Validación y Verificación de Software

Testing

La Crisis del Software

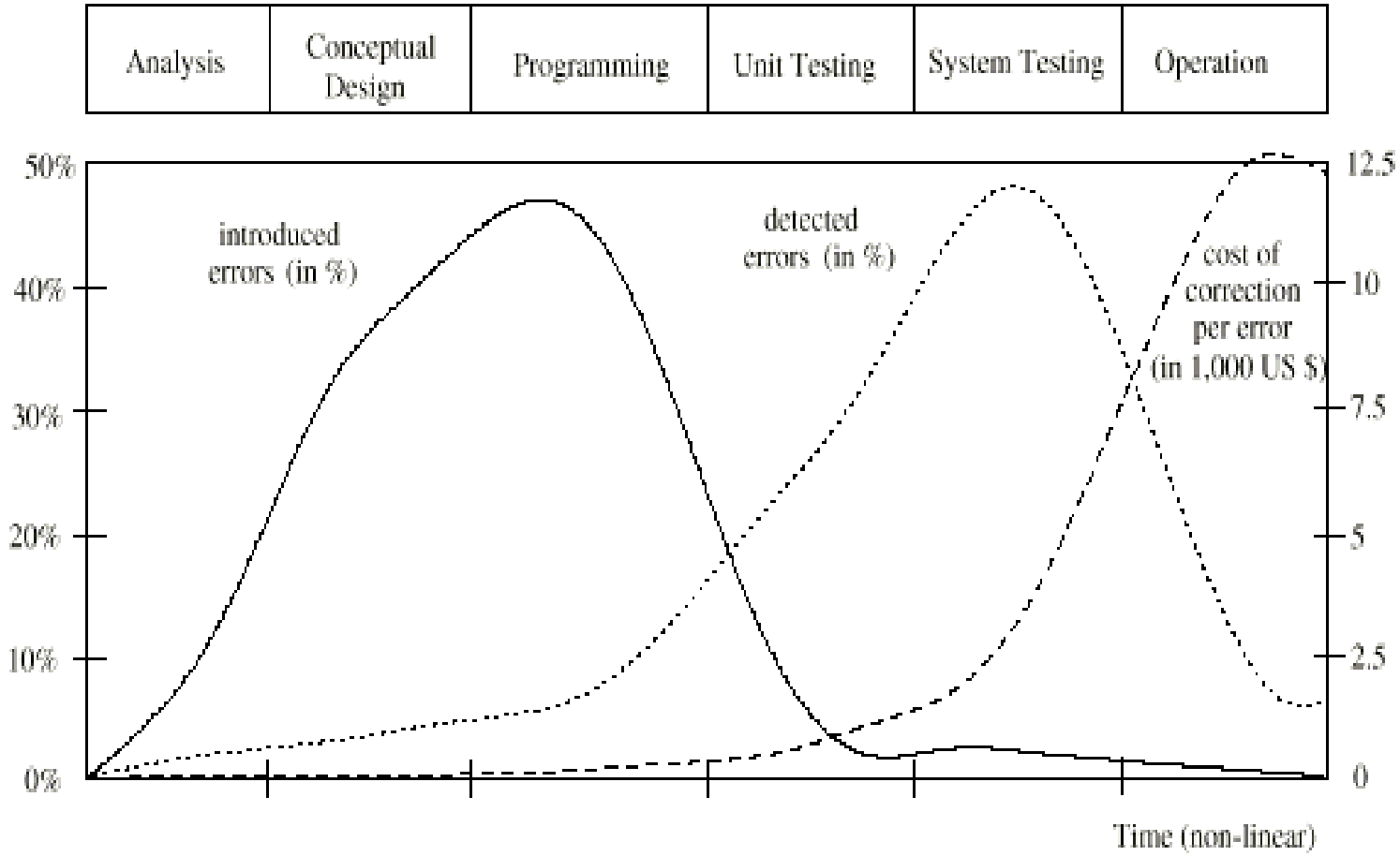
- Fenómeno mundial "decretado" hace muchos años
  - Muchos dicen que es crónica
  - Algunos casos traumáticos…
    - Therac-25 (6 vidas)
    - Arianne 5 (500 mill.)
    - AT&T Switching System, etc, etc…
    - Denver Airport (9 meses penalidad de 1.1 x día)
  - Miles de fracasos en Information Systems

- Cada vez + software crítico y + complejo

¿Por qué es tan difícil?

- No existen soluciones mágicas, o "Balas de Plata"
  - Dificultades esenciales y accidentales
- Inmadurez de la Disciplina
- Abismo entre el "Estado del Arte" y el "Estado de la Práctica" (Ing. en Verificación)
- Problemas de Gerenciamiento
- Confusión Calidad Proceso → Calidad Producto

Ciclo de Vida y la introducción, detección y eliminación de defectos



Algunos números sobre los defectos

- 50 % de los defectos se introducen durante la programación
- Hoy, no más del 15% de los defectos iniciales son detectados antes del testing
- Al comienzo del test de unidad la densidad es de 20 defectos x cada 1000 líneas de código (no comentadas)
- 80% de los defectos de prog. se encuentran en el 20% de los módulos de programación. Muchos se evidencian durante la integración
- Costo de reparación (1000 en test de unidad a 12500 durante operación)

Calidad en software

- Confiabilidad
- Corrección
- Robustez
- Seguridad (en datos, en acceso, Safety)
- Funcionalidad
- Usabilidad
- Facilidad de Mantenimiento
- Reusabilidad
- Verificabilidad + Claridad
- Interoperabilidad

Asegurar la calidad vs. controlar la calidad

Una vez definidos los requerimientos de calidad, tengo que tener en cuenta que:

- Las calidad no puede "inyectarse" al final.
- La calidad del producto depende de tareas realizadas durante todo el proceso.
- Detectar errores en forma temprana ahorra esfuerzos, tiempo, recursos.

Testear antes de codificar

- *"El acto de diseñar tests es uno de los mecanismos conocidos más efectivos para prevenir errores…*
  El proceso mental que debe desarrollarse para crear tests útiles puede descubrir y eliminar problemas en todas las etapas del desarrollo"

  B. Beizer
- *"Test-Driven Development": Kent Beck. XP*

Definiciones

- Validación
  - ¿Estamos haciendo el producto correcto?
  - Basada en el uso de modelos
- Verificación
  - ¿Estamos haciendo el producto correctamente?
  - Necesariamente es en relación a un componente anterior, que describe nuestro producto

Aspectos de los "errores" de software

- *Falla (failure)*
  - Diferencia entre los resultados esperados y reales
- *Defecto (defect o fault)*
  - Está en el texto del programa, una especificación, un diseño, y desde allí se hace visible una falla
- *Error* = equivocación humana
- Un error lleva a uno o más defectos, que están presentes en un producto de software
- Un defecto lleva a cero, una o más fallas: la manifestación del defecto

Verificación estática y dinámica

- *Dinámica*: trata con ejecutar y observar el comportamiento de un producto

- *Estática*: trata con el análisis de una representación estática del sistema para descubrir problemas

Técnicas de Verificación Estática

- Inspecciones, Revisiones, Walkthrough

(31 a 93 % media de 60%)

- Análisis de Reglas Sintácticas sobre código
- Análisis Data Flow sobre código
- Model Checking
- Theorem Proving (prueba de teoremas), etc.

Técnicas de Verificación Dinámica

- Testing
- Run-Time Monitoring. (pérdida de memoria, performance)
- Run-Time Verification

Testing

- Verificación Dinámica de la adecuación del sistema a los requerimientos (de distinto tipo)

- Objetivo: encontrar los errores importantes

Mundialmente: 30 a 50% del costo de un software confiable

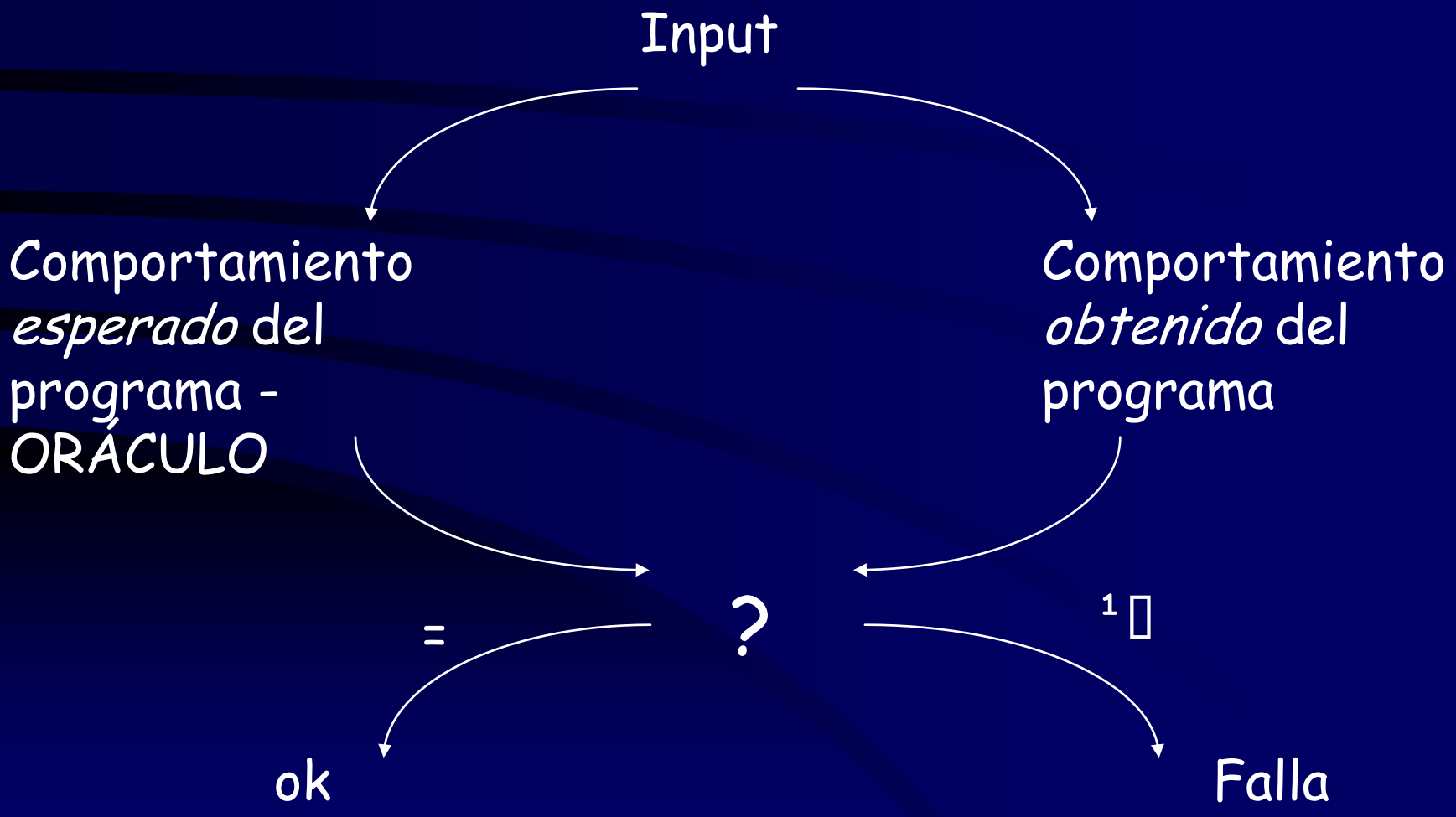
Testing

- Es el proceso de ejecutar un producto para
  - verificar que satisface los requerimientos
  - identificar diferencias entre el comportamiento real y el comportamiento esperado

(IEEE Standard for Software Test Documentation, 1983)

No prueba la corrección del software!

Cómo se hace testing?



Estamos asumiendo...

que se puede ejecutar el programa

que se conoce el resultado esperado

y que el resultado esperado puede compararse con el resultado obtenido (problema del oráculos: visibilidad y comparación)

Limitación del testing

- El testing puede demostrar la presencia de errores, nunca su ausencia [Dijkstra]
  - entonces, el testing NO puede probar que el software funciona ok
- Una de las mayores dificultades es encontrar un conjunto de tests adecuado:
  - suficientemente grande para abarcar el dominio y maximizar la probabilidad de encontrar errores
  - suficientemente pequeño para poder ejecutar el proceso de testing con cada elemento del conjunto y minimizar el costo del testing

Test de requerimientos no funcionales - Ejemplos

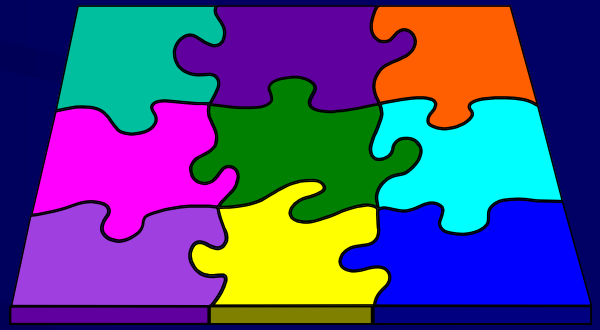
- Test de seguridad, validando disponiblidad, integridad y confidencialidad de datos y servicios
- Test de performance, validando los tiempos de acceso y respuesta del sistema
- Test de stress, validando el uso del sistema en sus límites de capacidad y verificando sus reacciones más allá de los mismos
- Test de Usabilidad

Niveles de test

- Test de sistema (o subsistema)
- Test de integración
- Test de unidad

Testing de Integración

- Test orientado a verificar que las partes de un sistema que funcionan bien aisladamente, también lo hacen en conjunto
  - Testeamos la interacción, la comunicación entre partes
  - No debe confundirse con testear un sistema-integrado (test de sistema)



El test de Integración

- Unimos y testeamos partes ya testeadas (que se asumen correctas)
- La "estrategia" de unión depende del tipo de sistema
  - Sistema organizado jerárquicamente
    - Top-down, bottom up, combinación de ambos
  - Sistema batch de procesamiento secuencial
    - Por partes del flow de corrida
  - Sistema sin jerarquía (p.e., objetos)
    - libre, salvo por el orden del desarrollo

Programa auxiliares: Stubs y drivers

- *Driver* simula las llamada
- *Stub* simula subprograma
- En total, exigen un esfuerzo considerable de programación

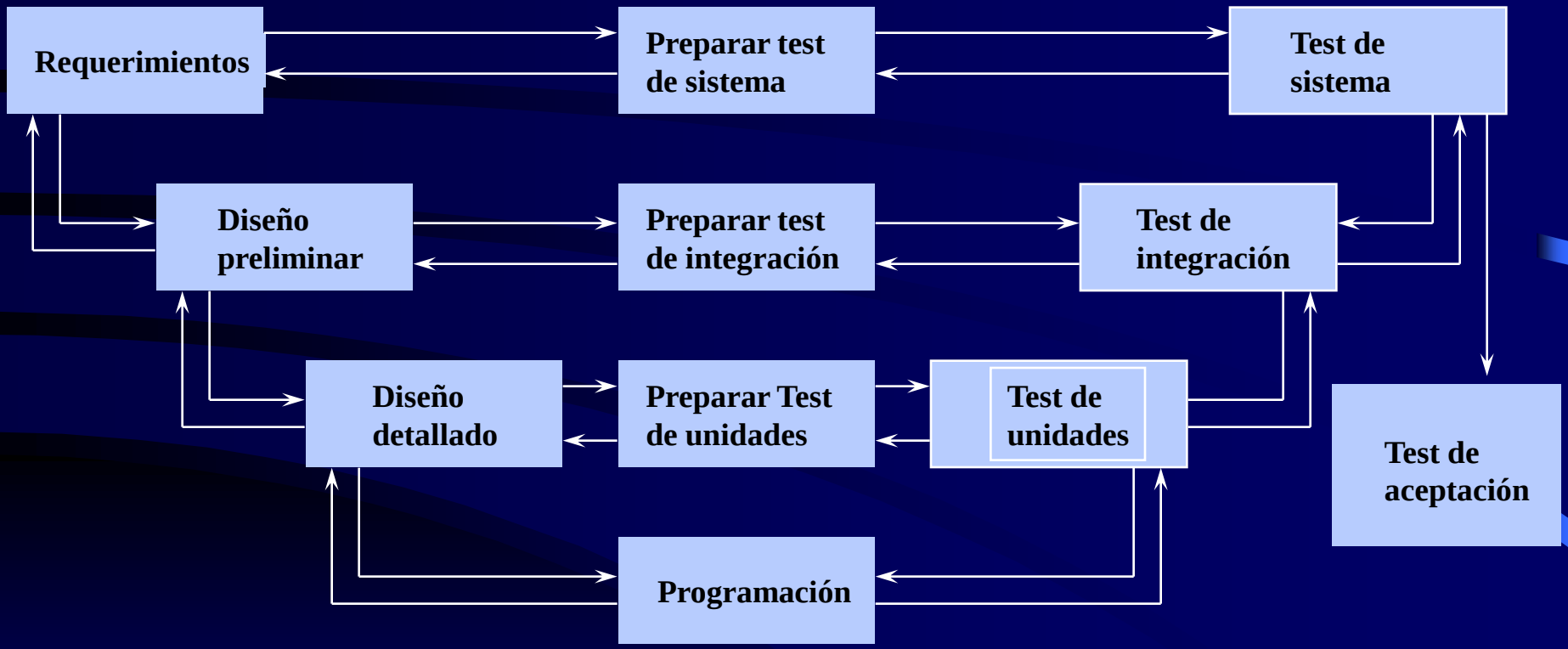
Testing de Unidad

- Se realiza sobre una unidad de código pequeña, claramente definida
  - qué es una unidad
- En general es llevado a cabo por los desarrolladores

Test de unidades

- ¿Qué es una unidad?
  - Un programa…
  - Una función o procedimiento…
  - Un form…
  - Un script de un form…
  - Un subsistema…
  - Una clase…

El Testing y el ciclo de vida de un sistema



Modelo del Ciclo de Vida en “V”

Testing Funcional

Test funcional - Corrección funcional

El programa *p* implementa correctamente a *f* si

$\forall\ i \in \mathrm{dom}(f)$: $p(i) = f(i)$

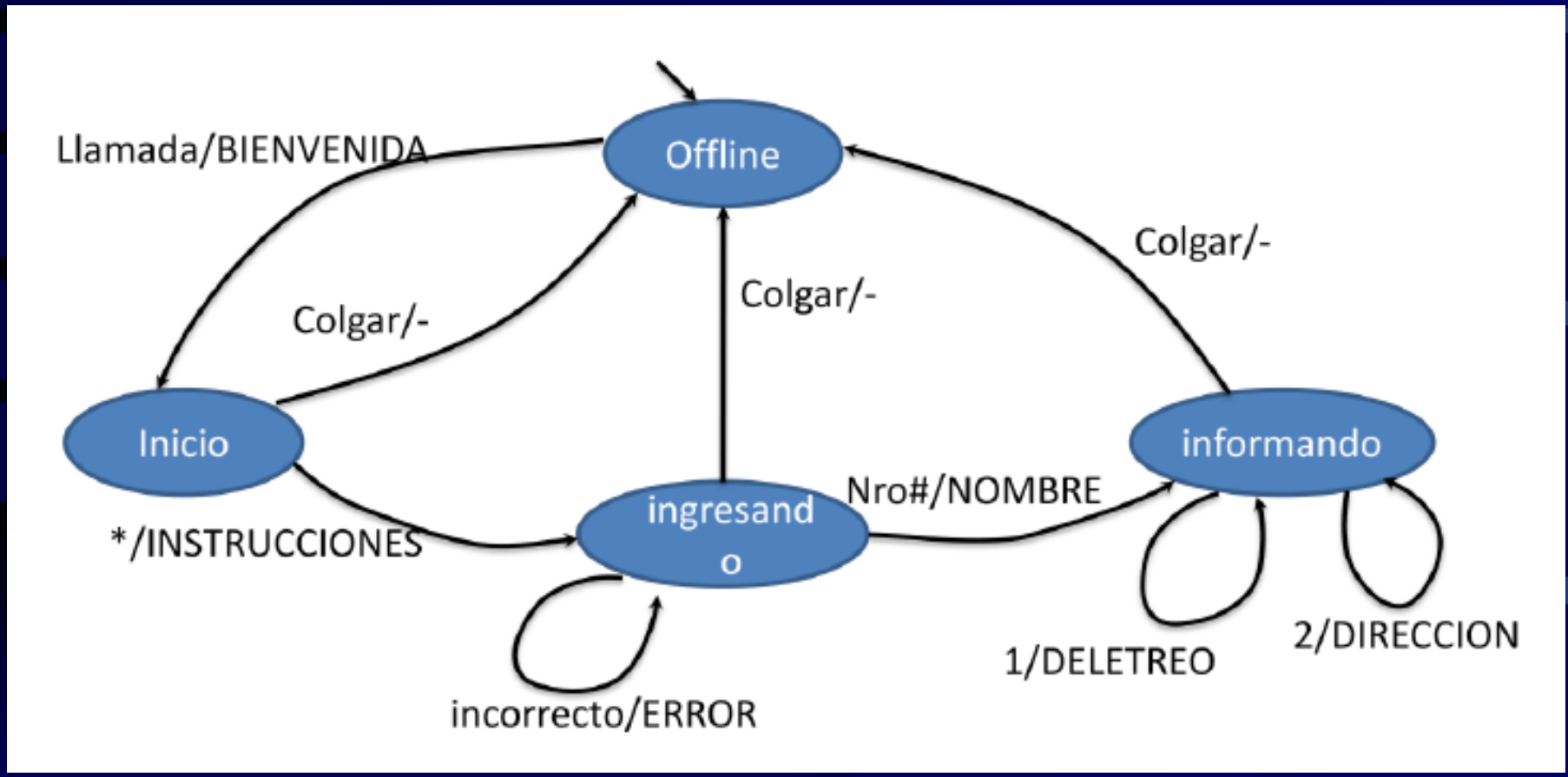
Qué debe testear el testing funcional?

- *p* implementa correctamente a *f*
- Además, lo hace en forma razonable
  - por ejemplo, si $i \in dom(f)$ y $f(i) = error$, *p* da un error razonable (por ejemplo, no se cuelga!)
- Además, $\forall\ i \notin dom(f)$, *p* avisa que $i \notin dom(f)$

Testing de Conformidad

- Es un sistema reactivo: mantiene una continua interacción con su entorno, respondiendo ante los estímulos externos en función de su estado interno
- Pensar en una especificación funcional de este tipo de sistemas
  - No es inmediato
  - Puede no ser clara
- Se usan otras abstracciones para describir su comportamiento

Testing de Conformidad



Testing de Conformidad

- Vamos a ver varias nociones de conformidad, casos de prueba, y algoritmos

  - Problemas de testing para Mealy Machines
  - Problemas de testing para modelos I/O generales
    - IOCO
    - Alternating Simulation

Cómo testear?: Random Testing

- Random testing
  - Generar al azar inputs con distribución uniforme o siguiendo un perfil operacional (no es trivial cuando el input es complejo)
  - Considerado base comparación de técnicas
  - Se usa en la práctica
  - Algunos estudios muestran que puede ser efectivo en algunas circunstancias

    (e.g., "Some observations on Partition Testing", Jeng, Weyuker ISSTA 1989; "Fomal Analysis of the Effectiviness and Predicatbility of Random Testing" Arcuri et.al. ISSTA 2010)

Cómo testear?: Partition Testing

- Familia de estrategias de testing
- Dividir el dominio del input en subconjuntos (no necesariamente una partición)
- Seleccionar uno o más elementos de cada subdominio

Cómo seleccionar datos de test para corrección funcional?...Intuiciones

- Agrupando los inputs
- Hipótesis: hay inputs "parecidos" (tratamiento), entonces testear el programa con uno de estos, equivaldría a testearlo con cualquier otro de estos
  - Esto no es verdad pero sirve y es la base de la mayor parte de las técnicas!!!

Observaciones (Jeng, Weyuker ´89)

- La probabilidad de encontrar fallas usando particiones se maximiza cuando hay un subdominio compuesto básicamente por elementos producen fallas
  - Fault-based strategies

- Hay casos dónde es igual (densidad pareja de fallas) o inclusive peor que random testing (muchos subdominios, fallas concentradas en un solo sub-dominio pero el subdominio es grande)

Criterios de Selección: La base Teórica de las Técnicas de Generación de Casos

Terminología

- *Casos de test*: descripciones de condiciones o situaciones a testear.
- *Dato de test:* asignación de valores concretos de parámetros para ejecutar un caso de test
- *Test Suite T*: conjunto de datos de test con los que se testea el programa
- Si P es correcto para todo elemento de T, se dice que T es *exitoso* para P
- Crear especificaciones y casos es un proceso creativo
- Crear datos de test es un proceso laborioso, y hay creatividad en cumplir económicamente con los casos

De manera Simple

- Caso1: $cond_1$, $cond_2$, $cond_3$,..., $cond_{n1}$ y ResEsp(1)

  ...
- Casok: $cond_1$, ..., ..., $cond_{nk}$ ResEsp(k)

Datos	$Param_1$	$Param_2$	$Param_3$	**Res. Esp.**
D_1	X_1	Y_1	Z_1	R_1
...	...	...	...	...
D_k	X_k	Y_k	Z_n	R_n

Criterios de selección de datos de test

- Un *criterio* es un subconjunto de conjuntos finitos del dominio de Inputs del programa P (puede estar expresado en términos de un conjunto de predicados también llamados *casos*)
- Se dice que un conjunto de datos *T* *satisface* un criterio *C* sii $T \in C$

Criterios "ideales"

- Un Criterio C es *Consistente* para P sii para todo par T1 y T2 de test sets que satisfacen C, T1 es exitoso para P sii T2 lo es

- Un Criterio C es *Completo* para P sii si P es incorrecto entonces hay un test set T que no es exitoso para P

Son estas nociones efectivas?

- es decir, hay un algoritmo que proponga casos (i.e., un criterio) tales que encuentren todos los errores en cualquier programa?

  NO!
- Ni siquiera un algoritmo que diga si un criterio es completo y/o consistente.
- Ninguna técnica puede ser efectiva para detectar todos los errores en un programa arbitrario
- Uso de heurísticas en forma de criterios de testing

Criterios de Partition Testing

Dado un programa P y una especificacion S, un criterio C define un conjunto no vacío de subdominios, SD_C (P,S)

Caso de prueba = Especificación de subdominio + resultado esperado

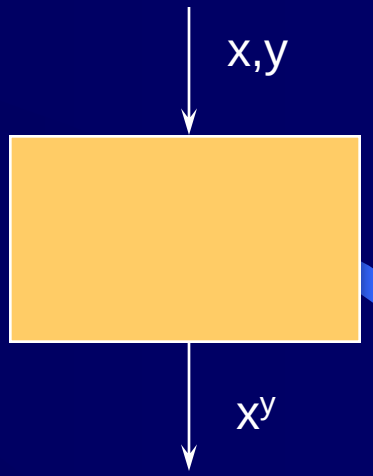
El criterio exige al menos un dato prueba por subdominio

Criterios"caja negra"

- También llamado testing producido por los datos, o testing producido por la entrada/salida
  - nos desentendemos completamente de la estructura interna del programa, pero no de su especificación

```
function fastexp(x,y: int): int;
{devuelve x a la y}
```

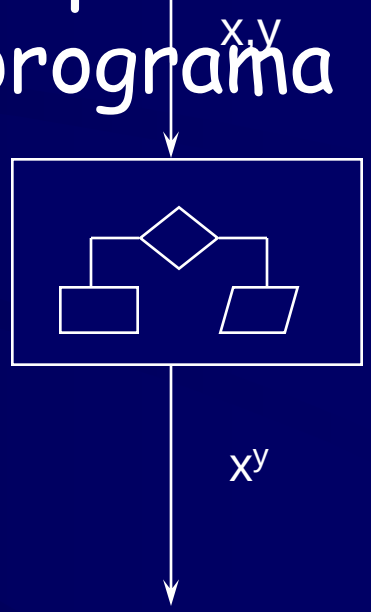
"¿Que casos probarían?"



Criterios estructurales o de "caja blanca o de cristal"

- Los casos de test se definen a partir de la estructura interna del programa

```
function fastexp (x,y: int):int;
% pre: y >= 0
% post: devuelve x a la y
z :int :=1;
while y ≠ 0 do
      if odd(y) then
         z :=z*x;
         y :=y-1
      endif;
      x := x*x; y :=y/2
endwhile
return(z)
```



"¿Qué pasa si y es potencia de 2?"
"¿Qué pasa si $y = 2^n - 1$?"

Técnicas de Selección de Casos: "Partir y Combinar"

Qué se busca con las Técnicas?

- alta probabilidad de encontrar errores
- a bajo costo (o al menos controlable)
- usando ideas generales, sistematizables y semiautomatizables
- y un modelo de fallas subyacente como "rationale"

Importante

- No sólo testear casos válidos!!!!

- Condiciones válidas y esperadas
  - hace lo que se supone que hace
- Condiciones inválidas o inesperadas
  - no hace lo que se supone que no hace

Técnicas de partición de Dominio

Técnica II: Category-Partition de T. Ostrand & M. Balcer -1988 (generalización de Técnica I)

Método de Partición en Categorías - Ejemplo

- FIND

Dado un nombre de archivo A y una expresión E, FIND devuelve la lista L de las palabras en A que comienzan con E, sin repeticiones. E no puede ser vacía ni contener blancos

Ejemplo
PASO 1

1. Elegir una funcionalidad que pueda testearse en forma independiente

Ya lo tenemos!

Ejemplo
PASO 2

2. Determinar sus parámetros u otros objetos del ambiente que pueden afectar su funcionamiento

-Archivo (nombre)

-Expresión

PASO 3

3. Determinar las características relevantes de cada objeto determinado en el punto 2 (Archivo, Expresión) y de la relación entre estos objetos en el output (relación Archivo/Expresión)

"Category" en el artículo original de Ostrand & Balcer 1988

Ejemplo - PASO 3
Características de E

- longitud de expresión
- blancos en la misma

Ejemplo - PASO 3 Características de A

- Tamaño de A

Ejemplo - PASO 3
Características de A/E

- número de ocurrencias de E en A
- número máximo de ocurrencias de E en una línea de A
- longitud de A vs. longitud de E
- repeticiones de palabras que contienen a E en A
- ubicación relativa de E en palabras de A
- ...

PASO 4

4. Determinar elecciones (“choices”) para cada característica de cada objeto

Ejemplo en Categorías de E

- longitud de expresión
  - vacía
  - no vacía
- contiene blancos
  - sí
  - no

Ejemplo en Categorías de *A*

- existe
  - sí
  - no
- tamaño
  - vacío
  - no vacío

Ejemplo en Categorías de A/E (I)

- número de ocurrencias de E en A
  - 0
  - al menos 1
- número máximo de ocurrencias de E en una línea de A
  - 0 a 1
  - más de 1

Ejemplo en Categorías de A/E (II)

- longitud de *A* vs. longitud de E
  - long *A* >= long E
  - long *A* < long E
- repeticiones de palabras que contienen a E en *A*
  - Sin repeticiones
  - *A* contiene repetida a p, palabra que comienza con E
- Ubicación relativa de E
  - p en *A*, E es prefijo de p
  - p en *A*, p contiene a E pero p no comienza con E…

PASO 5 - Clasificación : errores, únicos, restricciones

E

- longitud
  - vacía - ERROR
  - no vacía
- blancos
  - sí - ERROR
  - no

A

- existe
  - sí
  - no - ERROR
- tamaño
  - vacío - ÚNICO
  - no vacío

PASO 5 - Clasificación : errores, únicos, restricciones

A/E

- número de ocurrencias de E en A
  - 0 - ÚNICO
  - al menos 1 (A no vacío)

- número máximo de ocurrencias de E en una línea de A
  - 0 o 1
  - más de 1 (A no vacío) ÚNICO

- ETC.

PASO 6
Armado de casos

- Ver una posible solución en archivo Word

Resumen del Método

1. Elegir una funcionalidad que pueda testearse en forma independiente
2. Determinar sus parámetros u otros objetos del ambiente que pueden afectar su funcionamiento
3. Determinar las características relevantes de cada objeto determinado en el punto 2 y de la relación entre estos objetos en el output
4. Determinar elecciones para cada característica de cada objeto
5. Clasificación: errores, únicos, relaciones, restricciones
6. Armado de casos

Información para identificar categorías

- Texto informal o formal de la especificación
- Características propias de los param. de I/O de la funcionalidad a probar
- Cardinalidad del modelo de datos, que define reglas del negocio
- Ciclo de Vida de las entidades del modelo de datos

Algunas Conclusiones

- El método se aplica a cualquier descripción de funcionalidad (formal, semiformal o informal)
- Es necesario usar los requerimientos para planificar el testing
- Esto genera preguntas de incompletitud, ambigüedad, inconsistencia, e incorrección en un momento en que es "fácil" corregir requerimientos

Algunas Conclusiones (cont.)

- No hay una única solución: dependiendo de cómo se realice el proceso se obtendrá un conjunto de casos de test particular
- Los casos de test pueden darse a conocer a los programadores antes de que implementen

Técnicas de partición de Dominio

Técnica para la identificación de Categorías

Especificaciones de Servicios o Requerimientos

- Aparecen de manera + o – clara los (parámetros implícitos y explícitos), su casuística, la relación entre ellos
- Ejemplo:en los casos de uso: hay escenarios distintos dependiendo de valores de parámetros o de acciones decididas por el actor. Son parámetros candidatos con sus categorías y elecciones

Análisis de condiciones de borde

- Los casos de test que exploran los bordes de las clases de equivalencia producen mejor resultado
  - Cada margen de la clase de equivalencia debe quedar sujeto a un test
  - input y output

1. Heurística

Si una condición de input especifica un rango de valores (intervalo), identificar una clase válida y dos inválidas

Ejemplo: el item puede variar entre 1 y 999

VÁLIDA: $1 < valor < 999$

INVÁLIDAS: $1 \geq valor$ y $valor \geq 999$

DE BORDE: $valor = 1$ y $valor = 999$

2. Heurística

Si una condición de input especifica un número de valores, identificar una clase válida y dos inválidas

Ejemplo: un automóvil puede tener entre uno y seis dueños

VÁLIDA: entre uno y seis dueños

INVÁLIDAS: ningún dueño y más de seis dueños

DE BORDE: 1 dueño y 6 dueños

3. Heurística

Si una condición de input especifica un conjunto de valores, y hay razones para pensar que cada uno es manejado por el programa en forma distinta, identificar una clase válida por cada elemento y una clase inválida

Ejemplo: tipo de vehículo: camión, taxi, auto, moto

VÁLIDAS: camión, taxi, auto, moto

INVÁLIDA: algún tipo que no es camión, taxi, auto ni moto

4. Heurística

Si una condición de input especifica una situación que *debe* ocurrir, identificar una clase válida y una clase inválida

Ejemplo: el primer caracter debe ser una letra

VÁLIDAS: es una letra

INVÁLIDA: no es una letra

5. Sobre clases inválidas

- ¿Qué me falta?
- Probar el ingreso de valores de otro tipo que la clase
  - numéricos en vez de alfabéticos
  - alfabéticos en vez de numéricos
  - combinaciones de alfabéticos y numéricos
  - fechas erróneas, etc.
- El entorno de desarrollo puede ahorrar este trabajo

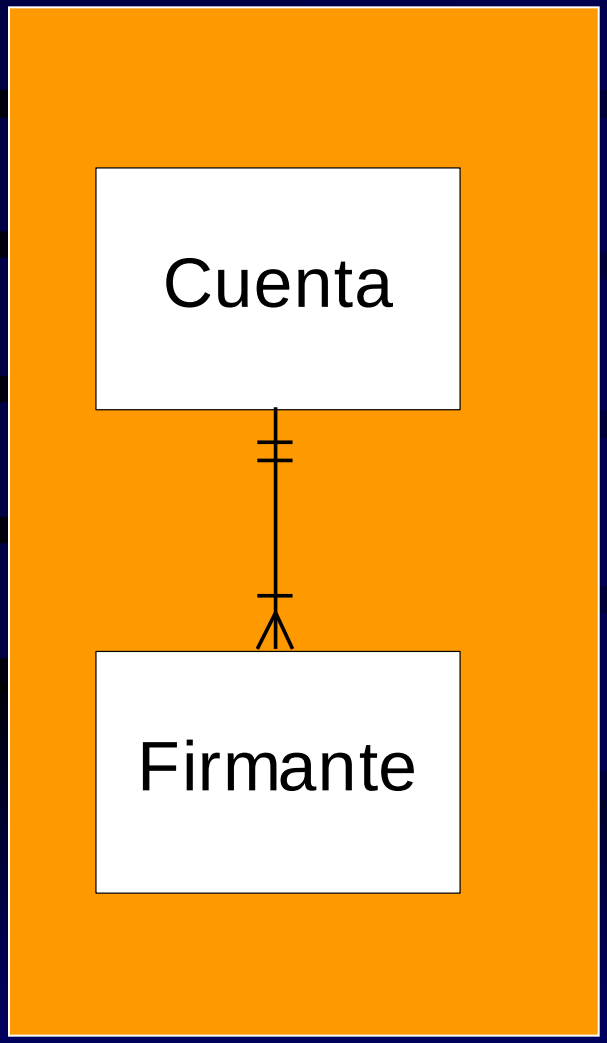
6. Tipos de Datos de input y output

- Características propias de los tipos de datos de entrada y salida de la funcionalidad testeada. Ejemplo: Fechas, listas, archivos de texto con determinado formato, etc.

8. Cardinalidad del modelo de datos

- La cardinalidad de las relaciones define reglas del negocio que deben ser probadas
- *Cardinalidad mínima*: cuántas instancias hay como mínimo de una entidad por cada instancia de una entidad relacionada con ella
- *Cardinalidad máxima*: cuántas instancias hay como máximo de una entidad por cada instancia de una entidad relacionada con ella

Ejemplo

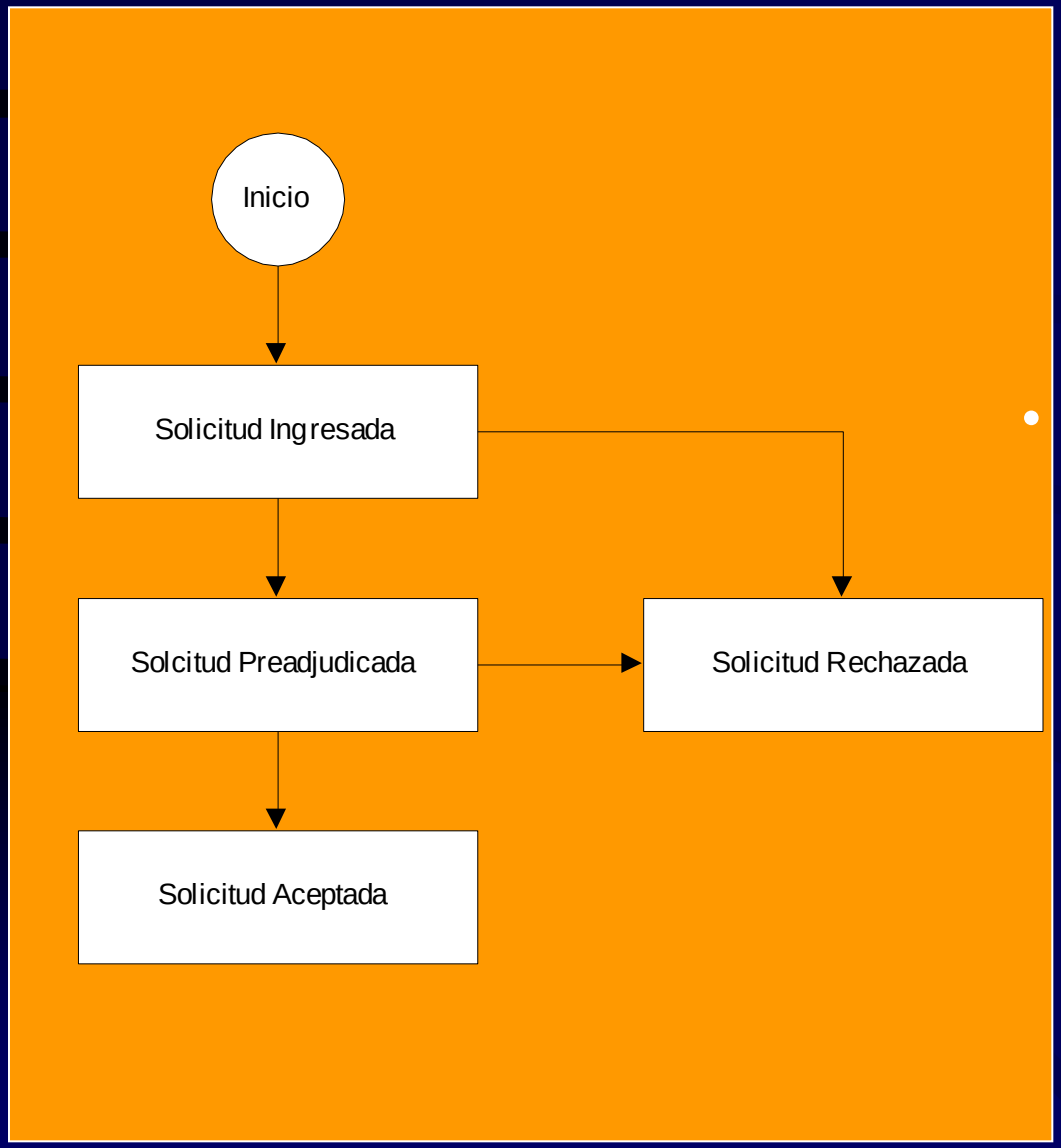


- Cuenta sin firmante (inválido)
- Cuenta con un firmante (válido)
- Cuenta con más de un firmante (válido)
- Cuenta con más firmantes del máximo permitido (inválido)

9. Ciclo de vida de las entidades

- Visión temporal del modelo de datos:
  - a partir de un diagrama del ciclo de vida de una entidad

Ejemplo



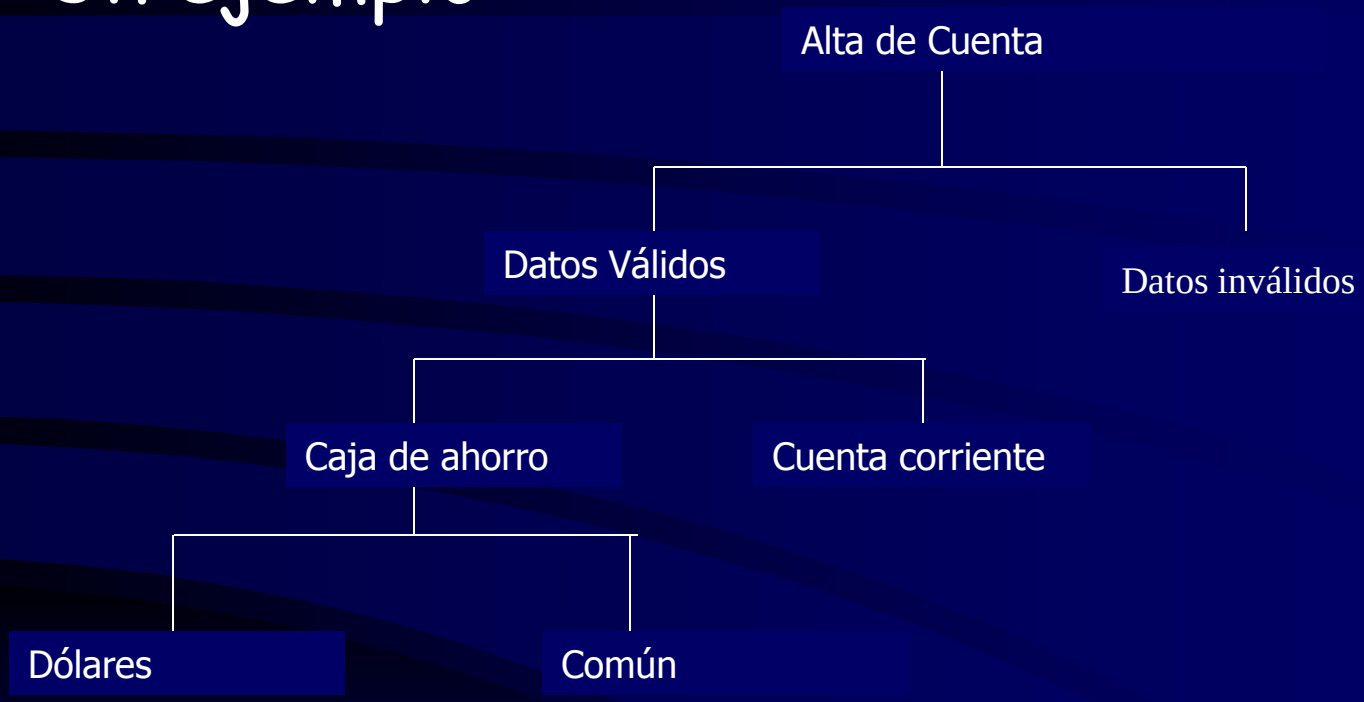
- Debo probar:
  - Transiciones válidas
  - Transiciones inválidas

Técnicas Combinatorias

Técnica III: Notación Arbórea

Definición Arbórea

- Un ejemplo:



Definición Arbórea

- Puede verse como category-partition especial donde el mismo diseñador de casos decide explicítimanete cómo combinar
- Por ejemplo, cuando hay idea de orden de ingreso de parámetros se puede minimizar casos sin sentido
- Se arma un árbol donde cada nodo indica "category" sobre un parámetro o combinación entre un parámetro y uno anterior descripto en el árbol
- Los ejes que salen son las choices compatibles con las condiciones acumuladas desde la raíz
- Los casos de tests especificados y ejecutados son los caminos del árbol

Definición Arbórea: Consejos

- Hay categorías que pueden ser irrelevantes en una rama, entonces no se abren
- Varios choices pueden agruparse
- Un error corta la rama
- Condiciones normales de un lado, errores del otro...
- Tener en cuenta que el orden en que aparecen los parámetros y sus categorías puede ser importante (ej: del año y día)
- Mantener el mismo orden parámetro-categoría en todas las ramas para facilitar la documentación posterior

Combinando Category-Partition Fases 1-5 con Definición Arbórea

- Si primero hago un category partition (salvo combinatoria):
  - conviene anotar la relevancia de cada categoría respecto a choices de otras categorías
  - Criterio de cobertura: hay que revisar que todos los choices sean ejercitados en el árbol

Técnicas Combinatorias

Técnica I: Grafo Causa-Efecto

Grafo de Causa-Efecto

- Especificaciones de I/O complejas (mucha dependencia entre Inputs, entre Outputs y entre I/O)
- Nos permite definir combinaciones relevantes de categorías binarias sobre inputs para definir casos. Allí difiere del la parte combinatoria de Category-Partition
- Provee un display visual de las relaciones entre una causa y la otra
- Ayuda a detectar ambigüedades o incompletitud en la especificación

Grafo de Causa-Efecto

- Idea: Generar todos los outputs admisibles con la siguiente heurística:
  - Si hay un "or" (todas las opciones con sólo una señal en True)
  - Si hay un "and" (todas con sólo una en falso)

- $O(n*k*o)$ en lugar de $O(2^n)$ dónde n es la cantidad de categorías binarias sobre el input, k la profundidad del DAG y o la cantidad de combinaciones del output válidas

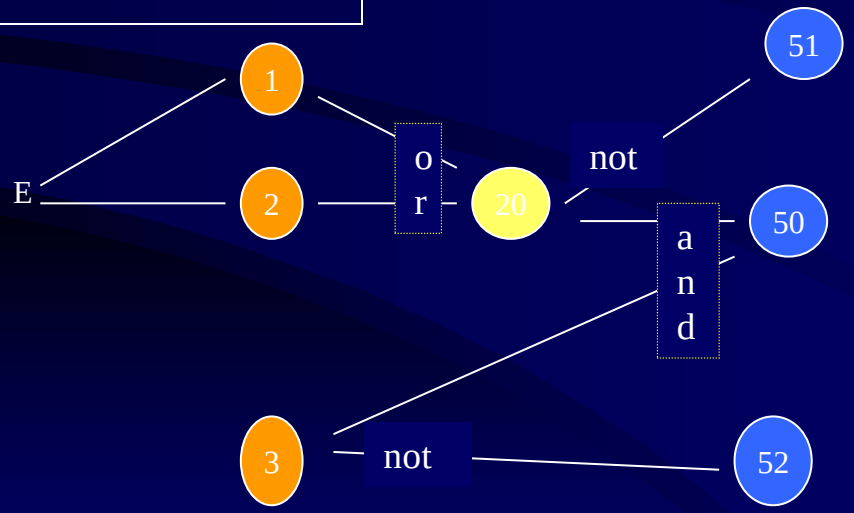
Ejemplo Grafo de Causa-Efecto

- Para mostrar una de las 10 posibles secciones del índice se debe entrar un comando que consiste en una letra y un dígito
  - El primer carácter debe ser una D para display o una L para List, y debe estar en la columna 1
  - El segundo carácter debe ser un número (0..9) en la columna 2
  - Si se ingresa un comando como éste, el índice de la sección identificada por el dígito es mostrada en la terminal. Si el primer caracter es incorrecto "comando inválido", y si el 2do caracter es incorrecto "número de índice inválido"

Grafo de Causa-Efecto: Ejemplo

- Causas
  1. Carácter en la 1er columna es D
  2. Carácter en la 1er columna es L
  3. Carácter en la 2da columna es un dígito

- Efectos
  50. El índice de la sección es mostrado
  51. Msg "comando inválido"
  52 Msg "número de indice inválido"



Grafo booleano

1=causa presente pasa, nulo no importa

Grafo de Causa-Efecto: Ejemplo

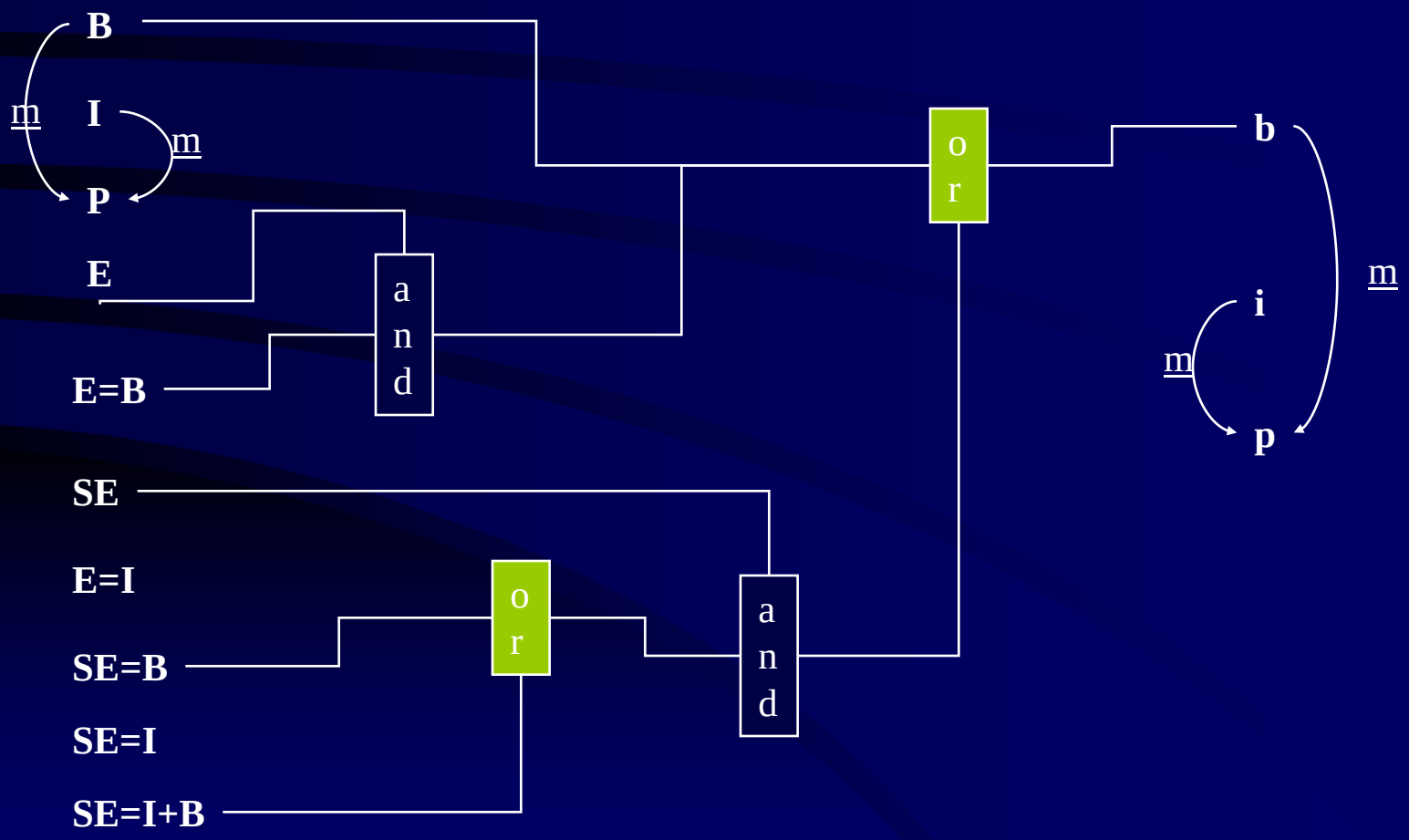
Tabla de decisión y casos de test

	Casos de test			
Causas	1	2	3	4
1	1	0	0	
2	0	1	0	
3	1	1		0
Efectos				
50	1	1	0	0
51	0	0	1	0
52	0	0	0	1

Ejemplo de datos de test derivados

Caso de test	Input	RE
1	D5	Se muestra indice 5
2	L4	Se muestra indice 4
3	B2	"Comando inválido"
4	DA	"Número de índice inválido"

Grafo de Causa-Efecto: Otro Ejemplo



Técnicas Combinatorias

Técnica II: 2-wise Partition & Arreglos Ortogonales

2-Wise Testing y OATS

- Aplicando **2-wise Testing**, se seleccionan casos de test de manera tal de testear las interacciones entre pares de parámetros (factores) independientes de una manera económica
- Ej.: 75 parámetros de 2 estados c/u se pueden testear con 28 tests!
- Inputs independientes enumerados
- Supuesto: errores son de tipo single o double mode.
- OATS (Orthogonal Array Testing Strategy)
- Herramientas de Telecordia y Freeware.

Arreglo Ortogonal: Ejemplo

- La sintaxis de un comando consiste en 3 posibles parámetros:
  - Command PARM1, PARM2, PARM3 (enter)
    - PARMx=1, 2, 3
  - En teoría un tester debería crear 3^3=27 combinaciones

1	1	1
1	1	2
1	1	3
1	2	1
1	2	2
1	2	3
1	3	1
1	3	2
1	3	3
...	...	...

  - En este caso hay 3 factores con 3 niveles cada uno. Entonces, con la técnica de arreglo ortogonal, sólo quedan 9 casos definidos

Arreglo Ortogonal: Ejemplo

- Las combinaciones entre 2 parámetros son 9:

1	1
1	2
1	3
2	1
2	2
2	3
3	1
3	2
3	3

- Una posible solución es la dada en el siguiente arreglo ortogonal
  - En el arreglo ortogonal las columnas se corresponden con los factores y las filas los casos de test. Las filas representan todas las posibles combinaciones de a pares entre los posibles niveles de los factores

Arreglo Ortogonal: Ejemplo

Caso de test	PARM1	PARM2	PARM3
1	1	1	3
2	1	2	2
3	1	3	1
4	2	1	2
5	2	2	1
6	2	3	3
7	3	1	1
8	3	2	3
9	3	3	2

Caso de test	PARM1	PARM2	PARM3
1	1	1	3
2	1	2	2
3	1	3	1
4	2	1	2
5	2	2	1
6	2	3	3
7	3	1	1
8	3	2	3
9	3	3	2

Caso de test	PARM1	PARM2	PARM3
1	1	1	3
2	1	2	2
3	1	3	1
4	2	1	2
5	2	2	1
6	2	3	3
7	3	1	1
8	3	2	3
9	3	3	2

Arreglo Ortogonal (OATS)

- Factors: Columnas=Parámetros
- Levels: cantidad de valores posibles para cada columna (e.g.,choices)
- Strength: Cantidad de columnas tales que las Levels ^{Strenght} posibilidades aparecen la misma cantidad de veces
- Tabulados como: L_{runs}(Levels ^{Factors})
- Ej: $L_4(2^3)$: cuatro corridas para cubrir 3 factores con dos elecciones posibles c/u
- $L_{18}(3^6 6^1)$: 18 corridas para 7 factores, 6 de 3 niveles y 1 de 6 niveles. Mejor que 216 tests!

Conclusiones sobre la Generación de Casos

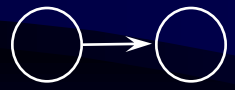
- Ninguna técnica es completa
- Las técnicas atacan distintos problemas
- Lo mejor es combinar varias de estas técnicas para complementar las ventajas de cada una
- Depende del programa a testear
- Sin especificaciones de req. todo es mucho más difícil
- Tener en cuenta la conjetura de defectos
- Ser sistemático y documentar las suposiciones sobre el comportamiento o el modelo de fallas

Testing Estructural de Unidades

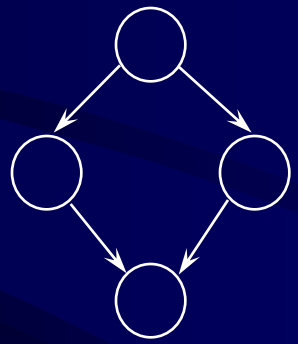
Representación del flujo de control

Representamos el flujo de control de un programa a través de un grafo de flujo o *flowgraph*

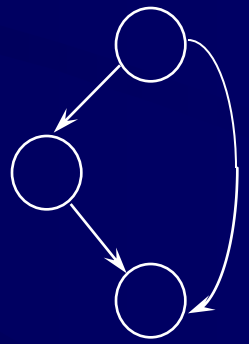
Programas secuenciales, con un único punto de ingreso y un único punto de terminación



Secuencia

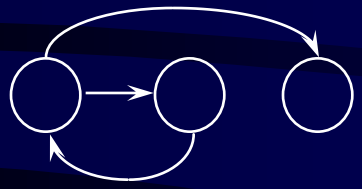


if then else

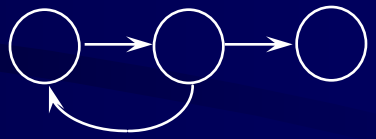


if then

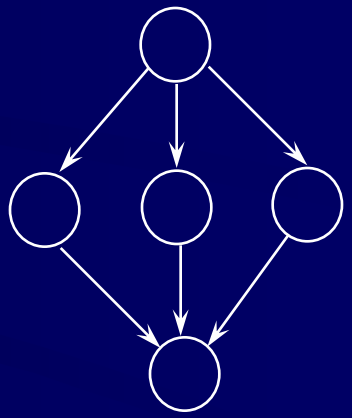
Representación del flujo de control



do while



repeat until

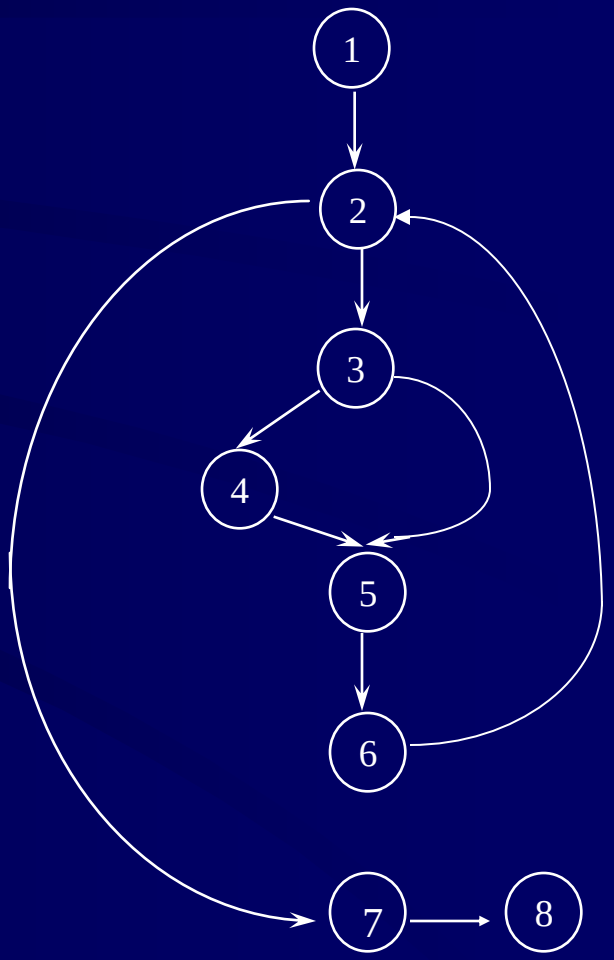


case, o selección múltiple

- Esta es una representación posible. Existen varias.

Flowgraph - Ejemplo

```
   begin
1:   read(x,y)
2:   while x ≠ y do
3:        if x > y
4:             then x := x-y
5:        end-if
6:   end-while
7:   return x
8: end
```



Caminos del flowgraph

- Un camino en un flowgraph desde el nodo asociado al inicio del programa hasta el nodo asociado a la terminación del programa se llama *camino completo*
- Una ejecución del programa que termina satisfactoriamente, está asociada a un camino completo en el flowgraph del programa
- Cada camino del flowgraph corresponde a una ejecución? Y los caminos completos?

De caminos a casos de test

- Dado un camino completo en un flowgraph, ¿cómo obtengo un caso de test que lo fuerce?
  - A partir de un camino completo, se puede obtener condiciones sobre los inputs para forzar dicho camino
    - ejecución simbólica
    - caso de test

Caminos no factibles

- Un camino en un flowgraph para el cual no existe input del programa que fuerce su ejecución, se dice *camino no factible*
  - Cada camino factible puede tener muchos inputs asociados que fuercen su ejecución
- Cuál es el problema de los caminos no factibles?

Criterios de Testing Estructural

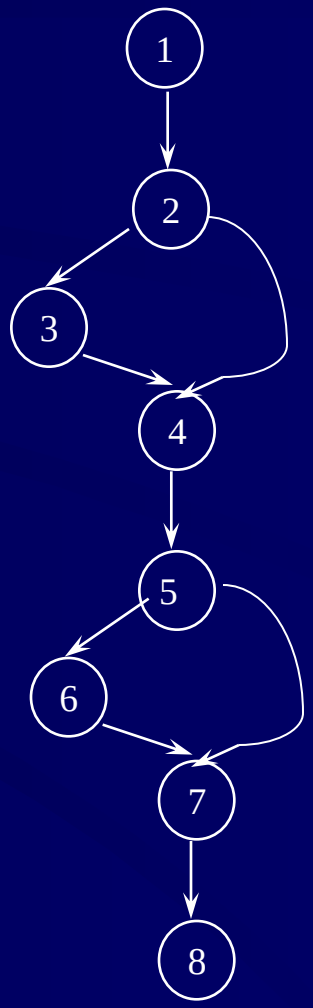
- Un *criterio de testing estructural* permite identificar entidades que deben cubrirse con los datos de test para satisfacer el criterio

Cubrimiento de sentencias o instrucciones

- *Todas las sentencias del programa deben ejercitarse*
- equivale a cubrir todos los nodos del flowgraph

Flowgraph - Ejemplo 2

```
   begin
1:  read(a,b,x)
2:  if (a>-1) and
(b=0)then
3:       x := x/a
4:  end if
5:  if (a=2) or (x >1)
then
6:       x := x+1
7:   end if
8:end
```



Ejemplo sentencias

- en el ejemplo, debemos cubrir los nodos 1, 2, 3, 4, 5, 6, 7 y 8
- el siguiente camino cubre todos los nodos

1, 2, 3, 4, 5, 6, 7, 8

- por ejemplo, con el input siguiente forzamos el camino
  - Test 0: a=2, b=0, x=1

Cubrimiento de decisiones o Branch

- Problema: el "else" del "if" no está testeado

- *Todas las decisiones en el control del programa deben ejercitarse al menos una vez por true, y al menos una vez por false*

- branch equivale a cubrir todos los arcos del flowgraph
- branch implica sentencias

Ejemplo cubrimiento de decisiones

- en el ejemplo, debemos cubrir todos los arcos (y en particular, los arcos 2-3, 2-4, 5-6 y 5-7)
- por ej., con los inputs siguientes
  - Test1: a=0, b=0, x=0
  - Test2: a=-2, b=0, x=2
- ejecutamos los siguientes caminos, que cubren todos los arcos
  - Camino1: 1, 2, 3, 4, 5, 5-7, 7, 8
  - Camino2: 1, 2, 2-4, 4, 5, 6, 7, 8

Problema...

- Una decisión puede estar compuesta de varias condiciones
- En el ejemplo, (a>-1) and (b=0)

Test1: a=0, b=0, x=0

- condiciones: $a>-1$, $b=0$, $a \neq 2$, $x \leq 1$

Test2: a=-2, b=0, x=2

- condiciones: $a \leq -1$, $b=0$, $a \neq 2$, $x>1$

Cubrimiento de condiciones

- *Todas las condiciones de todas las decisiones en el control del programa deben ejercitarse al menos una vez por true, y al menos una vez por false*

Condiciones

- Para satisfacer condiciones hay que ejecutar: (a>-1) (a≤-1) (b=0) (b≠0) (a=2) (a≠2) (x>1) (x≤1)
- Por ejemplo, Test1, Test2 y Test3, donde:

  Test3: a=2, b=1, x=0
  - camino: 1, 2, 4, 5, 6, 7, 8
  - condiciones: a>-1, b≠0, x≤1 , a=2

se satisface el criterio condiciones

Relación entre branch y condiciones

Consideramos la guarda *If A&B*

- branch: (A & B), (¬(A & B))
  - puede faltar ¬A o ¬B
    - ➢ branch no implica condiciones
- condiciones: A, ¬A, B, ¬B
  - (A & ¬B) (¬A & B)
    - ➢ condiciones no implica branch

❖ *es decir, ninguno garantiza al otro!*

❖ *Lo mismo sucede entre sentencias y condiciones*

MC/DC

- Modified Condition/Decision Coverage
- DO-178B
- Exige:
  - cada punto de entrada y salida del programa se invoco al menos una vez
  - Cada condición en una decisión tomó sus posibles resultados al menos una vez
  - Cada condición se mostró que afecta el resultado de la decisión independientemente del resto

MC/DC

- Entre N+1 a 2N test cases
- P(N,M) = probabilidad de encontrar un error en una implementación incorrecta de una expresión Booleana de N condiciones: 1-(2^(2^N-M) – 1/ 2^2^N)
- N crece, converge a 1- 1/ 2^M

Data Flow Testing Motivación

- Programas = control + data

Definiciones y Usos

- una sentencia que guarda un valor en la posición de memoria de una variable, crea una definición
- una sentencia que trae el valor de la posición de memoria de una variable es un uso de la definición activa de esa variable
  - un uso de x es un un uso predicado o p-uso si aparece en el predicado de una sentencia que representa una bifurcación de control
  - en otro caso, se llama uso computacional o c-uso (aparece del lado derecho de una asignación)

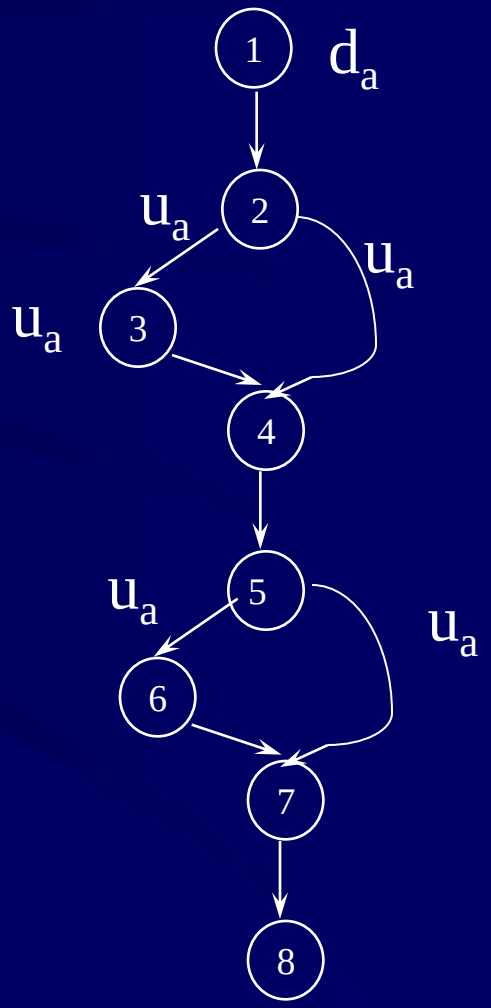
Def-Use flowgraph

Dado un programa *P* y una variable *x* en *P*, el def-use flowgraph es el flowgraph de *P* anotado de la siguiente manera:

- por cada *definición* de x, el nodo asociado está etiquetado con una definición de x
- por cada *c-uso*, el nodo asociado está etiquetado con un uso de x
- Por cada *p-uso* todos los arcos salientes del nodo asociado están etiquetados con un uso de x

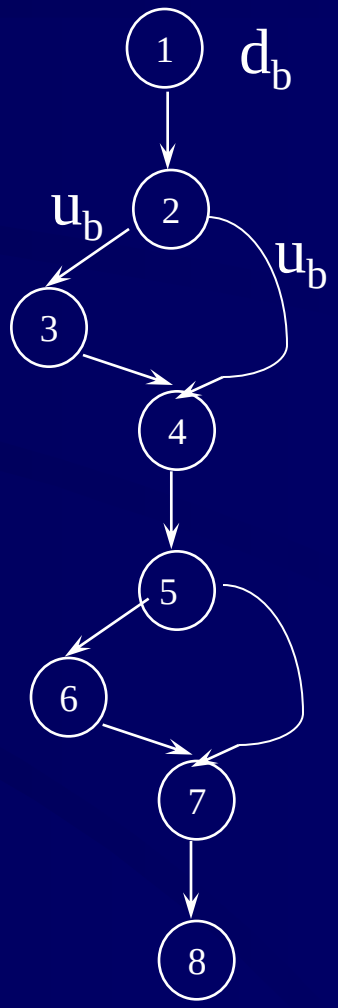
Def-use flowgraph para *a* Ejemplo

```
   begin
1:   read(a,b,x)
2:   if (a>-1) and (b=0)then
3:        x := x/a
4:   end if
5:   if (a=2) or (x >1) then
6:        x := x+1
7:    end if
8:end
```



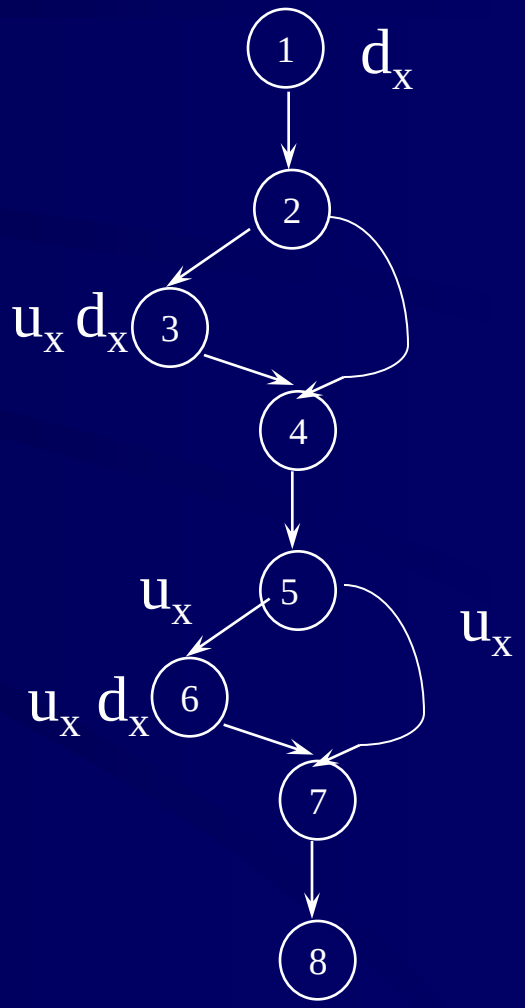
Def-use flowgraph para *b* Ejemplo

```
   begin
1:   read(a,b,x)
2:   if (a>-1) and (b=0)then
3:        x := x/a
4:   end if
5:   if (a=2) or (x >1) then
6:        x := x+1
7:    end if
8:end
```



Def-use flowgraph para *x* Ejemplo

```
   begin
1:   read(a,b,x)
2:   if (a>-1) and (b=0)then
3:        x := x/a
4:   end if
5:   if (a=2) or (x >1) then
6:        x := x+1
7:    end if
8:end
```



DUA (definition-use association)

Una DUA es una terna [*d*, *u*, *x*] tal que

- la variable *x* está definida en el nodo *d*
- la variable *x* se usa en el nodo *u* o en el arco *u*
- hay al menos un camino desde *d* hasta *u* que no contiene otra definición de *x* además de la de *d* (*def-clear para x* o *libre de definiciones para x*)

Ejemplo - DUAS ejemplo 2

- [1, 2-3, a]
- [1, 2-4, a]
- [1, 3, a]
- [1, 5-6, a]
- [1, 5-7, a]

- [1, 2-3, b]
- [1, 2-4, b]
- [1, 3, x]
- [1, 5-6, x]
- [1, 5-7, x]
- [1, 6, x]
- [3, 5-6, x]
- [3, 5-7, x]
- [3, 6, x]

Criterios estructurales basados en flujo de datos

Data flow testing

- all defs
- all c-use
- all p-use
- all uses
- all c-use some p-uses
- all p-uses some c-uses
- all du-paths

Cubrimiento All-uses

- *Para cada variable en el programa, deben ejercitarse toda las asociaciones entre cada definición y toda uso de la misma (tal que esa definición esté activa)*

- All-uses equivale a cubrir todas las DUAS del programa

Ejemplo All-Uses

- Test 0: a=2, b=0, x=1
  - Cubre las DUAS [3, 5-6, x], [3, 6, x]
- Test1: a=0, b=0, x=0
  - Cubre las DUAS [1, 2-3, a], [1, 3, a], [1, 5-7, a], [1, 2-3, b], [1, 3, x], [3, 5-7, x]
- Test2: a=-2, b=0, x=2
  - Cubre las DUAS [1, 2-4, a], [1, 5-6, a], [1, 2-4, b], [1, 5-6, x], [1, 6, x]
- Test3: a=2, b=1, x=0
  - Cubre las DUAS [1, 2-4, a], [1, 5-6, a], [1, 2-4, b], [1, 5-6, x], [1, 6, x]

Ejemplo (cont.)

Falta cubrir la DUA [1, 5-7, x]

Por ejemplo, el test

- Test 4: a=1, b=1, x=0

cubre esta DUA

Testing estructural - proceso

1- Con el código como base, dibujamos el grafo de flujo de control

2- Determinamos un conjunto de caminos que cumple el *criterio*

3- Preparamos los datos de test que forzarán la ejecución de cada camino

- p.e., usando ejecución simbólica

4- Evaluamos si satisfacemos el criterio

5- Eventualmente, iteramos

Observaciones

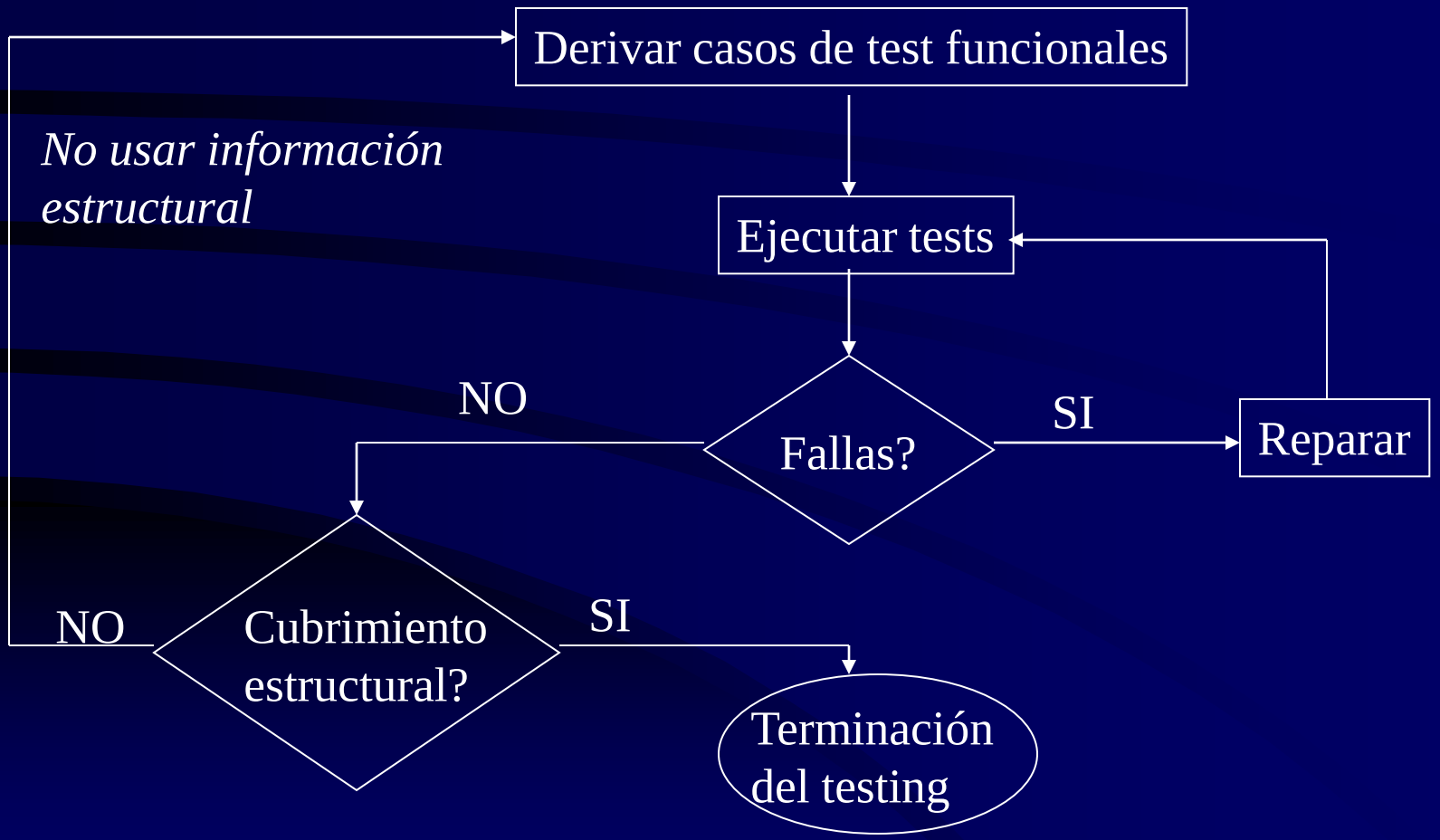
- El testing basado en código encuentra muchos errores
- Se ha realizado mucha investigación para determinar qué técnica estructural es la mejor

- Pero…

Problemas

- Cualquier técnica de selección de casos que no está basada en el comportamiento funcional, está mal guiada desde el comienzo
  - La gente no usa el software para ejecutar sus instrucciones, sino para invocar sus funcionalidades
  - Es fácil ejecutar todas las instrucciones y sin embargo no invocar ciertas funciones

Técnicas estructurales como *criterio de adecuación*



Compararación de Efectividad de Criterios para encontrar fallas

Modelo de Efectividad en laDetección de Fallas

- SD_C (P,S)
- Selección Random en cada D_i en SD_C (P,S)
- Sea d_i el cardinal de D_i
- Sea m_i el conjunto de inputs que revelan una falla en D_i

$$M(C,P,S) = 1 - ((1- m_1/d_1)*...*(1- m_n/d_n))$$

Subsumption

- Un criterio C_1 subsume (subsumes) a otro criterio C_2 si un conjunto de datos de test T satisface un criterio C_1, entonces T satisface C_2 (para todo par (P,S))

Subsumption y efectividad relativa de criterios

- C_1 subsumes a C_2 **no** implica
  $M(C_1, P, S) >= M(C_2, P, S)$

D domínio de P = {0,1,2}. P falla en el 0

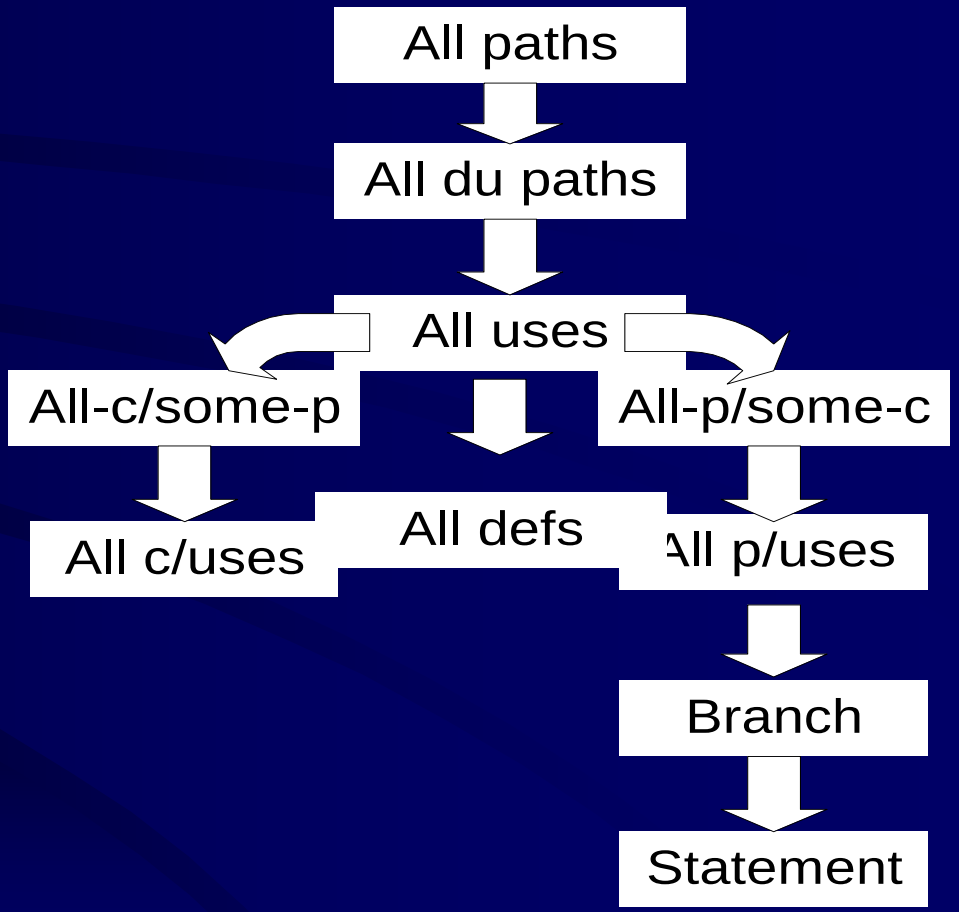
C1 requiere un test case de {0,1} y uno del subdominio {2}

C2 requiere un test case de {0,1} y uno del subdominio {0,2}

C1 subsume a C2 pero $\frac{1}{2}$ de las test suites adecuadas para C1 fallan. Mientras que $\frac{3}{4}$ de las test suites adecuadas para C2 fallan.

Subsumption en Criterios estructurales

- Un criterio *A* subsume a otro criterio B si un conjunto de datos de test *T* satisface un criterio *A*, entonces *T* satisface B



Properly Covers

Un criterio C_1 properly covers a otro criterio C_2 para (P,S) si vale que

SD_{C1} (P,S) = $\{D_1^1,\ldots, D_m^1\}$

SD_{C2} (P,S) = $\{D_1^2,\ldots, D_n^2\}$

Entonces existe

$M = \{D_{1.1}^1,\ldots, D_{1.k1}^1,\ldots, D_{n.1}^1,\ldots, D_{n.kn}^1\}$

- Tal que M es un sub-bag de SD_{C1} (P,S) y

  $D_1^2 = D_{1.1}^1 \cup \ldots \cup D_{1.k1}^1$

  …

  $D_n^2 = D_{n.1}^1 \cup \ldots \cup D_{n.kn}^1$

Properly Covers: Ejemplo

- P cuyo dominio es el intervalo de naturales [1,10]
- C2 un criterio son subdominios D21= [1,6], D22=[4,10]
- C1 con subdominios D11=[1,4],D12=[3,6],D13=[3,8], D14=[4,8], D14=[6,10]
- C1 properly covers C2
- M={D11,D12,D14,D15}

Propiedad del Properly Covers

- Teorema [Frankl,Weyuker '93]:
  Si C_1 properly covers C_2 entonces $M(C_1, P, S) >= M(C_2, P, S)$
- La relación es reflexiva y transitiva
- All-p-uses, all-uses, cobertura de condiciones (y otras) properly cover decisiones

Ejecución del testing

Ejecución del testing

- selección de datos
- dónde testear: ambiente de test
- terminación del testing
- documentación
- reporte y seguimiento de errores
- test de regresión

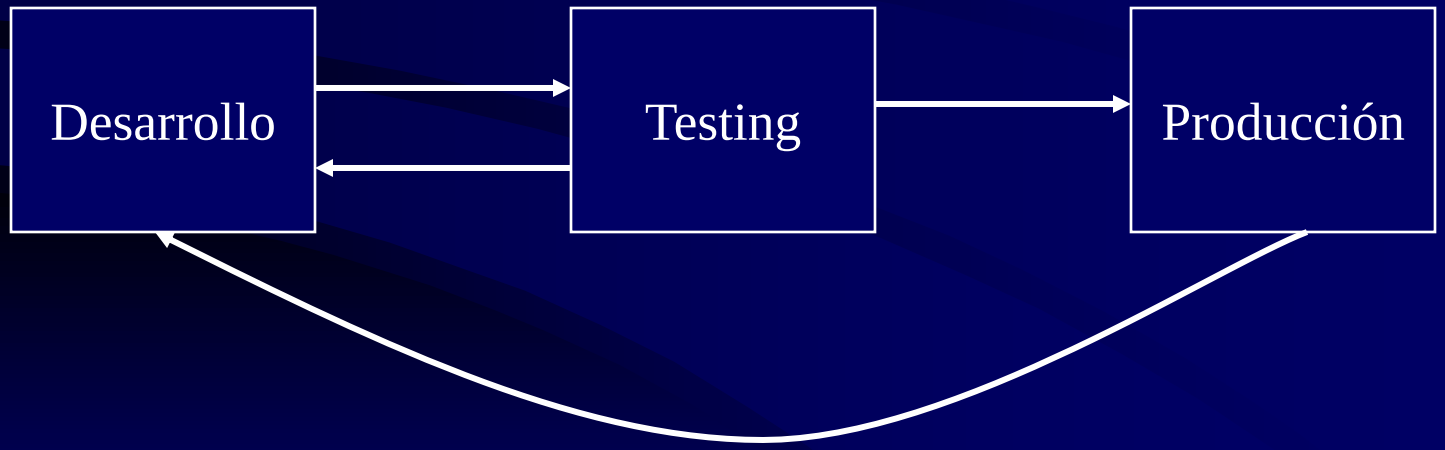
Ejecución del testing

- selección de datos
  - dónde testear: ambiente de test
- terminación del testing
- documentación
- reporte y seguimiento de errores
- test de regresión

Movimiento de componentes

programación
testing de unidades

integración,
testing del sistema
testing de aceptación



Ejecución del testing

- selección de datos
- dónde testear: ambiente de test
- terminación del testing
- documentación
- reporte y seguimiento de errores
- test de regresión

Terminación del testing

- Se terminó el tiempo o recursos
- Se corrieron todos los tests derivados sin detectarse ningún error
- % de cubrimiento de ciertas técnicas elegidas
- Fault-rate más bajo que un cierto valor especificado (# de errores por unidad de tiempo de testing)
- Se encontró un número predeterminado de errores (% del número total de errores estimado)

Ejecución del testing

- selección de datos
- dónde testear: ambiente de test
- terminación del testing
  - documentación
  - reporte y seguimiento de errores
- test de regresión

Documentos de Casos de test

- Criterios de derivación de casos empleados
- Casos de test organizados por
  - Módulo, sistema, unidad, etc.
  - Incluyendo el resultado esperado
  - con referencia al requerimiento que prueban
- Criterios de terminación del testing
- Datos de prueba

Documentos de reporte de ejecución

- ambiente de test
- selección de datos
  - referencia a casos de test que prueban
- ejecución y resultado obtenido
  - reporte de errores
- re-ejecución luego de las modificaciones pertinentes

Gerenciamiento de errores

- Determinación del origen del error
  - una vez detectado un error, se busca el problema que lo originó haciendo *debugging*
- Clasificación de errores
  - prioridad
  - severidad
- Seguimiento de errores

Gerenciamiento de errores

- Reparación del software
- Testing y testing de regresión
- Iterar hasta terminar

Ejecución del testing

- selección de datos
- dónde testear: ambiente de test
- terminación del testing
- documentación
- reporte y seguimiento de errores
- test de regresión

Test de Regresión

- Tareas de testing luego de que un sistema ha sido modificado
  - Algunos estudios dicen que la probabilidad de introducir un error al hacer un cambio es entre 50-80% [Hetzel: *The complete guide to software testing*, QED Info. Sciences, Wellesley, 1984]

¿Cuándo hacer test de regresión?

- Durante el desarrollo de software
- En tareas de mantenimiento, por
  - corrección
  - adaptación a nuevo ambiente
  - mejora de las prestaciones

Selección de casos de regresión

- Casos reusables: corresponden a partes del software no modificadas (ni especificación, ni implementación)
- Retesteables: la especificación no cambia, pero sí la implementación
- Obsoletos: no pueden seguir usándose
  - p.e., test case que especifica una relación I/O incorrecta, por modificación de la especificación

Bibliografía

- Beizer: *Software Testing Techniques*, Van Nostrand Reinhold, 1990.
- Myers: *The art of Software Testing*, John Wiley & Sons, 1979.
- Ghezzi et al.: *Fundamentals of Software Engineering*, Prentice Hall, 1992.
- Rapps, Weyuker: *Selecting software test data using data flow information*, IEEE Trans. on Software Engineering, SE-11(4):367-375, April 1985.
- Michal Young, Mauro Pezze: Software Testing and Analysis. 2006.

Terminología

- *Requerimientos de Test:* Qué quiero testear. En el marco del testing de sistemas reactivos se llama el Test Purpuse.
- *Especificaciones de Test* :Supuestos y definiciones que sirven para generar los casos de test para el requerimiento de test