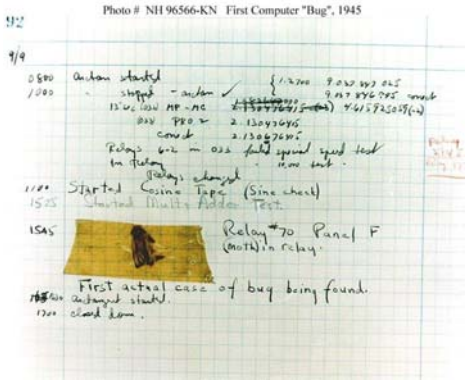
Laboratorio de Programación

Pruebas de Programas

Julio Villena Román
jvillena@it.uc3m.es

Introducción
Errores de software

- Un error en un programa puede ser algo muy serio



http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=all
(http://www.elmundo.es/elmundo/2005/11/11/ciencia/1131725311.html)
http://www5.in.tum.de/~huckle/bugse.html
http://www.cs.tau.ac.il/~nachumd/verify/horror.html

Introducción
¿Qué es probar?

- Probar es descubrir el mayor número posible de errores en un programa
- **¡Ojo!** No es convencerse de que el programa está bien...
  - Eso ya se hace en las fases anteriores del desarrollo del programa (o lo intentan)
- Probar un programa es buscarle fallos "a mala leche"
- El objetivo es poner en evidencia al programa

Introducción
La prueba exhaustiva es imposible

- Lo ideal sería probar el programa en todas las situaciones posibles
- Esto es imposible desde cualquier punto de vista:
  - Humano
  - Económico
  - Incluso matemático: bucles infinitos
- Explosivo número de combinaciones
- Habría que probar todas las posibles variaciones en los datos de entrada
- No podemos alcanzar la perfección... pero algo habrá que hacer

Introducción
Organización: Fases de prueba

- Prueba de Unidades
  - ***Caja Blanca***
    - Cobertura de segmentos
    - Cobertura de ramas
    - Cobertura de condición/decisión
    - Cobertura de bucles
  - ***Caja Negra***
    - Cobertura de requisitos
- Pruebas de Integración
- Pruebas de Aceptación

Prueba de Unidades

- Se trata de probar **módulos sueltos**
- Fase informal previa:
  - Ejecutar pequeños ejemplos
  - Hay herramientas que **analizan automáticamente** la sintaxis de los programas (¡e incluso "opinan" sobre fuentes potenciales de error!)
- Fase de prueba sistemática:
  - **Pruebas de caja blanca** (o transparente): Se mira con lupa la estructura del código escrito
  - **Pruebas de caja negra**: Se prueba la funcionalidad del módulo sin atender a su contenido

Prueba de Unidades
Caja blanca

- También llamadas:
  - pruebas **estructurales**
  - pruebas **de caja transparente**
- Su objetivo es probar exhaustivamente la **estructura** del **código**
- Cobertura: es una medida del porcentaje de código que ha sido probado o "cubierto" con las pruebas. Hay varios tipos:
  - Cobertura de segmentos
  - Cobertura de ramas
  - Cobertura de condición/decisión
  - Cobertura de bucles

Prueba de Unidades
Caja blanca

- Cobertura de segmentos **(o de sentencias)**
  - **Segmento**: secuencia de sentencias sin puntos de decisión
  - El nº de sentencias es finito. Hay que ser precavido a la hora de elegir cuándo parar
  - No se suele pasar por todas las sentencias sino por una mayoría elegida adecuadamente

Prueba de Unidades
Caja blanca

- Cobertura de ramas
  - ¿Qué ocurre con los segmentos opcionales?

```
if (condición) {
  EjecutaEsto()
}
```

  - Hay que probar cuándo la condición se cumple y cuándo falla
  - Refinamiento: hay que recorrer todas las posibles salidas de los puntos de decisión (y excepciones)

Prueba de Unidades
Caja blanca

- Cobertura de condición/decisión
  - ¿Qué ocurre cuando la expresión booleana es más compleja?

```
if (condición1 || condición2) {
  HazEsto()
}
```

  - Sólo 2 ramas, pero 4 posibles combinaciones
  - Hay que definir un criterio de cobertura sobre las combinaciones

Prueba de Unidades
Caja blanca

- Cobertura de bucles
  - Los bucles son una fuente inagotable de errores.
  - Pruebas para un bucle WHILE:
    1. 0 ejecuciones
    2. 1 ejecución
    3. Más de 1 ejecución
  - Pruebas para un bucle DO/WHILE:
    1. 1 ejecución
    2. Más de 1 ejecución
  - Pruebas para un bucle FOR:
    - Son muy seguros: basta ejecutarlos una vez (si dentro no se altera la variable de control o algo así)
    - … pero conviene examinarlos por si acaso

Prueba de Unidades
Caja blanca

- ¿Qué hacer en la práctica?
  - Acercarse al 100% de segmentos
  - Lograr una buena cobertura de ramas
  - La buena cobertura depende del tipo de programa:
    - Juegos basta con probar 60-80% del código
    - Aplicaciones críticas (sanitarias, centrales nucleares, aplicaciones militares, etc.) >90%
    - La cobertura requerida suele crecer con el grado de distribución del programa
  - Las pruebas de caja blanca se pueden hacer con un depurador *(debugger)*

Prueba de Unidades
Caja blanca

- Limitaciones
  - Las pruebas de caja blanca comprueban la **estructura** no la funcionalidad
  - Un programa puede estar bien, y sin embargo no servir a la función que pretende (lo que hace, lo hace bien, pero no hace lo que queríamos que hiciese)
  - Necesitamos comprobar la **funcionalidad** con pruebas de caja negra

Prueba de Unidades
Depuración *(debugging)*

- Ejecutar el programa paso a paso, conociendo el valor de las variables en todo momento
- El depurador se utiliza como consecuencia de la detección de un error
- Antes de entrar en el depurador hay que delimitar el error y sus posibles causas

Prueba de Unidades
Caja negra

- También llamadas:
  - Pruebas de **caja opaca**
  - Pruebas **funcionales**
  - Pruebas de **entrada/salida**
  - Pruebas **inducidas por los datos**
- Su objetivo es probar la funcionalidad del código
- Intentan encontrar casos en los que el módulo no atiende a su especificación
- Especialmente indicadas en los módulos que van a ser interfaz con el usuario

Prueba de Unidades
Caja negra

- Cobertura de especificación: nº de requisitos que se han probado
- **Problema**: el conjunto de datos es muy amplio
- **Solución**: analizar clases de equivalencia
  - Casos generalizados de uso
- **Recetas** para identificarlas:
  - Por debajo, en y por encima de un rango dado
  - Por debajo, en y por encima de un valor concreto
  - En un conjunto dado o fuera de él
  - Verdadero o falso

  (y lo mismo para las salidas)

Prueba de Unidades
Caja negra

- **Ejemplo**: Entero para identificar el día del mes.
  - Valores posibles: [1..31]
  - Tendremos tres clases de equivalencia:
    - Números menores que 1
    - Números entre 1 y 31
    - Números mayores que 31
- Atención a los valores singulares
- Se escoge un valor de cada clase (que no esté al límite)
- Probar también los valores frontera

Prueba de Unidades
Caja negra

- **Limitaciones:**
  - Las pruebas de caja negra comprueban la funcionalidad (el programa hace lo que queremos) pero...
  - No comprueban si el programa hace además cosas que no debería... (ej: un virus)
  - No es suficiente con sólo pruebas de caja negra

Pruebas de Integración

- Involucran varios módulos
- Pueden ser estructurales o funcionales
  - Estructurales: como las de caja blanca, pero analizando llamadas entre módulos
  - Funcionales*:* como las de caja negra, pero comprobando funcionalidades conjuntas
  - Se siguen utilizando clases de equivalencia **y** análisis de valores frontera
- Las **pruebas finales** consideran todo el sistema, cubriendo plenamente la especificación de requisitos del usuario

Pruebas de Integración
Algunas técnicas de diseño

- Diseño descendente: se comienza probando los módulos más generales
  - **Ventaja**: Piensa en términos de funcionalidad global
  - **Inconveniente**: No dispongo de los módulos inferiores
- Diseño ascendente: se comienza probando los módulos de base
  - **Ventaja**: No hay que construir módulos ficticios
  - **Inconveniente**: Se centra más en el desarrollo que en las expectativas del cliente
- Codificación incremental: se codifican sólo las partes de cada módulo necesarias para cada funcionalidad; una vez probada, se van añadiendo funcionalidades
  - **Ventaja**: Cuando hay mucha interacción con el usuario
  - **Inconveniente**: No tenemos módulos completos hasta el final

Pruebas de Aceptación

- Pruebas funcionales que realiza el **cliente** antes de poner la aplicación en producción
- Hay errores que sólo el cliente puede detectar...
  - "**el cliente siempre tiene razón**"
- Técnicas:
  - Pruebas alfa: Se invita al cliente al entorno de desarrollo, trabajando sobre un entorno controlado
  - Pruebas beta: Se desarrollan en el entorno del cliente, que se queda sólo con el producto en un entorno sin controlar
  - Ambas pruebas son habituales cuando se va a distribuir el programa a **muchos clientes**

Otros tipos de pruebas

- Recorridos *(walkthroughs)*
  - Se reúne a desarrolladores y críticos: los críticos se leen el código línea a línea y piden explicaciones a los desarrolladores
  - Eficaz para errores de naturaleza local
  - Pésima para localizar fallos en interacciones entre partes alejadas
- Aleatorias *(random testing)*
  - Se basa en que la probabilidad de descubrir un error es similar si se eligen pruebas al azar que si se utilizan criterios de cobertura
  - Razonable empezar las pruebas probando al azar
  - Pero es insuficiente en programas críticos

Otros tipos de pruebas

- Solidez *(robustness testing)*
  - ¿Cómo reacciona el sistema ante datos de entrada erróneos?
  - Por ejemplo, probar:
    - Todas las órdenes correctas
    - Órdenes con errores de sintaxis
    - Órdenes correctas, pero fuera de lugar
    - La orden nula
    - Órdenes con datos de más
    - Provocar una interrupción después de introducir una orden
    - Órdenes con delimitadores inapropiados
    - Órdenes con delimitadores incongruentes ( ]

Otros tipos de pruebas

- Aguante *(stress testing)*
  - Consiste en probar “hasta dónde aguanta” un programa por razones internas
  - Por ejemplo:
    - ¿Es capaz de trabajar con un disco al 95%?
    - ¿Aguanta una carga de CPU al 90%?
    - ¿Con qué porcentaje de la memoria ocupada es capaz de trabajar, etc.?
    - ¿Cuántos accesos concurrentes soporta?

Otros tipos de pruebas

- Prestaciones *(performance testing)*
  - Pueden ser importantes los parámetros de consumo: tiempo de respuesta, memoria ocupada, espacio en disco, ...
- Conformidad u Homologación *(conformance testing)*
  - Normas a las que el programa debe atenerse
  - Son pruebas de caja negra para comprobar que el programa se ajusta a las normas debidas
- Interoperabilidad *(interoperability testing)*
  - Buscan problemas de comunicación entre nuestro producto y otros con los que debe trabajar

Otros tipos de pruebas

- Regresión *(regression testing)*
  - Una nueva versión exige una nueva pasada de pruebas (¡de todo!)
  - Para reutilizar las pruebas, es necesario documentar éstas muy bien
- Mutación *(mutation testing)*
  - Introducir errores a propósito para verificar la bondad de las pruebas

Plan de pruebas

- Es un conjunto de pruebas
- Cada prueba debe:
  - Dejar claro **qué tipo de propiedades** se quieren probar
  - Dejar claro **cómo se mide** el resultado
  - Especificar **en qué consiste** la prueba
  - Definir **cuál es el resultado** que se espera
- Las pruebas “angelicales” carecen de utilidad

Plan de pruebas

- Un caso de prueba consta de:
  1. El propósito de la prueba
  2. Los pasos de ejecución de la prueba
  3. El resultado que se espera
- Todos estos puntos deben quedar perfectamente documentados
- Esto es laborioso, tedioso, aburrido y desagradable

Plan de pruebas

- Orden de las pruebas:
  1. Pasar pruebas de **caja negra** analizando valores límite
  2. Identificar clases de equivalencia de datos y añadir más pruebas de caja negra
  3. Añadir pruebas basadas en presunción de error: “¡Me lo temía!”
  4. Medir la cobertura de **caja blanca** que se ha logrado con las fases previas y añadir más pruebas de caja blanca

Aspectos psicológicos y de organización del trabajo

1. Probar es ejercitar un programa para **encontrarle fallos**, no para demostrar que funciona
2. Una prueba tiene **éxito** cuando encuentra un fallo
3. Las pruebas debe diseñarlas y pasarlas una **persona distinta** de la que ha escrito el código
4. **No se debe esperar** a que todo el código esté escrito para empezar a pasar las pruebas
5. Si en una sección se encuentran muchos fallos, hay que seguir **insistiendo** sobre ella
6. Si se detectan muchos fallos en un módulo, lo mejor es **desecharlo**, diseñarlo de nuevo y recodificarlo
7. **Las pruebas pueden encontrar fallos pero jamás demostrar que no los hay**
8. Las pruebas también tienen **fallos**

Conclusiones

- Probar es buscarle fallos a un programa
- La fase de pruebas absorbe una buena porción de los costes de desarrollo de software
- Hay herramientas, pero al final hace siempre falta el artista humano

Caso práctico

- Desarrollar un plan de pruebas para el método siguiente:

  public boolean Busca (char C, char V [ ])

- El procedimiento devuelve TRUE si C está en V, y FALSE si no

Caso práctico

- Primero identificamos **clases de equivalencia**:
  - C: Está poco especificado
  - V: No se dice nada de las dimensiones del array ni del criterio de ordenación
  - Lo que devuelve: esto sí está claro
- Antes de nada, hay que **aclarar estos puntos** en la especificación del problema

Caso práctico

- **Nueva versión** de la especificación:
  - A este procedimiento se le proporciona un carácter C y un array V de caracteres. Se admitirá cualquier carácter de 8 bits de los representables en un PC con Java. El array podrá tener entre 0 y 10.000 caracteres y deberá estar ordenado alfabéticamente, en orden ascendente. Es admisible cualquier cadena de caracteres. El procedimiento devuelve TRUE si C está en V, y FALSE si no. Trabajamos en Java.

Caso práctico

- Ahora podemos identificar las **clases de equivalencia**:
  - C:
    1. Cualquier carácter
  - V:
    1. El array vacío.
    2. Un array entre 1 y 10.000 elementos, ordenado
    3. Un array entre 1 y 10.000 elementos, desordenado
  - Resultado:
    1. TRUE
    2. FALSE
- Cabe considerar **combinaciones significativas** de datos de entrada: que C sea el primero o el último del array

Caso Práctico

Pruebas de caja negra

1. Pruebas de **caja negra**: valores límite
   1. Buscar el carácter “k” en el ARRAY “” → debe devolver FALSE
   2. Buscar el carácter “k” en el ARRAY “k” → debe devolver TRUE
   3. Buscar el carácter “k” en el ARRAY “j” → debe devolver FALSE
   4. Buscar el carácter “k” en el ARRAY “kl” → debe devolver TRUE
   5. Buscar el carácter “k” en el ARRAY “jk” → debe devolver TRUE
   6. Buscar el carácter “k” en el ARRAY de 10.000 “a” → debe devolver FALSE

   (… y pruebas referentes a la ordenación del array)

Caso Práctico
Pruebas de caja negra

2. Pruebas de **caja negra**: valores normales
   1. Buscar el carácter “k” en el ARRAY “abc” → debe devolver FALSE
   2. Buscar el carácter “k” en el ARRAY “jkl” → debe devolver TRUE

Caso Práctico
Pruebas de caja blanca

```
1.  public boolean Busca ( char C, char V[] ) {
2.    int a, z, m;
3.    a = 0;
4.    z = V.length – 1;
5.    while ( a <= z ) {
6.      m = ( a + z ) / 2;
7.      if ( V[m] == C )
8.        return true;
9.      else {
10.       if ( V[m] < C )
11.         a = m + 1;
12.       else
13.         z = m – 1;
14.     }
15.   }
16.   return false;
17.}
```

Para las pruebas de **caja blanca** necesitamos conocer el código interno

Caso Práctico
Pruebas de caja blanca

- Si ejecutamos las pruebas anteriores marcando por dónde vamos pasando sobre el código, sólo nos queda por probar la **rama de la línea 12**
- Hay que añadir un caso **adicional** para pruebas de caja blanca:
  1. Buscar el carácter “k” en el ARRAY “l” → debe devolver FALSE
- Cobertura del 100% de segmentos y de condiciones

Caso Práctico
Pruebas de caja blanca

- **Formalización** del banco de pruebas:
  - `If (Busca(“k”, “”)) System.out.println(“falla 1.1”);`
  - `If (!Busca(“k”, “k”)) System.out.println(“falla 1.2”);`
  - `If (Busca(“k”, “j”)) System.out.println(“falla 1.3”);`
  - `If (!Busca(“k”, “kl”)) System.out.println(“falla 1.4”);`
  - `If (!Busca(“k”, “jk”)) System.out.println(“falla 1.5”);`
  - `If (Busca(“k”, “aaaa”)) System.out.println(“falla 1.6”);`
  - `If (Busca(“k”, “abc”)) System.out.println(“falla 2.1”);`
  - `If (!Busca(“k”, “jkl”)) System.out.println(“falla 2.2”);`
  - `If (Busca(“k”, “l”)) System.out.println(“falla 3.1”);`

Caso Práctico
Pruebas de solidez

- Podríamos pasar **pruebas de solidez**:
  - ¿Qué ocurre si sobrepasamos el límite de 10.000 en el array?
  - ¿Qué ocurre si el array estuviese desordenado?