{PROG} dit UPM

Programación
Tema 8: Estilo y pruebas

{PROG} dit UPM

Contenidos

- Estilo
- Pruebas
- Depuración

Calidad del Software

- Estilo, prueba, depuración: aspectos relacionados con el…
- Aseguramiento de la **Calidad del Software** (QA, Quality Assurance):
  Conjunto de políticas y actividades para:
  - Definir objetivos de calidad
  - Ayudar a asegurar que los productos y procesos software cumplen esos objetivos
  - Saber en qué medida lo hacen
  - Mejorar la calidad progresivamente

{PROG} dit UPM

Estilo de código

- Comprensible
- Cohesión
- Desacoplamiento
- Formato (sangrado…)
- Nombres consistentes
- Ocultamiento de información
- Comentarios significativos

Estilo: objetivos

- Disminuir errores
- Asegurar **legibilidad**
  - El código se escribe para ser leído por otros (no solo por el compilador)
- Abaratar/**facilitar mantenimiento**
- Estándares de codificación en Java:
  - The Elements of Java Style
    Vermeulen et.al. SIGS Books

Estilo: reglas generales

- Simplicidad
  - **Clases y métodos simples**
- Claridad
  - Propósito claro de cada elemento: dónde, cuándo, porqué y cómo usarlo
- Completitud
  - Documentar todos los aspectos necesarios
- Consistencia
  - Entidades similares deben tener aspecto y comportamiento similar
- Robustez
  - Comportamiento predecible y documentado en respuesta a errores y excepciones
  - **No ocultar errores, ni forzar a los clientes a descubrirlos**

Estilo: nombrado

- Clases, Interfaces: Mayuscula-Mayuscula
  - Ej. Punto, PolinomioGrado2
- Métodos: minuscula-Mayuscula
  - Ej. getX(), distancia(), buscaMayor()
- Atributos: minuscula-Mayuscula
  - Ej. nombrePropio, x, y, apellidos
- Constantes: MAYUSCULA + “_”
  - Ej. ORIGEN_X
- Evitar nombres con caracteres acentuados, ü, ñ

Estilo: parámetros

- Orden de parámetros:
  - Entrada/modificación/salida
  - Consistente en todos los métodos
- Usar todos los parámetros
- Limitar número de parámetros (nunca >7)
- Documentar suposiciones sobre parámetros:
  - **Tipo de acceso, unidades, rangos, valores no válidos**

Estilo: comentarios

- Comentarios de código

```
/* ...
*/
// comentario táctico: hasta fin de línea
```

- Comentarios de documentación

```
/**
**/
```

- Herramienta **javadoc** para generar páginas HTML de documentación (ya explicado)

{PROG} dit UPM

Contenidos

- Estilo
- Pruebas
- Depuración

{PROG} dit UPM

Pruebas

- Valen para detectar errores
- Una prueba es tanto mejor cuantos menos errores pasan desapercibidos
- Hay que programar, ejecutar y documentar las pruebas antes de dar por acabado una clase
- Permiten ver cómo se usa la clase
- Al hacer pruebas hay que **intentar que falle** el programa o la clase por todos los medios
- Normalmente hay que hacer varias pruebas: una **batería de pruebas**

Tipos de pruebas funcionales

- **Pruebas unitarias**: prueban el menor elemento posible, una clase o un método
- **Pruebas de integración**: interacción entre clases o paquetes
- **Pruebas de sistema**: prueban el programa en contexto real
- **Pruebas de aceptación**: prueba del programa para ver que satisface los requisitos

Enfoques de prueba

- De **caja negra**
  - cuando conocemos la parte pública de una clase
- De **caja blanca**
  - cuando conocemos la parte privada y forzamos la ejecución de todo el código

Casos de prueba

- ¿Qué hay que probar?
  - ejecutar al menos una vez cada sentencia
    - cobertura de sentencias
  - ejecutar al menos una vez cada condición con resultado cierto y falso
    - cobertura de ramas
- Método
  - si va tachando el código probado 100% cuando todo esté tachado

Casos de prueba

- if (...)
  - si T, si F
- switch (...)
  - cada ‘case’ + ‘default’
- cobertura de bucles
  - for -> 3 pruebas: 0 veces, 1 vez, n>1 veces
  - repeat -> 2 pruebas: 1 vez, n>1 veces
  - while -> 3 pruebas: 0 veces, 1 vez, n>1 veces

Datos de prueba

- ¿Con qué datos se prueba?
  - divida el espacio de datos en **clases de equivalencia**
    - clase de equivalencia:
      datos que provocan el "mismo comportamiento"
    - no parece que deban provocar comportamientos diferentes
  - elija un **dato "normal"** de clase de equivalencia
  - pruebe con todos los **datos "frontera"**:
    valores extremos de la clase de equivalencia
- Añada aquellos **casos en los que sospeche** que el programador puede haberse equivocado
- Pruebe todas las combinaciones
  { datos × comportamiento }

Pruebas automáticas: JUnit

- **Framework** para pruebas unitarias de clases
  - Facilita la escritura de pruebas
  - Automatiza su ejecución
  - Uso independiente (desde línea de comandos) o **integrado en IDEs** (Eclipse, Netbeans)
  - Autores: Kent Beck and Erich Gamma.
  - Web: junit.org
  - Versión estable: 4.11 (nov.2012)

Casos de prueba

- Conjunto de **métodos java (anotados)**:
  - secuencia de operaciones, entradas y valores esperados (escritos por quien prueba)
  - Sustituyen el uso de main() para probar
- **Añade aserciones** (clase) para comprobar:
  - Si dos objetos son iguales
  - si dos referencias a objetos son idénticas
  - Si una referencia a un objeto es nula o no

Prueba con JUnit 4.x

- **Importar clases** JUnit

```
import org.junit.*;
import static org.junit.Assert.*;
```

(import static: para referenciar métodos/atributos sin citar la clase)

- Declarar clase de prueba (normal)

```
public class MiClaseTest {
```

- Declarar variables para prueba:

```
MiClase c;…
```

- Uso de **anotaciones** (metadatos) @...
  - Instrucciones para el compilador y otras herramientas
  - Igual que para JavaDoc: (@author, @param...)

Anotación de métodos

- **@Test**: método que representa un caso de prueba
  - Comprueba si el resultado esperado es igual al real
  - Métodos Assert: assertTrue, assertFalse, assertEquals, assertNull, assertNotNull, assertSame, assertNotSame
- **@Before/@After**: métodos que deben ejecutarse antes/después de cualquier caso de prueba
- **@BeforeClass/@AfterClass**: se ejecutan **una vez**, **antes/después** de todos los test de la clase
  - Puede haber varios métodos etiquetados con @Before y @After
  - Solo un @BeforeClass y un @AfterClass

Ejemplo 1

```
import org.junit.*;
import static org.junit.Assert.*;
@Before
public void setUp() {
 ej = new ClaseEjemplo();
}
@Test
public void testSuma( ) {
 int a=3 , b=6;
 int resultEsperado = (a+b);
 int resultReal = ej.suma(a, b);
 assertEquals(reusultEsperado, resultReal);
}
@After
public void tearDown() {
 ej = null;
}
```

Métodos de clase de prueba

- Sugerencia: nombrar métodos que comiencen por **test**
  - Llama al método que se desea probar y obtiene el resultado
  - Comprueba que el resultado es el esperado
  - Se repiten estos pasos tantas veces como sea necesario

Aserciones

- **assertEquals (esperado, obtenido)**
  - Comprueba si los objetos (o variables de tipo primitivo) son iguales, empleando el método equals() si está definido. Si no, emplea ==
- **assertArrayEquals (esperado, obtenido)**
  - arrays de la misma longitud
  - para cada valor del índice, se comprueba:

    assertEquals(expected[i],actual[i])

    o

    assertArrayEquals(expected[i],actual[i])

Aserciones (cont.)

- **assertTrue** (boolean comprobacion) - **assertFalse**
  - Éxito si la comprobación es cierta (falsa)
- **assertSame** (Objeto esperado, Objeto obtenido) - **assertNotSame**
  - Éxito si referencias iguales empleando == (o distintas)
- **assertNull** (Objeto objeto)
  - Éxito si el objeto es null (o no lo es)
- **fail** (String mensaje)
  - Causa fallo en la prueba. Para comprobar si una excepción se lanza y se captura.

Ejemplo 2

```
public class Contador {
    private int contador = 0;

    public int incrementa() {
        return ++contador;
    }

    public int decrementa() {
        return --contador;
    }

    public int getContador() {
        return contador;
    }
}
```

Los métodos triviales como getContador() no necesitan prueba asociada.

Ejemplo 2 (cont)

```
import static org.junit.Assert.*;
public class TestContador {
     private Contador miContador;

    @Before
     public void setUp() throws Exception {
          miContador= new Contador();}

    @After
     public void tearDown() throws Exception {}

    @Test
     public void testIncrementa1() {
          assertEquals(1, miContador.incrementa());}

    @Test
     public void testIncrementa2() {
          miContador.incrementa();
          assertEquals(2, miContador.incrementa());}

    @Test
     public void testDecrementa() {
          assertEquals(-1, miContador.decrementa());
     }
}
```

Cada prueba comienza con un contador *nuevo,* con lo que no hay que preocuparse del orden de las pruebas

Cada prueba permite comprobar un elemento específico=> un *assert*

Aserciones: mensajes

- Cualquier aserción tiene una signatura adicional con un **parámetro String inicial**.
  - Es un mensaje adicional que se incluirá en el mensaje de fallo (si lo hubiera)
- Ejemplos:

```
assertTrue (mensaje, comprobacion)
assertEquals (mensaje, esperado, real)
```

Aserciones sobre reales

- En la comparación de resultados reales se requiere un **parámetro adicional (delta)**
  - Para evitar problemas de redondeo, se evalúa:

```
Math.abs( resEsperado - resReal) <= delta
```

- Ejemplo:

```
assertEquals (resEsperado, resReal, 0.0001)
```

Prueba de excepciones

- Pasando parámetro a la **anotación @Test con el tipo de excepción esperada**:

```
@Test(expected=ImporteNoDisponibleException.class)
  public void reintegroExcesivo () throws
      ImporteNoDisponibleException {
      Cuenta c = new Cuenta("José Pérez", 1000);
      c.reintegro(1001);
  }
```

- El test falla si no se lanza excepción (o si se lanza otra distinta de la esperada)

Tiempo de ejecución

- Para evitar bucles infinitos, se pueden establecer **límites de tiempo** (en milisegundos)
- La prueba falla si el método se demora en exceso:

```
@Test (timeout=10)
 public void testTimeout() {
    Cuenta c = new Cuenta ("José Pérez", 1000);
    c.bucleInfinito();
}
```

{PROG} dit UPM

Contenidos

- Estilo
- Pruebas
- Depuración

{PROG} dit UPM

Depurar

- Consiste en encontrar y arreglar los errores detectados durante las pruebas
- Actividad difícil, manual
- Es necesario imaginar el flujo de ejecución y seguirlo para encontrar los errores
- Varias técnicas:
  - Revisión manual: revisar el listado del programa, llevando la cuenta de cómo están los objetos
  - Revisión en voz alta: leer el programa a otros
  - Inserción de trazas: sentencias System.out.println();
  - **Uso del depurador**