Programación II

Bloque temático 1. Lenguajes de programación

Bloque temático 2. Metodología de programación

Tema 2. Programación dirigida por eventos

Tema 3. Verificación de programas

Bloque temático 3. Esquemas algorítmicos

Tema 3. Verificación de programas

3.1. Verificación y Validación
3.2. Pruebas del software
3.3. Caja negra: particiones de equivalencia
3.4. Herramienta JUnit
3.5. Prueba de estados
3.6. Bibliografía

3.1 Verificación y Validación

VyV **(o V&V): procesos de comprobación y análisis para tratar de asegurar que el software esté acorde con sus requisitos y cumpla las necesidades de los clientes**

- ***Verificación***: ¿estamos construyendo el producto *correctamente*?
  - conjunto de actividades que aseguran que el software cumple sus requisitos (hace lo que se supone que tiene que hacer)
- ***Validación***: ¿estamos construyendo el producto *correcto*?
  - conjunto (diferente) de actividades que aseguran que el software construido se corresponde con los requisitos reales del cliente

Ejemplo de Verificación y Validación

VyV de un programa que permite gestionar los alumnos matriculados en los cursos de una academia

Verificación:

- ¿las operaciones de gestión de las listas de alumnos funcionan correctamente?
- ¿las operaciones de gestión de cursos funcionan correctamente?

Validación:

- ¿El programa proporciona todas las operaciones que el usuario necesita?
- ¿Las operaciones hacen lo que el usuario espera?
- ¿La interfaz con el usuario es la apropiada?

© Mario Aldea Rivas

Técnicas de Verificación y Validación

El proceso VyV utiliza dos técnicas de comprobación y análisis:

- *Inspecciones del software*:
  - revisión sistemática de todos los documentos generados en el proceso de desarrollo del software (requisitos, diseño, código)
  - es una técnica estática: no se necesita que el sistema se ejecute
- *Pruebas del software*:
  - contrasta la respuesta del sistema con los resultados esperados
  - es una técnica dinámica: se necesita ejecutar el sistema completo (o al menos un prototipo o alguna de sus partes)

© Mario Aldea Rivas

Depuración

- *VyV*: proceso que permite detectar la existencia de defectos
- *Depuración*: proceso que localiza y corrige esos defectos
  - es un proceso difícil: los defectos no siempre se detectan cerca del punto que los provocó
  - suele ser necesario diseñar programas de prueba que permitan repetir y observar el defecto
  - se utilizan los depuradores

Después de corregir un defecto es necesario volver a probar el sistema (*pruebas de regresión*)

- para detectar nuevos fallos introducidos al tratar de corregir un defecto ("*regresiones*")
- lo ideal sería repetir todas las pruebas, cuando es muy costoso se repiten únicamente las relacionadas con el fallo corregido

© Mario Aldea Rivas

Realización de las pruebas

Se debe probar con la mayor frecuencia que sea posible

Hay que dedicar los suficientes recursos:

- las pruebas suelen ser más costosas que el propio programa

Importancia del informe de pruebas:

- debe permitir comparar con resultados anteriores para detectar posibles regresiones

Ejemplos:

- Compilador GNAT: probado 2 veces al día en decenas de computadores, miles de programas de prueba, se prueban todas las configuraciones posibles y todas las arquitecturas
- MaRTE OS: probado 2 veces al día, se prueban todas las arquitecturas, 4000 programas de prueba, tardan 2 horas

3.2 Pruebas del software

Las pruebas se realizan en cuatro *etapas*:

1. Prueba de componentes (prueba de métodos y clases)
   - se prueba cada método y clase de forma independiente
2. Prueba de integración o de subsistemas
   - se prueban agrupaciones de clases relacionadas
3. Prueba de sistema
   - se prueba el sistema como un todo
4. Prueba de validación
   - prueba del sistema en el entorno real de trabajo
   - con intervención del usuario final

El descubrimiento de un defecto en una etapa requerirá la repetición de las etapas de prueba anteriores

Tipos de pruebas de software

- *Pruebas de defectos*
  - Buscan diferencias entre lo que el programa hace y lo que debería hacer (su especificación)
  - Una prueba exitosa es la que descubre un defecto
  - Sirven para demostrar la presencia, *no la ausencia*, de defectos
- *Pruebas estadísticas*
  - Miden la fiabilidad (número de caídas por unidad de tiempo) y los tiempos de respuesta del sistema
  - Se realizan exponiendo el programa a la misma carga de trabajo que soportará en su entorno final

Nos centraremos en las *pruebas de defectos* realizadas durante la *prueba de componentes* en sistemas orientados a objetos

Prueba de defectos en componentes (clases)

Idealmente se realiza en dos fases:

1. **Prueba de métodos:**
   - **se analiza el resultado de ejecutar cada método bajo distintas condiciones (distintos valores de los parámetros de entrada y de los atributos de la clase)**
2. **Prueba de estados:**
   - **se analiza la evolución del estado de la clase (valor de sus atributos) bajo distintas combinaciones de llamadas a sus métodos**

Normalmente ambas fases no son independientes:

- **probar un método bajo distintas condiciones requiere llevar al objeto a distintos estados (invocando otros métodos)**
- **p.e. la operación "busca" de una lista debería probarse cuando la lista esté vacía y cuando contenga elementos**

Prueba de métodos

Principales mecanismos para la prueba de métodos:

- **Pruebas funcionales o de "*caja negra*"**
  - **los casos de prueba se basan en la especificación del método**
  - **no se requiere conocimiento de su estructura interna**
  - **no es necesario disponer del código fuente**
- **Pruebas estructurales o de "*caja blanca*" o "caja de cristal"**
  - **los casos de prueba se seleccionan en función del conocimiento que se tiene de la estructura del método**
  - **es necesario disponer del código fuente**

No se trata de técnicas alternativas, más bien enfoques complementarios que permiten detectar distintos tipos de errores

3.3 Caja negra: particiones de equivalencia

En general es imposible probar un método para todas las combinaciones posibles de entradas

En lugar de eso los datos de entrada se dividen en *particiones de equivalencia*:

- **conjunto de datos de entrada que se espera que tengan un comportamiento equivalente**

Los casos de prueba se eligen en función de las particiones:

- **se elige un caso central por partición: caso típico**
- **se elige casos correspondientes a las fronteras con otras particiones: casos atípicos**

Las particiones se identifican utilizando la especificación del método y nuestra propia experiencia

Ej.: particiones de equivalencia con caja negra

Especificación de un método de búsqueda:

```
método Busca(Elemento ele, Elemento[] t)
    retorna Booleano encontrado, Entero pos
  {Pre: longitud de t > 0}
    busca elemento en tabla
  {Post:(t[pos]=ele & encontrado=true) |
        (ele no en t & encontrado=false) |
        (eleva excepción SECUENCIA_VACÍA)}
fmétodo
```

Particiones de equivalencia

- **“el elemento está en la secuencia” / “el elemento no está en la secuencia”**
- **“secuencia vacía” / “la secuencia tiene un elemento” / “la secuencia tiene más de un elemento”**

© Mario Aldea Rivas

Criterios de prueba generales para tablas:

- **utilizar tablas de distintos tamaños**
- **acceder a los elementos primero, central y último**

Casos de prueba basados en las particiones y los criterios:

Elección del caso de prueba	Casos de prueba		Salida esperada	
	t	ele	encontrado	pos
Secuencia vacía	vacía	20	excepción	
Secuencia de un elemento	17	17	true	0
	17	18	false	-
Frontera entre uno y más elementos	17, 18	18	true	1
	11, 4	1	false	-
Secuencia de más de un elemento: encontrado (primero, central y último) y no encontrado	17, 29, 21, 23	17	true	0
	41, 18, 9, 31, 30, 16, 45	45	true	6
	17, 18, 21, 23, 29, 41, 38	23	true	3
	21, 23, 29, 33, 38	25	false	-

© Mario Aldea Rivas

3.4 Herramienta JUnit

- **JUnit es una herramienta de código abierto que permite ejecutar conjuntos de pruebas de forma rápida y sistemática**
  - **integrada en el entorno Eclipse**
  - **para *prueba de clases***
  - **http://www.junit.org**
- **Pensada para realizar pruebas repetidamente cada vez que se realiza un pequeño cambio en el código:**
  - **siguiendo el lema de la metodología *Extreme Programming* (XP): “*Code a little, test a little*”.**
- **Configuración del proyecto para usar JUnit:**
  1. **Botón derecho del ratón sobre el proyecto y elegir: `Build Path => Configure Build Path`**
  2. **En la ficha `Libraries` elegir `Add Library`, seleccionar `JUnit` y pulsar `Next`, elegir la versión `JUnit 4`**

© Mario Aldea Rivas

Clase probadora

Creación de una *clase probadora*:

1. Pulsar con el botón derecho sobre la clase a probar y elegir `New => Other => Java => JUnit => JUnit Test Case`
2. Seleccionar los métodos a probar y pulsar `Finish`
3. Se crea la clase probadora con un conjunto de *métodos de prueba* que comienzan por la palabra "*test*"
4. Podemos añadir más métodos de prueba (también deberán comenzar por la palabra "*test*"

Métodos de prueba

- Son ejecutados automáticamente por la herramienta JUnit
  - no debemos asumir ningún orden de ejecución (podrían ejecutarse en cualquier orden)
- Debemos escribir en ellos nuestros casos de prueba
- Para cada método se diferencian tres comportamientos:
  - correcto: si finaliza sin lanzar ninguna excepción
  - fallo: si lanza la excepción `AssertionFailedError`
  - error: si lanza cualquier otra excepción
- Para detectar los fallos usaremos el método:

```
void assertTrue(String msj, Boolean cond)
```

  - lanza `AssertionFailedError` con `msj` como mensaje asociado cuando `cond` es false

Ejemplo de clase probadora sencilla

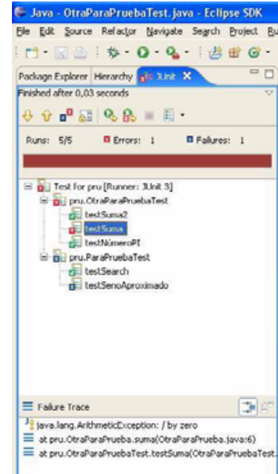
```
public class MiClaseBajoPruebaTest extends TestCase{

  // objeto de la clase que quiero probar
  MiClaseBajoPrueba bajoPrueba =
                              new MiClaseBajoPrueba();

  public void testNúmeroPI() {
    // mis casos de prueba
    final double mxError = 0.0001;
    double pi = bajoPrueba.númeroPI();
    assertTrue("Esperado "+Math.PI+
        "+/-"+mxError+" Obtenido:"+pi,
        Math.abs(Math.PI-pi)<mxError);
  }

}
```

Resultados y Depuración de métodos de prueba



Obtención de los resultados:

- Ejecutando la clase probadora ("*run*")

Resumen de resultados con:

- "*Failures*": fallo detectado con `assertTrue`
- "*Errors*": excepción inesperada

Es posible depurar un método de pruebas individualmente:

1. situar al menos un punto de ruptura en el método a depurar
2. pulsar con el botón derecho sobre el método de prueba y elegir `debug`

3.5 Prueba de estados

La prueba completa de una clase se divide en:

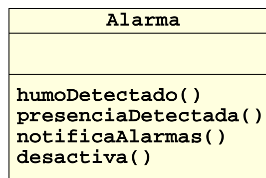
- Prueba de métodos: realizada con los mecanismos de "caja negra" (y/o "caja blanca" ← no visto en la asignatura)
- *Prueba de estados*: prueba todas las transiciones entre todos los estados en que se puede encontrar el objeto

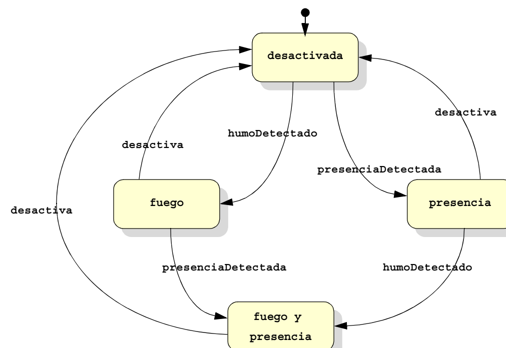
La prueba de clase o de estados:

- se apoya en el diagrama de estados de la clase
- persigue diseñar casos de prueba que permitan asegurara que:
  - se alcanzan *todos los estados*
  - se siguen *todas las transiciones* entre estados
- cuando el número de casos de prueba resulta inabordable se tratan de agrupar las transiciones en particiones de equivalencia

Ej.: Prueba de estados

Clase Alarma y su diagrama de estados:





Casos de prueba

- **al realizarles en el orden que aparecen se consigue alcanzar todos los estados pasando por todas las transiciones**

Casos de prueba	Salida esperada
notifica	desactivada
humoDetectado, notifica	fuego
desactiva, notifica	desactivada
humoDetectado, presenciaDetectada, notifica	fuego y presencia
desactiva, notifica	desactivada
presenciaDetectada, notifica	presencia
desactiva, notifica	desactivada
presenciaDetectada, humoDetectado, notifica	fuego y presencia

3.6 Bibliografía

[1] **Eric J. Braude, “Ingeniería de Software”. Alfaomega, 2003.**

[2] **Ian Sommerville, “Ingeniería de software” (6ª edición). Pearson Educación de México, 2002.**

[3] **JUnit.** `http://www.junit.org/`