Validación y Verificación de software

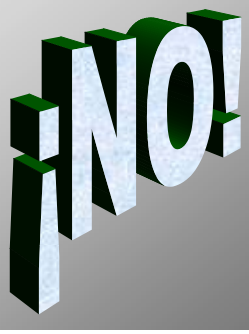
1. Introducción
2. Validación y Verificación
3. Proceso de prueba
4. Técnicas de prueba

Existe el mito de que no habría errores que pescar si fuésemos realmente buenos en programación. Si realmente nos pudiéramos concentrar, si todo el mundo usara programación estructurada, diseño descendente … entonces no habría errores. Ese es el mito. Hay errores, dice el mito, porque somos malos en lo que hacemos; y si lo somos, deberíamos sentirnos culpables por ello. Por tanto, la aplicación de pruebas y el diseño de casos de prueba es una admisión del fracaso, que inspira una buena dosis de culpa. Y el tedio de las pruebas es un justo castigo por nuestros errores. ¿El castigo por qué? ¿Por ser humanos? ¿Culpa por qué? ¿Por fracasar en lograr la perfección inhumana? ¿Por no distinguir entre lo que otro programador piensa y lo que dice? ¿Por no ser telépatas? ¿Por no resolver problemas de comunicación humana a los que se les ha dado la vuelta … durante siglos?

B. Beizer, 1990

¿Las pruebas deben inspirar culpa?

¿Las pruebas son realmente destructivas?



INTRODUCCIÓN

Las pruebas de software tienen dos objetivos principales

- Demostrar al desarrollador y al cliente que el software cumple con los requisitos
  - Prueba de validación
    Probar que el sistema funciona de manera correcta mediante un conjunto de casos de prueba, que refleje el uso esperado del sistema
    - Se descubrirán defectos en el sistema
- Encontrar situaciones en las que el comportamiento del software sea incorrecto, indeseable o no esté de acuerdo con su especificación
  - Prueba de defectos
    Los casos de prueba se diseñan para detectar defectos
    - Probarán que el programa cumple con los requisitos

Las pruebas pueden mostrar sólo la presencia de errores, más no su ausencia

VALIDACIÓN Y VERIFICACIÓN

Las pruebas son parte de un proceso más amplio de validación y verificación

VALIDACIÓN: ¿Construimos el producto correcto?

Finalidad
Garantizar que el software cumpla con las expectativas del cliente

VERIFICACIÓN: ¿Construimos bien el producto?

Finalidad
Comprobar que el software cumple con su funcionalidad y con los requisitos no funcionales establecidos

Comienzan

Tan pronto están disponibles lo requisitos y continúan en todas las etapas del desarrollo

VALIDACIÓN Y VERIFICACIÓN

El objetivo final de los procesos de validación y verificación es establecer confianza en que el sistema software es "adecuado"

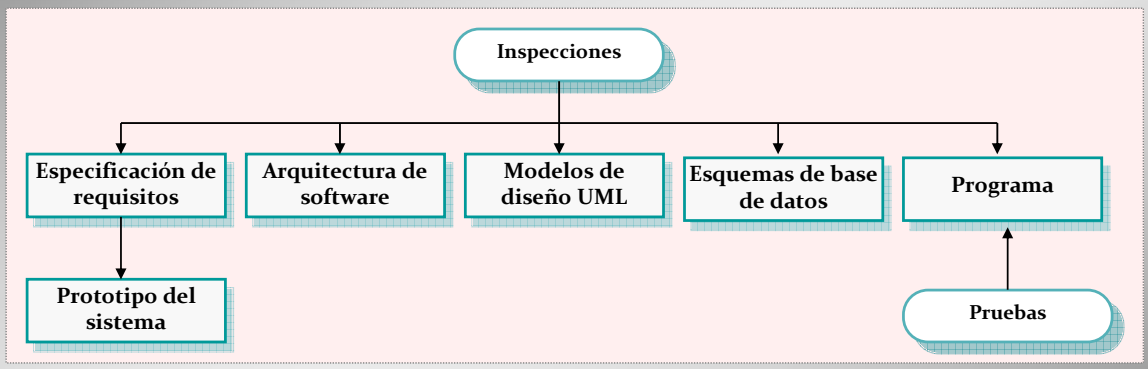
El nivel de confianza depende de:

- **Propósito del software**
  Cuanto más crítico sea el software, más importante debe ser su confiabilidad
- **Expectativas del usuario**
  Debido a su experiencia con software no confiable y cargado de errores, muchos usuarios tienen pocas expectativas sobre su calidad, por lo que no se sorprenden cuando falla
- **Entorno de mercado**
  Cuando un sistema se comercializa, los vendedores deben considerar los productos competitivos, el precio que los clientes están dispuestos a pagar y la fecha requerida de entrega

VALIDACIÓN Y VERIFICACIÓN

El proceso de validación y verificación implica **inspecciones y revisiones** de software

Técnicas estáticas
No es necesario ejecutar el software para verificarlo



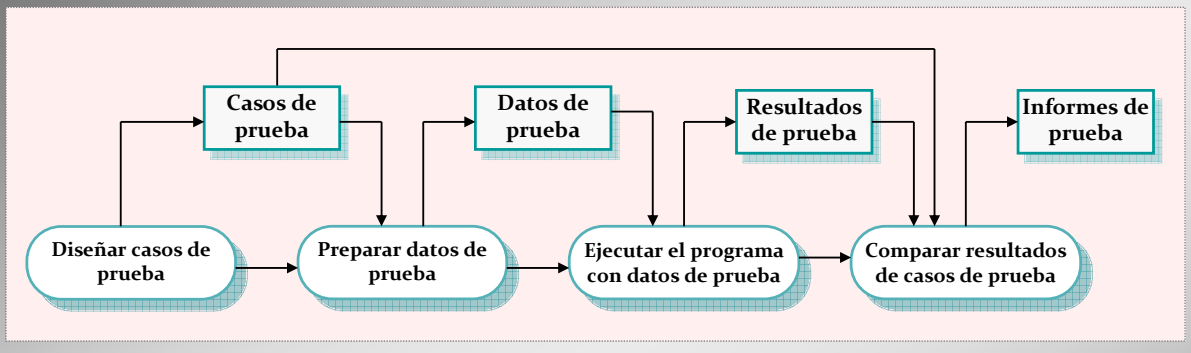
Ventajas de las inspecciones sobre las pruebas

- Durante las pruebas, los errores pueden enmascarar otros fallo
- Las versiones incompletas de un sistema se pueden inspeccionar sin coste añadido
- Además de buscar defectos de programa, pueden considerar también otros atributos de calidad

Desventajas

- No son eficaces para descubrir defectos debidos a interacciones entre las partes de un programa

PROCESO DE PRUEBA

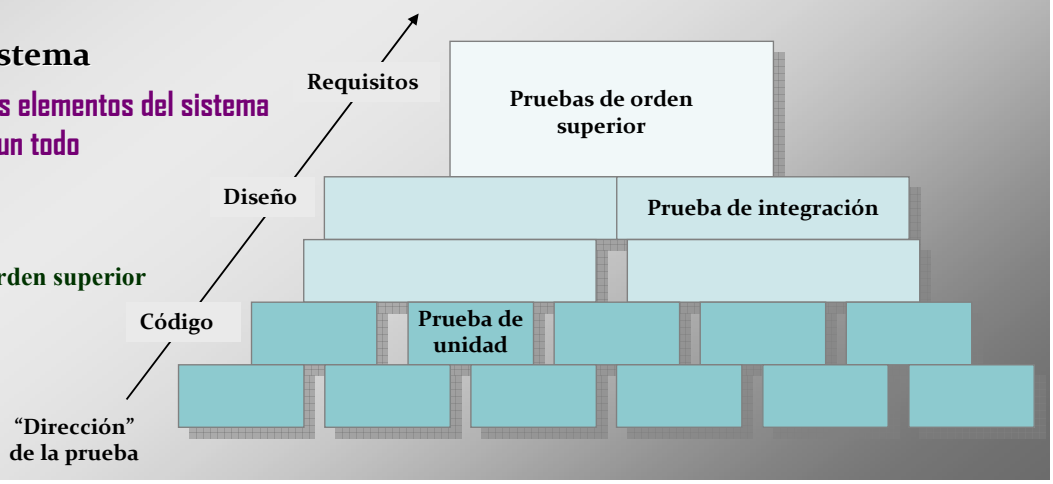


- **Casos de prueba**
  - Especificaciones de las entradas a la prueba y la salida esperada del sistema, además de información sobre los que se pone a prueba
  - Imposible generarlos de manera automática
- **Datos de prueba**
  - Entradas que se diseñan para probar un sistema
  - En ocasiones se pueden generar de forma automática

PROCESO DE PRUEBA

- **Prueba de unidad**
  Se concentra en cada unidad del software tal y como se implementó en el código fuente
- **Prueba de integración**
  Se concentra en el diseño y la construcción de la arquitectura del software
- **Prueba de validación** ⇢ Primera prueba de orden superior
  Se validan los requisitos establecidos como parte de su modelado confrontándose con el software que se ha construido
- **Prueba del sistema**
  El software y otros elementos del sistema se prueban como un todo

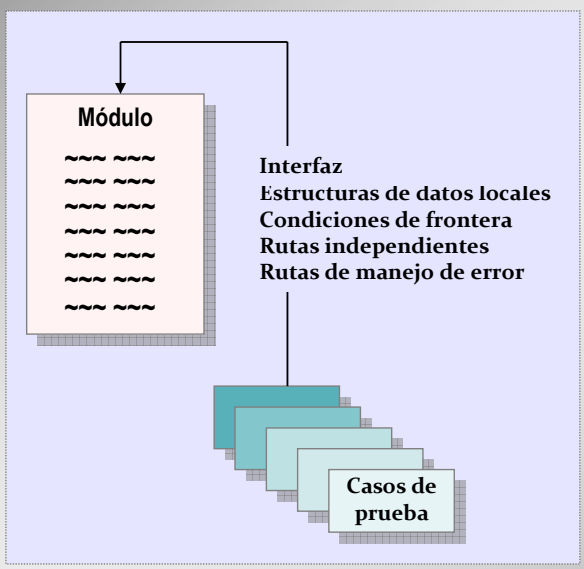
⇣ Segunda prueba de orden superior



PROCESO DE PRUEBA

Prueba de unidad

Enfoca los esfuerzos de verificación en la unidad más pequeña del diseño de software: el componente o módulo



- Se prueba la interfaz para garantizar que la información fluya de forma adecuada hacia y desde el módulo
- Se examinan las estructuras de datos locales para asegurar que los datos almacenados temporalmente mantienen su integridad durante toda la ejecución
- Se ejercitan todas las rutas de control independientes para asegurar que todas las instrucciones se ejecutan al menos una vez
- Se prueban las condiciones de frontera para asegurar que el módulo opera adecuadamente en los límites establecidos para restringir el procesamiento
- Se prueban todas las rutas para le manejo de errores

PROCESO DE PRUEBA

Procedimiento de prueba de unidad

- Las pruebas de unidad se diseñan antes o después de la codificación
- Revisar el diseño proporciona una guía para establecer casos de prueba
- Cada caso de prueba debe acoplarse con un conjunto de resultados esperados

Controlador

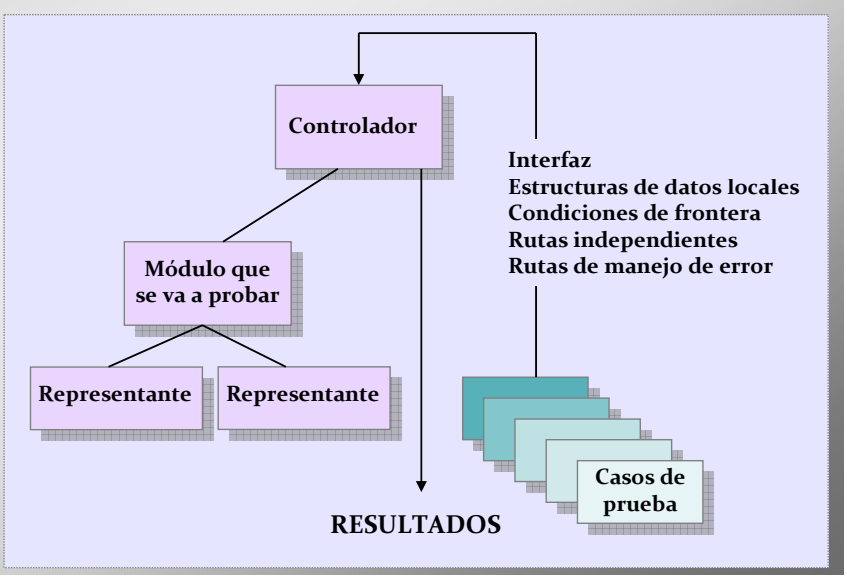
"Programa principal" que:

- acepta datos de casos de prueba
- pasa esos datos al módulo
- imprime resultados relevantes

Representante

Sustituye a módulos subordinados

- usa la interfaz de módulo subordinado
- realiza mínima manipulación de datos
- imprime verificación de entradas
- regresa el control al módulo



PROCESO DE PRUEBA

Prueba de integración

Técnica sistemática para construir la arquitectura del software mientras se llevan a cabo pruebas para descubrir errores asociados a la interfaz

Integración incremental

El programa se construye y prueba en pequeños incrementos:

- Los errores son más fáciles de aislar y corregir
- Las interfaces tienen más probabilidades de probarse por completo
- Puede aplicarse un enfoque de prueba sistemático

Estrategias de integración incremental

- Integración descendente

  Los módulos se integran de abajo hacia arriba, empezando con el módulo principal. Los módulos subordinados se incorporan en una forma primero en profundidad o primero en anchura

- Integración ascendente

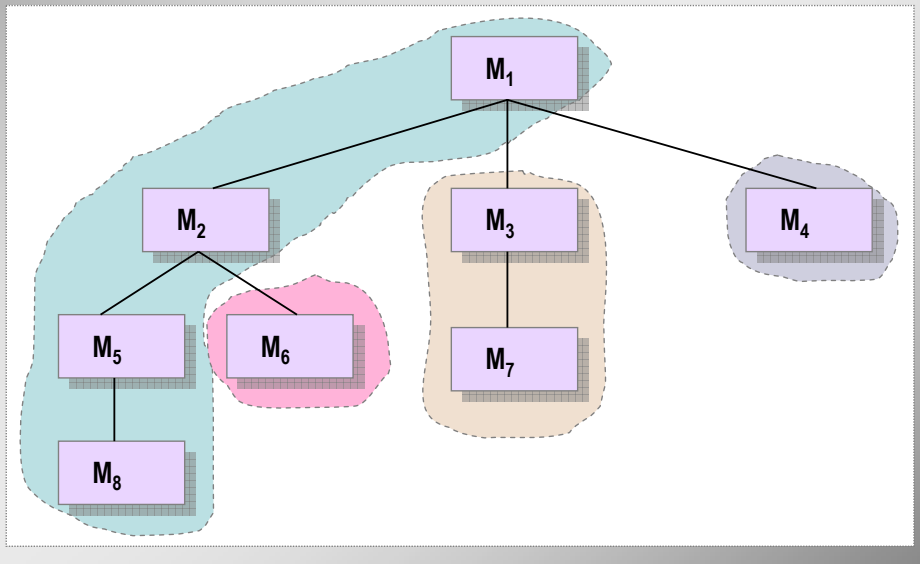
  La construcción y la prueba comienza con módulos atómicos

PROCESO DE PRUEBA

Integración descendente

La **integración primero en profundidad** integra todos los componentes sobre una ruta de control mayor de la estructura de programa

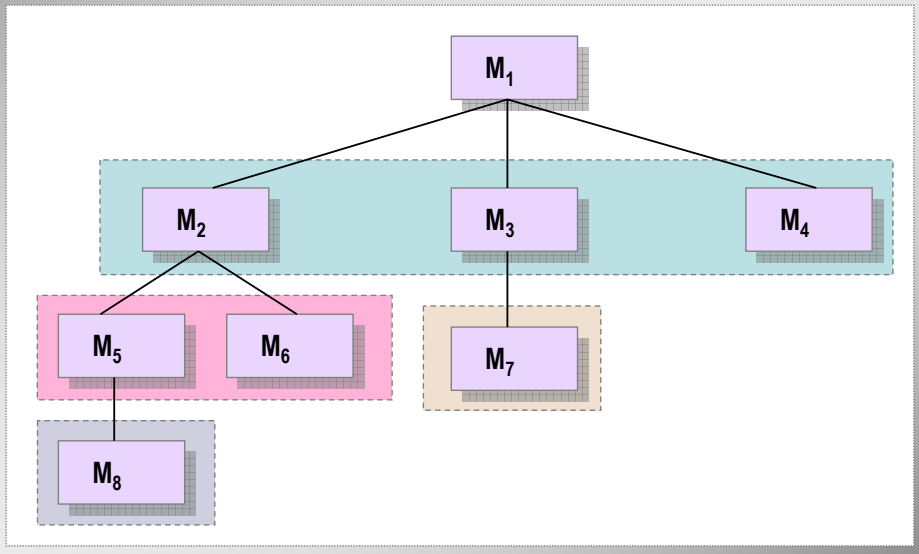
Ejemplo



PROCESO DE PRUEBA

La **integración primero en anchura** incorpora todos los módulos directamente subordinados en cada nivel y se mueve horizontalmente en la estructura

Ejemplo



PROCESO DE PRUEBA

Procedimiento

1. Recorrer la estructura de arriba hacia abajo, avanzando en profundidad o en anchura
2. Para probar un módulo
   usar el módulo del que depende como controlador, sustituir los módulos subordinados por representantes y realizar las pruebas específicas del módulo
3. Progresar substituyendo representantes por módulos reales realizando pruebas específicas para cada nuevo módulo y repitiendo las realizadas previamente (pruebas regresivas)

Ventajas

- Se prueban antes los módulo más importantes
- Si se avanza en profundidad se pueden probar subsistemas completos

Desventajas

- La necesidad de preparar los representantes

PROCESO DE PRUEBA

Integración ascendente

Procedimiento

1. Recorrer la estructura de abajo hacia arriba
2. Agrupar lo módulos inferiores
3. Preparar un controlador para cada grupo y realizar sus pruebas
4. Progresar sustituyendo los controladores por módulos reales realizando pruebas específicas y regresivas

Ventajas

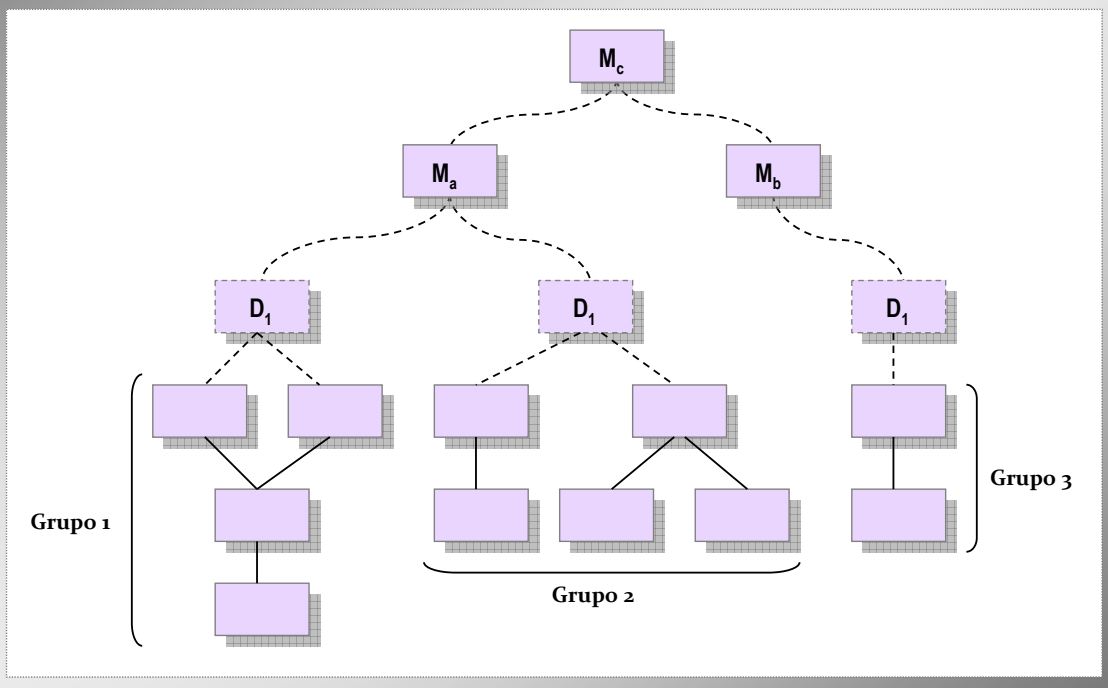
- No es necesario preparar representantes

Desventajas

- Los módulos más importantes se prueban al final, lo cual genera incertidumbre

PROCESO DE PRUEBA

Ejemplo



PROCESO DE PRUEBA

Prueba de validación

En el nivel de validación, las pruebas se enfocan en las acciones visibles para el usuario y las salidas del sistema reconocibles por el usuario

- Diseñar casos de prueba para garantizar que (criterios de validación)
  - Se satisfacen todos los requisitos de funcionamiento
  - Se logran todas las características de comportamiento
  - Todo el contenido es preciso y se presenta de manera adecuada
  - Se logran todos los requisitos de rendimiento
  - La documentación es correcta y se satisfacen la facilidad de uso y otros requisitos
- Prueba alfa
  - Se lleva a cabo en el sitio del desarrollador por un grupo representativo de usuarios finales
- Prueba beta
  - Se realiza en uno o más sitios del usuario final. Por lo general el desarrollador no está presente
- Prueba de aceptación
  - El software se entrega a un cliente bajo contrato

PROCESO DE PRUEBA

Prueba del sistema

Es una serie de diferentes pruebas cuyo propósito principal es ejecutar por completo el sistema basado en computadora

- **Pruebas de recuperación**
  Fuerzan al sistema a fallar en varias formas y verifican que la recuperación se realiza adecuadamente
- **Pruebas de seguridad**
  Verifican que los mecanismos de seguridad del sistema lo protegerán de cualquier penetración impropia
- **Pruebas de esfuerzo**
  Ejecutan un sistema de forma que demanda recursos en cantidad, frecuencia o volumen anormales
- **Pruebas de rendimiento**
  Con frecuencia se aparean con las pruebas de esfuerzo y, por lo general, requieren instrumentación hardware y software para medir el uso de recursos
- **Pruebas de despliegue**
  Ejercita el software en cada entorno en el que debe operar

TÉCNICAS DE PRUEBA

La meta de una técnica de prueba es encontrar errores y una buena técnica es aquella que tiene una alta probabilidad de encontrar uno

Características de una buena técnica de prueba

- **Alta probabilidad de encontrar un error**
  El examinador debe comprender el software e intentar desarrollar una imagen mental de cómo puede fallar
- **No es redundante**
  El tiempo y los recursos de la prueba son limitados
- **Es "la mejor de la camada"**
  Debe usarse la técnica de prueba que tenga mayor probabilidad de descubrir toda una clase de errores
- **No debe ser demasiado simple o demasiado compleja**
  Cada prueba debe ejecutarse por separado

TÉCNICAS DE PRUEBA

Clasificación de las técnicas de prueba

- **Pruebas de caja blanca**
  - Se basan en el examen cercano de los detalles de procedimiento
    - Se revisan conjuntos específicos de condiciones y/o bucles

Garantizan que todas las operaciones internas se realizan según las especificaciones

- **Pruebas de caja negra**
  - Se llevan a cabo en la interfaz del software
    - No se preocupan por la lógica interna del software

Demuestran que cada función es completamente operativa

TÉCNICAS DE PRUEBA

Pruebas de caja blanca

Filosofía de diseño de casos de prueba que usa la estructura de control descrita como parte del diseño a nivel de componentes para derivar los casos de prueba

Características de los casos de prueba que se derivan

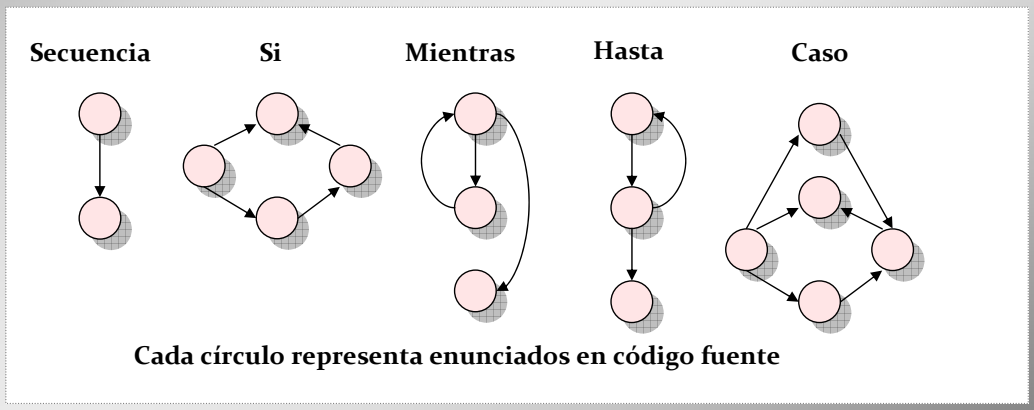
- Garantizan que todos los caminos independientes dentro de un módulo se revisan al menos una vez
- Revisan todas las decisiones lógicas en sus lados verdadero y falso
- Ejecutan todos los bucles en sus fronteras y dentro de sus fronteras operativas
- Revisan estructuras de datos internas para garantizar su validez

TÉCNICAS DE PRUEBA

Prueba de los caminos básicos

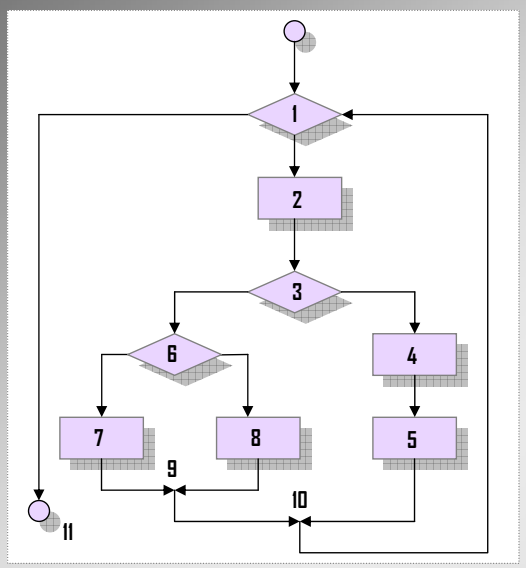
Técnica que permite derivar una medida de la complejidad lógica del diseño de un procedimiento y usarla como guía para definir un conjunto básico de caminos de ejecución

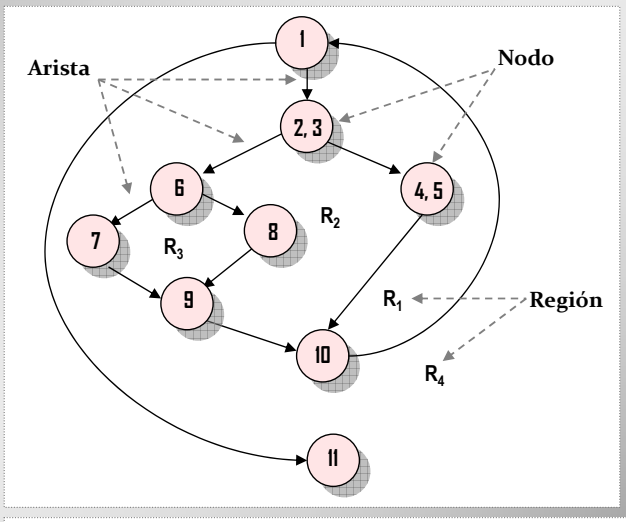
Grafo de flujo



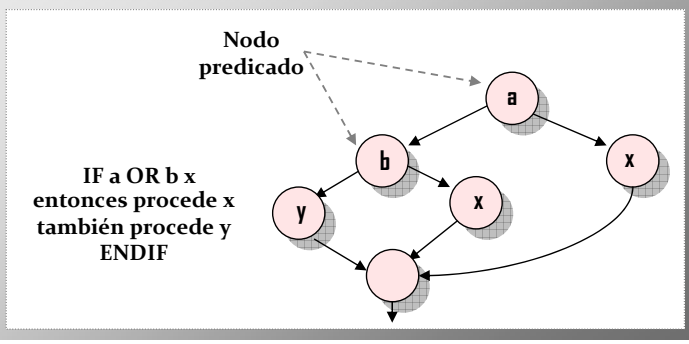
Cada círculo representa enunciados en código fuente

TÉCNICAS DE PRUEBA





Lógica compuesta



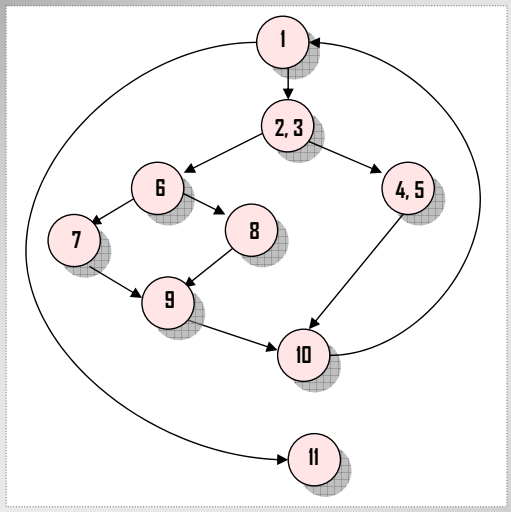
TÉCNICAS DE PRUEBA

Caminos de programa independientes

Un **camino independiente** es cualquier camino que introduce al menos un nuevo conjunto de enunciados de procesamiento o una condición en el programa

Debe moverse a lo largo de al menos una arista que no se haya recorrido antes de definir el camino

Ejemplo



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

Conjunto básico

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

¿Es independiente?

¿Cuántos caminos buscar?

TÉCNICAS DE PRUEBA

Complejidad ciclomática

Medición del software que proporciona una evaluación cuantitativa de la complejidad lógica de un programa

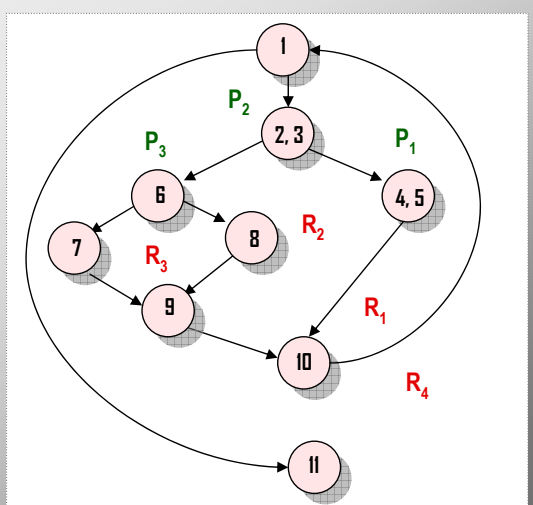
Define el número de caminos independientes del conjunto básico

Se calcula como:

- El número de regiones del grafo de flujo V(G)
- V(G) = E – N + 2
  E = número de aristas
  N = número de nodos
- V(G) = P + 1
  P = número de nodos predicado

- El grafo de flujo tiene 4 regiones
- V(G) = 11 aristas – 9 nodos + 2 = 4
- V(G) = 3 nodos predicado + 1 = 4

Ejemplo



TÉCNICAS DE PRUEBA

Derivación de casos de prueba

Pasos a seguir

1. Representar el flujo de control de módulo mediante un grafo de flujo
2. Calcular la complejidad ciclomática del grafo de flujo resultante
3. Determinar un conjunto básico de caminos independientes
4. Preparar casos de prueba que fuercen la ejecución de cada camino

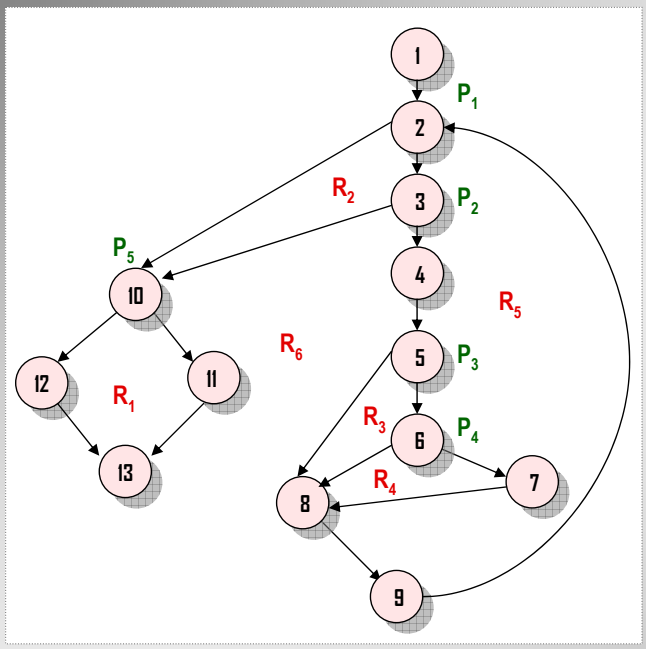
TÉCNICAS DE PRUEBA

EJEMPLO

```
PROCEDIMIENTO media;
  *  Este procedimiento calcula la media de 100 o menos
     números que se encuentran entre unos límites; también
     calcula el total de entradas válidas
  INTERFACE RETURNS media, total.input, total.valid;
  INTERFACE ACCEPTS value, minimum, maximun;
  TYPE value[1:100] IS SCALAR ARRAY;
  TYPE media, total.input, total.valid;
     minimum, maximun, sum IS SCALAR;
  TYPE i IS INTEGER;
1 i = 1;                                          2
  total.input = total.valid = 0;
  sum = 0;
  DO WHILE value[i] <> -999 AND total.input < 100   3
4   increment total.input by 1;
      IF value[i] > = minimum AND value[i] < = maximum   6
5       THEN increment total.valid by 1;
7          sum = sum + value[i];
        ELSE skip
8   ENDIF
    increment i by 1;
9 ENDDO
  IF total.valid > 0;                             10
11   THEN media = sum /total.valid;
12 ELSE media = -999;
13 ENDIF
END media;
```

TÉCNICAS DE PRUEBA

1 Representar el grafo de flujo



2 Calcular complejidad ciclomática

- V(G) = 6 regiones
- V(G) = 17 aristas – 13 nodos + 2 = 6
- V(G) = 5 nodos predicado + 1 = 6

3 Determinar conjunto básico

Camino 1: 1-2-10-11-13

Camino 2: 1-2-10-12-13

Camino 3: 1-2-3-10-12-13

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-...

Los ... significan que es aceptable cualquier camino a través de la estructura de control

TÉCNICAS DE PRUEBA

4 Preparar casos de prueba

Camino 1: 1-2-10-11-13

value (k) = entrada válida, con k < i

value (i) = -999, donde 2 ≤ i ≤ 100

Resultados esperados

Media correcta sobre los k valores y totales adecuados

Se debe probar como parte de las pruebas de los caminos 4, 5 y 6

Camino 2: 1-2-10-12-13

value (1) = - 999

Resultados esperados

Media = - 999

Otros totales con sus valores iniciales

```
1  i = 1;                                          2
   total.input = total.valid = 0;
   sum = 0;
   DO WHILE value[i] <> -999 AND total.input < 100   3
4     increment total.input by 1;
      IF value[i] > = minimum AND value[i] < = maximum   6
5        THEN increment total.valid by 1;
7             sum = sum + value[i];
         ELSE skip
8     ENDIF
      increment i by 1;
9  ENDDO
   IF total.valid > 0;                             10
11    THEN media = sum /total.valid;
12 ELSE media = -999;
13 ENDIF
   END media;
```

TÉCNICAS DE PRUEBA

4 Preparar casos de prueba

Camino 3: 1-2-3-10-12-13

Intento de procesar 101 o más valores
Los primeros 100 valores deben ser válidos

Resultados esperados

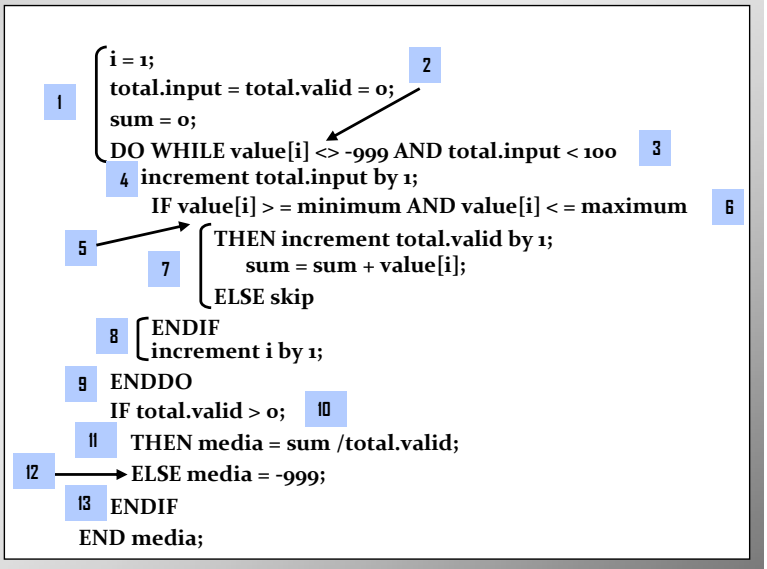
Media correcta sobre los k valores y totales adecuados

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

value (i) = entrada válida, con i < 100
value (k) < minimum, para k < i

Resultados esperados

Media correcta sobre los k valores y totales adecuados



TÉCNICAS DE PRUEBA

4 Preparar casos de prueba

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

value (i) = entrada válida, con i < 100
value (k) > maximun, para k ≤ i

Resultados esperados

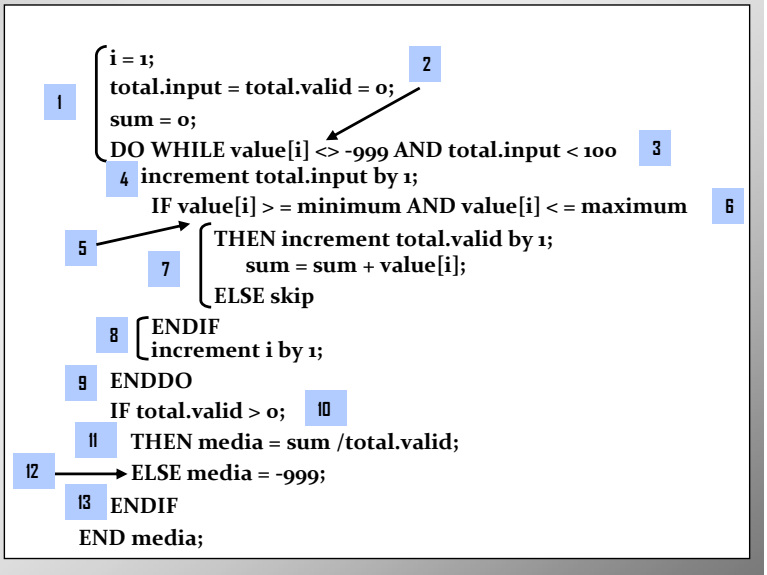
Media correcta sobre los n valores y totales adecuados

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

value (i) = entrada válida, con i < 100

Resultados esperados

Media correcta sobre los n valores y totales adecuados



TÉCNICAS DE PRUEBA

Pruebas de caja negra

Permiten derivar conjuntos de condiciones de entrada que revisarán por completo todos los requisitos funcionales de un programa

Intentan detectar

- Funciones incorrectas o faltantes
- Errores de interfaz
- Errores en las estructuras de datos o en el acceso a base de datos
- Errores de comportamiento o rendimiento
- Errores de inicialización o terminación

Los casos de prueba satisfacen los siguientes criterios

- Reducen el número de casos de prueba adicionales que se deben diseñar para lograr pruebas razonables
- Dicen algo sobre la presencia o ausencia de clases de errores

---

TÉCNICAS DE PRUEBA

Prueba de la partición de equivalencia

Técnica que divide el dominio de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba

- **El diseño de casos de pruebas**
  se basa en la evaluación de las clases de equivalencia para una condición de entrada
- **Una clase de equivalencia**
  representa un conjunto de estados válidos o inválidos para condiciones de entrada
- **Las clases de equivalencia se definen con los siguientes criterios**
  Si una condición de entrada especifica un rango
  se define una clase de equivalencia válida y dos inválidas
  Si una condición de entrada requiere un valor específico
  se define una clase de equivalencia válida y dos inválidas
  Si una condición de entrada especifica un miembro de un conjunto
  se define una clase de equivalencia válida y una inválida
  Si una condición de entrada es booleana
  se define una clase de equivalencia válida y una inválida

TÉCNICAS DE PRUEBA

Ejemplo

Se tiene un módulo con los siguientes argumentos:

NombreArtículo

String entre 2 y 15 caracteres alfanuméricos

Peso [5]

array de 5 elementos reales que representan los posibles pesos, entre 0 y 10.000 gramos, del artículo en cuestión. Estos pesos están ordenados de menor a mayor y si el artículo sólo está disponible en tres pesos, los dos primeros elementos estarán a 0 y los tres últimos a valores distinto de cero

TÉCNICAS DE PRUEBA

- Definir clases de equivalencia

NombreArtículo ····→ Requiere un miembro de un conjunto

- Clase válida — Alfanumérico: AcdEf4
- Clase inválida — No Alfanumérico: A$%!1

Longitud de NombreArtículo ····→ Requiere un rango

- Clase válida — 2 ≤ L ≤ 15: afdHteKJM14
- Clase inválida — L < 2: a
- Clase inválida — L > 15: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Rango de valores para el peso ····→ Requiere un rango

- Clase válida — 0 ≤ Peso ≤ 10.000: 500
- Clase inválida — Peso < 0: -2
- Clase inválida — Peso > 10.000: 11.000

Orden de los valores en el array ····→ Requiere un valor booleano

- Clase válida — Elementos ordenados: [0, 0, 1, 5, 10]
- Clase inválida — Elementos no ordenados: [1, 0, 10, 5, 0]

TÉCNICAS DE PRUEBA

- Derivar casos de prueba

Caso 1

"abcd", [0, 1, 2, 3, 4]

Resultados esperados

Ejecución del módulo sin problemas

Caso 2

"abcd", [0, 100, 200, 300, 400, 11.000]

Resultados esperados

Salida de error "peso no válido"

Caso 3

"abcd", [-1, 0, 2, 3, 4]

Resultados esperados

Salida de error "peso no válido"

Caso 4

"abcd", [1, 0, 10, 5, 4]

Resultados esperados

Salida de error "valores desordenados"

Caso 5

"$%3", [0, 0, 0, 0, 1]

Resultados esperados

Salida de error "nombre no válido"

Caso 6

"a", [0, 0, 0, 0, 1]

Resultados esperados

Salida de error "longitud de nombre no válido"

Caso 7

"aaaaaaaaaaaaaaaaaa", [0, 0, 0, 0, 1]

Resultados esperados

Salida de error "longitud de nombre no válido"