Departamento de Informática | Universidad de Valladolid

Programación II (I.T.I de Gestión)

Prueba de programas

Félix Prieto
Esperanza Manso

Curso 2009/10

Introducción

INFORMÁTICA

Los fallos de 'software' cuestan 59.000 millones de dólares

EE UU recomienda mejorar los test durante el desarrollo

R. C.

Los errores del *software* son molestos y muy caros. El departamento de Comercio de EE UU ha calculado que este problema cuesta al país 59.500 millones de dólares (63.095 millones de euros) al año. El gobierno norteamericano recomienda mejorar la validación del *software*: abaratería el problema en 22.000 millones de dólares. Los programas que identifican y corrigen defectos suponen el 80% del coste de desarrollo. Sin embargo, la mitad de los errores se descubren al final del proceso o una vez comercializado. La National Academy of Sciences ha pedido crear una ley de garantía del *software*.

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY: www.nist.gov

El País 4/07/2002

Primeros conceptos

- La prueba de programas es un proceso muy costoso
- Para justificarlo debemos aumentar la fiabilidad del software probado
- Sólo podemos garantizar el aumento de la fiabilidad si localizamos errores ocultos en el software

Definición 1 Llamamos prueba al proceso de «ejecutar» un programa con el fin de encontrar errores en él. Diremos que la prueba fué *positiva* cuando localizó algún error y *negativa* en caso contrario.

Consecuencias de la definición

- La definición presenta la prueba como un proceso destructivo
- Con esta definición planteamos una meta realizable
- Aunque la localización de errores es «desagradable» la prueba positiva justifica la inversión realizada
- Con esta definición ponemos de manifiesto que debemos conseguir que el programa *haga lo que debe hacer* y también que *no haga lo que no debe*
- Los procesos de prueba son caros y deben ser reutilizados

Principios básicos de la prueba de programas

- La definición del resultado esperado es parte integrante de la prueba
- Un programador «nunca» probará su propio programa
- Una empresa «nunca» probará su propio programa
- Los errores más difíciles de localizar se refieren a malas interpretaciones de las especificaciones originales
- El ojo se acostumbra al error

Clasificación de los tipos de pruebas

- *Indirectas:* No utilizan una máquina real para la «ejecución» del programa en busca de errores
- *Directas:* Se basan en la selección de un conjunto de *casos de prueba* y su posterior ejecución en una máquina real
  - Pruebas de *Caja Negra*: Los casos de prueba se seleccionan en función de la especificación del programa
  - Pruebas de *Caja Blanca*: Los casos de prueba se seleccionan en función del código del programa

Pruebas indirectas

- Se basan en la lectura del elemento a probar
- Podemos usarlas en cualquier fase del desarrollo
- Se detectan las causas junto a los errores
  - Ahorran tiempo en depuración
  - Se descubren grupos de errores
- Capaces de detectar entre un 30 % y un 70 % de los errores de codificación
- Útiles antes de las pruebas directas (detección de los bloques más propensos a error)
- Permiten probar diseños, documentación,...

Inspecciones

- Reuniones de tipo formal
- Tres o cuatro personas, incluído el programador
  - Programadores experimentados
  - Expertos del dominio
- Moderador encargado de:
  - Nombrar a los inspectores
  - Repartir tareas y material
  - Fijar fecha y duración de las reuniones
  - Controlar que el programador resuelve los errores
  - Coordinar el informe final para los gestores

Inspecciones (II)

- Los inspectores disponen de *listas de chequeo*
- Mecánica de la inspección:
  - Lectura individual a la vista de las listas de chequeo
  - Puesta en común de lo detectado
  - El programador resuelve los errores
  - Se elabora un informe y se mejoran las listas de chequeo
- La inspección puede aceptar o rechazar el producto
- El resultado de la inspección no debería afectar laboralmente al programador

Recorridos y pruebas de escritorio

- Recorridos
  - Menos formales que las inspecciones
  - Sólo aplicables a código
  - Además de las listas de chequeo, el autor debe proporcionar casos de prueba suficientemente «simples»
  - La ejecución del código se realiza de forma manual
- Pruebas de escritorio
  - Lectura individual del propio programador
  - La peor de las estrategias de prueba indirecta

Pruebas de caja negra

- Forman parte de las pruebas directas
- Consisten en la elaboración de un conjunto de casos de prueba
- La salida esperada forma parte del caso de prueba
- Los casos de prueba deben contener tanto datos válidos como datos no válidos para el programa
- La única información disponible para su elaboración es la especificación original del programa
- La falta de acceso al código permite detectar errores en la interpretación de la especificación

Partición de equivalencia

- Definición 2 Llamamos *clase de equivalencia* para un programa a un conjunto de datos de entrada para el mismo de los que *podemos esperar* un comportamiento homogéneo. Una clase de equivalencia es *válida* cuando los datos representan entradas válidas para el programa, y es *no válida* en otro caso
- El comportamiento «esperado» del programa lo es a la vista de la especificación y la experiencia del programador
- La división en clases se realiza en función de las condiciones de entrada que aparecen en la especificación

Identificación de las clases de equivalencia

La condición de entrada especifica...

- *un rango:* Una válida (dentro del rango) y dos no válidas (una por encima y otra por debajo)
- *un número de valores:* Una válida (el número de valores especificados) y dos no válidas (más o menos valores)
- *un valor entre un conjunto de valores posibles:* Tantas clases validas como valores posibles y una no válida (valores no declarados como posibles)
- *un dato «debe ser...»:* Una válida (lo es) y otra no válida (no lo es)
- *Pero:* si es razonable estas clases se pueden subdividir

Identificación de las clases... (II)

- Las clases identificadas se almacenan en una tabla

Condición	Clases válidas	Clases no válidas
Cond1	Clase 1	Clase 4.1
		Clase 4.1
Cond2	Clase 2	Clase 5
Cond3	Clase 3	Clase 6

- Es importante describir de forma precisa condiciones y clases de equivalencia
- Es útil numerar las clases para la siguiente fase
- La numeración estructurada facilita la modificación posterior de la tabla

Derivación de los casos de prueba

- Mientras queden clases válidas por cubrir, elegiremos un nuevo caso caso de prueba que cubra *tantas clases válidas como sea posible*
- Mientras queden clases no válidas por cubrir, elegiremos un nuevo caso de prueba que cubra *una sola clase no válida*
- El criterio para las clases no válidas impide que los errores se enmascaren mutuamente
- Los casos de prueba se almacenan en una tabla, con indicación de las clases que cubren y salida esperada

Un ejemplo concreto

Un programa busca errores sintácticos en la construcción de identificadores de un lenguaje de programación. El programa devuelve un mensaje (OK o Error) conforme a las siguientes reglas:

- No tiene más de 15 ni menos de 5 caracteres
- Utiliza letras (mayúsculas y minúsculas), dígitos y el guión
- Se distinguen mayúsculas de minúsculas
- El guión no puede estar al principio o final del identificador
- Debe contener al menos un carácter alfabético
- No puede ser una palabra reservada (if, data, real,...)

Partición en clases de equivalencia

Descripciones tan precisas como sea necesario

Condición	C. Válidas		C. No válidas	
Entre 5 y 15	Entre 5 y 15	(1)	Menos de 5	(6.1)
			Más de 15	(6.2)
Car. permitidos	Todos permitidos	(2)	Alguno no permitido	(7)
Posición del guión	Guión interior	(3)	Guión al ppio.	(8.1)
			Guión al final	(8.2)
Al menos 1 letra	Al menos una	(4)	Ninguna	(9)
Palabras reservadas	No lo es	(5)	if	(10.1)
			data	(10.2)
			real	(10.3)

Qué pasó con diferencia entre mayúsculas y minúsculas

¿Guión ausente?

Tantas clases como palabras reservadas

Batería de pruebas

Distinguir casos válidos y no válidos

En los casos no válidos sólo reflejamos las clases no válidas cubiertas

Caso	Clases cubiertas	Salida esperada
Ident−01	1,2,3,4,5	OK
I−01	6.1	Error
Identificador−001	6.2	Error
Ident−$1	7	Error
−Ident01	8.1	Error
Ident01−	8.2	Error
0−0001	9	Error
if	10.1, 6.1	Error
data	10.2,6.1	Error
real	10.3, 6.1	Error

Todas las palabras reservadas resultan demasiado cortas. En todo caso las probamos, pero tambien probamos palabras cortas que no son reservadas

Sin la salida esperada la tabla no es útil

Análisis de valores límite

- La experiencia indica que los errores tienden a aparecer en los extremos de los campos de entrada
- Debemos elegir los casos de prueba con la mayor posibilidad de localizar los errores ocultos
- Estas afirmaciones no afectan a la selección de las clases de equivalencia sino a la elección de los casos de prueba
- Debemos definir una nueva forma de elaborar la batería
- La técnica completa se denomina "Partición de equivalencia con análisis de valores límite"

Análisis de valores límite

Condiciones	Derivación de los casos de prueba
1.-Rango de entrada	Valor máximo
	Valor mínimo
	Justo sobre el máximo
	Justo bajo el mínimo
2.- Número de valores	Número pedido
	Uno mas
	Uno menos
3.-Conjunto	Primer elemento
	Último elemento
Utilizar 1,2 y 3 en las condiciones de salida	

Batería de pruebas

Aplicamos análisis de valores límite al número de caracteres

Caso	Clases cubiertas	Salida esperada
Id−01	1,2,3,4,5	OK
Identificado−01	1,2,3,4,5	OK
I−01	6.1	Error
Identificador−001	6.2	Error
Ident−$1	7	Error
−Ident01	8.1	Error
Ident01−	8.2	Error
0−0001	9	Error
if	10.1, 6.1	Error
data	10.2,6.1	Error
real	10.3, 6.1	Error

¿Rango de letras? Deberíamos probar con A, a y las anteriores Z, z y las siguientes

¿Rango de números? Deberíamos probar con 0,9 anterior y siguiente

Conjetura de error

- Ciertas personas tienen especial habilidad para detectar errores
- La experiencia ayuda a localizar errores
- Si se supone que un caso de prueba puede detectar un error, no permitiremos que una técnica concreta de prueba impida aplicarlo.

Pruebas de caja blanca

- Se realizan a la vista del programa
- Los casos de prueba se seleccionan en función de la lógica del programa, no de la especificación
- No son útiles para detectar errores sobre la especificación original
- Permiten asegurar una prueba razonable de todo el código
- Se utilizan para complementar los resultados de las técnicas de caja negra
- ¿Qué es una prueba razonable del código?

¿Prueba razonable?

```
mientras |X − X'| > ε hacer
    X' ← X
    Calcular nuevamente X
fin_mientras
```

- ¿Ejecutar al menos una vez cada sentencia del programa?
- ¿Ejecutar todas las sentencias del programa en todas las secuencias posibles?
- ¿Ejecutar todas las sentencias del programa en un conjunto razonablemente amplio de secuencias?

Cobertura de sentencia

```
si (a>b)
  entonces a:= a−b
  si_no b:= b−a
fin_si
...
si (a>b)
  entonces a:= a−b
  (*Rama vacía*)
fin_si
```

- **Definición 3** Diremos que una batería de pruebas proporciona *cobertura de sentencia* cuando garantiza que se ejecuta cada sentencia del programa al menos una vez
- Cobertura necesaria pero *insuficiente*

Cobertura de decisión

```
si (a>b)
  entonces a:= a−b
  (*Rama vacía*)
fin_si
...
i:= 1
mientras (v[i] <> b)
     y (i<>5) hacer
  i:=i+1
fin_mientras
```

- **Definición 4** Diremos que una batería de pruebas proporciona *cobertura de decisión* cuando garantiza que en cada nodo de decisión se toma al menos una vez cada salida
- En el segundo ejemplo con b=7 v=(1,1,1,1,1) tenemos cobertura de decisión y no hemos probado ningún caso en que se encuentra el elemento
- *Suficiente sólo si las condiciones son simples*

Cobertura de condición

```
i:= 1
mientras (v[i] <> b)
     y (i<>5) hacer
  i:=i+1
fin_mientras
...
si (a>b) y (b primo)
  entonces a:= a−b
  (*Rama vacía*)
fin_si
```

- **Definición 5** Diremos que una batería de pruebas proporciona *cobertura de condición* cuando garantiza que en cada decisión cada condición toma todos los valores posibles
- En el primer ejemplo con dos casos de prueba (a=0, b=7) (a=5, b=4) tenemos la cobertura de condición pero no de decisión
- *No garantiza la cobertura de decisión, luego es insuficiente*

Cobertura de condición-decisión

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

- **Definición 6** Diremos que una batería de pruebas garantiza *cobertura de condición–decisión* cuando garantiza las coberturas de condición y decisión simultáneamente
- En el ejemplo con un solo caso de prueba (b=7, v=(1,1,1,1,7)) tenemos ambas coberturas, pero no hemos probado la situación en que 'b' no se encuentra en el vector.
- *Insuficiente, como la cobertura de decisión*

Cobertura de condición múltiple

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

- **Definición 7** Diremos que una batería de pruebas proporciona *cobertura de condición múltiple* cuando garantiza que se toman todas las combinaciones posibles de las condiciones dentro de cada decisión
- Tres casos de prueba como (a=7;v=(1,1,1,1,1)) (a=7;v=(7,1,1,1,1)) (a=7;v=(1,1,1,1,7))
- *Mínima para programas con condiciones mútiples*

Cobertura de condición múltiple (II)

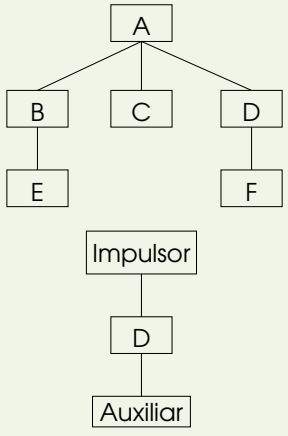
```
Si (x>1) y (x<2)
  entonces (*...*)
fin_si
```

- Las combinaciones deben ser analizadas dentro de cada decisión
- Cuando las decisiones tienen una sola condición las cuatro últimas coberturas son equivalentes
- No siempre son posibles todas las combinaciones de condiciones

Pruebas y estructura modular

- La estructura modular de los programas tiene relación con las pruebas puesto que
  - Permite manejar mejor las posibilidades combinatorias de las pruebas
  - Facilita la depuración posterior
  - Permite las pruebas en paralelo
- Es preferible probar los módulos de forma independiente y realizar una posterior prueba de integración

Herramientas necesarias



- *Módulos impulsores* que llaman al módulo objetivo con parámetros adecuados
- *Módulos auxiliares* que hacen el papel de los subordinados al módulo objetivo

Estrategias de prueba modular

- *No incremental* o gran explosión: Probamos cada módulo independientemente y luego realizamos la prueba de integración de todo el sistema
- *Incremental*: Utilizamos los módulos ya probados como impulsores o auxiliares
  - Ascendente, si comenzamos por los módulos inferiores hasta llegar al nivel del módulo principal
  - Descendente, si comenzamos probando el módulo principal, para descender a los módulos inferiores

Estrategias de prueba modular (II)

- La prueba no incremental permite trabajo mucho más «paralelo» pero requiere muchos auxiliares/impulsores
- La prueba incremental tiende a ser más económica, aunque errores no detectados en módulos ya probados pueden afectar a la prueba de otros.
- Se recomienda una mezcla entre las dos estrategias incrementales para disponer cuanto antes de módulos de entrada/salida

Estrategia general de prueba

- Elaboración de las baterías
  1. Pruebas indirectas
  2. Pruebas de caja negra: Análisis de valores límite
  3. Conjetura de error
  4. Completar con el método de caja blanca más adecuado
- ¿Cuándo detenerse?
  - No se encuentran errores
  - Cuando el plan de pruebas lo determine

Estrategia general de prueba (II)

- Pruebas de módulos
- Pruebas de integración: Centradas en la arquitectura del sistema
- Pruebas de validación: ¿Satisfacemos los requisitos?
- Pruebas de sistema: Interacción entre software y otros elementos
- Pruebas de instalación y aceptación

Tras el proceso de prueba...

Dificultades en el proceso de depuración

- El síntoma y la causa pueden estar físicamente alejados
- El síntoma puede no estar asociado a un error del programa (Errores de redondeo...)
- El síntoma puede desaparecer temporalmente enmascarado por otro error
- El síntoma puede estar relacionado por un error humano difícilmente detectable (Una suposición implícita)
- En ocasiones es difícil reproducir el error (Programas críticos, tiempo real...)

Estrategias de depuración: Fuerza bruta

- Volcado de memoria (fichero core)
  - Exceso de información estática
- Sembrado de sentencias
  - Modificamos el programa
  - Estructura `debug ... end` en Eiffel
  - Parámetros de compilación: `-debug_check`
- Programas depuradores (debugger)
  - Organizan la información de modo más dinámico

Estrategias de depuración: Vuelta atrás

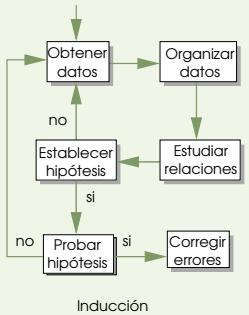
- Partimos del punto en que aparece el síntoma
- Recorremos el flujo de ejecución en sentido contrario hasta localizar la fuente del error
- El número de alternativas distintas crece rápidamente cuando el programa se hace grande

Estrategias de depuración: Método científico

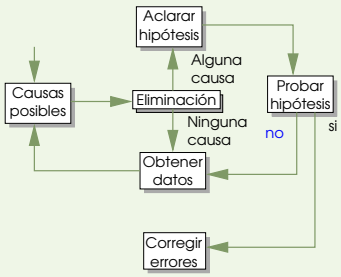
- Adecuado cuando los programas se hacen muy grandes
- Contrastar las hipótesis de error con experimentos (nuevos casos de prueba)
- Es preciso organizar la información disponible

	es	no es
¿Qué?	mediana inf. 3 erronea	Medianas inf. 1 y 2 ni medias de inf. 1,2,3
¿Donde?	informe 3	resto de informes
¿Cuando?	51 alumnos	2 o 50 alumnos
¿Alcance?	mediana 26 por 13	

Inducción y Deducción



Inducción



Deducción