

# *Fundamentos de Ingeniería del Software*



## *Capítulo 5. Prueba del software*



“Bubbles don’t crash”

Bertrand Meyer

# Prueba del software.

## Estructura



1. Objetivos de la prueba
2. Importancia de la prueba
3. Principios de la prueba
4. El proceso de prueba
5. Métodos de diseño de casos de prueba
6. Enfoque estructural
  - Prueba del camino básico
    - Notación de grafo de flujo
    - Complejidad ciclomática
    - Derivación de los casos de prueba
  - Prueba de bucles
7. Enfoque funcional
  - Particiones o clases de equivalencia
  - Análisis de Valores Límite (AVL)
8. Prueba de interfaces gráficas de usuario
9. Estrategias de prueba del software
  - Relación entre productos de desarrollo y niveles de prueba
  - Organización para la prueba del software
  - Prueba de unidad
  - Prueba de integración
  - Integración incremental descendente
  - Integración incremental ascendente
  - Módulos críticos
  - Prueba de aceptación

# *Prueba del software.*

## *Bibliografía*



- (Piattini et *al.* 04) Capítulo 11
- (Piattini et *al.* 96) Capítulo 12
- (Pressman 06) Apto. 5.5.2, capítulos 13 y 14
- (Pressman 02) Capítulos 17 y 18
- (MAP 95) Ministerio de Administraciones Públicas. Guía de Técnicas de Métrica y Guía de Referencia. v.2.1. 1995

# 1. *Objetivos de la prueba*

⇒ *¿Labor destructiva y rutinaria?*

Objetivos de las pruebas (Myers 79)(Pressman 06):

1. La prueba es el proceso de *ejecución* de un programa con la intención de descubrir un error.
  2. Un buen *caso de prueba* es aquel que tiene una alta probabilidad de descubrir un error no encontrado hasta entonces.
  3. Una prueba tiene *éxito* si descubre un error no detectado hasta entonces.
- Actividad de V&V (Validación y Verificación)
    - Implica la ejecución del código
    - En otras actividades de V&V el código no se ejecuta:
      - revisiones (como *walkthroughs*) y pruebas formales

# *Objetivos de la prueba (II)*



- No sólo se prueba el código: tb. documentación y ayuda.
- La prueba no puede asegurar la ausencia de defectos: sólo puede demostrar que existen defectos en el software.
- Índice de la fiabilidad del software:
  - ⇒ se comporta de acuerdo a las especificaciones funcionales y no funcionales.

## 2. *Importancia de la prueba*

- No es una actividad secundaria:
  - Típicamente 30-40% del esfuerzo de desarrollo
  - En aplicaciones críticas (p.ej. control de vuelo, reactores nucleares),
    - ¡de 3 a 5 veces más que el resto de pasos juntos de la ingeniería del software!
  - El coste aproximado de los errores del software (*bugs*) para la economía americana es el equivalente al 0,6% de su PIB, unos 60.000 millones de dólares (NIST, *National Institute of Standards and Technology*, 2002)
    - ⇒ más de 1/3 podría evitarse si la prueba se efectuara mejor

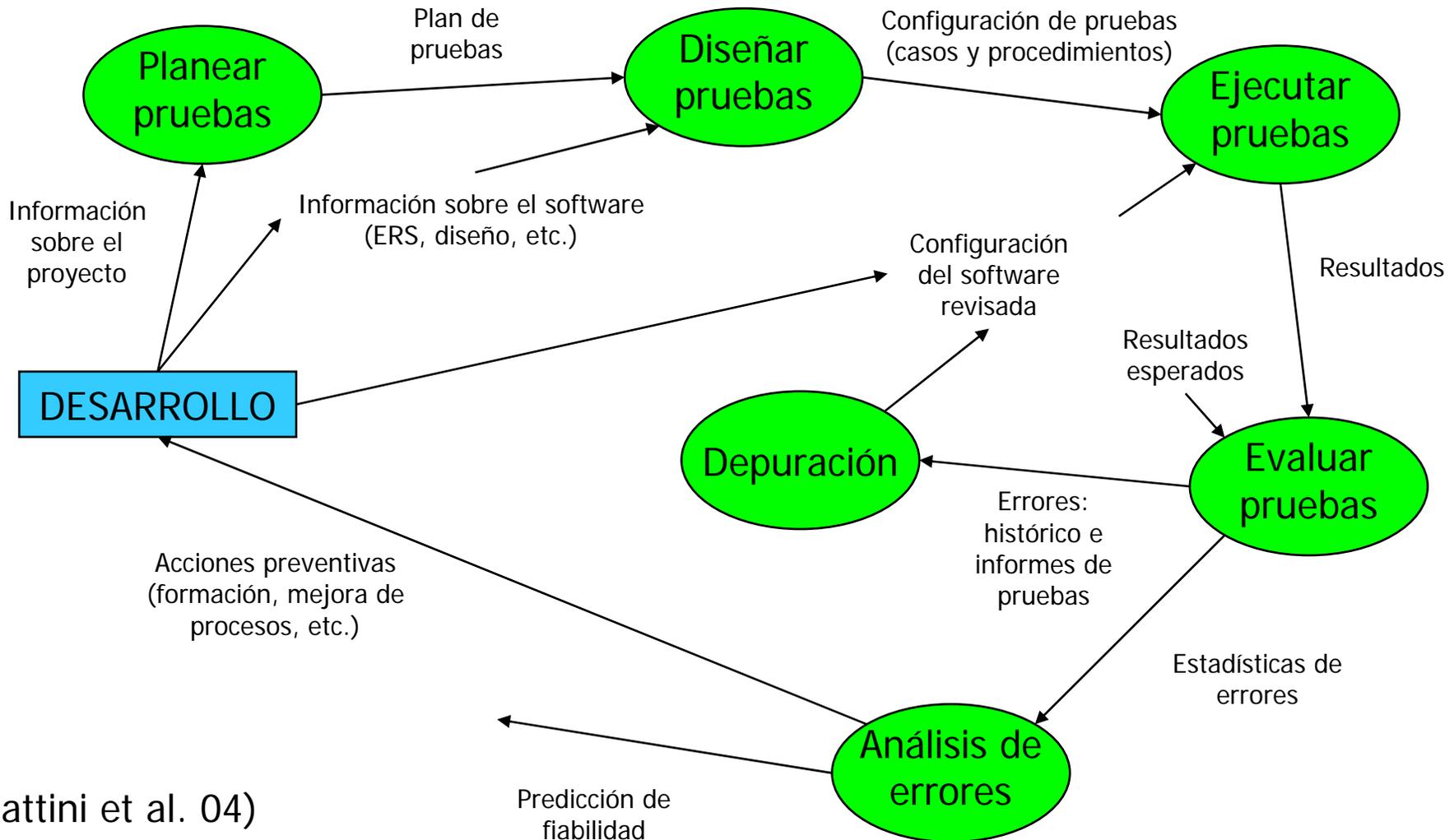
# 3. Principios de la prueba

(Davis 95, Myers 79) (Pressman 06)



- No son posibles las pruebas exhaustivas.
- Las pruebas deberían planificarse antes de que empiecen.
- No deben realizarse planes de prueba suponiendo que prácticamente no hay defectos en los programas y, por tanto, dedicando pocos recursos a las pruebas.
- Se deben evitar los casos de prueba no documentados ni diseñados con cuidado.
- A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos de los clientes (*trazabilidad*).
- El principio de Pareto es aplicable a la prueba del software (*“donde hay un defecto, hay otros”*).
- Las pruebas deberían empezar por “lo pequeño” y progresar hacia “lo grande”.
- Para ser más efectivas, las pruebas deberían ser conducidas por un equipo independiente.

# 4. El proceso de prueba

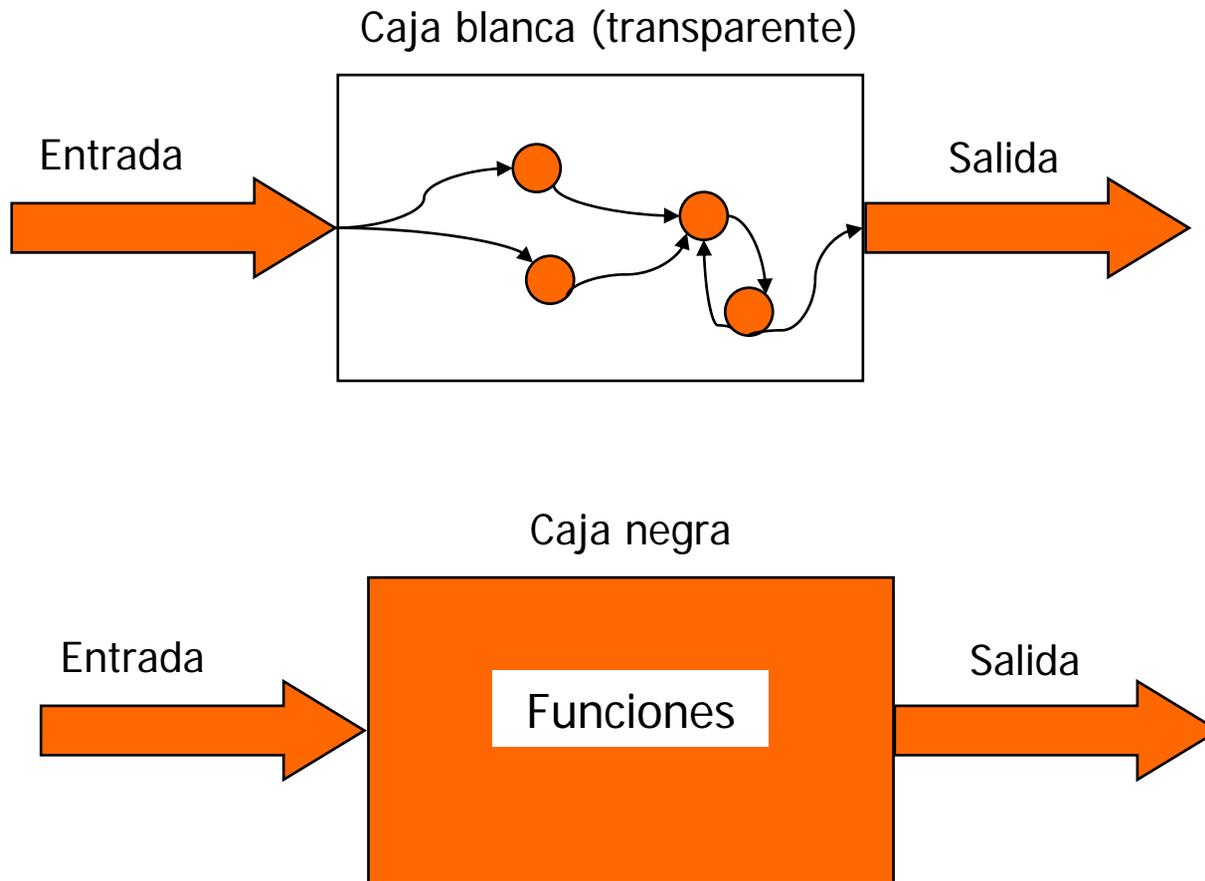


# 5. Métodos de diseño de casos de prueba

- Seguridad total: prueba exhaustiva
  - ⇒ *no practicable*
  - Por ejemplo, en cuanto al número de caminos posibles
    - (Pressman) Programa en C de 100 líneas con 2 bucles, de 1 a 20 iteraciones máximo, dentro del bucle interior, 4 if-then-else
      - ⇒  $10^{14}$  caminos posibles
    - (Myers) Programa de 50 líneas con 25 IF en serie:
      - ⇒  $2^{50} = 33,5$  millones de secuencias potenciales.
- Objetivo: conseguir confianza aceptable en que se encontrarán todos los defectos existentes, sin consumir una cantidad excesiva de recursos.
- *“Diseñar las pruebas que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y tiempo posible.”*

# *Diseño de casos de prueba.*

## *Enfoques principales*



# *Diseño de casos de prueba.*

## *Enfoques principales (II)*

### *a) Enfoque estructural o de caja blanca (transparente):*

- se centra en la estructura interna del programa para elegir los casos de prueba
- la prueba exhaustiva consistiría en probar todos los posibles caminos de ejecución
- $n^\circ$  caminos lógicos  $\uparrow\uparrow \Rightarrow$  buscar los más importantes

### *b) Enfoque funcional o de caja negra:*

- para derivar los casos, se estudia la especificación de las funciones, la entrada y la salida.
- No son excluyentes.

# 6. Enfoque estructural

⇒ *¿Porqué no dedicar todo el esfuerzo a probar que se cumplen los requisitos?*

⇒ *¿Porqué usar pruebas de caja blanca?*

- Los errores lógicos y las suposiciones incorrectas son inversamente proporcionales a la probabilidad de que se ejecute un camino del programa.
- A veces creemos que un camino lógico tiene pocas posibilidades de ejecutarse cuando puede hacerlo de forma normal.
- Los errores tipográficos son aleatorios.

*“Los errores se esconden en los rincones y se aglomeran en los límites” (Beizer 90)*

# *¿Bastaría con una prueba exhaustiva de caja blanca?*

Contraejemplo:

```
IF (X + Y + Z) / 3 = X THEN  
    print ("X, Y, Z son iguales")  
ELSE  
    print ("X, Y, Z no son iguales");
```

Caso 1: X=5, Y=5, Z=5

Caso 2: X=2, Y=3, Z=7

*Son casos de prueba que cumplen **cobertura de sentencias** pero no detectan ningún problema en el programa  $\Rightarrow$  las pruebas de caja blanca no son suficientes*

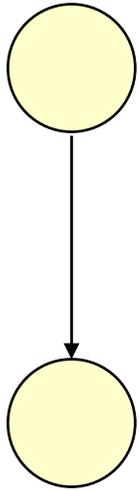
# Prueba del camino básico

(Mc Cabe 76) (Pressman 06) (Piattini et al. 04)

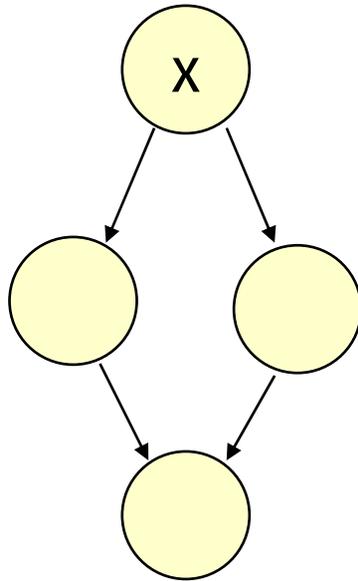
## Objetivos:

1. Obtener una medida de la complejidad lógica de un diseño procedimental  
⇒ *Complejidad ciclomática de Mc Cabe*
  2. Usar esa medida como guía para la definición de un conjunto básico de caminos de ejecución
- Los casos de prueba obtenidos garantizan que durante la prueba se ejecuta al menos una vez cada sentencia del programa (cobertura de sentencias).
  - Se puede automatizar.
  - Es una prueba de caja blanca y de caja negra  
⇒ *Si se aplica sobre la especificación, es de caja negra*
  - Otras pruebas de caja blanca son las siguientes:
    - Prueba de condición
    - Prueba de flujo de datos

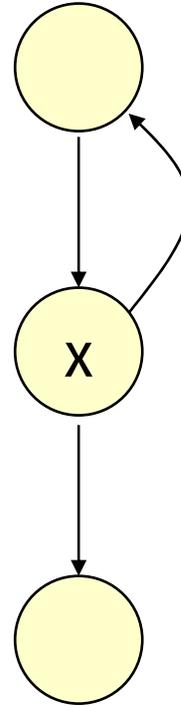
# Notación de grafo de flujo



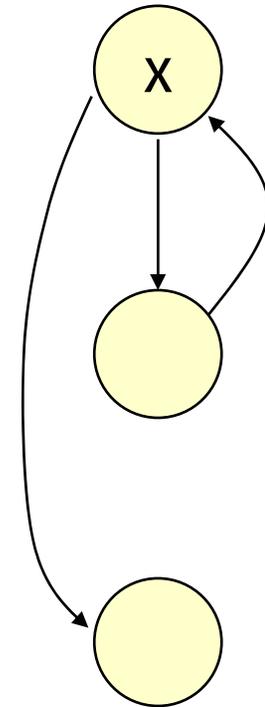
Secuencia



Si x  
entonces...

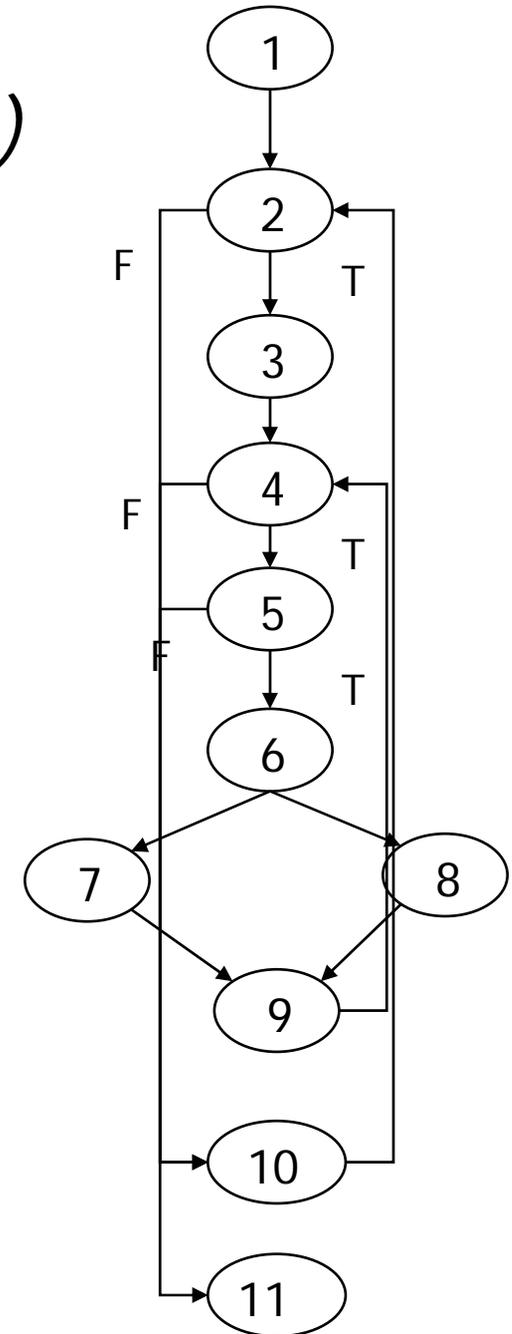
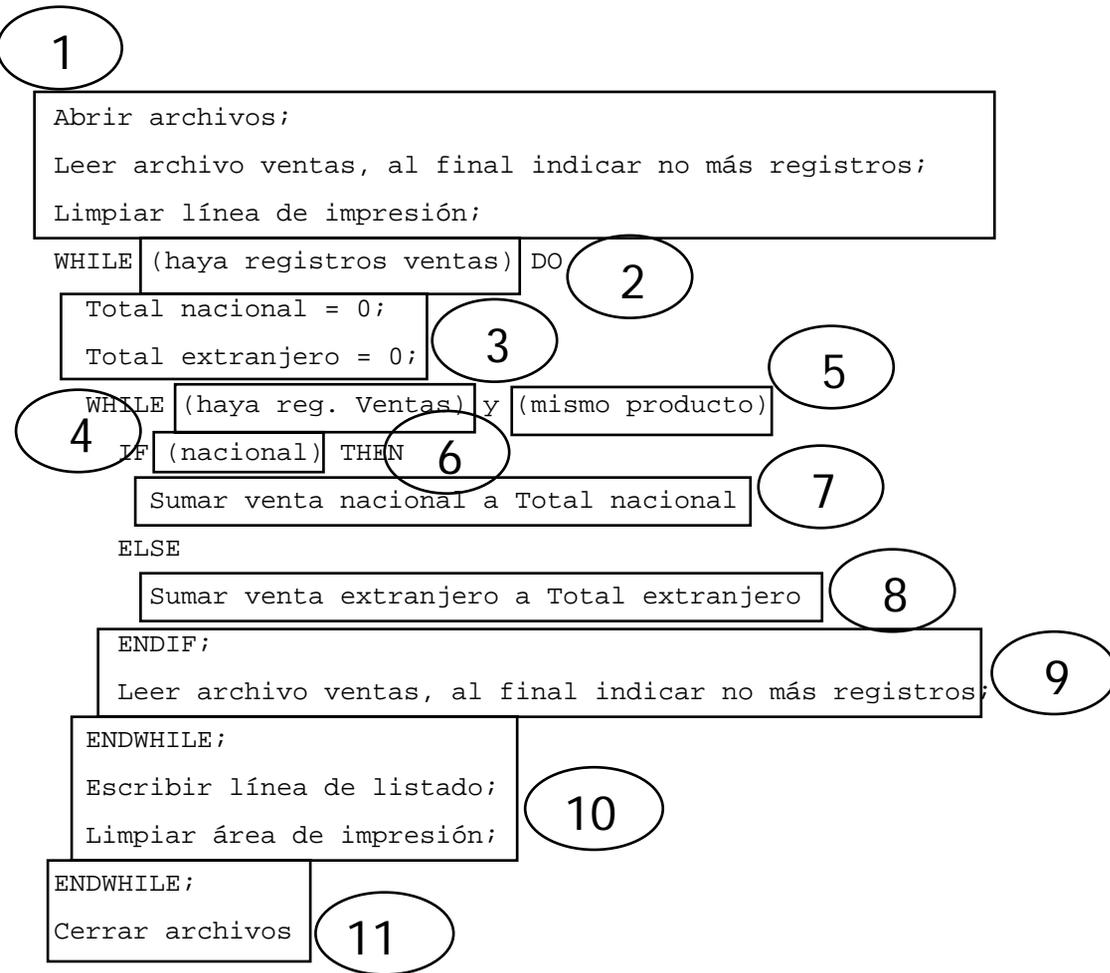


Hacer ...  
hasta x



Mientras x  
hacer...

# Grafo de flujo de un programa (pseudocódigo)



# *Prueba del camino básico.*

## *Definiciones*



- *Camino*: secuencia de sentencias encadenadas desde la sentencia inicial del programa hasta su sentencia final.
- *Camino de prueba*: un camino que atraviesa, como máximo, una vez el interior de cada bucle que encuentra.
  - Algunos autores hablan de pasar 3 veces:
    - una sin entrar en su interior
    - otra entrando una vez
    - otra entrando dos veces
- *Camino linealmente independiente* de otros: introduce por lo menos un nuevo conjunto de sentencias de proceso o una nueva condición
  - ⇒ en términos del grafo de flujo, introduce una nueva arista que no haya sido recorrida anteriormente a la definición del camino



# *Derivación de casos de prueba*



1. Usando el diseño o el código como base, dibujamos el correspondiente grafo de flujo.
2. Determinamos la complejidad ciclomática del grafo de flujo resultante,  $V(G)$ .
3. Determinamos un conjunto básico de **hasta**  $V(G)$  caminos linealmente independientes.
4. Preparamos los casos de prueba que forzarán la ejecución de cada camino del conjunto básico.

# Derivación de casos de prueba (II)

Algunos consejos:

- numerar los nodos del grafo secuencialmente
- dividir las condiciones compuestas en distintos nodos, uno por cada condición simple (así se obtiene cobertura de sentencias y de condiciones)
- describir el conjunto de caminos independientes (subrayar aristas que los hacen independientes de los anteriores)
  - 1-2-11
  - 1-2-3-4-...
  - ...
- crear los caminos “incrementalmente”: cuando se escogen caminos largos al principio, es más probable que no se puedan ejecutar en la práctica, y además se cubren todas las aristas del grafo con un número de caminos menor que  $V(G)$  (que recordemos es un valor máximo)
- a partir de los caminos, analizar el código para ver qué datos de entrada son necesarios, y qué salida se espera
- algunos caminos pueden ser imposibles  $\Rightarrow$  seleccionar otros
- algunos caminos no se pueden ejecutar solos y requieren la concatenación con otro

# *Complejidad ciclomática.*

## *Conclusiones*



Según (Beizer 90):

- $V(G)$  marca un límite **mínimo** del nº de casos de prueba, contando cada condición de decisión como un nodo individual.
- Parece que cuando  $V(G)$  es mayor que 10 la probabilidad de defectos en el módulo crece bastante si dicho valor alto no se debe a sentencias CASE

*⇒ en ese caso, es recomendable replantearse el diseño modular*

# Prueba de bucles (Beizer 90) (Pressman 06)

- Bucles simples (“n” es el n° máx. de iteraciones):
  - 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.
- Bucles no estructurados:
  - rediseñar primero.
- Bucles anidados (para evitar progresión geométrica):
  - Llevar a cabo las pruebas de bucles simples con el bucle más interior. Configurar el resto de bucles con sus valores mínimos.
  - Progresar hacia afuera, llevando a cabo pruebas para el siguiente bucle, manteniendo los bucles externos en sus valores mínimos y los internos en sus valores “típicos”.
- Bucles concatenados:
  - si no son independientes, como con los bucles anidados.

# 7. Enfoque funcional

- Estudia la especificación del software, las funciones que debe realizar, las entradas y las salidas.
- Busca tipos de errores diferentes a las pruebas de caja blanca:
  - *¿es el sistema particularmente sensible a ciertos datos de entrada?*
  - *¿qué volumen de datos tolerará el sistema?*
  - *¿qué efectos tendrán determinadas combinaciones de datos sobre el funcionamiento del sistema?*
- Un caso de prueba está bien elegido si:
  - reduce el n° de casos de prueba adicionales para alcanzar una prueba razonable
  - nos dice algo sobre la presencia o ausencia de *clases de errores*.

# *Particiones o clases de equivalencia* (Piattini et al. 04)



- Cada caso de prueba debe cubrir el máximo  $n^{\circ}$  de entradas.
- Debe tratarse el dominio de valores de entrada dividido en un  $n^{\circ}$  finito de clases de equivalencia
  - ⇒ la prueba de un valor representativo de la clase permite suponer “razonablemente” que el resultado obtenido será el mismo que probando cualquier otro valor de la clase
- Método de diseño:
  1. Identificación de clases de equivalencia.
  2. Creación de los casos de prueba correspondientes.

# *Paso 1. Identificar las clases de equivalencia*



1.1. Identificar las condiciones de las entradas del programa

1.2. A partir de ellas, se identifican las clases de equivalencia:

- De datos válidos
- De datos no válidos

1.3. Algunas reglas para identificar clases:

1. Por cada rango de valores, se especifica una clase válida y dos no válidas.
2. Si se especifica un n° de valores, se creará una clase válida y dos no válidas.
3. Si se especifica una situación del tipo “debe ser” o booleana, se identifica una clase válida y una no válida.
4. Si se especifica un conjunto de valores admitidos, **y el programa trata de forma distinta cada uno de ellos**, se crea una clase válida por cada valor, y una no válida.
5. **Si se sospecha que ciertos elementos de una clase no se tratan igual que el resto de la misma, debe dividirse en clases menores.**

# *Paso 1. Identificar las clases de equivalencia (II)*

## 1.3. Algunas reglas para identificar clases:

1. Por cada rango de valores, se especifica una clase válida y dos no válidas
  - (válida)  $1 \leq \text{número} \leq 49$ ; (no válidas)  $\text{número} < 1$ ,  $\text{número} > 49$
2. Si se especifica un n° de valores, se creará una clase válida y dos no válidas
  - (válida)  $\text{num propietarios} = 3$ ;  
(no válidas)  $\text{num propietarios} < 3$ ,  $\text{num propietarios} > 3$
3. Si se especifica una situación del tipo "debe ser" o booleana, se identifica una clase válida y una no válida
  - (válida) "El primer carácter es una letra"; (no válida) "(...) no es una letra"
  - (válida) "X es un número"; (no válida) "X no es un número"
4. Si se especifica un conjunto de valores admitidos, y el programa trata de forma distinta cada uno de ellos, se crea una clase válida por cada valor, y una no válida
  - Tres tipos de inmuebles: (válidas) "pisos", "chalets", "locales comerciales";
  - (no válida) "jklñ"

# *Paso 2. Identificar los casos de prueba*



- 2.1. Asignación de un número único a cada clase de equivalencia.
- 2.2. Hasta que todas las clases de equivalencia hayan sido cubiertas por casos de prueba, se tratará de escribir un caso que cubra tantas clases válidas no incorporadas como sea posible.
- 2.3. Hasta que todas las clases de equivalencia no válidas hayan sido cubiertas por casos de prueba, escribir un caso para una ÚNICA clase no válida sin cubrir.

*⇒ Ejemplo p.436 (Piattini et al. 04),  
"Aplicación bancaria"*

# Análisis de Valores Límite (AVL) *(Piattini et al. 04)*



- Los casos de prueba que exploran las condiciones límite producen mejores resultados  
*“Los defectos del software se acumulan en las situaciones límite”*
- Diferencias de AVL con particiones de equivalencia:
  - No se elige “cualquier elemento” de la clase de equivalencia, sino uno o más de manera que los márgenes se sometan a prueba.
  - Los casos de prueba se generan considerando también el espacio de salida.

# *Análisis de Valores Límite.*

## *Reglas para identificar casos*



1. Si una condición de entrada especifica un rango delimitado por los valores  $a$  y  $b$ , se deben generar casos para  $a$  y  $b$  y casos no válidos justo por debajo y justo por encima de  $a$  y  $b$ , respectivamente.
2. Si una condición de entrada especifica un número de valores, se deben desarrollar casos de prueba que ejerciten los valores máximo y mínimo, uno más el máximo y uno menos el mínimo.
3. Aplicar las directrices 1 y 2 a las condiciones de salida.
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.

# *Análisis de Valores Límite.*

## *Reglas para identificar casos (II)*

1. Si una condición de entrada especifica un rango delimitado por los valores  $a$  y  $b$ , se deben generar casos para  $a$  y  $b$  y casos no válidos justo por debajo y justo por encima de  $a$  y  $b$ , respectivamente
  - Rango de entrada:  $[-1.0, 1.0]$ 
    - Casos de prueba para  $-1.0, +1.0, -1.001, +1.001$  (si se admiten 3 decimales)
2. Si una condición de entrada especifica un número de valores, se deben desarrollar casos de prueba que ejerciten los valores máximo y mínimo, uno más el máximo y uno menos el mínimo
  - “El fichero de entrada tendrá de 1 a 255 registros”
    - Casos para 0, 1, 255, 256 registros
3. Aplicar las directrices 1 y 2 a las condiciones de salida
  - “El programa podrá mostrar de 1 a 4 listados”
    - Casos para intentar generar 0, 1, 4 y 5 listados

# 8. Prueba de Interfaces Gráficas de Usuario (GUI, Graphical User Interface)

Uso de una lista de chequeo preestablecida (Pressman 98), p.319:

- Para ventanas:
  - ¿Se abrirán las ventanas mediante órdenes basadas en el teclado o en un menú?
  - ¿Se puede ajustar el tamaño, mover y desplegar la ventana?
  - ¿Se regenera adecuadamente cuando se escribe y se vuelve a abrir?
  - ...
- Para menús emergentes y operaciones con el ratón:
  - ¿Se muestra la barra de menú apropiada en el contexto apropiado?
  - ¿Es correcto el tipo, tamaño y formato del texto?
  - ¿Si el ratón tiene varios botones, están apropiadamente reconocidos en el contexto?
  - ...
- Entrada de datos:
  - ¿Se repiten y son introducidos adecuadamente los datos alfanuméricos?
  - ¿Funcionan adecuadamente los modos gráficos de entrada de datos? (p.e. barra deslizante)
  - ¿Se reconocen adecuadamente los datos no válidos?
  - ¿Son inteligibles los mensajes de entrada de datos?

# 9. Estrategias de prueba del software *(Piattini et al. 04)*

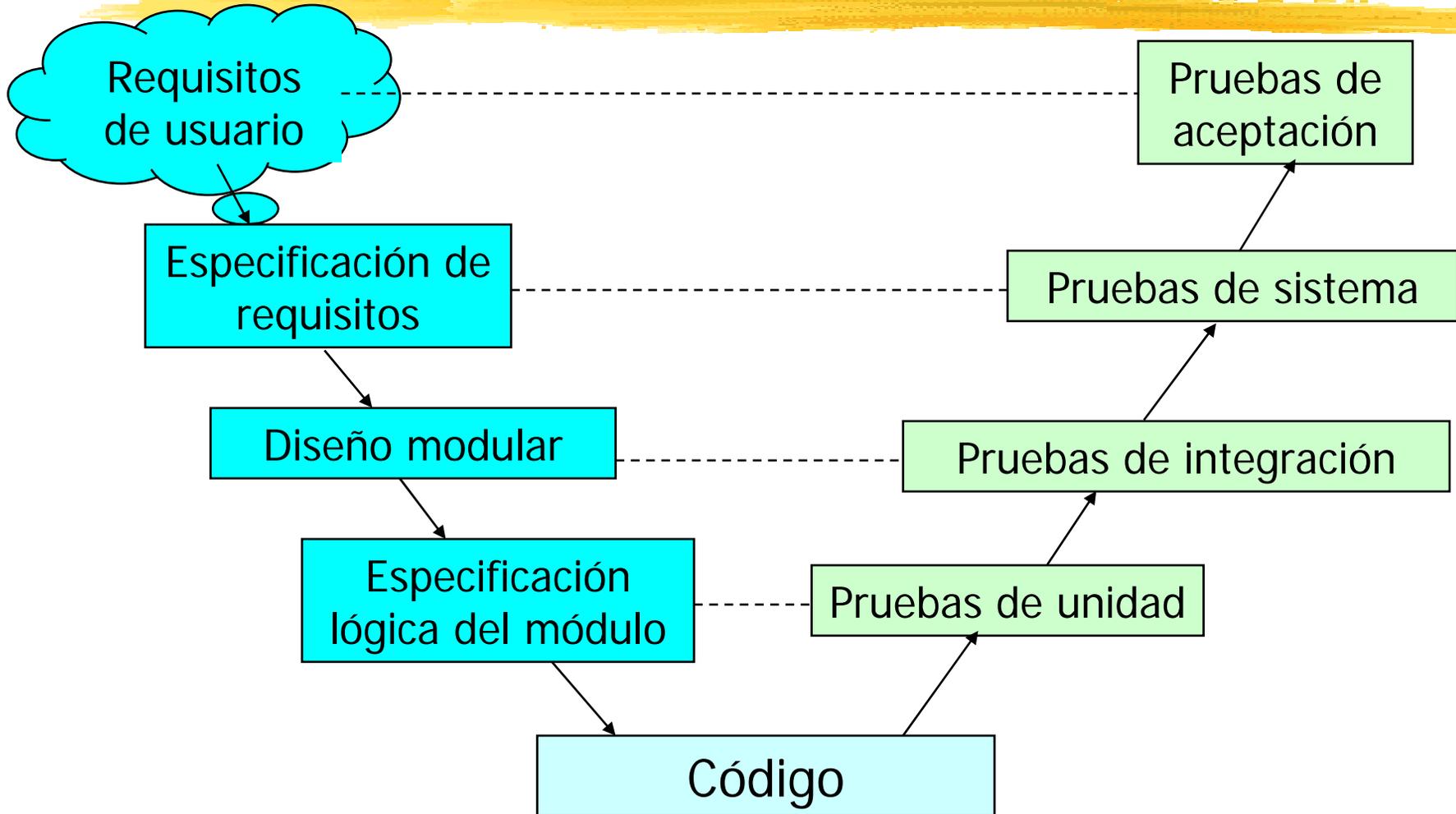


## Niveles de prueba:

1. **Prueba de unidad:** es la prueba de cada módulo, que normalmente realiza el propio personal de desarrollo en su entorno
2. **Prueba de integración:** con el esquema del diseño del software, los módulos probados se integran para comprobar sus interfaces en el trabajo conjunto
3. **Prueba de validación:** el software totalmente ensamblado se prueba como un todo para comprobar si cumple los requisitos funcionales y de rendimiento, facilidad de mantenimiento, recuperación de errores, etc. Coincide con la de sistema cuando el sw. no forma parte de un sistema mayor.
4. **Prueba del sistema:** el sw. ya validado se integra con el resto del sistema (rendimiento, seguridad, recuperación y resistencia)
5. **Prueba de aceptación:** el usuario comprueba en su propio entorno de explotación si lo acepta como está o no

# Relación entre productos de desarrollo y niveles de prueba

(Modelo en V del ciclo de vida)

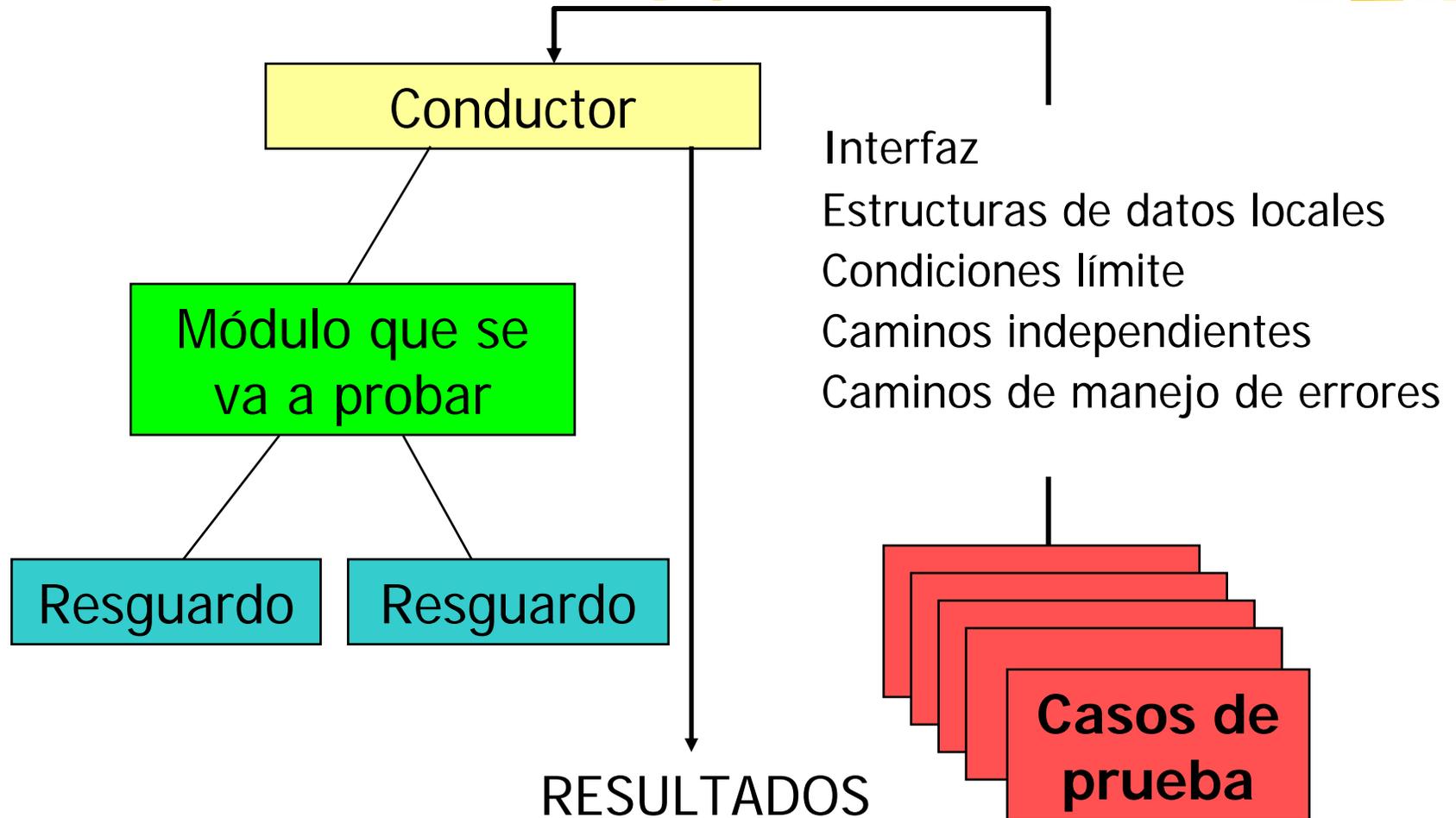


# *Organización para la prueba del software* (Pressman 06)



- El responsable de la codificación del sw. es siempre responsable de probar las unidades que ha escrito del programa.
- Muchas veces también se encarga de la prueba de integración.
- Cuando hay una arquitectura del sw. completa, debe entrar en juego un *Grupo Independiente de Prueba* (GIP), que garantice independencia (ha debido participar también en la especificación).
- El GIP es ayudado por el desarrollador.

# Prueba de unidad



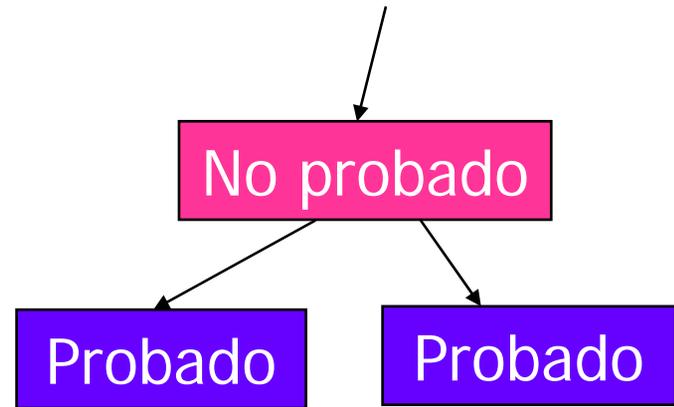
# *Prueba de unidad (II)*



- A menudo, se dice que está “orientada a caja blanca, aunque se puede completar con caja negra”
- Un enfoque posiblemente más realista:
  1. Diseñar un conjunto de pruebas de caja negra, basadas en la especificación del módulo
  2. Ejecutar las pruebas y comprobar la cobertura obtenida, usando una herramienta de pruebas:
    - Buscamos al menos  $C1 + C2$
    - C1 – Cobertura de sentencias
    - C2 – Cobertura de condición
  3. Completar con pruebas de caja blanca hasta alcanzar la cobertura deseada

# Prueba de integración

- Objetivo: prueba de los interfaces entre módulos
- Tipos de integración:
  - Incremental:
    - Se sigue el diseño modular
    - Se solapa con la prueba de unidad
    - Tipos:
      - ascendente
      - descendente
  - No incremental (*Big-bang*)
    1. Realizar todas las pruebas unitarias
    2. Realizar toda la integración
    - En este caso las pruebas unitarias no se solapan con las de integración
    - No recomendable en cualquier aplicación de tamaño medio o grande



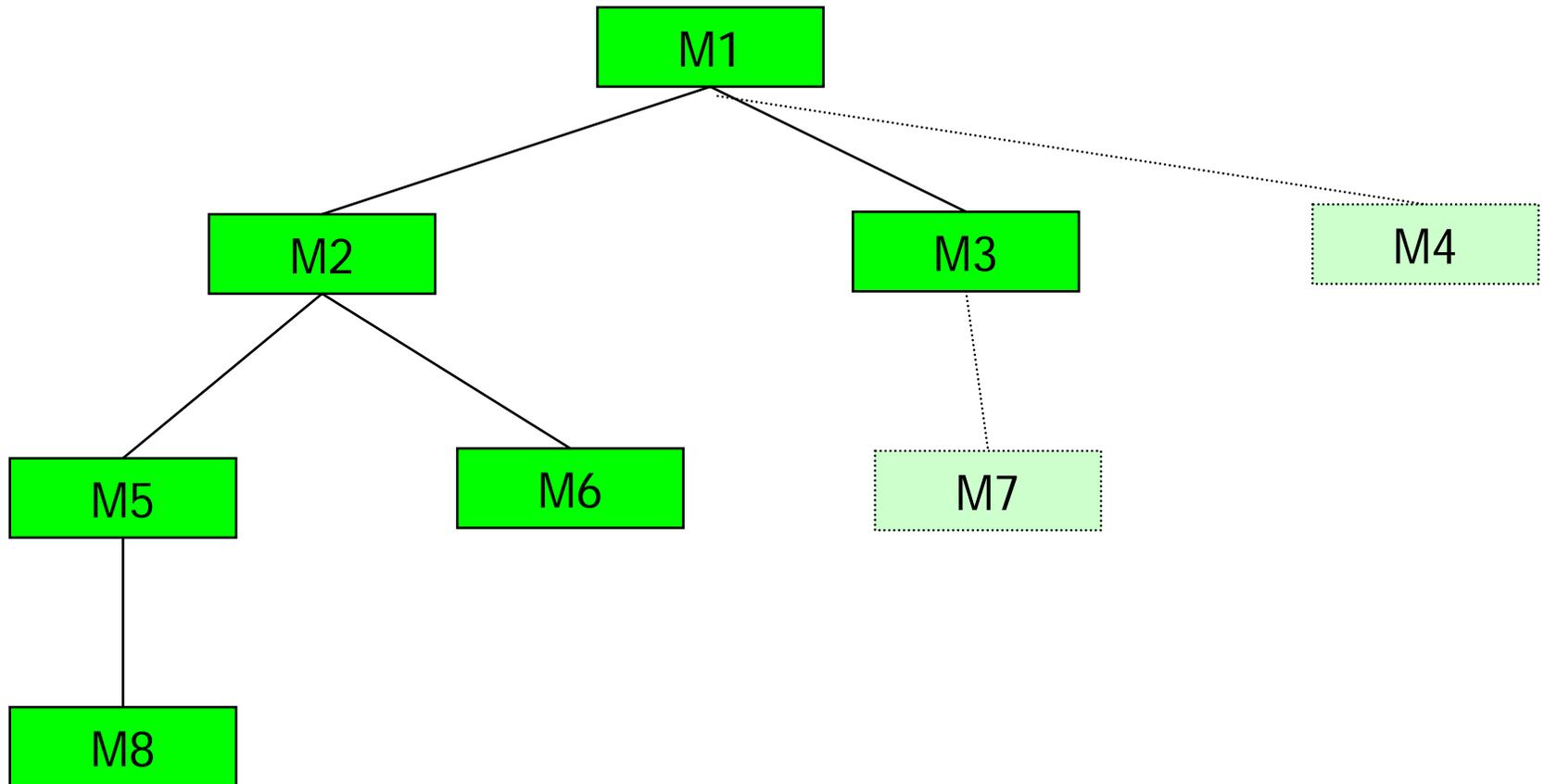
# *Integración incremental descendente*



- Se empieza con el módulo principal
- Se sustituyen los resguardos uno a uno
- Enfoques:
  - primero-en-profundidad
  - primero-en-anchura
  - Aunque a veces se puede retocar el enfoque...
    - Conviene probar pronto los módulos de E/S
    - Módulos críticos
- Se llevan a cabo las pruebas (uso de *resguardos*)
- Se hace la *prueba de regresión*

# *Integración descendente.*

## *Ejemplo* (Primero en profundidad)



# *Resguardo*



Es un “subprograma simulado” que usa la interfaz del módulo subordinado, lleva a cabo una mínima manipulación de datos, imprime una verificación de entrada y devuelve el control al módulo de prueba que lo invocó.

## Tipos de resguardos

- Mostrar un mensaje de traza
- Mostrar el parámetro pasado
- Devolver valor (de una tabla o archivo externo)
- Hacer una búsqueda en una tabla de un parámetro de entrada y devolver el parámetro de salida asociado

# Prueba de regresión

(Pressman 02) p.314 (Pressman 06) p.398

- Cada vez que se añade un nuevo módulo como parte de una prueba de integración, el sw. cambia  
⇒ Estos cambios pueden producir errores con funciones que antes trabajaban perfectamente
- Las pruebas de regresión consisten en volver a ejecutar un subconjunto de las pruebas realizadas anteriormente, con el fin de asegurarse de que los cambios no han propagado efectos laterales no deseados
- Fuera de la prueba de integración, las pruebas de regresión también se utilizan cuando cambia la *configuración del software (programa, documentación o datos)*  
⇒ p.ej. durante el mantenimiento correctivo

# *Integración descendente.*

## *Ventajas e inconvenientes*



### ■ Ventajas:

- Lo más importante es probar el control cuanto antes
  - Si la jerarquía está bien diseñada, es lo que se hace con esta aproximación

### ■ Inconvenientes:

- Si se requiere un procesamiento no trivial en los niveles más bajos de la jerarquía para probar los niveles superiores,

*¿qué se hace con los resguardos?*

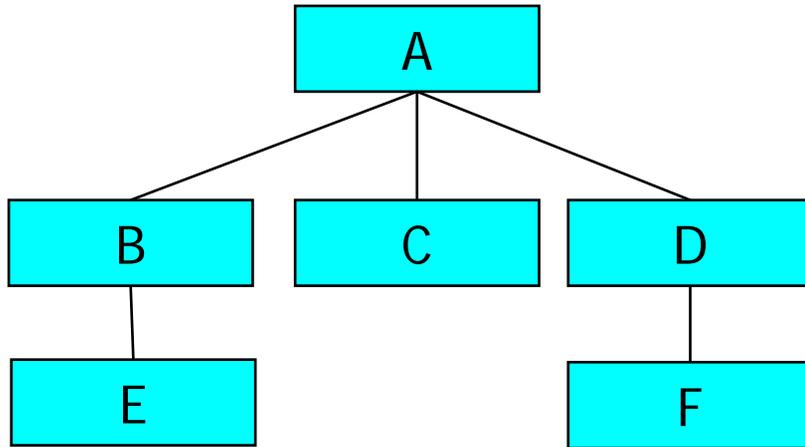
- Desarrollar resguardos con funciones limitadas ⇒ esfuerzo++
- Retrasar las pruebas hasta que los resguardos sean sustituidos por módulos reales
- Integrar el sw. desde abajo ⇒ integración ascendente

# *Integración incremental ascendente*

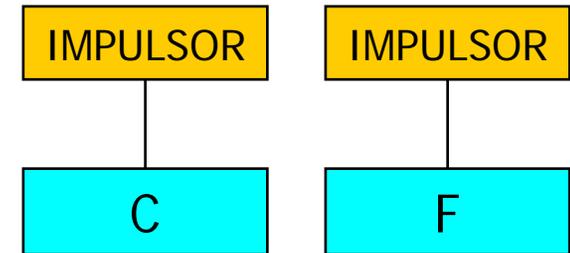


1. Se combinan los módulos de bajo nivel en grupos que realicen alguna subfunción
2. Se escribe (o se genera automáticamente) un controlador (impulsor) para coordinar la E/S de los casos de prueba
3. Se prueba el grupo
4. Se eliminan los controladores y se combinan los grupos moviéndose hacia arriba por la estructura del programa

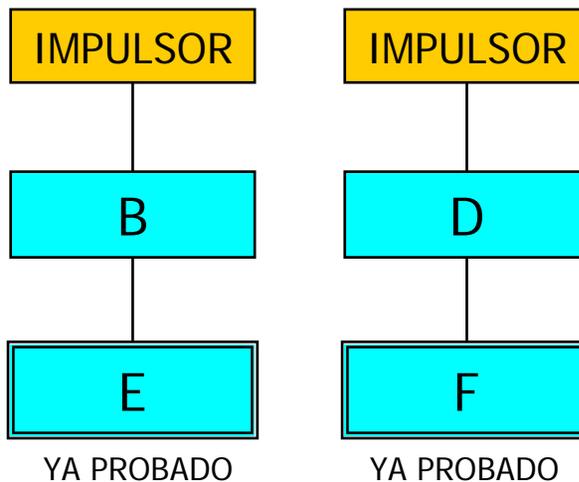
# Integración ascendente. Ejemplo



1ª FASE



2ª FASE

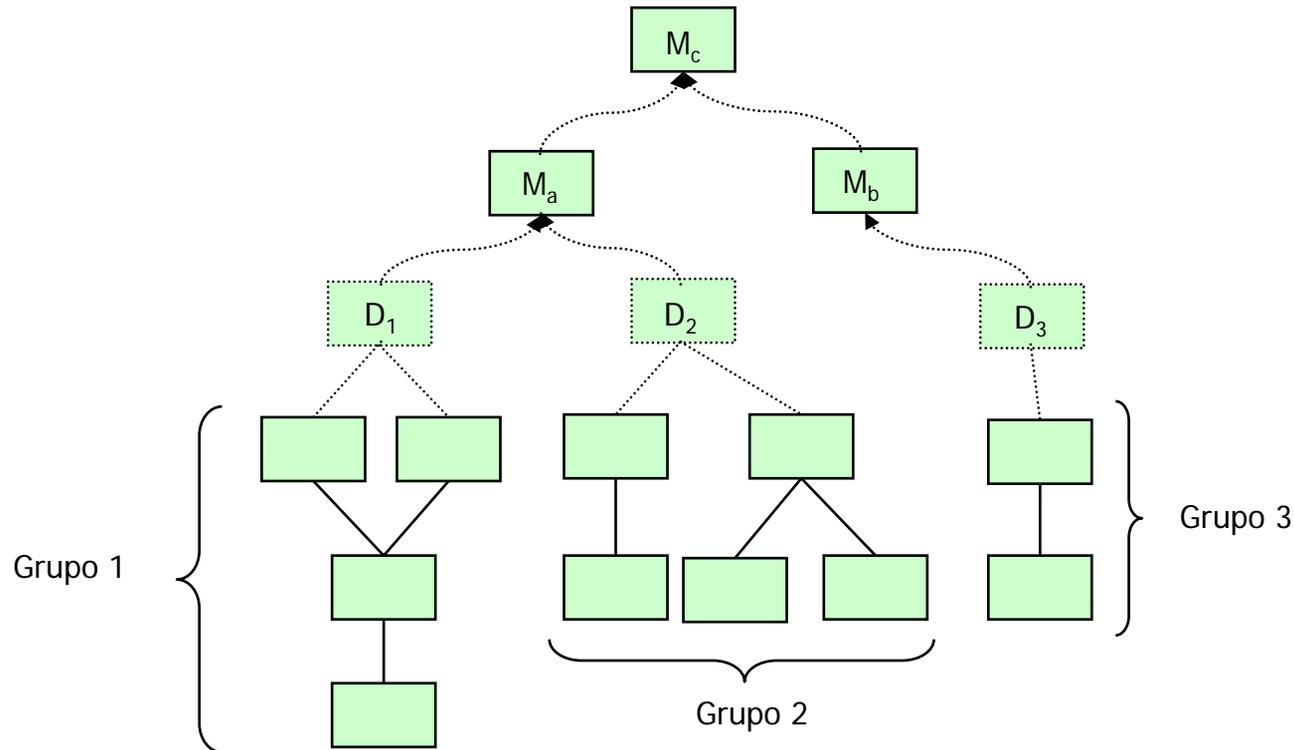


3ª FASE

Incorporación de A y prueba del programa completo. No requiere impulsor, pues todo el código de E/S está incorporado

# Integración ascendente. Ejemplo (II)

Se pueden formar grupos



# *Integración ascendente.*

## *Ventajas e inconvenientes*



### ■ Ventajas:

- No se necesitan resguardos

### ■ Inconvenientes:

- Hay que escribir controladores (impulsores)
  - Son menos complejos que los resguardos
  - Se pueden automatizar más fácilmente

⇒ *¿Hace falta la prueba de regresión en este tipo de integración?*

# *Integración mixta o "sandwich"*



- Se comienza en los módulos superiores de forma descendente
- Simultáneamente, se empieza de forma ascendente desde los módulos inferiores
- Se continúa hasta que ambas aproximaciones se encuentran en un nivel intermedio
- En la integración ascendente,
  - ⇒ si los dos niveles superiores de la estructura se prueban de forma descendente, se elimina la necesidad de muchos controladores

# *Módulos críticos*



- Deben probarse lo antes posible
- Son aquellos que:
  - Están dirigidos a varios requisitos del sw.
  - Tienen un mayor nivel de control
  - Son complejos o propensos a errores (p.ej., usar complejidad ciclomática como indicador)
  - Tienen unos requisitos de rendimiento muy definidos
- Las pruebas de regresión deben centrarse en los módulos críticos

# *Prueba de aceptación*



- La realiza el usuario
- Si el sw. se desarrolla como un producto que se va a usar por muchos usuarios, no es práctico realizar pruebas de aceptación formales para cada uno de ellos:
  - *pruebas alfa*: en el lugar de desarrollo pero por un cliente. El desarrollador observa y registra los problemas y errores.
  - *pruebas beta*: por los usuarios finales (o un conjunto limitado de ellos) en sus entornos de explotación. El cliente registra los errores e informa al desarrollador.