + 2$$

donde A es el número de aristas del grafo de flujo y N es el número de nodos del mismo.

■ Fórmula de las decisiones (P = nodos de decisión):

$$V(G) = P + 1$$

donde P es el número de nodos predicho contenidos en el grafo de flujo G .

■ El número de regiones del grafo:

- $V(G) = R$
- $R = A - N + 2 = C + 1$

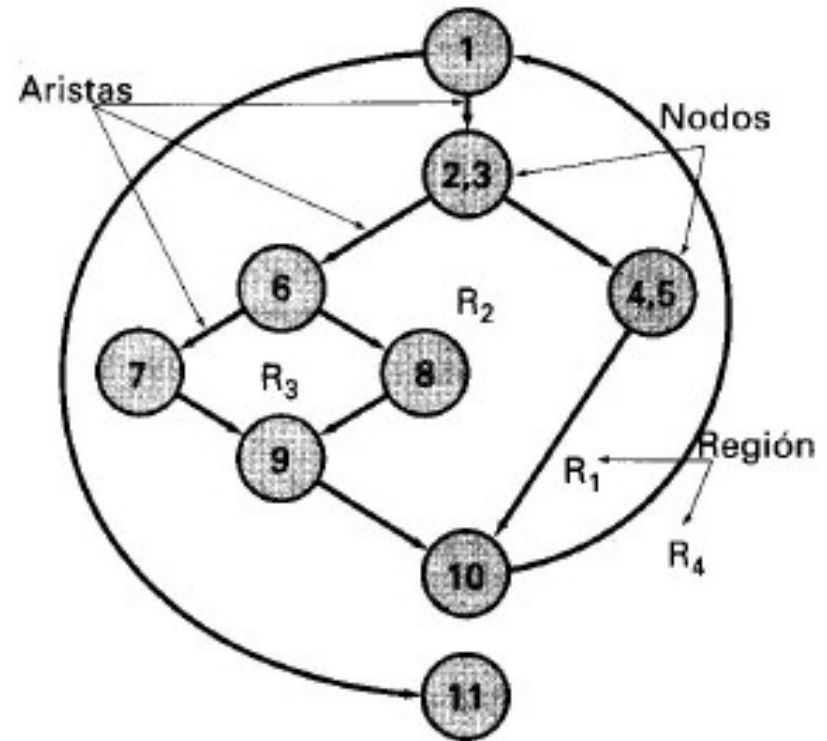
Pruebas de unidad

Enfoque de *Caja Blanca*

❑ Prueba del camino básico: Cálculo de la complejidad ciclomática

✓ Considerando el ejemplo anterior:

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



b) Grafo de flujo

✓ Luego, la complejidad ciclomática es 4

Pruebas de unidad

Enfoque de *Caja Blanca*

❑ Prueba del camino básico: **Procedimiento de diseño de prueba**

1. *Dibujar el grafo de flujo.*
2. *Calcular la complejidad ciclomática.*
3. *Determinar el conjunto de caminos independientes.*
4. *Preparar el caso de prueba para cada camino.*

Pruebas de unidad

Enfoque de *Caja Blanca*

❑ Prueba del camino básico: Otro ejemplo

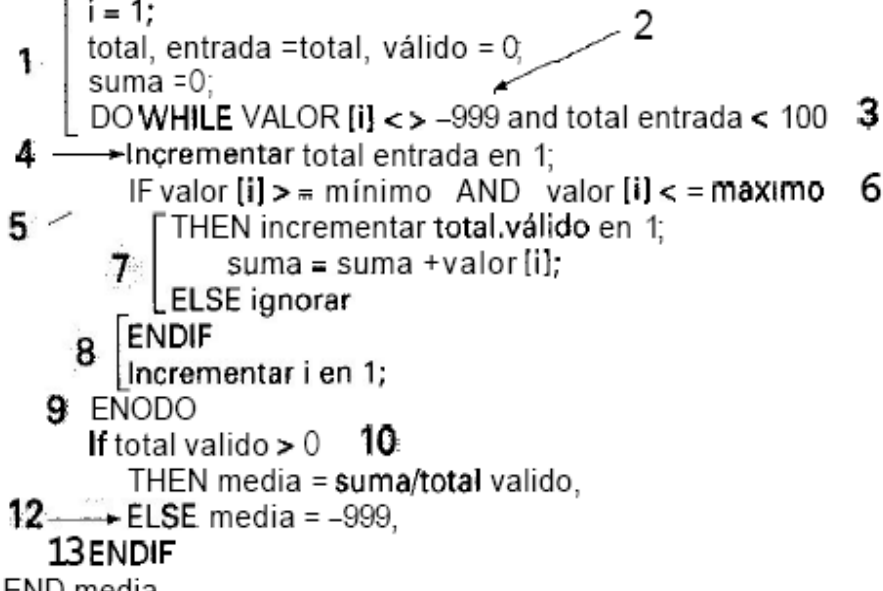
PROCEDURE media;

* Este procedimiento calcula la media de 100 o menos números que se encuentren entre unos límites; también calcula el total de entradas y el total de números válidos.

INTERFACE RETURNS media, total. entrada, total. válido;
INTERFACE ACCEPTS valor, mínimo, máximo;

TYPE valor [1:100] IS SCALAR ARRAY;
TYPE media, total. entrada, total. válido;
Mínimo, máximo, suma IS SCALAR;
TYPE i IS INTEGER;

```
1  i = 1;  
   total, entrada = total, válido = 0;  
   suma = 0;  
   DO WHILE VALOR [i] <> -999 and total entrada < 100 3  
4  → Incrementar total entrada en 1;  
   IF valor [i] >= mínimo AND valor [i] <= máximo 6  
5  / THEN incrementar total.válido en 1;  
   7  suma = suma + valor [i];  
   ELSE ignorar  
   8  ENDIF  
   Incrementar i en 1;  
9  ENODO  
   If total valido > 0 10  
   THEN media = suma/total valido,  
12 → ELSE media = -999,  
13 ENDIF  
END media
```

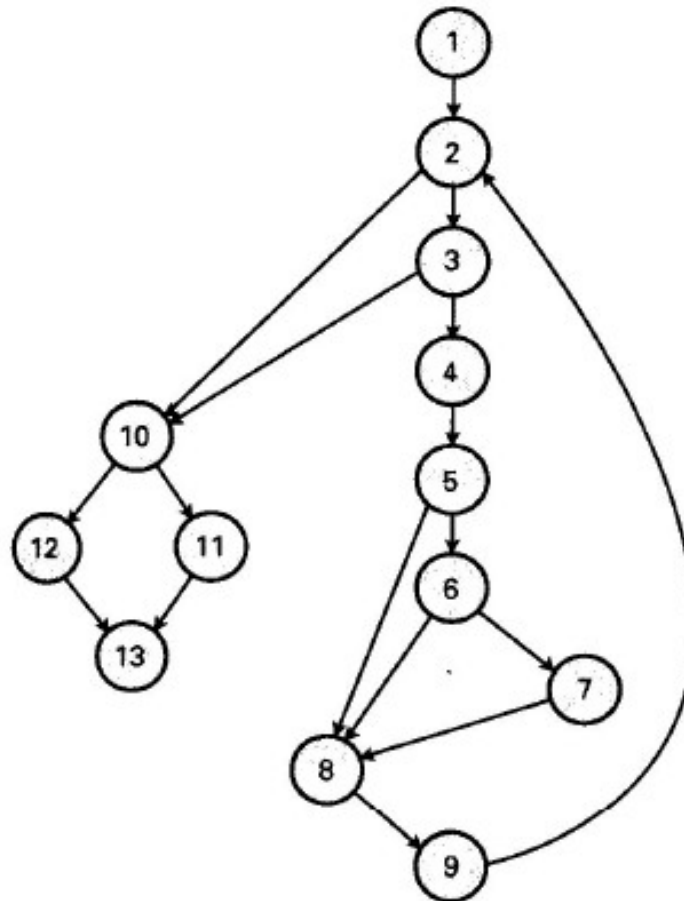


Pruebas de unidad

Enfoque de *Caja Blanca*

❑ Prueba del camino básico: otro ejemplo

1. Grafo de flujo



Pruebas de unidad

Enfoque de *Caja Blanca*

- ❑ Prueba del camino básico: **otro ejemplo**
 - 2. *Cálculo de la complejidad ciclomática*

$$V(G) = 6 \text{ regiones}$$

$$V(G) = 17 \text{ aristas} - 13 \text{ nodos} + 2 = 6$$

$$V(G) = 5 \text{ nodos predicado} + 1 = 6$$

Pruebas de unidad

Enfoque de *Caja Blanca*

❑ Prueba del camino básico: **otro ejemplo**

3. *Conjunto básico de caminos linealmente independientes*

camino 1: 1-2-10-11-13

camino 2: 1-2-10-12-13

camino 3: 1-2-3-10-11-13

camino 4: 1-2-3-4-5-8-9-2-...

camino 5: 1-2-3-4-5-6-8-9-2-...

camino 6: 1-2-3-4-5-6-7-8-9-2-...

Pruebas de unidad

Enfoque de *Caja Blanca*

❑ Prueba de la estructura de control:

- ✓ Aunque la prueba del camino básico es sencilla y altamente efectiva, **no es suficiente** por sí sola
- ✓ Estas variantes **amplían la cobertura** de la prueba y **mejoran la calidad** de la prueba de caja blanca

❑ Criterios de Cobertura:

- De Sentencia
- De Decisiones
- De Condiciones
- De Decisión / Condición
- De Condición Múltiple

❑ De Sentencias:

- ✓ *Se trata de generar los casos de prueba necesarios para que cada sentencia o instrucción del programa se ejecute, al menos, una vez.*

❑ De decisiones:

- ✓ *Consiste en escribir casos suficientes para que cada decisión tenga, por lo menos una vez, un resultado verdadero y, al menos una vez, uno falso.*

Pruebas de unidad

Enfoque de *Caja Blanca*

❑ De condiciones:

- ✓ *Se trata de diseñar tantos casos como sea necesario para que cada condición de cada decisión adopte el valor verdadero al menos una vez y el falso al menos una vez.*

❑ De decisión/condición.

- ✓ *Consiste en exigir el criterio de cobertura de condiciones obligando a que se cumpla también el criterio de decisiones.*

❑ De condición múltiple.

- ✓ *En el caso de que se considere que la evaluación de las condiciones de cada decisión no se realiza de forma simultánea.*

Pruebas de unidad

Enfoque de *Caja Blanca*

❑ Ejemplo

- ✓ *Crea el grafo de flujo para siguiente código:*

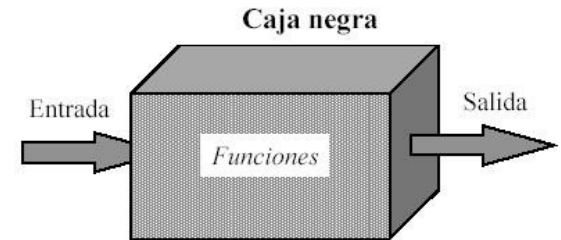
```
If ((i==0) AND (j==1)){  
    ....  
} else {  
    ...  
}
```

- ✓ *Lista el conjunto de caminos linealmente independientes.*
- ✓ *Planifica las pruebas para proporcionar cobertura:*
 - Sentencia
 - Decisión
 - Condición
 - Decisión / Condición
 - Condición Múltiple

Pruebas de unidad

Enfoque de *Caja Negra*

❑ Prueba de caja negra: Principios



- ✓ *Ó prueba de comportamiento*
- ✓ *Se centran en los **requisitos funcionales** del software.*
- ✓ *La prueba de caja negra permite al ingeniero del software obtener **conjuntos de condiciones de entrada** que ejerciten completamente todos los requisitos funcionales de un programa*
- ✓ ***No es una alternativa** a las técnicas de prueba de caja blanca.*
- ✓ *Se trata de un **enfoque complementario** que intenta descubrir diferentes tipos de errores que los métodos de caja blanca*

❑ Prueba de caja negra: **Principios**

✓ *Intenta encontrar errores de las siguientes categorías:*

- funciones incorrectas o ausentes
- errores de interfaz
- errores en estructuras de datos o en accesos a bases de datos externas
- errores de rendimiento
- errores de inicialización y de terminación

Pruebas de unidad

Enfoque de *Caja Negra*

❑ Prueba de caja negra: **Principios**

- ✓ *La prueba de caja negra tiende a aplicarse durante **fases posteriores** de la prueba*
- ✓ *La prueba de caja negra **ignora intencionadamente** la estructura de control*
- ✓ *Se obtiene un conjunto de casos de prueba que satisfacen los siguientes criterios:*
 - Casos de prueba que reducen el número de **casos de prueba adicionales** que se deben diseñar para alcanzar una prueba razonable
 - Casos de prueba que nos dicen algo sobre la **presencia o ausencia de clases de errores** en lugar de errores asociados solamente con la prueba que estamos realizando

Pruebas de unidad

Enfoque de *Caja Negra*

❑ Prueba de caja negra: **Técnicas**

- ✓ *Partición equivalente (clases de equivalencia)*
- ✓ *Análisis de valores límite (AVL)*

Pruebas de unidad

Enfoque de *Caja Negra*

❑ Partición en clases de equivalencia:

- ✓ *Es un método de prueba de caja negra que **divide el campo de entrada** de un programa en **clases de datos** de los que se pueden derivar casos de prueba*
- ✓ *Puede descubrir de forma **inmediata** una clase de errores (por ejemplo, proceso incorrecto de todos los datos de carácter) que, de otro modo, requerirían la ejecución de muchos casos antes de detectar el error genérico*
- ✓ *El diseño de casos de prueba 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 no válidos** para condiciones de entrada*

Pruebas de unidad

Enfoque de *Caja Negra*

❑ Partición en clases de equivalencia:

- ✓ ***Clase de equivalencia:** representa un conjunto de estados válidos o no válidos para condiciones de entrada*
- ✓ *Las clases de equivalencia se pueden definir de acuerdo con las siguientes **directrices**:*
 1. Si una condición de entrada especifica un **rango**, se define una clase de equivalencia válida y dos no válidas
 2. Si una condición de entrada requiere un **valor específico**, se define una clase de equivalencia válida y una no válida
 3. Si una condición de entrada especifica un **miembro de un conjunto**, se define una clase de equivalencia válida y una no válida
 4. Si una condición de entrada es **lógica**, se define una clase de equivalencia válida y una no válida

Pruebas de unidad

Enfoque de *Caja Negra*

❑ Partición en clases de equivalencia:

✓ *Ejemplos:*

- **Rango:** Alquiler para personas entre 18 y 80 años
 - Válida: clase que agrupe los valores entre 19 y 79 (inclusive)
 - Inválidas: una clase para los < 18 años y otra para los > 80 años
- **Valor:** El nombre debe tener 10 caracteres
 - Válida: clase para 10 caracteres exactamente
 - Inválida: clase para cualquier valor distinto de 10
- **Conjunto:** Las formas de compra serán “pagaré”, “tarjeta” y “efectivo”
 - Válida: clase para los valores del conjunto
 - Inválida: clase para los valores que no pertenezcan al conjunto
- **Lógico:** El código postal es obligatorio
 - Válida: clase en la que el código está presente
 - Inválida: clase para el caso en el que no se especifica el código

✓ *Es posible que haya que definir más de una clase sobre la misma condición de entrada (p.ej. Código postal)*

- ❑ **Partición en clases de equivalencia (Ejemplo):**
 - ✓ *Diseñar casos de prueba de partición equivalente para un software que capte estos datos de entrada (clases válidas y no válidas):*
 - **Código de área:** *Un número de tres dígitos.*
 - **Prefijo:** *Número de tres dígitos, que no comiencen por 0 ó 1*
 - **Sufijo:** *Número de cuatro dígitos.*
 - **Órdenes:** *"Cheque", "Depósito", "Pago factura"*
 - **Palabra clave:** *Valor alfanumérico de 6 dígitos*

❑ Análisis de valores límite (AVL)

- ✓ *Es una técnica **complementaria** a la partición en clases de equivalencia*
- ✓ *Los errores tienden a darse más en los límites del campo de entrada que en el «centro»*
- ✓ *Explora las **condiciones límites** de un programa. Se eligen valores en los **márgenes de la clase***
- ✓ *No sólo se fija el dominio de entrada sino también el **dominio de salida***

Pruebas de unidad

Enfoque de *Caja Negra*

❑ Análisis de valores límite: **Principios**

1. **Rango:** 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
 - para los valores justo por debajo y justo por encima de a y b , respectivamente
2. **Valor:** Si la condición de entrada especifica un valor se deben generar casos para:
 - Valor
 - Valor + 1
 - Valor - 1
3. Aplicar las directrices 1 y 2 a las **condiciones de salida**:
 - **Por ejemplo**, supongamos que se requiere una tabla de «temperatura / presión» como salida de un programa de análisis de ingeniería
 - Se deben diseñar casos de prueba que creen un informe de salida que produzca el máximo (y el mínimo) número permitido de entradas en la tabla
4. Si las estructuras de datos internas tienen límites preestablecidos, hay que asegurarse de diseñar un caso de prueba que ejercite la estructura de datos en sus límites.

Conjetura de errores y Enfoque aleatorio

❑ Conjetura de errores:

- ✓ *Enumerar una lista de **errores comunes***
- ✓ *En función de ella elaborar los casos de prueba*

❑ Enfoque aleatorio:

- ✓ *Se crean datos que sigan **la frecuencia y la secuencia** que tendrían en la práctica.*
- ✓ *Se **simula** la entrada de datos*
- ✓ *Se usan **generadores** de prueba.*

Pruebas de Integración

❑ Pruebas de Integración

- ✓ *La prueba de integración es una técnica **sistemática** para construir la estructura del programa*
- ✓ *El objetivo es coger los módulos probados mediante la prueba de unidad y construir una **estructura** de programa que esté de acuerdo con lo que dicta el **diseño***
- ✓ *La prueba de integración deberá ser conducida **incrementalmente***
 - El programa se construye y se prueba en pequeños segmentos
 - Los errores son más fáciles de aislar y de corregir
- ✓ *Enfoques de integración:*
 - Integración descendente
 - Integración ascendente
- ✓ *Pruebas de integración:*
 - Pruebas de regresión
 - Pruebas de humo

❑ Integración descendente

- ✓ *Es un planteamiento **incremental** para la construcción de la estructura de programas.*
- ✓ *Se integran los módulos moviéndose hacia abajo por la jerarquía de control*
 - Comenzando por el módulo de control principal (**programa principal**).
 - Los módulos subordinados (**subordinados** de cualquier modo) al módulo de control principal se van incorporando en la estructura, bien de forma primero-en-profundidad, o bien de forma primero-en-anchura.

Pruebas de Integración

Integración descendente

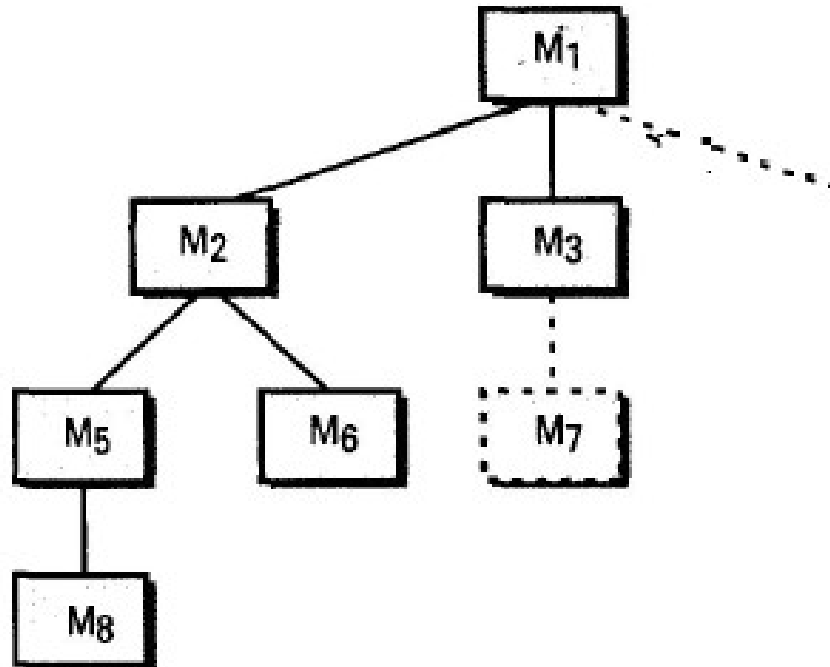
❑ Estrategia top-down

- ✓ *La integración primero-en-profundidad integra todos los módulos de un camino de control principal de la estructura.*
- ✓ *La selección del camino principal es arbitraria y dependerá de las características específicas de la aplicación*
- ✓ *Pasos para una integración top-down:*
 1. Se usa el módulo de control principal como **controlador** de la prueba
 2. Dependiendo del enfoque de integración elegido (es decir, primero-en-profundidad o primero-en-anchura) se van sustituyendo uno a uno los **resguardos** subordinados por los módulos reales
 3. Se llevan a cabo **pruebas cada vez que se integra** un nuevo módulo
 4. Tras terminar cada conjunto de pruebas, se **reemplaza** otro resguardo con el módulo real
 5. Se hace la **prueba de regresión** para asegurarse de que no se han introducido errores nuevos

Pruebas de Integración

Integración descendente

❑ Estrategia top-down



❑ Integración ascendente

- ✓ *Es otro planteamiento **incremental** a la construcción de la estructura de programas.*
- ✓ *Comienza la construcción y la prueba con los módulos atómicos*
- ✓ *El proceso requerido de los módulos subordinados siempre está disponible y se elimina la necesidad de resguardos*

Pruebas de Integración

Integración ascendente

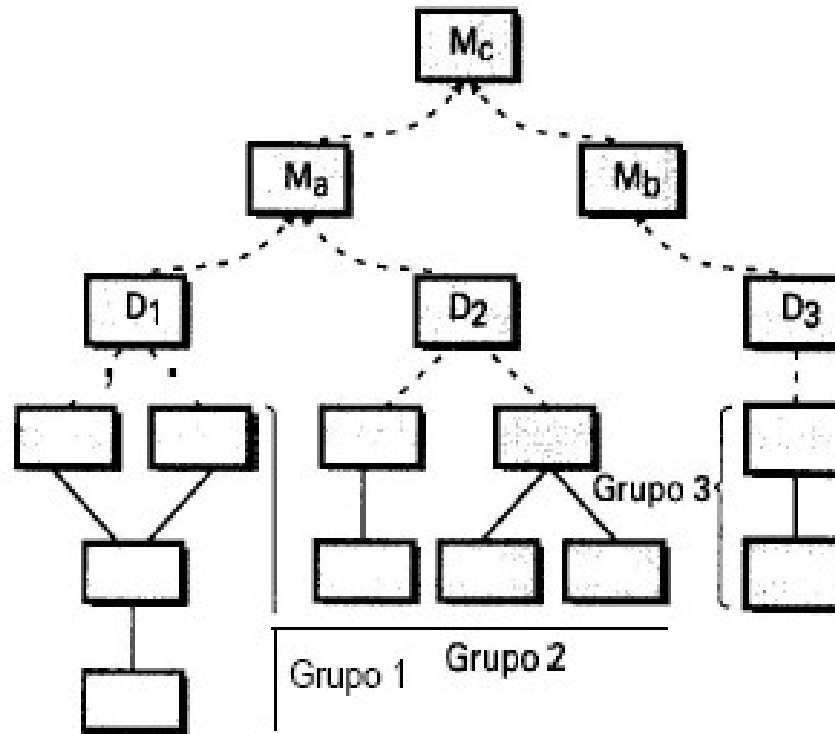
❑ Pasos para una integración ascendente:

1. *Se combinan los módulos de bajo nivel en grupos (construcciones) que realicen una sub-función específica del software*
2. *Se escribe un **controlador** (un programa de control de la prueba) para coordinar la entrada y la salida de los casos de prueba*
3. *Se prueba el grupo*
4. *Se eliminan los controladores y se combinan los grupos moviéndose hacia arriba por la estructura del programa*

Pruebas de Integración

Integración ascendente

❑ Estrategia bottom-up:



Pruebas de Integración

Pruebas de Regresión

❑ Prueba de Regresión

✓ Definiciones:

- Cualquier tipo de pruebas de software que intentan descubrir nuevos errores inducidos por cambios recientemente realizados en partes de la aplicación que anteriormente al citado cambio no eran propensas a este tipo de error
 - Volver a ejecutar un subconjunto de pruebas que se han llevado a cabo anteriormente para asegurarse de que los cambios no han propagado efectos colaterales no deseados
 - La prueba de regresión es la actividad que ayuda a asegurar que los cambios no introducen un comportamiento no deseado o errores adicionales
- ✓ *Cada vez que se añade un nuevo módulo como parte de una prueba de integración, el software **cambia**:*
- Se establecen nuevos caminos de flujo de datos
 - Pueden ocurrir nuevas E/S
 - Se invoca una nueva lógica de control
- ✓ *Estos cambios pueden causar problemas con funciones que **antes** trabajaban perfectamente*
- ✓ *La prueba de regresión es una estrategia importante para reducir **efectos colaterales**. Se deben ejecutar pruebas de regresión cada vez que se realice un cambio importante en el software (incluyendo la integración de nuevos módulos)*

Pruebas de Integración

Pruebas de Regresión

- ❑ El conjunto de pruebas de regresión contiene **tres clases** diferentes de casos de prueba:
 - ✓ *Una muestra representativa de pruebas que ejercite **todas las funciones** del software*
 - ✓ *Pruebas adicionales que se centran en las funciones del software que se van a ver probablemente **afectadas por el cambio***
 - ✓ *Pruebas que se centran en los **componentes** del software que han cambiado*

- ❑ Tipos de regresión según el ámbito:
 - ✓ **Local** - *Los cambios introducen nuevos errores en un módulo concreto*
 - ✓ **Desenmascarada** - *Los cambios revelan errores previos*
 - ✓ **Remota** - *Los cambios vinculan algunas partes del programa (módulo) e introducen errores en ella*

❑ Otras consideraciones de la prueba de regresión

- ✓ *A medida que progresa la prueba de integración, el número de pruebas de regresión puede **crecer demasiado***
- ✓ *Incluir sólo pruebas que traten **una o más clases de errores** en cada una de las funciones principales*
- ✓ *No es práctico ni eficiente volver a ejecutar cada prueba*

❑ Prueba de humo

- ✓ *Es un método de prueba de integración*
- ✓ *Comúnmente utilizada para software «empaquetado»*
- ✓ *Es diseñado como un mecanismo para proyectos críticos por tiempo*
- ✓ *Definición:*
 - Es un procedimiento de prueba diseñado para cubrir una **amplia porción** de la funcionalidad del sistema
 - Las pruebas así diseñadas dan prioridad al porcentaje del total del sistema que se pone a prueba, sacrificando la precisión de estas de ser necesario

Pruebas de Integración

Pruebas de Humo

❑ Actividades

1. *Los componentes software que han sido traducidos a código se integran en una «construcción»*
 - Una construcción incluye ficheros de datos, librerías, módulos reutilizables y componentes de ingeniería que se requieren para implementar una o más funciones del producto
2. *Se diseña una serie de pruebas para descubrir errores que impiden a la construcción realizar su función adecuadamente.*
 - El objetivo será descubrir errores «bloqueantes» que tengan la mayor probabilidad de impedir al proyecto de software el cumplimiento de su planificación
3. *Es habitual en la prueba de humo que la construcción se integre con otras construcciones y que se aplica una prueba de humo al producto completo.*
 - La integración puede hacerse bien de forma descendente (top-down) o ascendente (bottom-up).

❑ Otras consideraciones:

- ✓ *La prueba de humo se caracteriza por una estrategia de integración continua*
- ✓ *La prueba de humo ejercita el sistema entero de principio a fin. No ha de ser exhaustiva, pero será capaz de descubrir importantes problemas*

❑ Beneficios

1. *Se minimizan los riesgos de integración.*
 - Dado que las pruebas de humo son realizadas frecuentemente, incompatibilidades y otros errores bloqueantes son descubiertos rápidamente
2. *Se perfecciona la calidad del producto final*
 - Es probable que descubra errores funcionales, además de defectos de diseño a nivel de componente y de arquitectura
3. *Se simplifican el diagnóstico y la corrección de errores*
4. *El progreso es fácil de observar*
 - Cada día que pasa, se integra más software y se demuestra que funciona

Pruebas de Sistema

❑ Características

✓ *A final, el software es incorporado a otros elementos del sistema:*

- Hardware
- Información
- Software de base
- Personas
- Procesos no sistematizados

✓ *Se realizan una serie de pruebas de **integración del sistema** y de **validación***

- Diseñar caminos de manejo de errores que prueben toda la información procedente de otros elementos del sistema
- Pruebas que simulen la presencia de datos en mal estado o posibles errores en la interfaz
- Registrar los resultados de las pruebas como «evidencia» en el caso de que se le señale con el dedo al responsable del módulo donde se encontró el error

❑ Tipos

- ✓ *Prueba de recuperación*
- ✓ *Prueba de seguridad*
- ✓ *Prueba de resistencia (Stress)*
- ✓ *Prueba de rendimiento*

❑ Prueba de recuperación

- ✓ *Recuperación ante fallos y continuar el proceso en un tiempo previamente especificado*
- ✓ **MTTR** (*Mean/Minimum/Maximum Time to Recovery/ Repair/Respond/Restore*)
- ✓ *La prueba de recuperación es una prueba del sistema que **fuerza el fallo** del software de muchas formas y verifica que la **recuperación** se lleva a cabo apropiadamente*

Pruebas de Sistema

❑ Prueba de seguridad

- ✓ *Entradas impropias o ilegales al sistema.*
- ✓ *Incluye un amplio rango de actividades:*
 - Piratas informáticos que intentan entrar en los sistemas por deporte
 - Empleados disgustados que intentan penetrar por venganza
 - Individuos deshonestos que intentan penetrar para obtener ganancias personales ilícitas.
- ✓ *La prueba de seguridad intenta verificar que los mecanismos de protección incorporados en el sistema lo protegerán, de hecho, de accesos impropios*
- ✓ *¡¡TODO VALE!!*
 - Durante la prueba de seguridad, el responsable de la prueba desempeña el papel de un individuo que desea entrar en el sistema
 - Al final, siempre se accede: el coste de la entrada ilegal debe ser mayor que el valor de la información obtenida

Pruebas de Sistema

❑ Prueba de resistencia (stress)

- ✓ *En pruebas anteriores, se buscaba la **evaluación** del funcionamiento y del rendimiento normales del programa*
- ✓ *Las pruebas de resistencia están diseñadas para enfrentar a los programas con **situaciones anormales***
- ✓ *Se busca responder a la pregunta: **¿A qué potencia puedo ponerlo a funcionar antes de que falle?***
- ✓ *Ejemplos:*
 1. Diseñar pruebas especiales que generen diez interrupciones por segundo, cuando las normales son una o dos
 2. Incrementar las frecuencias de datos de entrada en un orden de magnitud con el fin de comprobar cómo responden las funciones de entrada
 3. Ejecutar casos de prueba que requieran el máximo de memoria o de otros recursos
 4. Diseñar casos de prueba que produzcan excesivas búsquedas de datos residentes en disco

Pruebas de Sistema

❑ Prueba de rendimiento

- ✓ *La prueba de rendimiento está diseñada para probar el rendimiento del software **en tiempo de ejecución** dentro del contexto de un **sistema integrado***
- ✓ *Se da durante todos los pasos del proceso de la prueba (incluso al nivel de unidad)*
- ✓ *Hasta que no están completamente integrados no se puede asegurar realmente el rendimiento del sistema*
- ✓ *A menudo, van emparejadas con las pruebas de resistencia*

Pruebas de Validación

❑ Principios de las Pruebas de Validación:

- ✓ *La validación se consigue cuando el software funciona de acuerdo con las **expectativas razonables** del cliente*
- ✓ *Las expectativas razonables están definidas en la **Especificación de Requisitos del Software***
- ✓ *La especificación debería contener una sección denominada «**Criterios de validación**»*
- ✓ *Como en otras etapas de la prueba, la validación permite descubrir errores, pero su enfoque está en el **nivel de requisitos** -sobre cosas que son necesarias para el usuario final*

Pruebas de Validación

❑ Aplicación de criterios en Pruebas de Validación

✓ *Conceptos:*

- La validación del software se consigue mediante una serie de pruebas de caja negra que demuestran la **conformidad con los requisitos**
- Un **plan de prueba** traza la clase de pruebas que se han de llevar a cabo
- Un **procedimiento de prueba** define los casos de prueba específicos

✓ *Tanto el plan como el procedimiento estarán diseñados para asegurar:*

1. Que se satisfacen todos los requisitos funcionales
2. Que se alcanzan todos los requisitos de rendimiento
3. Que la documentación es correcta e inteligible
4. Que se alcanzan otros requisitos (ej. portabilidad)

✓ *Puede darse una de las dos condiciones:*

- Las características de funcionamiento o de rendimiento están de acuerdo con las especificaciones y son aceptables
- Se descubre una **desviación** de las especificaciones y se crea una lista de **deficiencias**

✓ *Las desviaciones o errores descubiertos en esta fase del proyecto **raramente se pueden corregir** antes de la terminación planificada.*

✓ *A menudo es necesario **negociar** con el cliente un método para resolver las deficiencias*

❑ Pruebas de Validación: **Técnicas**

✓ *Revisión de la configuración*

✓ *Pruebas alfa y beta*

❑ Revisión de la configuración

- ✓ *Asegurarse de que **todos los elementos** de la configuración del software se han desarrollado apropiadamente*
- ✓ *Se han **catalogado** y están suficientemente detallados para soportar la fase de **mantenimiento** durante el ciclo de vida del software*
- ✓ *La revisión de la configuración a veces se denomina **auditoría***

❑ Pruebas alfa y beta

- ✓ *Es **virtualmente imposible** que un desarrollador de software pueda prever cómo utilizará el usuario realmente el programa.*
 - Se pueden malinterpretar las instrucciones de uso
 - Se pueden utilizar combinaciones de datos extrañas
 - Una salida que puede parecer clara para el responsable de las pruebas y puede ser ininteligible para el usuario
- ✓ *En proyectos de desarrollo de software a medida, se llevan a cabo una serie de **pruebas de aceptación***

Pruebas de Validación

Pruebas Alfa y Beta

❑ Pruebas de aceptación

- ✓ *Las realiza el usuario final en lugar del responsable del desarrollo del sistema. Puede ir desde:*
 - Un informal «paso de prueba»
 - Hasta la ejecución sistemática de una serie de pruebas bien planificadas
- ✓ *La prueba de aceptación puede tener lugar a lo largo de semanas o meses*
- ✓ *Si va a ser usado por muchos clientes, no es práctico realizar pruebas de aceptación formales para cada uno de ellos*
- ✓ *Llevan a cabo un proceso denominado “de prueba alfa y beta”:*
 - Descubrir errores que parezca que sólo el usuario final puede descubrir

❑ Pruebas alfa:

- ✓ *Se lleva a cabo, por un cliente, en el **lugar de desarrollo***
- ✓ *Se usa el software de forma natural con el desarrollador como observador del usuario*
- ✓ *Registrando los errores y los problemas de uso.*
- ✓ *Las pruebas alfa se llevan a cabo en un **entorno controlado***

Pruebas de Validación

Pruebas Alfa y Beta

❑ Pruebas beta:

- ✓ *Se lleva a cabo por los usuarios finales del software en los lugares de trabajo de los clientes*
- ✓ *A diferencia de la prueba alfa, el desarrollador no está presente*
- ✓ *Es una aplicación «**en vivo**» del software en un entorno que no puede ser controlado por el desarrollador*
- ✓ *El **cliente**:*
 - Registra todos los problemas (reales o imaginarios) que encuentra
 - Informa a intervalos regulares al desarrollador

Depuración

❑ Proceso de depuración:

- ✓ *La depuración no es una prueba*
- ✓ *Pero siempre ocurre como consecuencia de la prueba*
- ✓ *Comienza con la ejecución de un caso de prueba*
- ✓ *Se evalúan los resultados detectando el error*
- ✓ *El proceso de depuración siempre tiene uno de los dos resultados siguientes:*
 - a. *Se encuentra la causa, se corrige y se elimina*
 - b. *No se encuentra la causa*

Depuración

❑ Otras consideraciones:

- ✓ *A medida que las consecuencias de un error aumentan, crece la presión por encontrar su causa*
- ✓ *A menudo la presión fuerza a un ingeniero del software a corregir un error introduciendo dos más*
 - Van Vleck (1989) sugiere tres preguntas sencillas antes de hacer la «corrección»:
 1. ¿Se repite la causa del error en otra parte del programa?
 2. ¿Cuál es el «siguiente error» que se podrá presentar a raíz de la corrección que hoy voy a realizar?
 3. ¿Qué podríamos haber hecho para prevenir este error la primera vez?

Depuración

❑ Enfoques de depuración:

- ✓ *La depuración tiene un objetivo primordial: encontrar y corregir el error*
- ✓ *El objetivo se consigue mediante una combinación de una **evaluación sistemática**, de **intuición** y de **suerte***
- ✓ *Diferentes enfoques de depuración:*
 1. **Fuerza bruta**
 2. **Vuelta atrás**
 3. **Eliminación de causas**
- ✓ *Consejos:*
 - **Fíjate un tiempo limitado, por ejemplo dos horas, en relación a la cantidad de tiempo que tu inviertes para depurar un problema de tu incumbencia.**
 - **Y si no... ¡pide ayuda!**
 - **Cualquier discusión sobre los enfoques para la depuración y sus herramientas no estaría completa sin mencionar un poderoso aliado: ¡otras personas!**