Pruebas del Software

Information Engineering Research Group

Contenido

- Repaso de los conceptos básicos de Prueba de Software vistos en la parte teórica del curso
- Descripción del framework JUnit
- Adaptaciones del framework JUnit
- Conceptos avanzados en JUnit
  - Pruebas de métodos no públicos
  - Pruebas de excepciones
- Pruebas de aislamiento con objetos simulados (*Mock Objects*)

Prueba de Software

- Definición

  Proceso para verificar que el software cumpla criterios específicos de calidad, corrección, compleción, exactitud, eficacia y eficiencia, para asegurar su buen funcionamiento y la satisfacción del cliente final.

  Software: conjunto de elementos de un sistema informático: programas, datos, documentación, procedimientos y reglas.

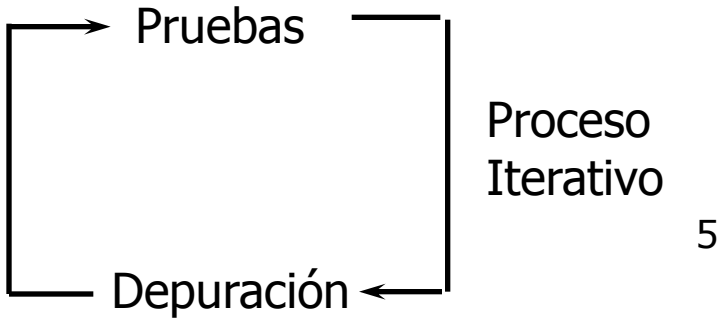
- Objetivo:
  - Se asume que todo software contiene defectos.
  - Los defectos generalmente se manifestarán en fallos.
  - Los fallos serán detectados o bien por las pruebas o bien por los usuarios.

Verificación vs. Validación

- Validación: ¿Estamos construyendo el sistema correcto?
  - Proceso de evaluación de un sistema o componente durante o al final del proceso de desarrollo para comprobar si se satisfacen los requisitos especificados (IEEE Std610.12-1990)
- Verificación: ¿Estamos construyendo correctamente el sistema?
  - Proceso de evaluar un sistema o componente para determinar si los productos obtenidos en una determinada fase de desarrollo satisfacen las condiciones impuestas al comienzo de dicha fase (IEEE Std610.12-1990)
  - Evaluación de la calidad de la implementación

Pruebas vs. Depuración

- *Pruebas* es el proceso de demostrar la presencia de ***fallos*** en el software
  - El defecto o error es una imperfección en el software, que se manifiesta en un fallo de ejecución.
- *Depuración* es el proceso de localizar errores y el subsecuente arreglo de los mismos.



Tipos de Pruebas de Software según el SWEBOK

Base de clasificación	Técnicas específicas
Técnicas basadas en la intuición y experiencia	Pruebas *ad hoc*
	Pruebas por exploración

Cada caso de prueba se aplica tanto al programa original como a los **mutantes** generados: si una prueba consigue identificar la diferencia entre el programa y el mutante, se dice que se ha "acabado" con este último. La base teórica de las pruebas de mutación dice que buscando errores sintácticos simples se deberían encontrar otros más complejos pero existentes

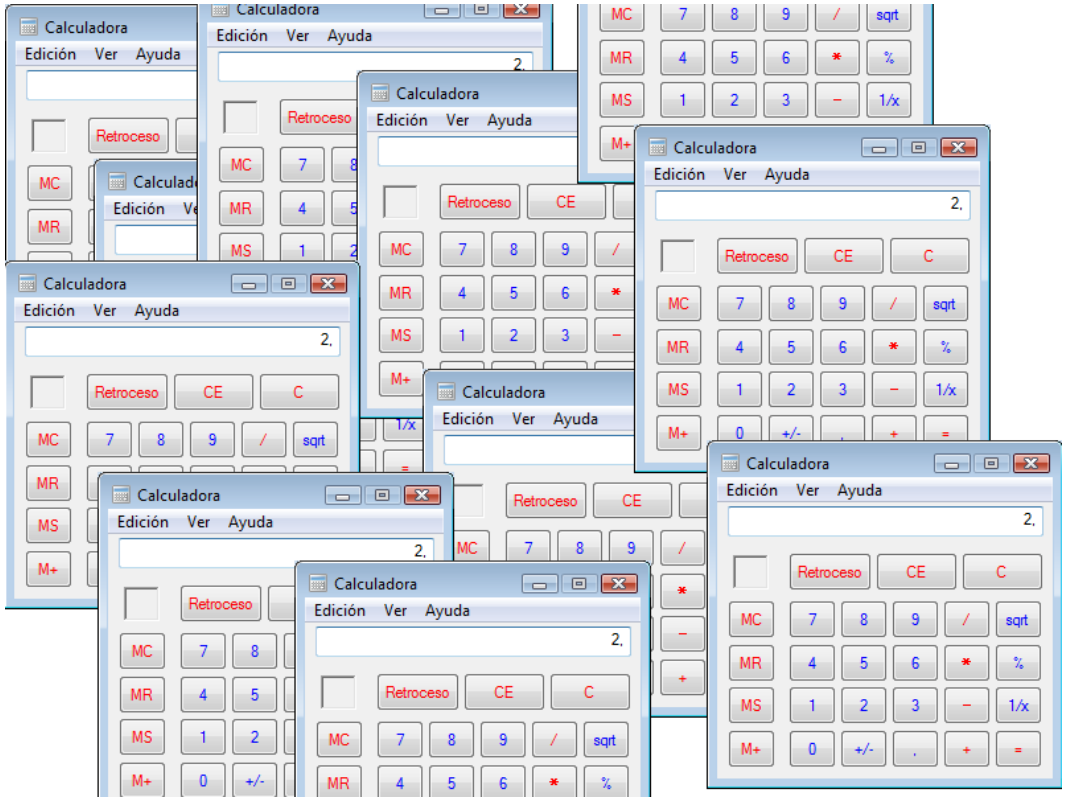
	ciones formales
	Pruebas aleatorias

Se toma un subconjunto significativo de todos los posibles usos del software, y se utiliza para crear casos de prueba. Los resultados obtenidos para dichos casos se consideran una muestra **estadística** de la que es posible inferir conclusiones acerca de la población completa, esto es, sobre el software en su conjunto

	Pruebas de mutación
Técnicas estadísticas	Pruebas de sala limpia
Técnicas basadas en el uso	Pruebas de perfil operativo
	Pruebas de fiabilidad del software

Prueba exhaustiva

- No puede hacerse una prueba exhaustiva, excepto en módulos triviales.



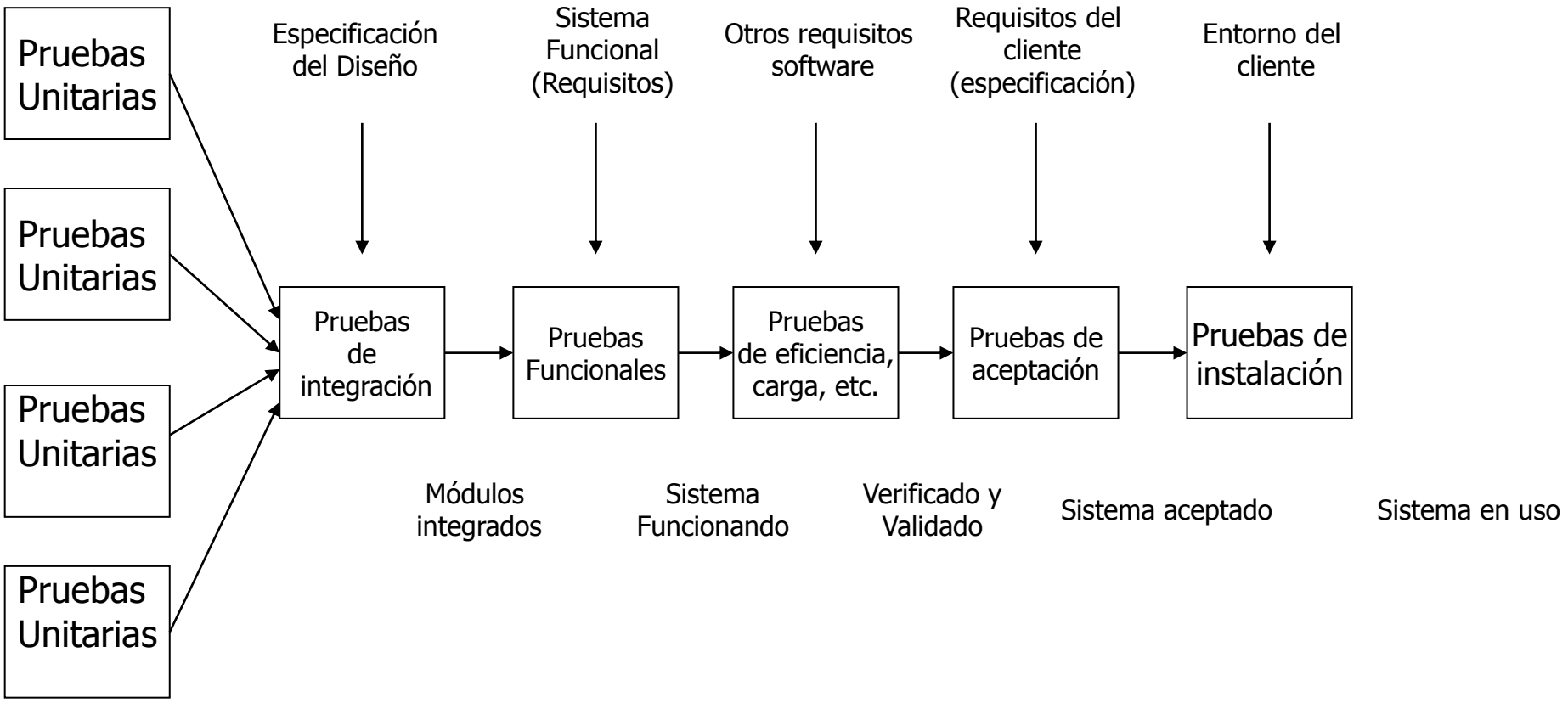
Técnicas de Prueba basadas en el Código

- **Pruebas de caja blanca**
  - basadas en el flujo de control
  - son pruebas unitarias que se usan cuando se conoce la estructura interna y el funcionamiento del código a probar y pueden ser:
    - Dinámicas
    - Estáticas
  - En Java, cuando "miramos el código"
- **Pruebas de caja negra**
  - basadas en el flujo de datos
  - se usan cuando se quiere probar lo que hace el software pero no cómo lo hace y pueden ser:
    - Dinámicas
    - Estáticas
  - En Java cuando "miramos el javadoc"

Proceso de fase de pruebas

- Para asegurar el correcto funcionamiento de un sistema de SW completo y garantizar la satisfacción del cliente se debe implementar una estrategia de pruebas que incluya:
  - Pruebas unitarias
  - Pruebas de integración
  - Pruebas de validación
  - Pruebas del sistema
  - Pruebas de aceptación

Proceso de fase de pruebas



Clases de equivalencia

- Conjuntos de valores equivalentes para las pruebas
  - si un parámetro de entrada debe estar comprendido en un cierto rango hay 3 clases de equivalencia: por debajo, en y por encima del rango.
  - si una entrada requiere un valor de entre los de un conjunto, aparecen 2 clases de equivalencia: en el conjunto o fuera de él.
  - si una entrada es booleana, hay 2 clases: si o no.
  - los mismos criterios se aplican a las salidas esperadas: hay que intentar generar resultados en todas y cada una de las clases.

Pruebas unitarias

- Es un proceso de prueba donde se estudia de manera aislada un componente de SW, como por ejemplo un clase de Java
- Se centran en las pruebas de los programas / módulos generados por los desarrolladores
  - Generalmente, son los propios desarrolladores los que prueban su código.
  - Diferentes técnicas: caja blanca y caja negra
  - Con caja blanca: medición de la cobertura, ramas, etc.

Caja blanca / Caja negra

```
int compararEnteros(int i, int j){
  /* Retorna 0 si i es igual a j, -1 si i es menor
   que j, y +1 si j es mayor que i */
  if (i==0) return -1;
  if (i==j) return 0;
  if (i<j) return -1;
  else return 1;
}
```

Incorrecto: funciona correctamente para muchas parejas de posibles valores, pero cuando i=0 el resultado es siempre -1 independientemente del valor de j

Pruebas unitarias: beneficios y limitaciones

- Beneficios
  - Permite arreglar los errores detectados sin cambiar la funcionalidad del componente probado
  - Contribuye al proceso de integración de componentes del Sistema de SW al permitir asegurar el funcionamiento correcto de los componentes de manera individual
- Limitaciones
  - No permite identificar errores de integración ni errores de desempeño del Sistema de SW
  - Son impracticas para probar todos los casos posibles para los componentes que se prueban
  - Su compatibilidad con los diferentes modelos de desarrollo de SW varía, siendo más compatible con el *Desarrollo dirigido por pruebas* (p.e., en metodologías ágiles como XP)

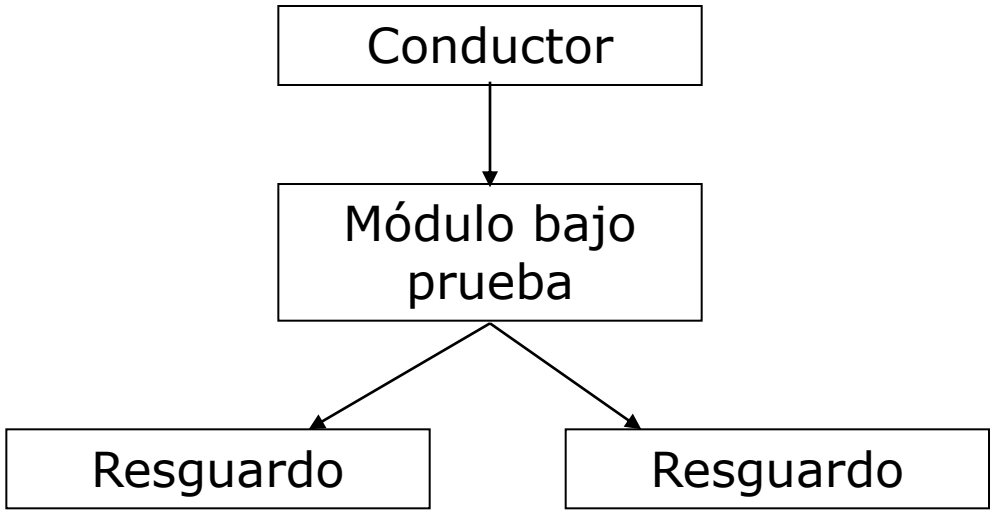
Pruebas de integración

- Unión de varios componentes unitarios y módulos y subsistemas
- Se comprueba que módulos ya probados en las pruebas unitarias de forma asilada, pueden interactuar correctamente.

Aproximaciones de integración

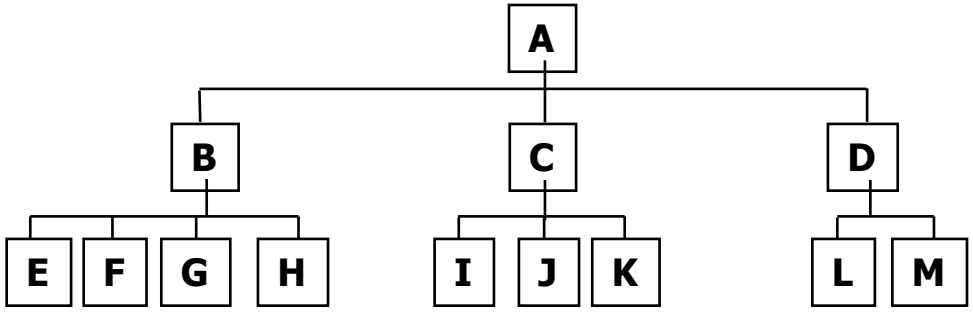
- Aproximaciones:
  - Big-bang!
  - Descendente (Top-down)
  - Ascendente (Bottom-up)
  - Sandwich: Combinación de las 2 anteriores.
- Pueden ser necesario escribir objetos simulados:
  - **Conductor**: simulan la llamada al módulo pasándole los parámetros necesarios para realizar la prueba
  - **Resguardos**: módulos simulados llamados por el módulo bajo prueba

Conductores y Resguardos



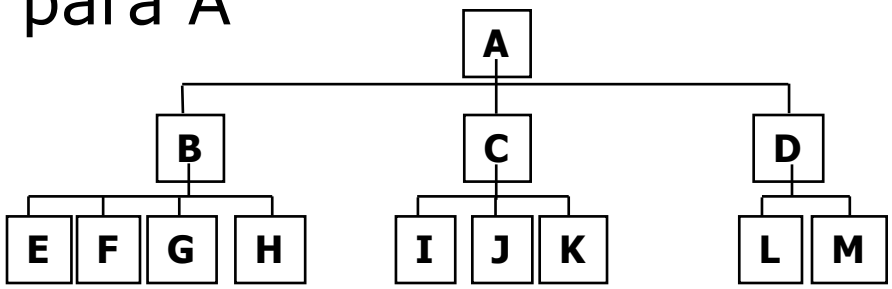
Integración descendente

- **Descendente (*Top-down*)**
  - Probar el módulo *A* primero
  - Escribir *módulos simulados, resguardos* (*stubs*) para B, C, D
  - Una vez eliminados los problemas con A, *probar el módulo* **B**
  - Escribir resguardos para E, F, G, H etc.



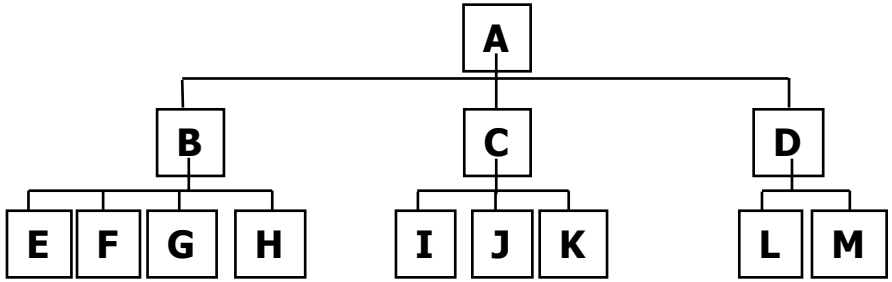
Integración ascendente

- **Ascendente (*Bottom-up*)**
  - Escribir *conductores* para proporcionar a E los datos que necesita de B
  - Probar E independientemente
  - Probar F independientemente, etc.
  - Una vez E, F, G, H han sido probados, el subsistema B puede ser probado, escribiendo un *conductor* para A



Integración Big-Bang! y Sandwich

- **Big-Bang**
  - Escribir y probar A, B, C, D, E, F, G, H, I, J, K, M *a la vez.*



- **Sandwich**
  - Combinación de ascendente y descende

Pruebas de sistema

- Pruebas para analizar que todo el sistema está de acuerdo con los requisitos y las especificaciones.
  - No está solamente relacionado con el software
  - Varios tipos de pruebas que incluyen:
    - Pruebas funcionales
    - Pruebas de eficiencia
    - Pruebas de aceptación
    - Pruebas de instalación
    - Otras:
      - Pruebas de recuperación (Recovery testing)
      - *Pruebas de seguridad (security)*
      - Pruebas de sistema es seguro (Safety testing)
      - *Pruebas de estrés (Stress testing)*
      - Pruebas de interfaces (Interface testing)

Pruebas unitarias con xUnit

- Para hacer pruebas unitarias se usan *frameworks* en entornos de pruebas automatizados
  - Un *framework* es un conjunto de clases relacionadas que proveen un conjunto de funcionalidades para realizar una tarea específica
- En el caso de la realización de pruebas unitarias xUnit es el *framework* más usado para hacer pruebas unitarias automatizadas

Pasos generales para realizar pruebas unitarias en xUnit

1. Cada prueba a realizar se programa como un método OO
   - Determinar la clase del sistema de SW que se va a probar, esté o no programada
   - Determinar los métodos de la clase que se van a probar
   - Definir los métodos de prueba usando aserciones (assert) para verificar que los métodos a probar producen las salidas esperadas al introducir ciertos valores de entrada
2. Se reúnen todos los métodos en una colección (*suite)* de pruebas
3. Se ejecuta la colección de pruebas
4. El framework xUnit invoca cada prueba de la colección de pruebas
   - Útil en las pruebas de regresión
5. El framework xUnit reporta los resultados

Los diferentes xUnit

- Para poder usar el Framework xUNit se necesita su implementación en el lenguaje de programación usado en las clases a probar. Las implementaciones de xUnit son:
  - JUnit: xUnit para Java
  - VbUnit: xUnit para Objetos Visual Basic y Objetos COM (Component Object Model)
  - Cunit: xUnit para lenguaje C
  - CPPUnit: xUnit para lenguaje C++
  - csUnit, MbUnit, NUnit: xUnit para lenguajes Microssoft .NET como C#, Vb.NET, J# y C++
  - DBUnit: xUnit para proyectos con Bases de Datos
  - Dunit: xUnit para Borland Delphi 4 y posteriores
  - PHPUnit: xUnit para PHP
  - PyUnit: xUnit para Python

Pruebas unitarias con JUnit

- ¿Qué es **JUnit**?
  - JUnit es la implementación del Framework xUnit para el lenguaje Java
  - Creado en 1997 por Erich Gamma y Kent Beck
  - Se usa para realizar pruebas unitarias de clases escritas en Java en un entorno de pruebas
  - Framework muy sencillo, con muy pocas clases
  - Puede aprenderse a usar muy rápidamente por lo que se ha hecho muy popular
  - Es *open source* y su código está disponible en:

    `http://www.junit.org/`

Pruebas unitarias con JUnit
Instalación

- Para instalar JUnit se deben seguir los siguientes pasos:

1. Descargar la versión con la que se va a trabajar (en este caso es la 3.8) del sitio Web de SourceForge

   `http://sourceforge.net/projects/junit/`

2. Descomprimir el archivo `.zip`.
   - La ruta del directorio donde se va a instalar JUnit no debe tener nombres con espacios en blanco ya que esto puede ocasionar problemas para al momento de ejecutarlo

Pruebas unitarias con JUnit
Instalación

3. Al descomprimir el archivo `.zip` en el directorio indicado se crea automáticamente una carpeta que contiene el archivo `junit.jar`.

   Este archivo como el directorio donde se encuentra deben añadirse a la variable de entorno CLASSPATH.

4.- Comprobar la instalación del JUnit haciendo lo siguiente:

   - Abrir una ventana de comandos
   - Colocarse en la carpeta donde se instaló el JUnit
   - Ejecutar las pruebas de ejemplo que trae escribiendo el siguiente comando:

   ```
   java junit.textui.TestRunner junit.samples.AllTests
   ```

   - Si JUnit está bien instalado se debe tener como resultado el mensaje `OK <119 tests>`

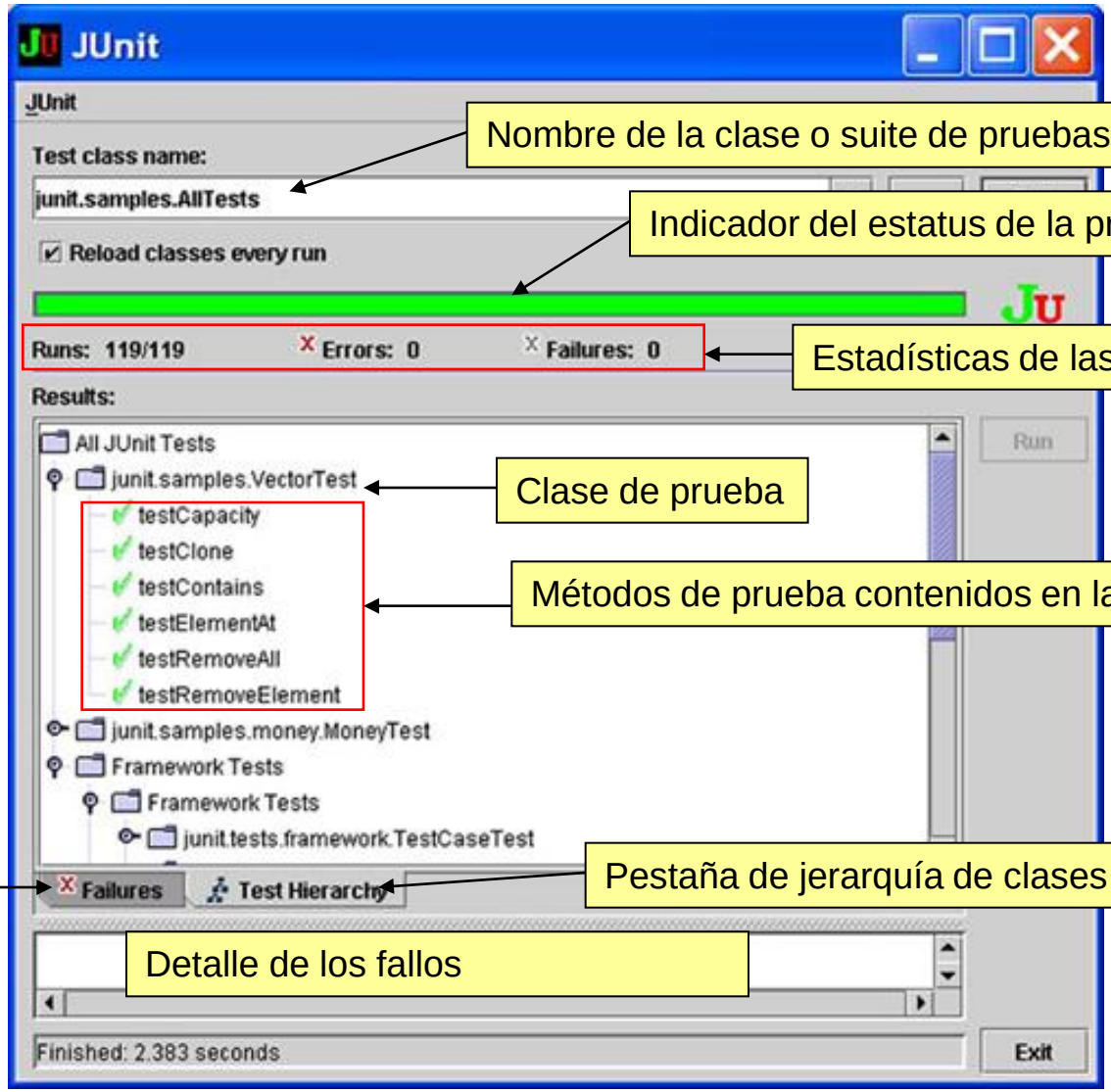
Pruebas unitarias con JUnit. Ejecución

- JUnit puede ejecutarse por si sólo o embebido en un entorno de desarrollo Java, como por ejemplo Netbeans o eclipse.
- Las formas de ejecutarlo de manera autónoma son:
  - En modo texto, como se mostró en la lámina anterior
  - En modo gráfico, abriendo la ventana de comandos, colocándose en el directorio de JUnit y escribiendo el siguiente comando:

    ```
    java junit.swingui.TestRunner junit.samples.AllTests
    ```

    Entonces aparecerá la interfaz gráfica de JUnit que permitirá ejecutar las pruebas y ver la información sobre estas.

Pruebas unitarias con JUnit
Ejecución en modo gráfico



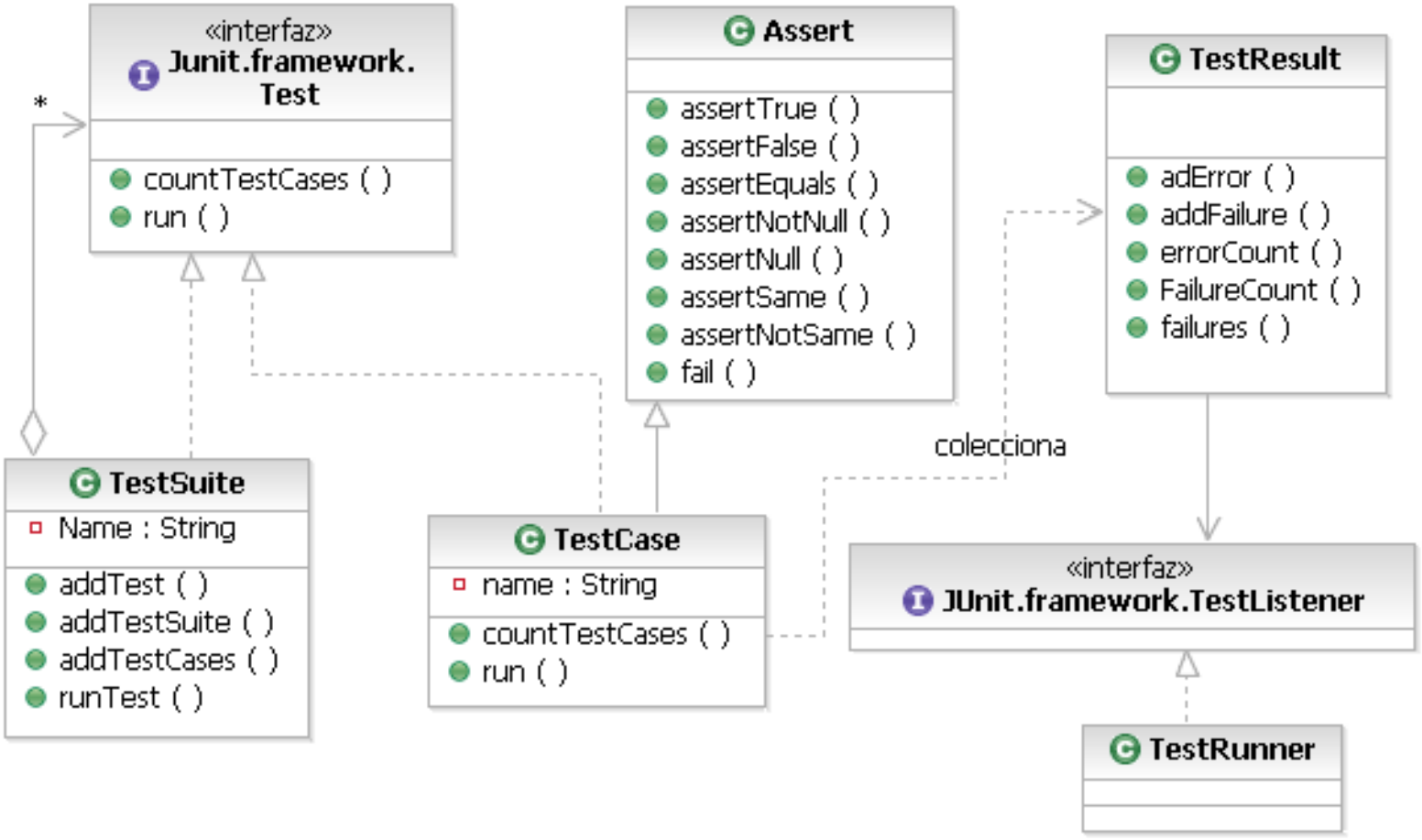
Resumen

- Concepto de prueba de Software
- Tipos de técnicas de pruebas de Software según el SWEBOK
- Técnicas de pruebas basadas en el código
  - Pruebas de caja blanca
  - Pruebas de caja negra
- Estrategias de prueba
- Pruebas unitarias
- Pruebas unitarias con el framework xUnit
- Implementaciones del framewrok xUnit
- Pruebas unitarias con JUnit

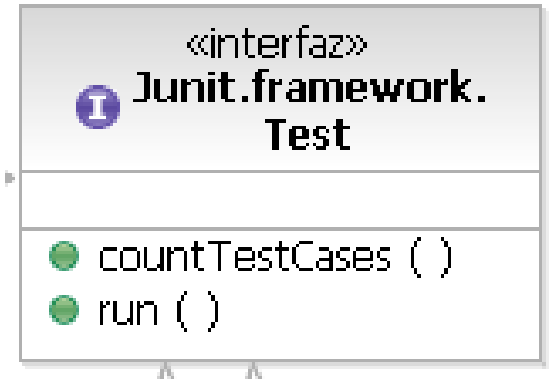
Pruebas del Software
Descripción del *framework* JUnit

Information Engineering Research Group

Clases principales del Paquete `junit.framework`

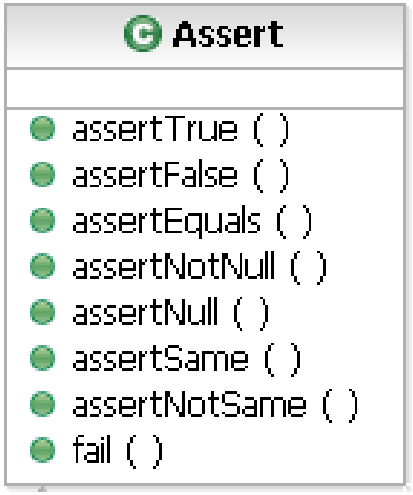


Interface `Test`



- Todos los tipos de clases **`Test`** del *framework* JUnit deben imlementar la interface **`Test`**
- Las clases **`TestCase`** y **`TestSuite`** implementan la inteface **`Test`**

Clase `Assert`



- Es la superclase de la clase **`TestCase`**
- Provee los métodos **`assertXXX()`** que se utilizarán a la hora de hacer las pruebas
- Los métodos **`assertXXX()`** son métodos que permiten comprobar si la salida del método que se está probando concuerda con los valores esperados

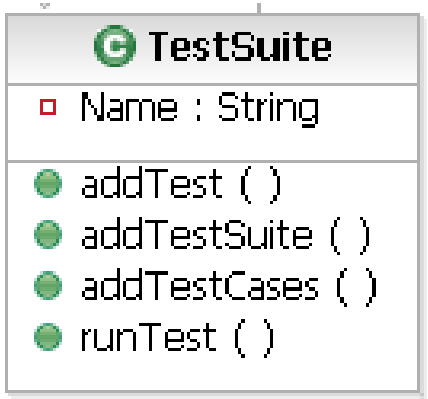
Clase `TestResult`

- Colecciona los resultados de una prueba (*Test*)
  - Indica el comienzo y el fin de una prueba
  - Proporciona información sobre los fallos y errores detectados durante la ejecución de una prueba
- Los fallos se anticipan y comprueban utilizando aserciones, mientras que los errores son problemas no anticipados como ArrayIndexOutOfBoundsException.
- `void addError(Test test, java.lang.Throwable t)`
- `void addFailure(Test test, AssertionFailedError t)`

Concepto de **Fallo** (*Failure*) en JUnit

- Un fallo se asocia con el caso en el cual una prueba no obtiene el resultado esperado
- Los fallos se detectan a través de los métodos **`assert()`** si no se cumplen las restricciones establecidas en estos métodos

Clase `TestSuite`

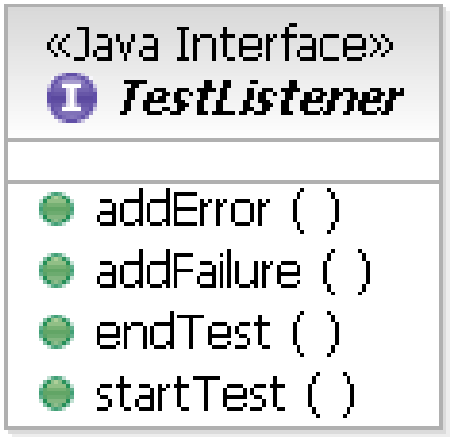


- Es una subclase de **`TestClass`**
- Puede contener instancias de las clases **`TestSuite`**, **`TestCase`**
- Permite organizar un conjunto de pruebas relacionados para manejarlos y ejecutarlos más fácilmente como una *Suite de Pruebas* (*TestSuite*)

Clase `TestCase`

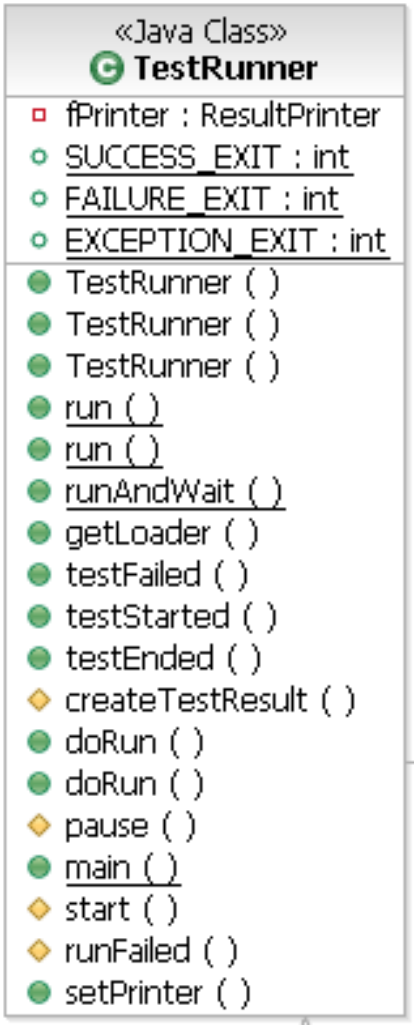
- Clase que se extiende al crear una prueba de una clase particular
- Permite definir cualquier caso de prueba en tres partes:
  - La inicialización, método **`setUp()`**
  - La aplicación de la prueba, a través del método **`runTest()`**
  - El limpiado de las estructuras de datos usadas, método **`tearDown()`**
  - El método **`run()`** llama a los tres métodos anteriores en ese orden

Interface `TestListener`



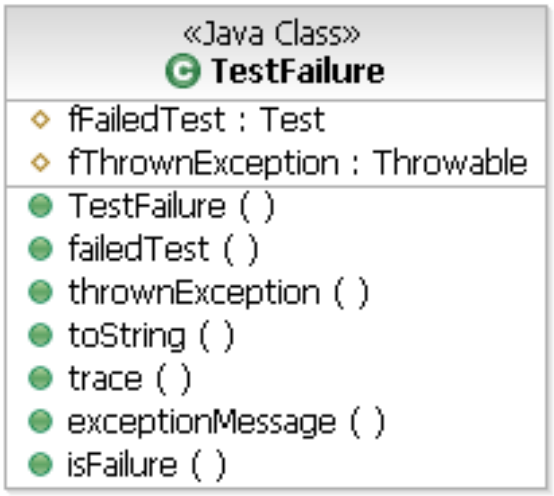
- Define los métodos que permiten seguir el progreso de la ejecución de una prueba
- La clase **`TestRunner`** implementa los métodos de la interfaz **`TestListener`**

Clase `TestRunner`



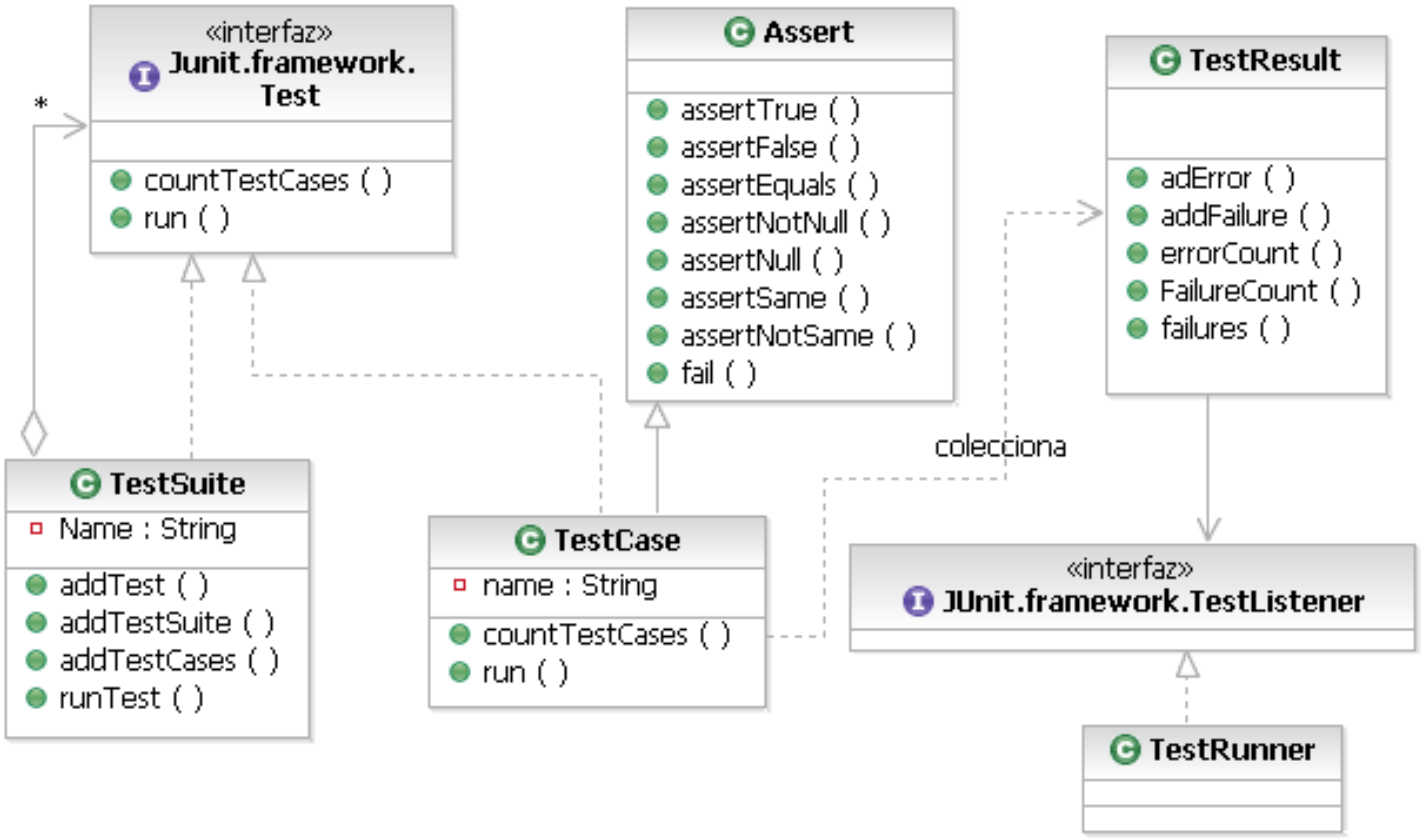
- Provee una traza y un resumen de los resultados de una prueba
- La clase **`TestRunner`** implementa los métodos de la interface **`TestListener`**

Clase `TestFailure`



- Encapsula los fallos y errores ocurridos durante la ejecución de una prueba
- Rastrea una instancia de la clase Test describiendo la prueba que falló y almacenando la excepción que causó el fallo

Resumen
Paquete `junit.framework`



Pruebas del Software
Tipos de aserciones con JUnit

Information Engineering Research Group

Aserciones en JUnit

- Aserción: “a statement that enables you to test your assumptions about your program”
- En JUnit representan condiciones para verificar si los resultados generados por el código corresponden con los resultados esperados:
  - **`assertTrue()`**
  - **`assertNull()`**
  - **`assertSame()`**
  - **`assertEquals()`**
  - **`assertFalse()`**
  - **`assertNotNull()`**
  - **`assertNotSame()`**
  - **`fail()`**

assertTrue()

- Sintaxis

```
void assertTrue(expresión booleana)
void assertTrue(String, expresión
  booleana)
```

- Evalúa una expresión booleana. La prueba pasa si el valor de la expresión es verdad, p.e.:

```
assertTrue("La temperatura debe ser
  mayor a 15 grados", actualTemp > 15);
```

assertFalse()

- Sintaxis:

```
void assertFalse(expresión booleana)
void assertFalse(String, expresión booleana)
```

- Evalúa una expresión booleana
- La prueba pasa si el valor de la expresión es falso, p.e.:

```
assertFalse("El coche no debería estar
  corriendo", coche.estaCorriendo());
```

assertNull()

- Sintaxis:

```
void assertNull(Object)
void assertNull(String, Object)
```

- Verifica si la referencia a un objeto es nula. La prueba pasa si el objeto es nulo:

```
assertNull("No exite ningún empleado con
  codigo"+id, bd.obtenerEmpleado(id));
```

assertNotNull()

- Sintaxis

```
void assertNotNull(Object)
void assertNotNull(String, Object)
```

- Verifica que la referencia a un objeto **no** es nula:
  - `assertNotNull("Se espera encontrar un empleado con código" + id, bd.obtenerEmpleado( id));`

assertSame()

- Sintaxis

```
void assertSame(Object esperado, Object actual)
void assertSame(String, Object esperado, Object
  actual)
```

- Compara dos referencias y asegura que los objetos referenciados tienen la misma dirección de memoria (i.e. que son el mismo objeto)
- La prueba pasa si los 2 argumentos son el mismo objeto o pertenecen al mismo objeto

```
assertSame("Se espera que sean la misma parte",
  part1, part2);
assertSame(contenedor.obtElem(2),
  contenedor.obtElem(4));
```

assertNotSame()

- Sintaxis

```
void assertNotSame(Objeto esperado Object actual)
void assertNotSame(String, Object esperado, Object
  actual)
```

- Compara dos referencias a objetos y asegura que ambas apuntan a diferentes direcciones de memoria, i.e., que son objetos distintos.
- La prueba pasa si los dos argumentos suplidos son objetos diferentes o pertenecen a objetos distintos

```
assertNotSame("Se espera que sean partes distintas",
  part1, part2);
assertNotSame("Los elementos en la posición 1 y 3 no
  pertenecen al mismo objeto", contenedor.obtElem(1),
  contenedor.obtElem(3));
```

assertEquals()

- Sintaxis

```
void assertEquals(valor esperado, valor actual)
void assertEquals(String, valor esperado, valor actual)
```

- El método `assertEquals()` contiene un método para cada tipo predefinido en java.
- Se usa para comprobar igualdad a nivel de contenidos
  - La igualdad de tipos primitivos se compara usando "=="
  - La igualdad entre objetos se compara con el método `equals()`
- La prueba pasa si los valores de los argumentos son iguales

```
assertEquals("El valor debe ser 28.0", 28.0, valor)
assertEquals("El apellido debe ser McKoy", "McKoy",
  paciente.obtApellido());
```

static void **assertEquals**(double expected, double actual, double delta)

fail()

- Sintaxis
  - `void fail`
  - `void fail(String)`
- Causa que la prueba falle inmediatamente. Se puede usar
  - Cuando la prueba indica que hay un error
  - Cuando se espera que el método que se está probando llame a una excepción

```
public void testExpresionInvalida() {
  String expresionAEvaluar = "3 + * 8";
  ExpresionEvaluada exp =
            new EvaluadorDeExpresiones(expresionAEvaluar);
  try {
    ContenedorDeExpresiones contenedor = exp.parse();
    fail ("Expresión de formato invalido");
  } catch(ExpresionExcepction e) {}
}
```

Resumen

- Las aserciones son condiciones lógicas que se usan para verificar si los resultados generados por el código corresponden con los resultados esperados
  - **`assertTrue()` y `assertFalse()`**
    - **Prueban condiciones booleanas**
  - **`assertNull()` y `assertNotNull()`**
    - **prueban valores nulos**
  - **`assertSame()` y `assertNotSame()`**
    - **prueban referencias a objetos**
  - **`assertEquals()`**
    - **prueba igualdad de contenidos**
  - **`fail()`**
    - **hace que la prueba falle inmediatamente**

Pruebas de Software

Pruebas de métodos constructores, `set`/`get` y métodos convencionales con el framework JUnit

Information Engineering Research Group

D. Rodríguez, G. López

Introducción

- Se estudiará en detalle cómo probar usando el *framework* JUnit, los diferentes tipos de métodos que puede tener una clase java como son:
  - Métodos constructores
  - Métodos **`get()`** y **`set()`**
  - Métodos convencionales

Prueba de métodos constructores

- EL objetivo de un método constructor es crear las instancias de la clase a la que pertenece e inicializar los valores de sus atributos
- En java se invoca al usar la palabra reservada **`new(…)`**.
- Los métodos constructores a probar son los definidos explícitamente en la clase

Prueba de métodos constructores

- Características de los métodos constructores importantes para realizar sus pruebas
  - Los métodos constructores no retornan ningún valor al terminar de ejecutarse
  - Pueden manejar excepciones para indicar algún problema a la hora de inicializar los valores de las propiedades del objeto construido
  - No es fácil comprobar que un método constructor se ha ejecutado correctamente

Prueba de métodos constructores

- Pasos para probar un método constructor a través de la técnica de caja blanca
  - Establecer los valores de los datos del caso de prueba
  - Invocar al método constructor para crear una instancia de la clase a la que pertenece
  - Establecer las aserciones necesarias para comprobar que los valores dados a los atributos por el método constructor corresponden con los valores esperados

Ejemplo prueba de métodos constructores

```
public void testPerson()  {
   //Se establecen los valores esperados en la prueba
   String name = "Gertrudis";
   String lastName1 = “López”;
        String lastName2 = “López”;

   //Se invoca al método constructor para crear una instancia

   Person instance = new Person ("Gertrudis", "Lopez", "Lopez");

   //Se verifica a través del método assert que los valores
   //asignados por el método constructor se corresponden con los
   // valores esperados

        assertEquals(instance.getName(), name);
        assertEquals(instance.getLastName1(), lastName1);
        assertEquals(instance.getLastName2(),lastName2);
}
```

Prueba de métodos `get()` y `set()`

- EL objetivo de un método set es asignar valores a los atributos de un objeto
- EL objetivo de un método **`get()`** es obtener los valores de los atributos de un objeto
- Permiten ocultar la estructura interna del objeto e implementar la encapsulación de atributos

Prueba de métodos `get()` y `set()`

- Pasos para probar un par de métodos **`get()`** y **`set()`**
  - Se invoca el constructor para crear una instancia de la clase
  - Se almacena un valor en la propiedad usando el método **`set(…)`**
  - Se obtiene el valor de la propiedad asignado usando el método **`get`** correspondiente
  - Se comprueba que el valor almacenado del atributo con el método **`set`** y el obtenido con el método **`get`** son iguales a través de **`assertEquals(…)`**

Ejemplo de una prueba de métodos `get/set` con la técnica de caja negra

```
public void testGetSetPerson() {
        String name = “Julieta";
        String lastName1 = “Alcántara”;
        String lastName2 = “Guitérrez”;
    //Se invoca al método constructor para crear una instancia

    Person instance = new Person ("Gertrudis", "Lopez", "Lopez");

    //Se modifican los valores de los atributos del objeto
    //usando los métodos set correspondientes
        instance = instance.setName(name);
        instance = instance.setLastName1(lastName1);
        instance = instance.setLasName2(lastName2);
    //Se verifica que los atributos tienen los valores esperados
    // usando los métodos get.
    //Esto se hace a través del método assert
        assertEquals(instance.getName(), name);
        assertEquals(instance.getLastName1(), lastName1);
        assertEquals(instance.getLastName2(), lastName2);
}
```

Prueba de métodos convencionales

- Para probar el correcto funcionamiento de los métodos que ejecutan las funcionalidades de las clases de un sistema se debe tomar en cuenta los factores que condicionan su ejecución y las consecuencias de su ejecución

Prueba de métodos convencionales

- Factores que condicionan la ejecución de un método
  - Parámetros de entrada
  - Estado interno del objeto
  - Estado de los objetos externos
  - Entidades externas que forman el entorno de ejecución
- Efectos obtenidos después de la ejecución de un método
  - Valor de retorno del método
  - Efectos producidos sobre los parámetros
  - Excepciones lanzadas por el método
  - Cambio del estado interno del objeto
  - Efectos externos al objeto

Resumen

- Para probar los métodos constructores se usa una prueba de caja blanca donde
  - se establece los valores de los datos del caso de prueba
  - se invoca al método constructor para crear una instancia de la clase a la que pertenece
  - se definen las aserciones necesarias para comprobar que los valores dados a los atributos por el método constructor corresponden con los valores esperados
- Para probar los métodos **`set`** y **`get`** se usa una prueba de caja negra donde
  - se invoca el método constructor para crear una instancia de la clase
  - se almacena un valor en la propiedad usando el método set
  - se obtiene el valor de la propiedad asignado usando el método **`get`** correspondiente
  - se comprueba, usando aserciones, que el valor almacenado del atributo con el método set y el obtenido con el método get son iguales.
- Para probar los métodos convencionales se deben tomar en cuenta los factores que condicionan su ejecución y las consecuencias de su ejecución

Pruebas del Software
Definición de casos de prueba sencillos con **JUnit**

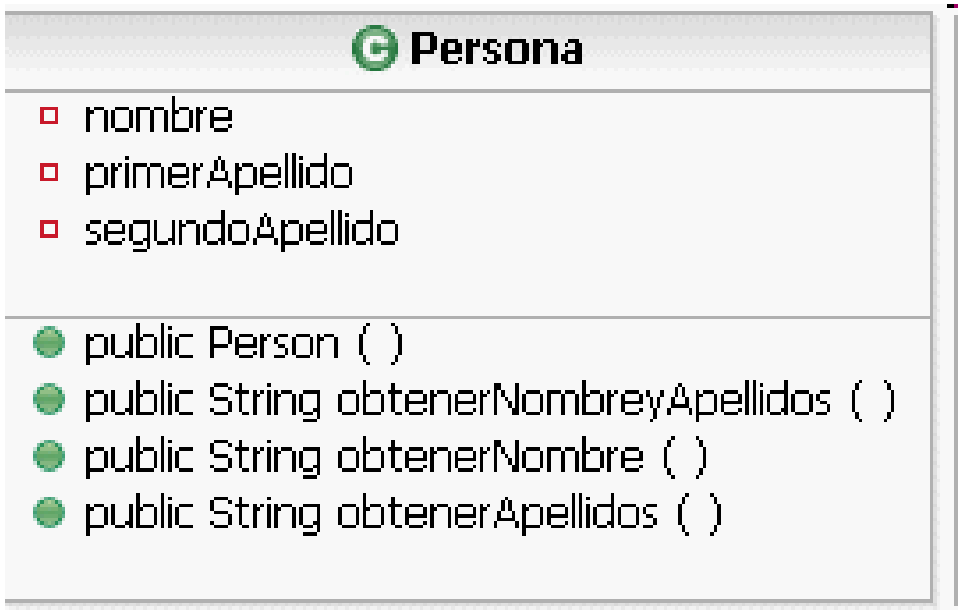
Information Engineering Research Group

Concepto de un Caso de Prueba (TestCase) en JUnit

- Un caso de prueba (TestCase) es un conjunto de métodos que prueban las funcionalidades de una clase Java.
- La clase **`TestCase`** del framework JUnit contiene los métodos básicos para escribir un caso de prueba particular.

Creación de un caso de prueba sencillo usando JUnit

- Para ver como crear un caso de prueba sencillo adaptando JUnit vamos a probar los métodos de la clase **`Persona`**.



Pasos para crear un caso de prueba en JUnit

- Para crear un caso de prueba particular hay que crear una subclase de la clase `TestCase` de JUnit.
- El nombre de la clase de prueba particular a crear debe contener con la palabra `Test` (versión 3.8)
  - Para el ejemplo de la clase persona tenemos:

```
import junit.framework.TestCase;

    public class PersonaTest extends TestCase {

    }
```

Pasos para crear los métodos de prueba

- Para escribir métodos de prueba se debe:
  1. Definir las precondiciones de la prueba y la funcionalidad a probar
  2. Especificar el resultado esperado de la prueba o postcondición para chequear si se cumple. Esto se hace usando los métodos `asserts…()` del framework JUnit.
  3. Implementar el código para que los métodos de prueba verifiquen las aserciones definidas.
  4. Ejecutar el método de prueba

Pasos para crear los métodos de prueba

- La signatura de los métodos de prueba en la versión 3.8 de Junit es:

```
public void testSuCasoDePrueba ( ) { }
```

- El nombre del método de prueba debe comenzar con la palabra `test` y a continuación el nombre del método de prueba particular.
  - La primera letra de cada palabra del nombre del método debe ser mayúscula

Prueba del método `ObtenerNombre` de la clase `Persona`

```
public void testObtenerNombre() {
    String expResult = "Gertrudis";
     Persona instancia =
                new Persona ("Gertrudis", "Lopez", "Lopez");
    String result = instancia.obtenerNombre();
     //Prueba si el método obtenerNombre de
     //la clase Persona funciona correctamente
    assertEquals(expResult, result);
}
```

Precondición

Código que prueba la funcionalidad

Postcondición

Prueba del método `ObtenerApellidos` de la clase `Persona`

```
public void testObtenerApellidos () {

    String expResult = "Lopez Lopez";

    Persona instancia =
              new Persona ("Gertrudis", "Lopez", "Lopez");

    String result = instancia.obtenerApellidos();

    //Prueba si el método obtenerNombre de
    //la clase Persona funciona correctamente

    assertEquals(expResult, result);

}
```

Prueba del método `ObtenerNombreyApellidos` de la clase `Persona`

```
public void testObtenerNombreyApellidos () {

   String expResult = "Gertrudis Lopez Lopez";

    Persona instancia =
              new Persona ("Gertrudis", "Lopez", "Lopez");

   String result = instancia.obtenerNombreyApellidos();

    //Prueba si el método obtenerNombre de
    //la clase Persona funciona correctamente

   assertEquals(expResult, result);

}
```

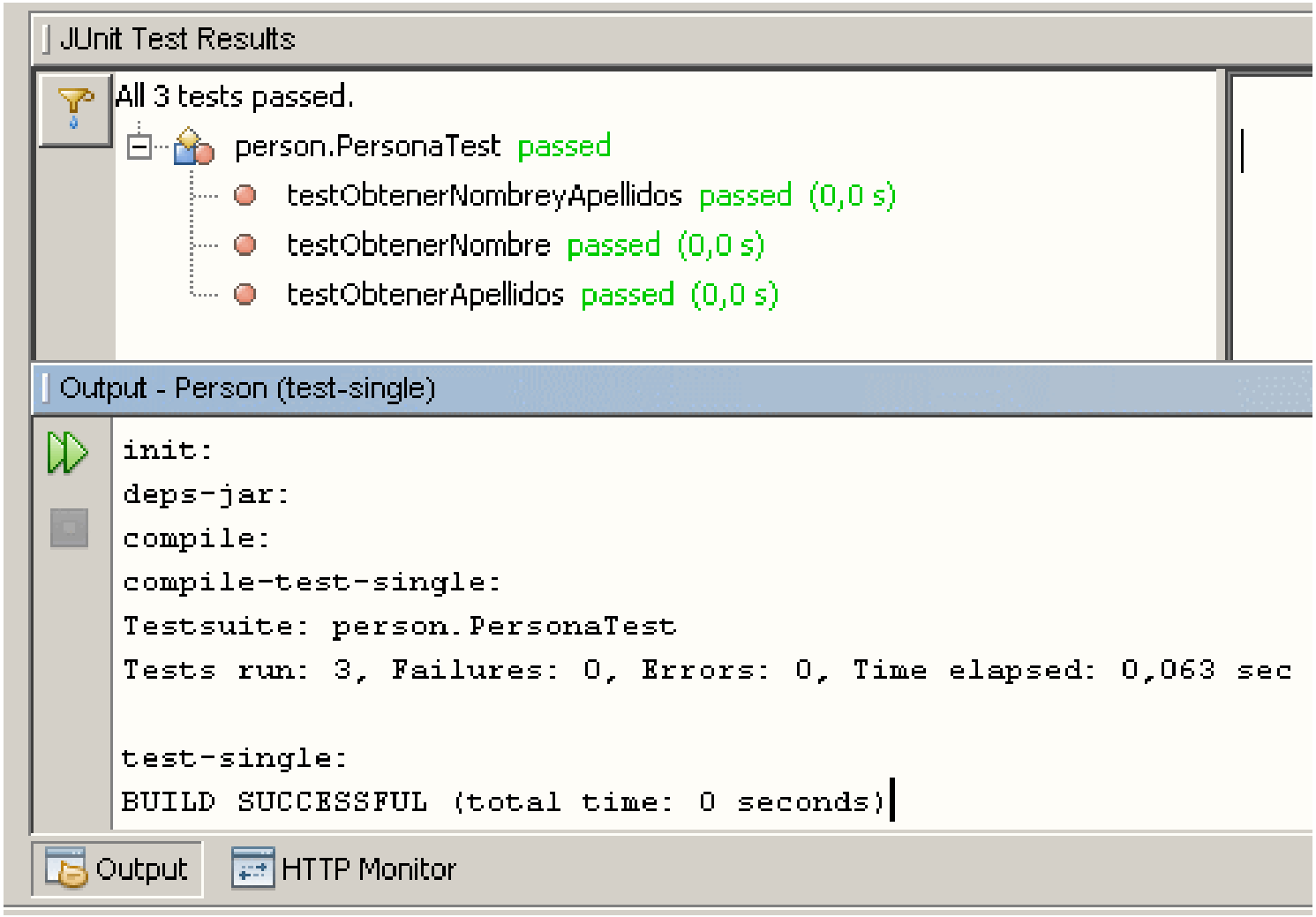
Código del caso de prueba de la clase `Persona`

```
import junit.framework.TestCase;

public class PersonaTest extends TestCase {

    public void testObtenerNombreyApellidos() {
        String expResult = "Gertrudis Lopez Lopez";
        Persona instancia = new Persona ("Gertrudis", "Lopez", "Lopez");
        String result = instancia.obtenerNombreyApellidos();
        assertEquals(expResult, result);
    }
    public void testObtenerNombre() {
        Persona instancia = new Persona ("Gertrudis", "Lopez", "Lopez");
        String expResult = "Gertrudis";
        String result = instancia.obtenerNombre();
        assertEquals(expResult, result);
    }
    public void testObtenerApellidos() {
        Persona instancia = new Persona ("Gertrudis", "Lopez", "Lopez");
        String expResult = "Lopez Lopez";
        String result = instancia.obtenerApellidos();
        assertEquals(expResult, result);
    }
}
```

Ejecución de los casos de prueba

- Para ejecutar un caso de prueba se usa el método `run()` de la clase `TestCase()`.
- En *Netbeans* un caso de prueba particular se ejecuta seleccionando en el menú `Run | Runfile` y en la ventana desplegable se selecciona el nombre del caso de prueba que se desea ejecutar
- Cuando se ejecuta un caso de prueba JUnit proporciona dos tipos de información
  - Información que indica si las pruebas pasaron o fallaron
  - Si la prueba falló indica la causa de la falla

Ejemplo de resultados en *Netbeans*



Resumen

- Se han visto los pasos para crear, ejecutar y ver los resultados de un caso de prueba sencillo escrito con JUnit
- Para escribir un caso de prueba y sus métodos constituyentes:
  - Se crea una subclase de la clase `TestCase` del framework JUnit
  - Para cada método de prueba
    - Se definen las precondiciones de la prueba y la funcionalidad a probar.
    - Se especifica el resultado esperado de la prueba o postcondición. Esto se hace usando los métodos `asserts` del framework Junit
    - Se implementa el código para que los métodos de prueba verifiquen las aserciones definidas.
  - Se ejecuta el caso de prueba.
  - Se ve la información del resultado de la ejecución en las ventanas correspondientes.

Ejercicio práctico

- **Escribir un caso de prueba con JUnit** para una clase java, en el IDE Netbeans.
- **Objetivo**: crear una clase Java con un conjunto de métodos y definir y ejecutar un caso de prueba sencillo (*TestCase*) para probar los métodos de dicha clase usando JUnit desde el IDE Netbeans.
- La clase que se va a construir es una calculadora sencilla con las cuatro operaciones básicas:
  - suma, resta, multiplicación y división con 2 argumentos.
- **Tiempo estimado**: 30 minutos.

Pruebas del Software

Definición de casos de prueba usando los métodos `setUp()` y `tearDown()` de JUnit

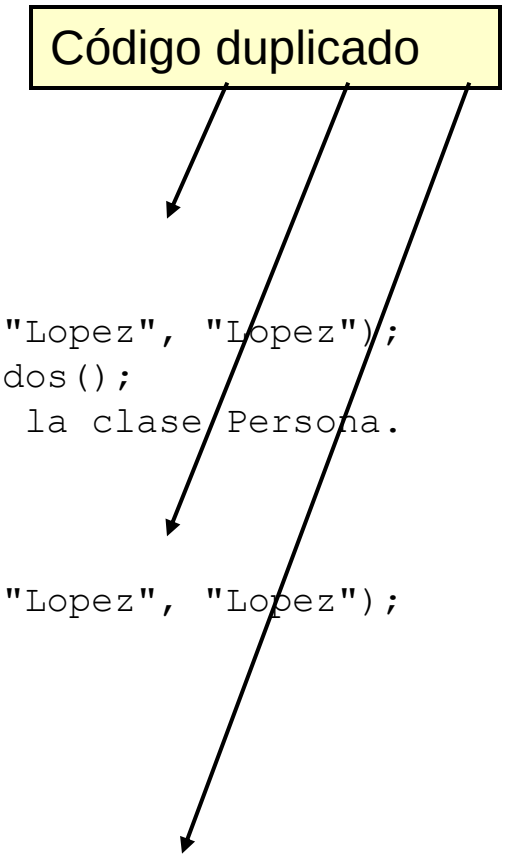
Information Engineering Research Group

Duplicación de código en los métodos de un Caso de Prueba

- En caso de prueba particular puede haber código duplicado en cada método de prueba al tener que crear objetos y estructuras de datos comunes
- Por ejemplo, en el caso de prueba de la clase persona, en cada uno de sus métodos de prueba se crea un instancia de persona.
- Esto podría hacerse una sola vez y definir esta instancia como el contexto para ejecutar cada uno de los métodos del caso de prueba **`testPersona()`**.

Código del caso de prueba de la clase **Persona**

```
import junit.framework.TestCase;

public class PersonaTest extends TestCase {

    public void testObtenerNombreyApellidos() {
        String expResult = "Gertrudis Lopez Lopez";
        Persona instancia = new Persona ("Gertrudis", "Lopez", "Lopez");
        String result = instancia.obtenerNombreyApellidos();
        //Prueba del método obtenerNombreyApellidos de la clase Persona.
        assertEquals(expResult, result);
    }
    public void testObtenerNombre() {
        Persona instancia = new Persona ("Gertrudis", "Lopez", "Lopez");
        String expResult = "Gertrudis";
        String result = instancia.obtenerNombre();
        assertEquals(expResult, result);
    }
    public void testObtenerApellidos() {
        Persona instancia = new Persona ("Gertrudis", "Lopez", "Lopez");
        String expResult = "Lopez Lopez";
        String result = instancia.obtenerApellidos();
        assertEquals(expResult, result);
    }
}
```



Uso de `setUp()` de la clase `TestCase`

- Para evitar duplicar código en los métodos de prueba de un caso de prueba se sobrescribe el método **`setUp()`**, heredado de la clase **`TestCase`** del *framework* JUnit
- Este es un método privado del caso prueba que provee el contexto de trabajo común para ejecutar los métodos del caso de prueba creando "*Fixtures*"
  - Los "*Fixtures*" son los objetos y estructuras de datos comunes a todos los métodos del Caso de Prueba

Uso de **setUp()** de la clase **TestCase**

- Para el caso de prueba de la clase Persona la sobre escritura del método **setUp()** incluiría la creación de la instancia de la clase **Persona** sobre la cual se ejecutarán todos los métodos de prueba:

```
protected void setUp() {
        //Se definen los objetos necesarios para ejecutar
        //los métodos del caso de prueba PersonaTest

        instancia = new Persona
("Gertrudis","Lopez",Lopez");

}
```

Uso del método tearDown( ) de la clase TestCase

- El método **`tearDown()`**, heredado de la clase **`TestCase`** del *framework* JUnit, permite liberar y limpiar los objetos y estructuras de datos del "*Fixture*" definido en el método **`setUp()`**
- Tanto setUp como tearDown se ejecutan una vez por método probado.
  - En Junit4 los métodos marcados como @BeforeClass y @AfterClass

Uso de `tearDown()` de la clase `TestCase`

- Cuando se ejecuta un caso de prueba particular, usando el método **`run()`** heredado de la clase **`TestCase`**, el orden de ejecución de los métodos se corresponde con las tres partes de un caso de prueba:
  - La inicialización, a través del método **`setUp()`**
  - La aplicación de las pruebas, a través del método **`runTest()`**
  - La eliminación de las estructuras de datos usadas, a través del método **`tearDown()`**

@XXX-Class

@BeforeClass and @AfterClass	@Before and @After
Only one method per class can be annotated.	Multiple methods can be annotated. Order of execution is unspecified. Overridden methods are not run.
Method names are irrelevant	Method names are irrelevant
Runs once per class	Runs before/after each test method
@BeforeClass methods of superclasses will be run before those of the current class. @AfterClass methods declared in superclasses will be run after those of the current class.	@Before in superclasses are run before those in subclasses. @After in superclasses are run after those in subclasses.
Must be public and static.	Must be public and non static.
All @AfterClass methods are guaranteed to run even if a @BeforeClass method throws an exception.	All @After methods are guaranteed to run even if a @Before or @Test method throws an exception.

Ejemplo **tearDown()**

- Para el Caso de Prueba de la clase **Persona** el método **tearDown()** libera los recursos reservados en el método **setUp()**, que es la instancia de la clase **Persona** creada en el método **setUp()**:

```
protected void tearDown()  {
        //Se liberan los objetos usados en
        //la ejecución de las pruebas
        instancia = null;
  }
```

Ejemplo completo con `setUp()` y `tearDown()`

```
import junit.framework.TestCase;
public class PersonaTest extends TestCase {
    private Persona instancia;
    @Override
    protected void setUp() {
     instancia = new Persona ("Gertrudis", "Lopez", "Lopez");
    }
    public void testObtenerNombreyApellidos() {
        String expResult = "Gertrudis Lopez Lopez";
        String result = instancia.obtenerNombreyApellidos();
        assertEquals(expResult, result);
    }
    public void testObtenerNombre() {
        String expResult = "Gertrudis";
        String result = instancia.obtenerNombre();
     assertEquals(expResult, result);
    }
    public void testObtenerApellidos() {
        String expResult = "Lopez Lopez";
        String result = instancia.obtenerApellidos();
        assertEquals(expResult, result);
    }
    @Override
    protected void tearDown() {
     instancia = null;
    }
}
```

Resumen

- Se ha visto como utilizar los métodos **`setUp()`** y **`tearDown()`** de la clase **`TestCase`** para establecer un contexto de ejecución de los métodos de un caso de prueba particular.
- El método **`setUp()`** es un método privado del caso prueba que provee el contexto de trabajo común para ejecutar los métodos del Caso de Prueba creando *Fixtures*
- Los *fixtures* son los objetos y estructuras de datos comunes a todos los métodos del caso de prueba
- El método privado **`tearDown()`**, heredado de la clase **`TestCase`** de JUnit, permite liberar y blanquar los objetos y estructuras de datos del *fixture* definido en el método **`setUp()`**, una vez ejecutados los casos de prueba

Ejercicio práctico

- **Escribir un caso de prueba con JUnit** para una clase java sobreescribiendo lo métodos **`setUp()`** y **`tearDown()`** de la clase **`TestCase`** en Netbeans.
- **Objetivo**: crear una clase Java con un conjunto de métodos y definir y ejecutar un caso de prueba usando los métodos **`setUp()`** y **`tearDown()`** para probar los métodos de dicha clase usando JUnit desde el IDE Netbeans.
- **Tiempo estimado**: 45 minutos

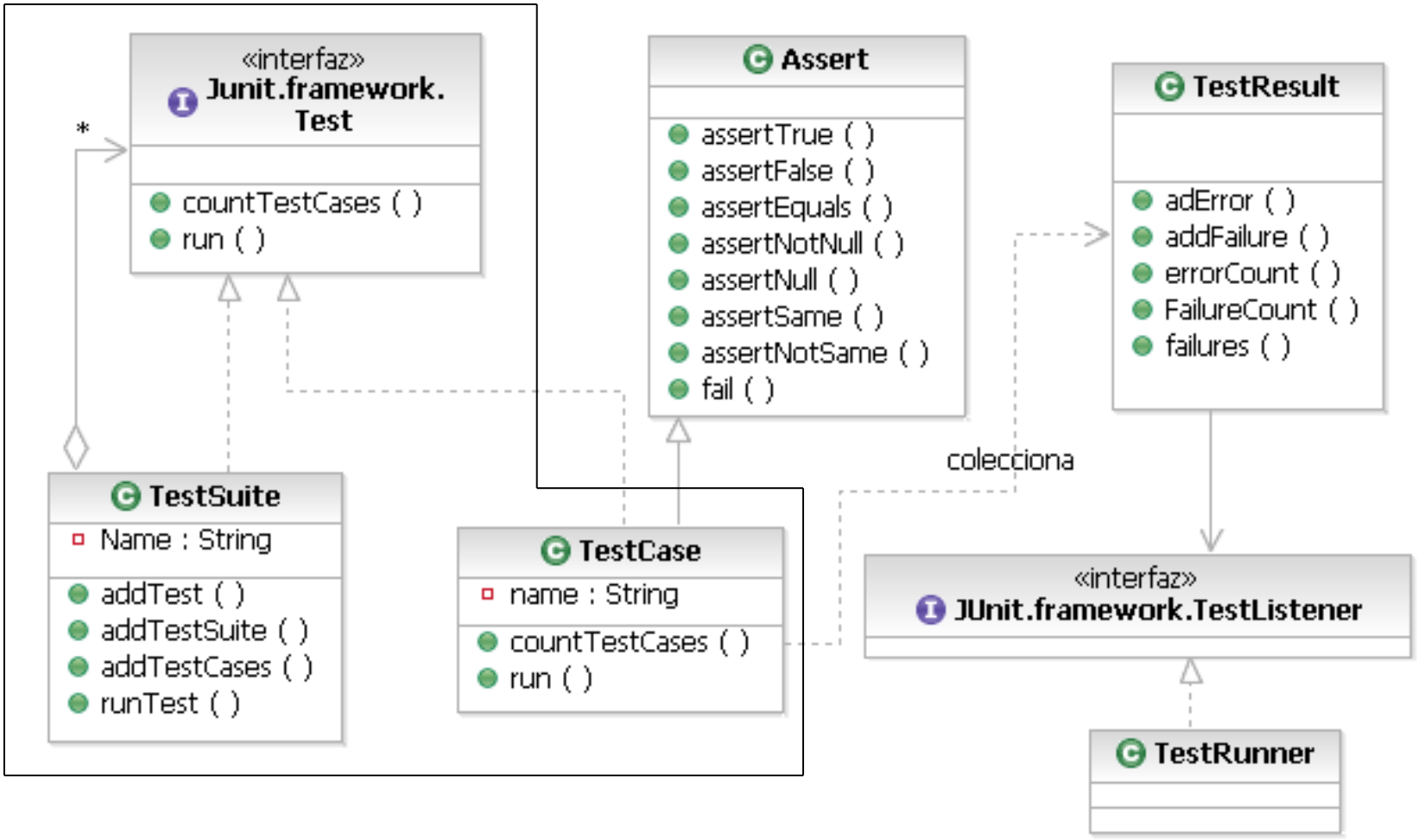
Pruebas del Software
Suites de Pruebas con JUnit

Information Engineering Research Group

Introducción

- Los casos de prueba (*test cases*) son un conjunto de métodos de prueba relacionados para probar las funcionalidades de una clase particular
- Para probar un sistema de software y/o cada uno de sus módulos es necesario elaborar varios casos de prueba
- Las *suites* de prueba (*test suites*) permiten ejecutar de manera colectiva casos, métodos y/o suites de pruebas relacionados

Estructura de la clase `TestSuite` en `JUnit`



Suites de pruebas (*Test Suites*)

- Las *suites* de prueba se usan para agrupar métodos, casos de prueba y/o *suites* de pruebas relacionados que se ejecutan secuencialmente para probar las funcionalidades de módulos y/o de sistemas de software
- Para crear un *suite* de prueba usando JUnit (Versión 3.8) se define una subclase de la clase **`TestSuite`**.
- Si el programador NO define una *TestSuite* para un **`TestCase`**, JUnit construye automáticamente una *TestSuite* que incluye todos las pruebas (tests) del caso de prueba.

Organizar casos de prueba en suites de pruebas

1. Identificar las pruebas y suites de pruebas a agrupar en base a sus características o propósito en común
2. Si ya existe una suite de pruebas, ¿se deben añadir pruebas nuevas a la suite?
3. Se crea un método estático Suite que retorna una instancia de la interface **`Test`** de JUnit
4. Dentro del método se crea un objeto **`TestSuite`**
5. Para añadir métodos de prueba la suite de manera individual, se usa el método **`addTest()`** indicando el nombre método de prueba a añadir
6. Para añadir casos de prueba completos a la suite se usa el método **`addTestCases()`** incicando el nombre del caso de prueba
7. Para añadir otras suites de pruebas ya existentes se usa el método **`addTestSuite()`** indicando el nombre de la suite de prueba a añadir

Ejemplo *Suite* de pruebas

- Para construir una suite de pruebas para un evaluador de expresiones matemáticas, que dado un **`String`** que representa la expresión matemática produce como resultado su evaluación. P.e.:

```
(8 * 3) + 2 = 26

8 * (3 + 2) = 40
```

Ejemplo *Suite* de pruebas

- Primero se crea un caso de prueba para el evaluador de expresiones.
- Su código es el siguiente:

```
import junit.framework.TestCase;
 public class TestExpressionEvaluator extends TestCase{
 }
```

Ejemplo *Suite* de pruebas

- Luego se crean múltiples métodos de pruebas, como por ejemplo:
  - un método para probar la evaluación de expresiones con dos operandos
  - uno para probar la evaluación de expresiones con tres operandos
  - otro para probar la evaluación de expresiones con tres operandos y paréntesis
  - otro para probar la evaluación de expresiones con cuatro operandos
  - Etc.
- Los códigos de estos métodos se muestran a continuación

Ejemplo *Suite* de pruebas

```
public void testTwoOperators() {
  String expressionToEvaluate = "3 + 2";
  ExpressionEvaluator exp =
      new ExpressionEvaluator(expressiontoEvaluate);

  double val = exp.evaluate();
  assertEquals(5.0,val);
}


 public void testThreeOperators() {
    String expressionToEvaluate = "3 + 2 * 5";
    ExpressionEvaluator exp =
       new ExpressionEvaluator(expressiontoEvaluate);
           double val = exp.evaluate();
                 assertEquals(13.0,val);
 }
```

Ejemplo *Suite* de pruebas

```
public void testThreeOperatorsWithBraces( ) {
   String expressionToEvaluate = "(3 + 2) * 5";
   ExpressionEvaluator exp =
        new ExpressionEvaluator(expressiontoEvaluate);
   double val = exp.evaluate();
   assertEquals(25.0,val);
}


public void testFourOperatorsWithBraces() {
   String expressionToEvaluate = "(3 + 2) * (5 + 1)";
   ExpressionEvaluator exp =
       new ExpressionEvaluator(expressiontoEvaluate);

   double val = exp.evaluate();
   assertEquals(30.0,val);
}
```

Ejemplo *Suite* de pruebas

- Ahora se quiere construir una *suite* de pruebas que agrupe a los métodos de prueba que prueben solamente las operaciones con paréntesis y otra *suite* con métodos que prueban operaciones sin símbolos de prioridad

Ejemplo *Suite* de pruebas

```
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

…

public static Test suite( ) {
 TestSuite testSuiteWithBraces = new TestSuite();

 testSuiteWithBraces.addTest(new
   TestExpressionEvaluator("testThreeOperatorsWithBraces "));
 testSuiteWithBraces.addTest(new
   TestExpressionEvaluator("testFourOperatorsWithBraces"));

 return testSuiteWithBraces;
}
```

Ejemplo *Suite* de pruebas

```
public static Test suite(){
  TestSuite testSuiteSimpleExp = new TestSuite();

  testSuiteSimpleExp.addTest(new
     TestExpressionEvaluator("testTwoOperators"));
  testSuiteSimpleExp.addTest(new
     TestExpressionEvaluator("testThreeOperators"));

  return testSuiteWithBraces;
}
```

Ejecución de *Suites* y casos de prueba

- Para ejecutar todos las pruebas de una suite o de un caso de prueba se usa un objeto **`TestRunner`**
- JUnit reune todos los tests y elabora una estructura de ejecución en forma de árbol donde los nodos son las clases **`TestSuites`** y/o **`TetsCases`** y las hojas son los métodos **`testxxx()`** que los componen y se ejecutan

Resumen

- Las *suites* de pruebas se definen para agrupar métodos de prueba, casos de prueba y/o suites de pruebas relacionadas por tener características o propósitos comunes que necesitan ser ejecutados de manera conjunta
- Para crear una suite…
  - Se crea un método estático **`suite()`** que retorna una instancia de la interface **`Test`** de JUnit.
  - Dentro del método se crea un objeto **`TestSuite`**
  - Se usa el método **`addTest()`** para agregar métodos de prueba y el método **`addTestSuite()`** para añadir otras *suites* de puebas ya creadas

Prueba de Software
Conceptos avanzados en JUnit

Information Engineering Research Group

Introducción

- Pruebas de excepciones
- Pruebas de métodos no incluidos en la *interface* pública de la clase

Pruebas de excepciones - Concepto de Excepción

- Una excepción es un evento anormal que ocurre durante la ejecución de un método y que se maneja explícitamente a través del lanzamiento de una excepción
- Cuando ocurre una excepción puede hacerse lo siguiente:
  - Capturarla y procesarla a través de las sentencias **`try`**, **`catch`** y **`finally`**
  - Lanzarla a través de la palabra reservada **`throw`**
  - No hacer nada si es una excepción no comprobada

Pruebas de excepciones – ¿Cuándo?

- Las excepciones manejadas explícitamente son una parte más del código del programa así que también deberían probarse para verificar que se están programando correctamente
- La prueba de excepciones se hace cuando un método lanza una excepción y al evento que la ha originado se le puede asociar un caso de prueba

Prueba de excepciones – Procedimiento general

- Se define un método de prueba donde:
  1. Se incluye el método que debe lanzar la excepción en un bloque **`try cacth`**
  2. Se invoca el método **`fail`** (que es uno de los tipos de métodos **`assert`** que provee JUnit) después de la llamada al método que debe lanzar la excepción. El método **`fail`** hará que el método de prueba falle inmediatamente si el método que debe lanzar la excepción no la lanza.
  3. Se crea un bloque **`catch`** en el que se captura la excepción esperada.

Ejemplo del procedimiento general de Prueba de excepciones

- Por ejemplo, en la aplicación del evaluador de expresiones aritméticas se quiere probar si el evaluador lanza una excepción cuando tiene como entrada una expresión matemática inválida. Entonces, usando el procedimiento general de pruebas de excepciones el método de prueba sería el siguiente:

```
public void testInvalidExpression()
{
        String expressionToEvaluate = "9 * - 5";
        ExpressionEvaluater exp1 = new
                ExpressionEvaluator(expressionToEvaluate);

        try {
        ExpressionHolder h1 = exp1.parse();
        fail("Invalid expression");
        } catch(ExpressionException e) {

        }
}
```

1
2
3

Ejemplo del procedimiento general de Prueba de excepciones

- Otro ejemplo. En la clase **`Person`**, el método constructor debería lanzar una excepción si sus argumentos son nulos. Entonces, usando el procedimiento general de pruebas de excepciones el método de prueba sería el siguiente:

```
public void testPassNullToPerson()
{
        try {
        Person p = new Person(null, null, null);
        fail("Expected BadArgumentsException");
        } catch(BadArgumentsException expected){}
}
```

1
2
3

Prueba de métodos no públicos

- Para aplicar el principio de la encapsulación java ofrece diferentes niveles de privacidad para los elementos de una clase:
  - **`public`**:
    - implica que el elemento puede accederse a cualquier nivel
  - **`protect`**:
    - cuando el elemento es visible desde la clase a la que pertenece y desde las subclases de la clase a la que pertenece
  - **`private`**:
    - el elemento es visible únicamente desde la clase a la que pertenece
  - **`por defecto`**:
    - el elemento es visible en el paquete al que pertenece

Prueba de métodos no públicos

- Generalmente la filosofía de la realización de pruebas unitarias OO consiste en invocar los métodos de la interfaz pública de la clase a probar
- En ciertos casos puede ser necesario probar métodos privados
- Los métodos privados son invocados por otros métodos de la clase para realizar implementar alguna funcionalidad interna de manera limpia y sin redundancia

Prueba de métodos no públicos de manera indirecta

- La prueba indirecta de métodos no públicos consiste en probarlos mediante pruebas de los métodos de la interfaz pública que los invocan
- La prueba indirecta implica que no es necesario definir casos de prueba explícitos para los métodos fuera de la interfaz pública de la clase
- En principio, los métodos no públicos deberían ser simples, por lo que pueden ser probados efectivamente desde los métodos que los invocan
  - Si lo anterior no se cumple existe un problema de diseño que implica definir el método como otra clase debido a su complejidad

Prueba de métodos no públicos de manera indirecta

- Razones que apoyan la realización de pruebas indirectas de los métodos que están fuera de la interfaz pública de una clase:
  - Por lo general estos métodos son simples, por lo que pueden ser probados efectivamente desde los métodos que los invocan
  - Si lo anterior no se cumple existe un problema de diseño que implica definir el método como otra clase debido a su complejidad

Prueba de métodos no públicos modificando el atributo de privacidad

- Se convierten los métodos no públicos de la clase a probar en métodos de paquetes (se asume que el método es visible en todo el paquete si no se especifica el modificador de acceso)
- Definir el caso de prueba para la clase que contiene el método siempre que esté en el mismo paquete donde está la clase a probar

Prueba de métodos no públicos modificando el atributo de privacidad

- Desventaja de este tipo de pruebas:
  - Se sacrifica el encapsulamiento y la propiedad de ocultamiento de información de la Orientación a Objetos
  - Se sacrifica la legibilidad del código fuente, ya que cuando un método es privado esto indica que es parte de la implementación de la clase

Resumen

- En este tema se abordaron conceptos avanzados en JUnit como la prueba de excepciones esperadas y la prueba de métodos no públicos
- La prueba de excepciones se hace cuando un método lanza una excepción y al evento que la ha originado se le puede asociar un caso de prueba. Para hacerlo
  1. Se incluye el método que debe lanzar la excepción en un bloque `try-cacth`
  2. Se invoca el método fail después de la llamada al método que debe lanzar la excepción
  3. Se crea un bloque `catch` en el que se captura la excepción esperada.
- La prueba de métodos no públicos puede hacerse de varias formas
  - Indirectamente probándolos mediante pruebas de los métodos de la interfaz pública que los invocan
  - Modificando el atributo de privacidad de los métodos a friendly o de paquete
  - Utilizando el API de *reflection* de Java

Pruebas del Software
Pruebas con objetos simulados

Information Engineering Research Group

Introducción

- A veces es necesario hacer pruebas de una clase aislando sus objetos colaboradores.
  - Por ejemplo, en las pruebas de integración
- Esta necesidad se presenta cuando la instanciación de los objetos colaboradores es muy costosa, p.e., si se requiere activar una base de datos
- Una de las técnicas para poder probar clases relacionadas con otras es a través de objetos simulados (*Mock Objects*)
  - Simulan el comportamiento de los objetos colaboradores para la realización de las pruebas

Objectos simulados (*Mock Objects*)

- Son objetos que permiten realizar pruebas de manera aislada de clases que tienen asociaciones con otras clases, llamadas clases colaboradoras
- Los *Mock Objects* permiten hacer la prueba de una clase aislándola de las clases con las que interactúa
- Los *Mock Objects* son objetos que van a simular a los objetos colaboradores de la clase que esté probando, adoptando la interfaz pública de los objetos colaboradores

¿Cuándo usar *Mock Objects*?

- Cuando en una aplicación no se puede probar los objetos que no tienen relaciones de asociación antes de probar los objetos que si las tienen
- Cuando no se cuenta con la implementación de los objetos colaboradores
- Cuando usar los objetos colaboradores para las pruebas es muy costoso, como por ejemplo, usar una base de datos

Ventajas de los *Mock Objects*

- Son de fácil instanciación, ya que su objetivo es comportarse como los objetos que simulan
- Son infalibles, ya que si se detecta un fallo haciendo la prueba de la clase con la que colabora se sabe que el fallo está en dicha clase
- Existen distintas herramientas como *JMock*, *NMock* y *Easymock*.
- Permiten reducir el tiempo de la pruebas al sustituir las clases colaboradoras por un código mucho más simple

Proceso para usar *Mock Objects*

1. Crear los *Mock Objects* necesarios
2. Definir el comportamiento esperado de los *Mock Objects*, estableciendo las expectativas
3. Crear la instancia de la clase que va a ser probada de manera que use las instancias de los *Mock Objects* en vez de los objetos colaboradores originales
4. Ejecutar el método que se vaya a probar realizando la prueba correspondiente
5. Decir a cada *Mock Object* involucrado en la prueba que verifique si se cumplieron las expectativas

Proceso para usar *Mock Objects*

1. Crear los *Mock Objects* necesarios
   - Se debe crear un *Mock Object* por cada objeto colaborador que utilice la clase a probar
   - Las clases de los *Mock Objects* deben ser subclases de las clases de los objetos colaboradores para evitar errores de compilación

Proceso para usar *Mock Objects*

2. Definir el comportamiento esperado de los *Mock Objects*, estableciendo las expectativas
   - Definir los métodos que van a ser llamados desde la clase que se va a probar
   - Definir con qué parámetros van a ser llamados los métodos y sus valores de retorno (opcional)
   - Definir las secuencias en las que se van a invocar los métodos
   - Definir el número de invocaciones a cada método

Proceso para usar *Mock Objects*

3. Crear la instancia de la clase a probar de manera que use las instancias de los *Mock Objects* en vez de los objetos colaboradores originales
   - Pasar los *Mock Objects* como parámetros en el método constructor de la clase a probar en vez de los objetos colaboradores originales
   - En el caso de que los objetos colaboradores sean creados dentro de la instancia de la clase a probar, se debe *refactorizar* el código de la clase a probar para encapsular dichas instanciaciones en métodos que estén fuera del objeto a probar

Proceso para usar *Mock Objects*

4. Ejecutar el método que se vaya a probar realizando la prueba correspondiente

Proceso para usar *Mock Objects*

5. Decir a cada *Mock Object* involucrado en la prueba que verifique si se cumplieron las expectativas
   - Comprobar al finalizar la prueba que los *Mock Objects* cumplieron las expectativas. Esto se puede hacer de manera automática usando las herramientas que automatizan la técnica de los *Mock Objects*

Herramientas de Mock Objects
Características

- Características:
  - Eliminan las tareas que repetitivas a los desarrollador debe repetir cuando usa la técnica de *Mock Objects*.
  - Permiten utilizar de manera estándar la técnica de los *Mock Objects*.
  - Generan código legible y entendible para los desarrolladores

Herramientas de Mock Objects Funcionalidades

- Crear *Mocks Objects* invocando un método que tiene como parámetro la clase del objeto colaborador para el que se va a crea el *Mock Object*
- Definir las expectativas de un *Mock Object* usando unas pocas líneas de código
- Almacenar información sobre evento de comportamiento del *Mock Object*
  - métodos invocados, parámetros usados y valores de retorno
- Verificar expectativas a través de la invocación de un método que compara las expectativas con los eventos de comportamiento almacenados

EasyMock

- EasyMock fue la primera herramienta de manejo de *Mocks Objects* para Java desarrollada en el año 2001
- Es una herramienta de software libre disponible en

  `http://www.easymock.org/`

JMock

- JMock es una de las herramienta de manejo de Mocks Objects más usada en la programación en Java
- Es una herramienta Software Libre disponible en:
  - **`http://www.jmock.org/`**

Uso de los objetos simulados con *EasyMock*

1. **Crear el *mock***, a partir de una interfaz.
   - A través del método **`createMock()`**.
2. **Preparar el *mock* estableciendo las expectativas**
   - Definir todas las invocaciones que se espera que ocurran.
   - Esto se hace a través del método **`expect()`** y otros métodos.
   - (Modo grabación)
3. **Iniciar el *mock***
   - Informar que ya se establecieron todas las expectativas y que comienza a funcionar el objeto mock.
   - Esto se hace a través del método **`replay()`**.
   - (Modo reproducción)
4. **Ejecutar el método a probar** realizando la prueba correspondiente
5. **Verificar el *mock***
   - Consiste en verificar que se hayan realizado todas las invocaciones que se esperaban, en el orden y cantidad correctas.
   - Esto se hace a través del método **`verify()`**.

Ejemplo EasyMock

```
import org.easymock.EasyMock;
import junit.framework.TestCase;

public class InternetRelayChatTest extends TestCase {
  public void testJoin() {
    String msg = "Hello";
    Client fred = EasyMock.createMock(Client.class);
    Client peter = EasyMock.createMock(Client.class);
    Client mary = EasyMock.createMock(Client.class);

    expect (mary.onMessage("homer",msg)).andReturn(true);
    peter.onMessage("homer",msg);
    expectLastCall().andReturn(true);

    replay(peter, fred, mary);

    InternetRelayChat irc = new InternetRelayChat();
    irc.join("the boss", peter);
    irc.join("cool", mary);
    Prompt prompt = irc.join("homer", fred);
    prompt.say(msg);

    verify(peter, fred, mary);
  }
}
```

1

2

3

4

5

Resumen

- Se ha visto como hacer pruebas de clases de manera aislada de sus objetos colaboradores.
  - En especial se vió la técnica de objetos simulados (*Mock Objects)*
- Uso de la herramienta *EasyMock*.
  - Y los pasos para hacer pruebas usando *mock objects* con EasyMock

Pruebas de Software
NUnit

Information Engineering Research Group

Introducción

- En este tópico se estudiará NUnit, que es una implementación para microsoft .NET del *framework* de pruebas unitarias XUnit

¿Qué es NUnit?

- NUnit es una implementación para todos los lenguajes Microsoft .NET del framework de pruebas unitarias XUnit
  - NUnit es el equivalente de JUnit pero para los lenguajes del entorno .NET
- NUnit es de código abierto

  http://www.nunit.org/

Características de NUnit?

- NUnit está escrito en C# y usa muchas de las características de .NET como atributos personalizados y capacidades relacionadas con la reflexión
- Basa su funcionamiento en:
  - Aserciones: métodos del *framework* de NUnit utilizados para comprobar y comparar valores.
  - Atributos personalizados: indican a NUnit que debe hacer con determinado método o clase, es decir, le indican como interpretar y ejecutar las pruebas implementadas en el método o clase.

Métodos Asserts en NUnit

- Son métodos que se usan para probar software
- Verifican si los resultados arrojados por el código que se está probando corresponden con los resultados esperados
- Son efectivos para detectar errores
- Existen diferentes tipos de métodos `Asserts` (aserciones)

Tipos de aserciones en NUnit

- Aserciones de:
  - igualdad
  - identidad
  - comparación
  - tipos
  - String
  - colecciones
  - archivos
  - pruebas de condición
  - métodos de utilidad (utility)

Aserciones de igualdad

- Sintaxis

```
Assert.AreEqual( value1, value2 );
Assert. AreEqual(value1, value2 , string message );
Assert. AreEqual(value1, value2 , string message,
  object[] parms );
```

- La aserción **`AreEqual`** verifica si dos argumentos son iguales
- La sobrecarga de métodos permite comparar dos valores numéricos de tipos diferentes, como se muestra en el siguiente ejemplo:

```
Assert.AreEqual(5, 5.0);
```

- Cuando se comparan valores numéricos punto flotante y double se usa un argumento adicional que indica la tolerancia con la que van a ser considerados valores iguales

Aserciones de identidad

- Existen tres tipos de métodos **`Asserts`** para probar la identidad de un objeto:

  `Assert.AreSame()`

  `AssertAreNotSame()`

  `Assert.Contains()`

  - Los métodos **`Assert.AreSame()`** y **`Assert.AreNotSame()`** verifican si dos argumentos referencian o no a un mismo objeto
  - El método **`Assert.Contains()`** verifica si un objeto está contenido en un *array* o en una lista

Aserciones de identidad

- Sintaxis

```
Assert.AreSame( object expected, object actual );
Assert.AreSame( object expected, object actual, string );
 Assert.AreSame(object expected, object actual, string, params
   object[] parms );

Assert.AreNotSame(object expected, object actual );
Assert.AreNotSame(object expected, object actual , string );
 Assert.AreNotSame(object expected, object actual , string,
   params object[] parms );

Assert.Contains( objeto, IList collection );
Assert.Contains( objeto, IList collection, string );
Assert.Contains( objeto, IList collection, string, params
   object[] parms );
```

Aserciones de condición

- Este tipo de aserciones prueban una condición específica
- Su nombre denota la condición que prueban
- Tiene como primer argumento el valor a probar y opcionalmente pueden tener un **`string`** para mandar un mensaje

Aserciones de condición

- Sintaxis

```
Assert.IsTrue( bool condition );
Assert.IsTrue( bool condition, string message );
Assert.IsTrue( bool condition, string message, object[] parms );

Assert.IsFalse( bool condition);
Assert.IsFalse( bool condition, string message );
Assert.IsFalse( bool condition, string message, object[] parms );

Assert.IsNull( object anObject );
Assert.IsNull( object anObject, string message );
Assert.IsNull( object anObject, string message, object[] parms );

Assert.IsNotNull( object anObject );
Assert.IsNotNull( object anObject, string message );
Assert.IsNotNull( object anObject, string message, object[] parms);
```

Aserciones de condición

- Sintaxis

```
Assert.IsNaN( double aDouble );
Assert.IsNaN( double aDouble, string message );
Assert.IsNaN( double aDouble, string message,
  object[] parms );

Assert.IsEmpty( string aString );
Assert.IsEmpty( string aString, string message );
Assert.IsEmpty( string aString, string message,
  params object[] args );

Assert.IsNotEmpty( string aString );
Assert.IsNotEmpty( string aString, string message
  ); Assert.IsNotEmpty( string aString, string
  message, params object[] args );
```

Aserciones de condición

- Sintaxis

```
Assert.IsEmpty( ICollection collection );
Assert.IsEmpty( ICollection collection, string
  message );
Assert.IsEmpty( ICollection collection, string
  message, params object[] args );

Assert.IsNotEmpty( ICollection collection );
Assert.IsNotEmpty( ICollection collection, string
  message );
Assert.IsNotEmpty( ICollection collection, string
  message, params object[] args );
```

Aserciones de comparación

- Los métodos **`Assert`** de comparación prueban condiciones de mayor, mayor o igual, menor o menor igual sobre sus argumentos:
  - **`Assert.Greater( x, y )`**
  - **`Assert.GreaterOrEqual( x, y )`**
  - **`Assert.Less( x, y )`**
  - **`Assert.LessOrEqual( x, y )`**
  - Existe cada uno de estos métodos para cada tipo predefinido en .NET

Aserciones de comparación

- Sintaxis

```
Assert.Greater( value1, value2 );
Assert.Greater(value1, value2 , string message );
Assert.Greater(value1, value2 , string message, object[] parms
  );

Assert.GreaterOrEqual( value1, value2 );
Assert.GreaterOrEqual(value1, value2 , string message );
Assert.GreaterOrEqual(value1, value2 , string message, object[]
  parms );

Assert.Less( value1, value2 );
Assert.Less(value1, value2 , string message );
Assert.Less(value1, value2 , string message, object[] parms );

Assert.LessOrEqual( value1, value2 );
Assert.LessOrEqual(value1, value2 , string message );
Assert.LessOrEqual(value1, value2 , string message, object[]
  parms );
```

Aserciones de tipos

- Este tipo de aserciones hacen pruebas sobre el tipo de un objeto.
- Sintaxis:

```
Assert.IsInstanceOfType( Type expected, object actual );
Assert.IsInstanceOfType( Type expected, object actual,
  string message );
Assert.IsInstanceOfType( Type expected, object actual,
  string message, params object[] parms );

Assert.IsNotInstanceOfType( Type expected, object actual );
Assert.IsNotInstanceOfType( Type expected, object actual,
  string message );
Assert.IsNotInstanceOfType( Type expected, object actual,
  string message, params object[] parms );
```

Aserciones de tipos

- Sintaxis:

```
Assert.IsAssignableFrom( Type expected, object actual );
Assert.IsAssignableFrom( Type expected, object actual,
  string message );
Assert.IsAssignableFrom( Type expected, object actual,
  string message, params object[] parms );

Assert.IsNotAssignableFrom( Type expected, object actual );
Assert.IsNotAssignableFrom( Type expected, object actual,
  string message );
Assert.IsNotAssignableFrom( Type expected, object actual,
  string message, params object[] parms );
```

Aserciones `Fail()` e `Ignore()`

- Los métodos **`Fail()`** and **`Ignore()`** permiten controlar el proceso de prueba
- El método **`Assert.Fail`** permite generar un fallo de manera explícita, como en el caso de pruebas de manejo de excepciones Su sintaxis es:

```
Assert.Fail();
Assert.Fail( string message );
Assert.Fail( string message, object[]
  parms );
```

Aserciones `Fail()` e `Ignore()`

- El método **`Assert.Ignore`** permite ignorar una prueba o una suite de pruebas a momento de ejecución, haciendo posible ejecutar o no pruebas cuando sea necesario. Su sintaxis es:
  - **`Assert.Ignore();`**
  - **`Assert.Ignore( string message );`**
  - **`Assert.Ignore( string message, object[] parms );`**

Aserciones de `Strings`

- Este tipo de método permite hacer distintas pruebas sobre **`Strings`** tales como:
  - inclusión, comienza con, termina con, igualdad y coincidencia.
- La sintaxis de estos métodos es:

```
StringAssert.Contains( string expected, string actual );
StringAssert.Contains( string expected, string actual, string
  message );
StringAssert.Contains( string expected, string actual, string
  message, params object[] args );

StringAssert.StartsWith( string expected, string actual );
StringAssert.StartsWith( string expected, string actual, string
  message );
StringAssert.StartsWith( string expected, string actual, string
  message, params object[] args );

StringAssert.EndsWith( string expected, string actual );
StringAssert.EndsWith( string expected, string actual, string
  message );
StringAssert.EndsWith( string expected, string actual, string
  message, params object[] args );
```

Aserciones de `Strings`

```
StringAssert.AreEqualIgnoringCase( string expected, string
   actual );
StringAssert.AreEqualIgnoringCase( string expected, string
   actual, string message );
StringAssert.AreEqualIgnoringCase( string expected, string
   actual, string message params object[] args );

StringAssert.IsMatch( string expected, string actual );
StringAssert.IsMatch( string expected, string actual,
   string message );
StringAssert.IsMatch( string expected, string actual,
   string message, params object[] args );
```

Aserciones de colecciones

- Los asserts de este tipo permiten hacer pruebas de colecciones y sus contenidos. Esto incluye métodos para probar que:
  - Todos los elementos de una colección sean de un tipo esperado. Su sintaxis es:

```
CollectionAssert.AllItemsAreInstancesOfType(
  IEnumerable collection, Type expectedType );
CollectionAssert.AllItemsAreInstancesOfType(
  IEnumerable collection, Type expectedType,
  string message );
CollectionAssert.AllItemsAreInstancesOfType(
  IEnumerable collection, Type expectedType,
  string message, params object[] args );
```

Aserciones de colecciones

- Esto incluye métodos para probar que:
  - Todos los elementos de una colección no sean nulos. La sintaxis de este método es:

```
CollectionAssert.AllItemsAreNotNull(
  IEnumerable collection );
CollectionAssert.AllItemsAreNotNull(
  IEnumerable collection, string message );
CollectionAssert.AllItemsAreNotNull(
  IEnumerable collection, string message,
  params object[] args );
```

Aserciones de colecciones

- Esto incluye métodos para probar que:
  - Todos los elementos de una colección sean únicos. La sintaxis de este método es:

```
CollectionAssert.AllItemsAreUnique( IEnumerable
  collection );
CollectionAssert.AllItemsAreUnique( IEnumerable
  collection, string message );
CollectionAssert.AllItemsAreUnique( IEnumerable
  collection, string message, params object[] args );
```

Aserciones de colecciones

- Esto incluye métodos para probar que:
  - Dos colecciones sean iguales. La sintaxis de este método es:

```
CollectionAssert.AreEqual( IEnumerable expected,
  IEnumerable actual );
CollectionAssert.AreEqual( IEnumerable expected,
  IEnumerable actual, string message );
CollectionAssert.AreEqual( IEnumerable expected,
  IEnumerable actual string message, params object[]
  args );
```

Aserciones de colecciones

- Esto incluye métodos para probar que:
  - Dos colecciones sean equivalentes. La sintaxis de este método es:

```
CollectionAssert.AreEquivalent( IEnumerable
  expected, IEnumerable actual);
CollectionAssert.AreEquivalent( IEnumerable
  expected, IEnumerable actual, string message );
CollectionAssert.AreEquivalent( IEnumerable
  expected, IEnumerable actual string message, params
  object[] args );
```

Aserciones de colecciones

- Esto incluye métodos para probar que:
  - Dos colecciones sean diferentes. La sintaxis de este método es:

```
CollectionAssert.AreNotEqual( IEnumerable expected,
  IEnumerable actual );
CollectionAssert.AreNotEqual( IEnumerable expected,
  IEnumerable actual, string message );
CollectionAssert.AreNotEqual( IEnumerableon
  expected, IEnumerable actual string message, params
  object[] args );
```

Aserciones de colecciones

- Esto incluye métodos para probar que:
  - Dos colecciones no sean equivalentes. La sintaxis de este método es:

```
CollectionAssert.AreNotEquivalent( IEnumerable
  expected, IEnumerable actual );
CollectionAssert.AreNotEquivalent( IEnumerable
  expected, IEnumerable actual, string message );
CollectionAssert.AreNotEquivalent( IEnumerable
  expected, IEnumerable actual, string message,
  params object[] args );
```

Aserciones de colecciones

- Esto incluye métodos para probar que:
  - Una colección contiene un objeto específico. La sintaxis de este método es:

```
CollectionAssert.Contains( IEnumerable
  expected, object actual );
CollectionAssert.Contains( IEnumerable
  expected, object actual, string message );
CollectionAssert.Contains( IEnumerable
  expected, object actual string message,
  params object[] args );
```

Aserciones de colecciones

- Esto incluye métodos para probar que:
  - Una colección no contiene un objeto específico. La sintaxis de este método es:

```
CollectionAssert.DoesNotContain( IEnumerable
  expected, object actual );
CollectionAssert.DoesNotContain( IEnumerable
  expected, object actual, string message );
CollectionAssert.DoesNotContain( IEnumerable
  expected, object actual string message,
  params object[] args );
```

Aserciones de colecciones

- Esto incluye métodos para probar que:
  - Una colección es subconjunto de otra colección. La sintaxis de este método es:

```
CollectionAssert.IsSubsetOf( IEnumerable
  subset, IEnumerable superset );
CollectionAssert.IsSubsetOf( IEnumerable
  subset, IEnumerable superset, string message
  );
CollectionAssert.IsSubsetOf( IEnumerable
  subset, IEnumerable superset, string message,
  params object[] args );
```

Aserciones de colecciones

- Esto incluye métodos para probar que:
  - Una colección es vacía. La sintaxis de este método es:
    - **`CollectionAssert.IsEmpty( IEnumerable collection );`**
    - **`CollectionAssert.IsEmpty( IEnumerable collection, string message );`**
    - **`CollectionAssert.IsEmpty( IEnumerable collection, string message, params object[] args );`**

Aserciones de colecciones

- Esto incluye métodos para probar que:
  - Una colección no es vacía. La sintaxis de este método es:
    - **`CollectionAssert.IsNotEmpty( IEnumerable collection );`**
    - **`CollectionAssert.IsNotEmpty( IEnumerable collection, string message );`**
    - **`CollectionAssert.IsNotEmpty( IEnumerable collection, string message, params object[] args );`**

Aserciones de archivos

- Este tipo de métodos permite comparar dos archivos y ver si son iguales o no
- Los archivos pueden referenciarse como streams, como **`FileInfo`** o como un **`string`** que indica la ruta donde se encuantra el archivo.

Aserciones de archivos

- Sintaxis:

```
FileAssert.AreEqual( File expected, File actual );
FileAssert.AreEqual( File expected, File actual, string
  message );
FileAssert.AreEqual( File expected, File actual, string
  message, params object[] args );

FileAssert.AreNotEqual( File expected, File actual );
FileAssert.AreNotEqual( File expected, File actual,
  string message );
FileAssert.AreNotEqual( File expected, File actual,
  string message, params object[] args );
```

Atributos

- Los atributos indican a NUnit la acción que debe ejecutar con un método o clase, es decir, le indican como interpretar y ejecutar las pruebas implementadas en el método o clase.
  - Todos los atributos que se pueden usar en las pruebas a definir están en el **`namespace`** de del *framework* Nunit
  - Cada clase de prueba que se construya debe incluir una instrucción para el **`namespace`** de Nunit y debe referenciar a **`nunit.framework.dll`**

Atributo: **`TestFixture`**

- El atributo **`[TestFixture]`** marca o identifica una clase que contiene pruebas y/o los métodos **`setup()`** y **`teardown()`** para establecer y liberar los *fixtures* de las pruebas
- Un *fixture* contiene todo los objetos y estructuras a usar en la clase de prueba.

```
namespace NUnit.Tests
{
        using System;
        using NUnit.Framework;
        [TestFixture]
        public class SuccessTests
        {
                // ...
        }
}
```

El atributo [TestFixture] indica que la clase SuccessTests contiene pruebas

Atributo: `SetUp`

- El atributo **`[SetUp]`** se usa dentro de un **`[TestFixture]`** para crear e inicializar los objetos y estructuras de un *fixture* para la clase de pruebas. Se ejecuta justo antes de llamar a cada método de la clase de pruebas

```
namespace NUnit.Tests
{
  using System;
  using NUnit.Framework;
  [TestFixture]
  public class SuccessTests
  {
        [SetUp] public void Init()
        { /* ... */ }
        [TearDown] public void Cleanup()
        { /* ... */ }
        [Test] public void Add()
        { /* ... */ }
  }
}
```

El atributo `[SetUp]` indica que el método `Init()` realizará el SetUp del TestFixture

Atributo: **TearDownAttribute**

- El atributo **[TearDown]** se usa dentro de un **TestFixture** para liberar los objetos y estructuras de un Fixture para la clase de pruebas. Un **TestFixture** debe tener un solo método **TearDown.**
- Se ejecuta justo después de llamar a cada método de la clase de pruebas:

```
namespace NUnit.Tests
{
    using System;
   using NUnit.Framework;
   [TestFixture]
   public class SuccessTests
   {
         [SetUp] public void Init()
         { /* ... */ }
         [TearDown] public void Cleanup()
         { /* ... */ }
         [Test] public void Add()
         { /* ... */ }
   }
}
```

El atributo `[TearDown]` indica que el método `Cleanup()` realizará el *TearDown* del *TestFixture*

Atributo: **Test**

- El atributo **[Test]** marca un método como un método de prueba. Esto se hace dentro de una clase que ya se ha marcado como una clase de prueba usando el atributo [TestFixture]

```
namespace NUnit.Tests
{
  using System;
  using NUnit.Framework;
  [TestFixture]
  public class SuccessTests
        {
                [SetUp]public void Init()
                { /* ... */ }
                [TearDown] public void Cleanup()
                { /* ... */ }
                [Test] public void Add()
                { /* ... */ }
                [Test] public void TestSubtract()
                { /* ... */ }
        }
 }
```

El atributo **[Test]** indica que el método **Add()** es un método de prueba

El atributo **[Test]** indica que el método **TestSubtract()** es un método de prueba

Atributo: **Suite**

- El atributo **[Suite]** se usa para definir una *suite* de pruebas que contiene todas las pruebas a ser ejecutadas en la *Suite*.
- Una *suite* de pruebas se ejecuta desde la línea de comandos usando la opción **/fixture**

```
namespace NUnit.Tests
 {
    using System;
    using NUnit.Framework;
    private class AllTests
    {
      [Suite]
      public static IEnumerable Suite
      {
      get
        {
          ArrayList suite = new ArrayList();
          suite.Add(new OneTestCase());
          suite.Add(new AssemblyTests());
          suite.Add(new NoNamespaceTestFixture());
          return suite; }
        }
      }
 }
```

El atributo `[Suite]` define una suite de pruebas que contiene los métodos `OneTestCase()`, `AssemblyTests()` y `NoNamespaceTestFixture()`

Atributo: **Category**

- El atributo **[Category]** se usa para agrupar pruebas en una categoría particular. Las categorías pueden ejecutarse desde los **test runners** de Nunit.
- Es una alternativa al uso de las *Suites* de pruebas

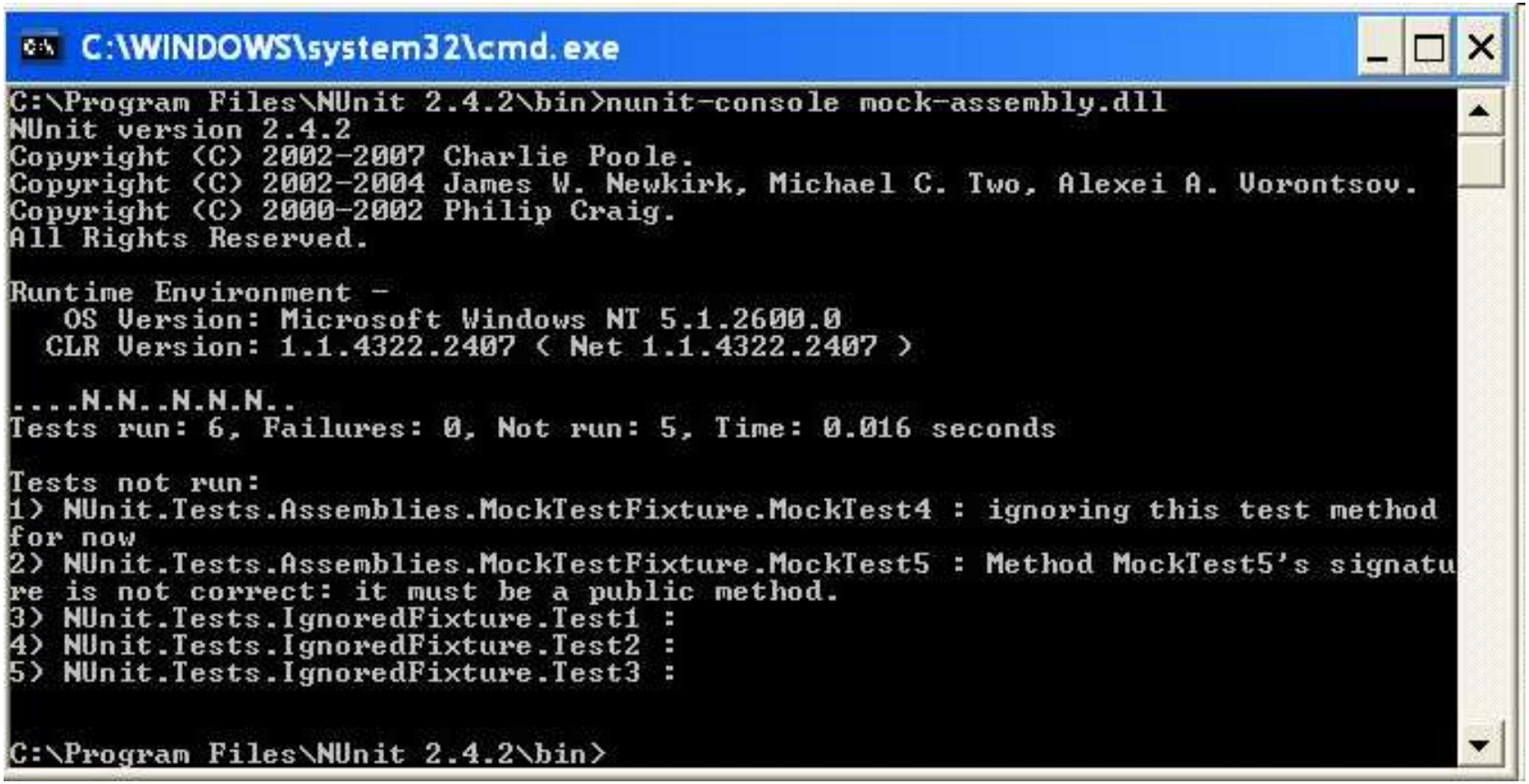
```
namespace NUnit.Tests
{
    using System;
    using NUnit.Framework;
      [TestFixture]
      [Category("LongRunning")]
        public class LongRunningTests
        {
          // ...
        }
 }
```

El atributo `[Category]` define que la clase `LongRunning` está formada por el conjunto de pruebas que se definen en su interior

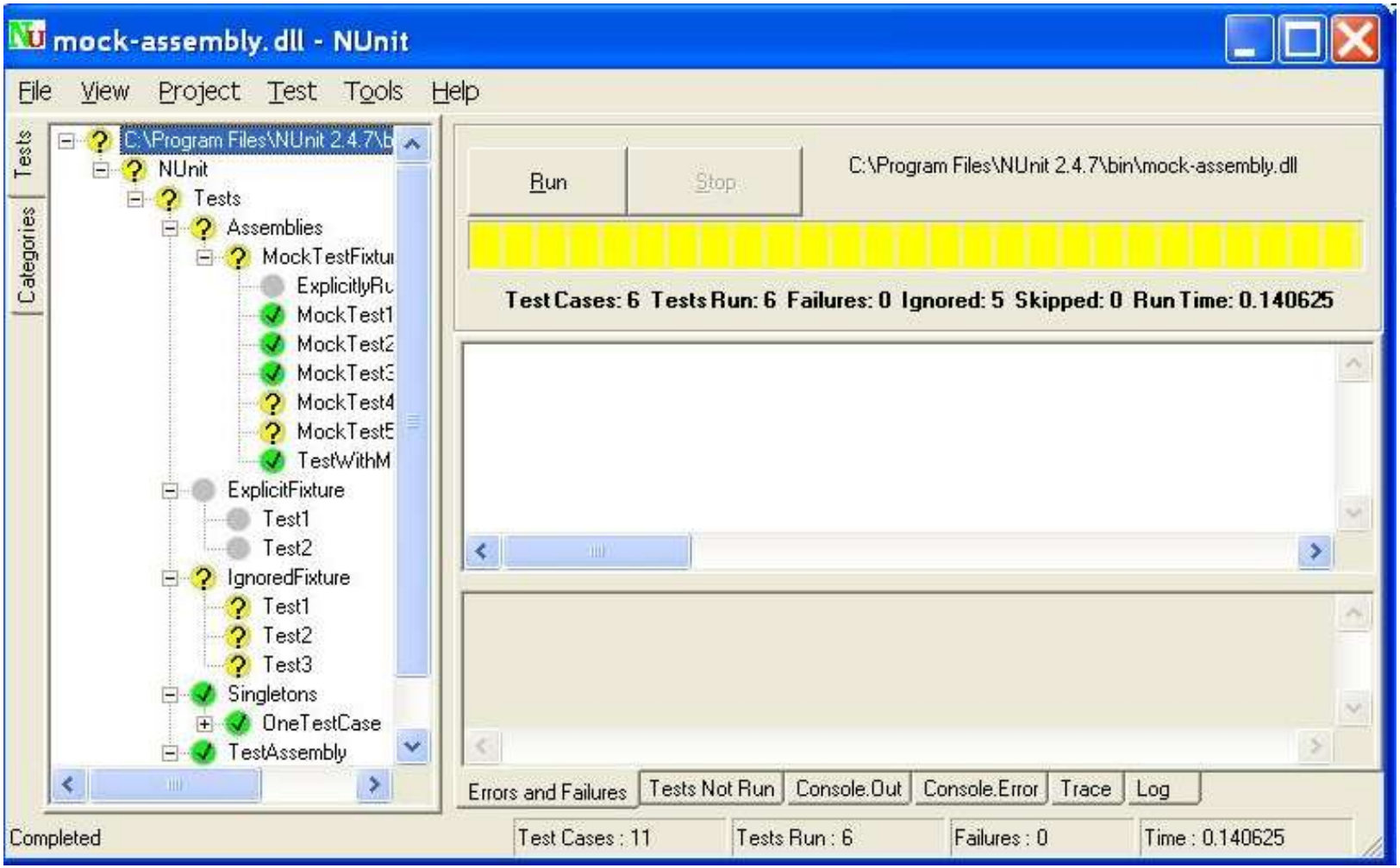
Ejecución de las pruebas con NUnit

- Nunit provee dos formas de ejecución:
  - En línea de comandos (**`nunit-console.exe`**)
    - Se usa cuando no se necesita un indicador gráfico de éxito o fallo de las pruebas
    - Almacena los resultados de las pruebas en formato XML
  - Con interfaz gráfico (**`nunit-gui.exe`**)
    - Muestra las pruebas en una ventana gráfica
    - Despliega un indicador visual del éxito o fallo de las pruebas
    - Permite seleccionar las pruebas a ejecutar
    - Permite recargar el código a probar cuando se modifica y/o se recompila para volver a ejecutar las pruebas

Nunit en línea de comandos



Nunit GUI


mock-assembly.dll - NUnit
File View Project Test Tools Help
Tests
Categories
C:\Program Files\NUnit 2.4.7\b
NUnit
Tests
Assemblies
MockTestFixtur
ExplicitlyRu
MockTest1
MockTest2
MockTest3
MockTest4
MockTest5
TestWithM
ExplicitFixture
Test1
Test2
IgnoredFixture
Test1
Test2
Test3
Singletons
OneTestCase
TestAssembly
Run
Stop
C:\Program Files\NUnit 2.4.7\bin\mock-assembly.dll
Test Cases: 6 Tests Run: 6 Failures: 0 Ignored: 5 Skipped: 0 Run Time: 0.140625
Errors and Failures
Tests Not Run
Console.Out
Console.Error
Trace
Log
Completed
Test Cases : 11
Tests Run : 6
Failures : 0
Time : 0.140625

Resumen

- En este tema se estudió NUnit, que es una implementación para todos los lenguajes microsoft .NET del framework de pruebas unitarias Xunit
- Se estudiaron la bases de funcionamients de NUnit que son:
  - Las aserciones que son métodos del framework de NUnit utilizados para comprobar y comparar valores
  - Los atributos personalizados que indican a NUnit como interpretar y ejecutar las pruebas implementadas en el método o clase.
- Se vió la forma de ejecutar las pruebas en NUnit
  - Usando el runner via consola
  - Usando el runner gráfico

Pruebas del Software
Análisis estático

Daniel Rodríguez

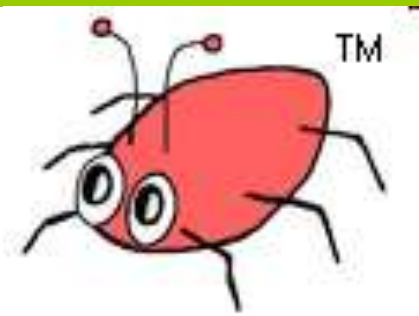
Contenido

- Análisis estático
  - ¿Qué es?
- Herramientas de análisis estático
  - Analisis de bytecode
    - FindBugs
  - Análisis de código fuente
    - PMD

Análisis estático

- Analiza los programas sin ejecutarlos
  - En Java se puede analizar código fuente
  - o *bytecod*e (ficheros `.class`)
- Independiente de los casos de prueba
  - Generalmente no se sabe lo que hace el programa
- Busca violaciones de "reglas" en lenguajes de programación.
- Multitud de herramientas
  - Código abierto:
    - FindBugs (bytecode) `http://findbugs.sourceforge.net/`
    - PMD (código fuente) – `http://pmd.sourceforge.net/`
    - CheckStyle – `http://checkstyle.sourceforge.net/`
  - Comerciales:
    - Fortify Software, KlocWork, Coverity, Parasoft, SureLogic

FindBugs™



- Análisis estático de programas en Java
  - **`http://findbugs.sourceforge.net/`**
  - Analiza *bytecode*, i.e., ficheros **`.class`**
    - Para ello utiliza BCEL (Byte Code Engineering Library)
      - **`http://jakarta.apache.org/bcel/`**
    - La salida es un fichero de texto o XML
  - Busca en el código patrones de errores
    - Su objetivo es indicar errores que realmente lo son y minimizar los *falsos positivos*, i.e.,
      - dar demasiados avisos de error que no lo son
  - Demostraciones on-line:
    - **`http://findbugs.sourceforge.net/demo.html`**

FindBugs: Características

- Hay muchos errores detectables con análisis estático.
  - Usado en proyectos importantes
  - JDK, Glassfish, Eclipse, etc (con millones de líneas de código)
  - Encontrado cientos de errores
- Integrable con IDEs:
  - Netbeans/Swing/Eclipse/Ant/SCA
  - Integración con Netbeans

https://grizzly.dev.java.net/tutorials/findBugs-nb-tutorial/index.html

FindBugs: Ejemplos Patrones de Errores I

- Bucles recursivos infinitos

```
public WebSpider() {
      WebSpider w = new WebSpider(); }
```

- Hascode/Equals
  - En Java es necesario redefinir equals() y hashCode()
    - Con este error, los objectos no funcionan bien en tablas de dispersión, maps, sets, etc.
- Valores de retorno ignorados (e.g., objectos inmutables)

```
String name= workingCopy.getName();
      name.replace('/', '.');
```

FindBugs: Ejemplos Patrones de Errores II

- Punteros `null`
  - Referenciando valores null genera `NullPointerException`.

```
if (name != null || name.length >0) {
    ...
```

- Nombre errones o que llevan a confusión
  - Herencia con mismo método, distinta capitalización

```
public abstract class Dialog extends Window {
  protected Button getOKButton () {...}
```

```
public abstract class InputDialog extends
Dialog {
  protected Button getOkButton () {...}
```

FindBugs: Patrones de Errores

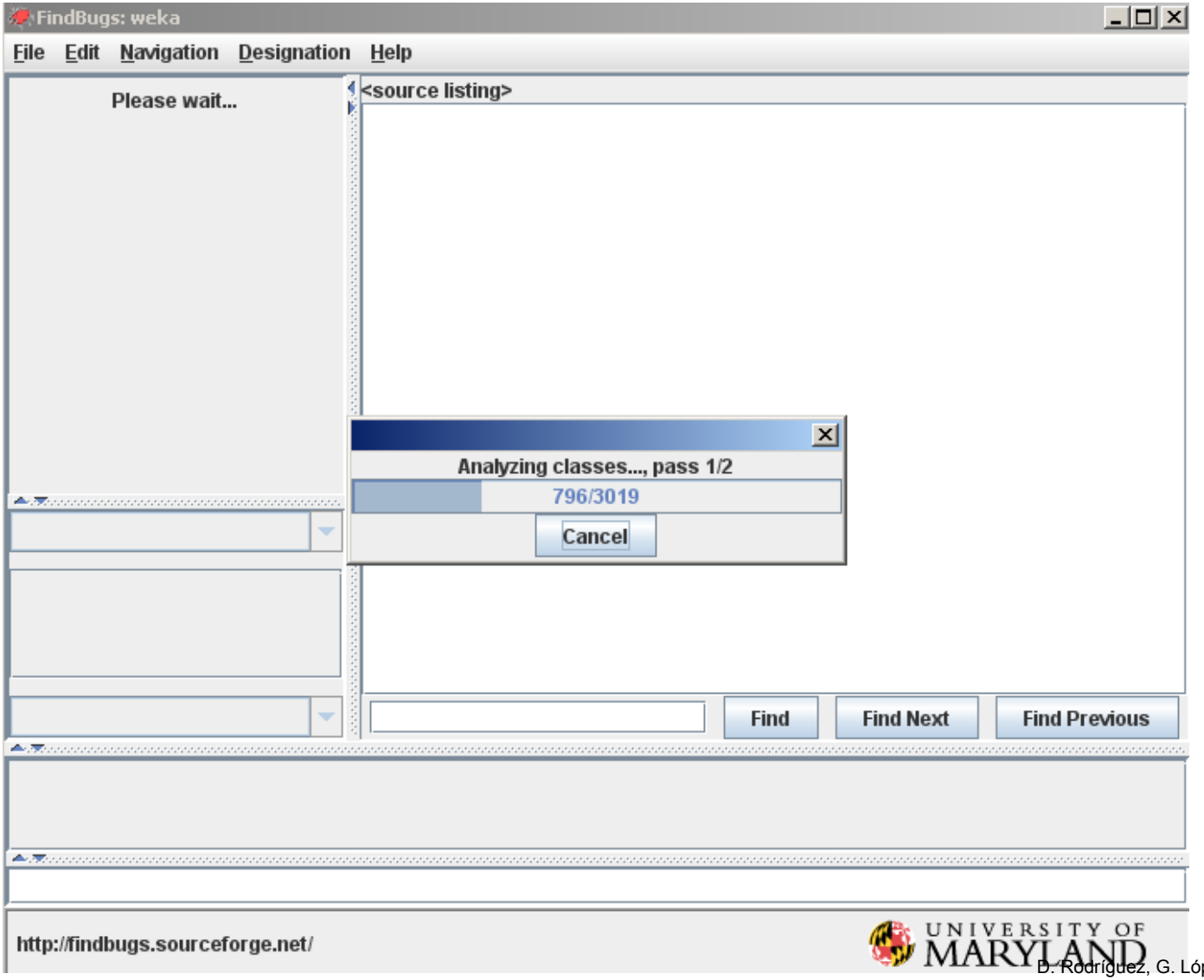
- Y muchos más (sobre 200 patrones de error):

`http://findbugs.sourceforge.net/bugDescriptions.html`

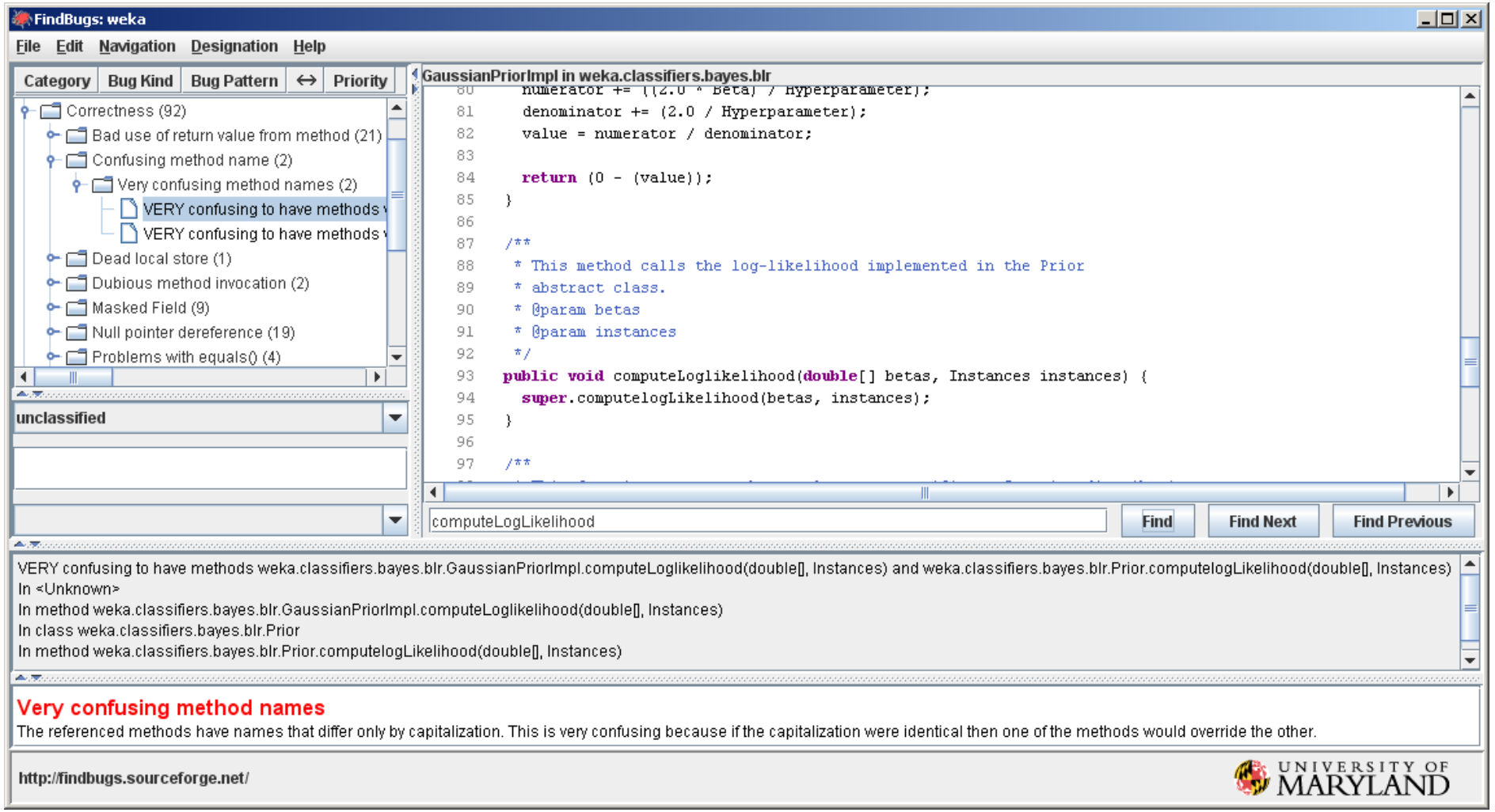
  - Se muestran en forma de tabla, agrupados por categoría:
    - Mala práctica, correctitud, internacionalización, código vulnerable, correctitud, correctitud en ejecución con hilos, eficiencia, seguridad y "cosas raras".

Description	Category
AM: Creates an empty jar file entry	Bad practice
AM: Creates an empty zip file entry	Bad practice
…	…

FindBugs: Ejecución I



FindBugs: Ejecución



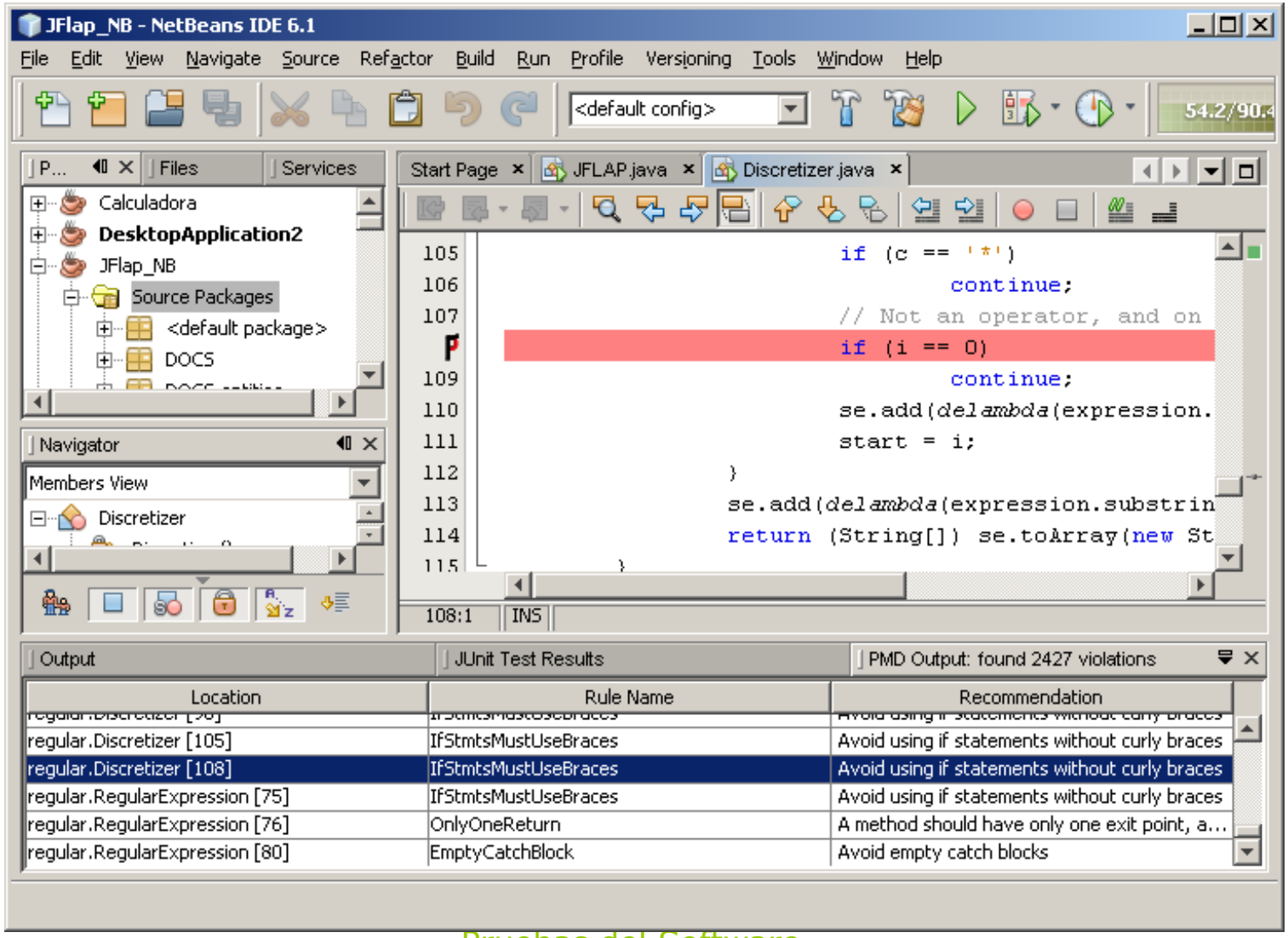
PMD

- PDM (Don't Shoot the Messenger)
  - **`http://pmd.sourceforge.net/`**
- Se ejecuta sobre *código fuente* buscando patrones (reglas) de errores.
  - Todas las reglas:
    - **`http://pmd.sourceforge.net/rules/index.html`**
  - Ejemplos:
    - Posibles *bugs* – con sentencias **`try`**/**`catch`**/**`finally`**/**`switch`**
    - Código muerto
      - variables no usadas, parámetros, métodos privados, etc.
    - Código no óptimo – mal uso de String/StringBuffer
    - Expresiones más complicadas de lo necesario
      - sentencias **`if`** inecesarias,
      - Bucles **`for`** que pueden sustituirse por bucles **`while`**
    - Código duplicado

PMD

- Se integra con IDEs
  - Netbeans, Eclipse, JEdit, JDeveloper, etc.
- O puede ser ejecutado desde la línea de comandos
  - Línea de comandos simple, Ant, Maven, etc.
- Además, proporciona alguna herramienta básica con GUI:
  - Encontrar código duplicado
  - Creación de reglas (patrones de errores)

PMD: Ejecución en Netbeans



PMD: Ejecución línea de comandos

- Necesita tres parámetros:
  - Fichero java fuente o directorio
  - Formato de salida
  - Fichero de reglas o un un string de reglas separadas por comas
    - Ej. Código muerto

```
c:\> java -jar pmd-4.2.3.jar c:\code text
  unusedcode,imports -targetjdk 1.6 -debug
```

PMD: Ejemplo

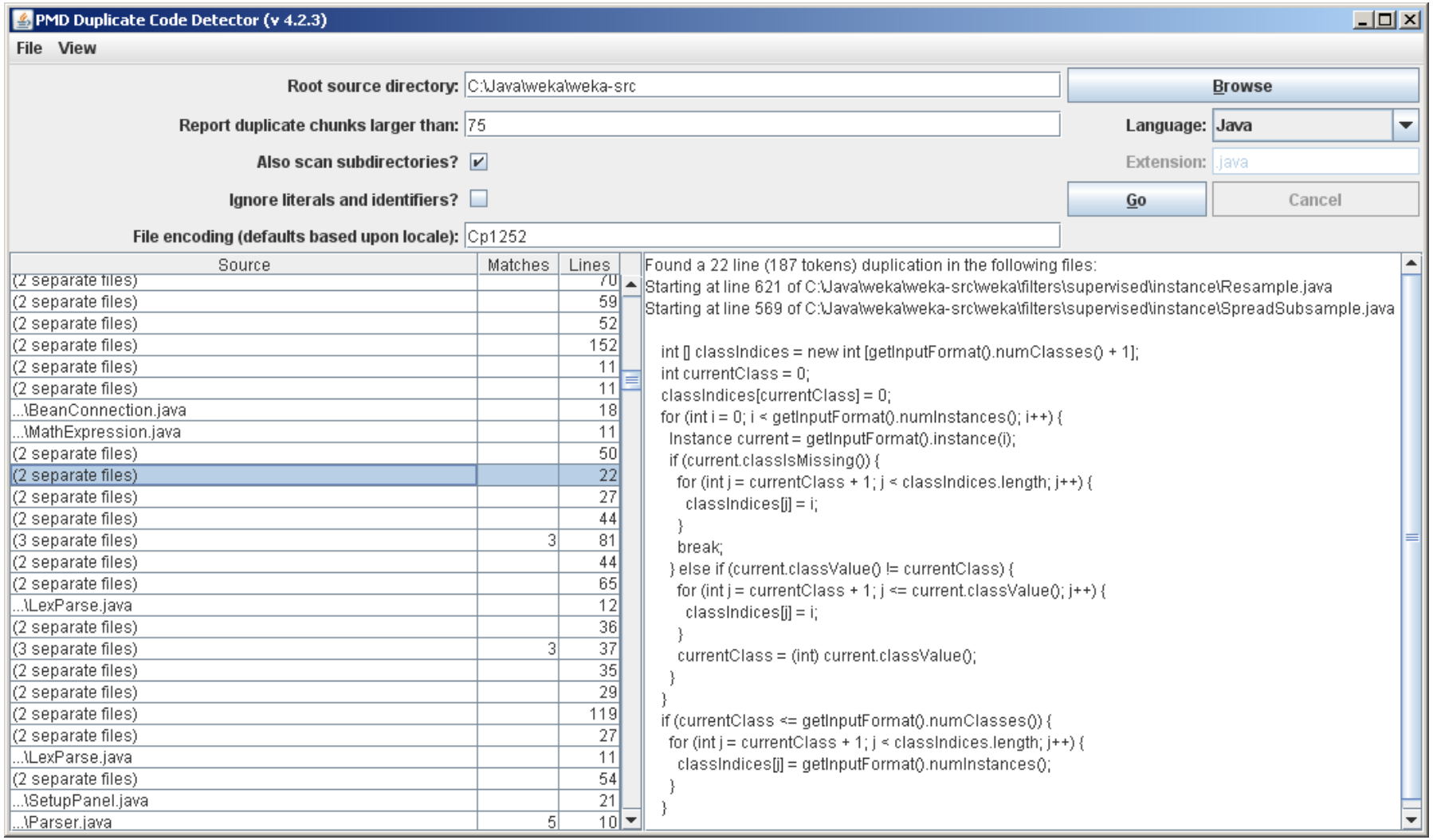
- Como entrada:

```
C:\>pmd C:\weka-src\weka\Filters text basic > salida.txt
```

- Salida:

```
C:\weka-src\weka\filters\Filter.java:542     These nested if
statements could be combined
...
C:\Java\weka\weka-
src\weka\filters\unsupervised\attribute\AddNoise.java:498   An
empty statement (semicolon) not part of a loop
...
```

PMD: Ejemplo código duplicado



Análisis estático con métricas

- Ciertas métricas de código, por ejemplo, profundidad de la herencia, complejidad ciclomática, número de líneas de código pueden ser indicadores de problemas.
- Con herramientas se comprueba que módulos, métodos, clases en ciertas métricas no excedan ciertos umbrales.
  - Cuando módulos, métodos, clases exceden el umbral definido en cierta métrica, ese módulos, métodos, clases tiene más posibilidades de contener errores.

JavaNCSS:

File Help

Packages | Classes | Methods

Sun, Nov 02, 2008 00:05:16 Europe/Paris

Nr.	NCSS	Functions	Classes	Javadocs	Class
1	6	2	0	3	gui.action.AboutAction
2	5	2	0	3	gui.action.AutomatonAction
3	14	2	1	3	gui.action.BruteParseAction
4	33	4	1	5	gui.action.BuildingBlockSimulateAction
5	30	3	1	4	gui.action.CloseAction
6	35	3	2	4	gui.action.CloseButton
7	8	2	0	3	gui.action.CloseWindowAction
8	55	3	0	4	gui.action.CombineAutomaton
9	53	7	6	8	gui.action.ConvertAutomatonToGrammarAction
10	28	2	1	3	gui.action.ConvertCFGLL
11	28	2	1	3	gui.action.ConvertCFGLR
12	36	4	0	5	gui.action.ConvertFSAToGrammarAction
13	17	2	1	3	gui.action.ConvertFSAToREAction
14	55	4	0	5	gui.action.ConvertPDAToGrammarAction
15	31	2	1	3	gui.action.ConvertRegularGrammarToFSA
16	95	8	1	8	gui.action.CYKParseAction

Pruebas del Software
JUnit 4

Daniel Rodríguez

JUnit 4

- En su versión más actual JUnit explota las ventajas de Java 5 (JDK):
  - *Import* estáticos
    - Permite a una clase hacer uso de métodos y variables estáticas de otra clase sin el incluir el cualificador (nombre de la clase a la que perteneces
  - Anotaciones
    - Son metadatos, que permiten añadir información para el compilador (u otras herramentas)
    - Es texto precedido por "`@`". Por ejemplo:

```
@test
public void comprobarValor(…){...}
```

JUnit 3 vs. JUnit 4

- Nombre de la clase de pruebas es el nombre de la clase más el sufijo **`Test`**
  - Sin heredar de la clase **`TestCase`**.
  - Los aserciones están disponibles meditante import estáticos. P.e.:

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

- Los métodos **`setUp()`** y **`tearDown()`** se sustiyen las anotaciones **`@Before`** y **`@After`** respenctivamente.
  - Los métodos pueden tener ahora cualquier nombre, aunque se recomienda mantener `setUp()` y tearDown().
  - Dos anotaciones nuevas @BeforeClass y @AfterClass
    - Se ejecutan sólo una vez en vez de ejecutarse con cada caso de prueba.
- Los métodos de prueba se definen con la anotación @Test.
  - Se recomienda llamar a los métodos de prueba igual que los métodos de la clase a probar.

`@Test` parámetros

- *Timeout* – tiempo máximo en milisegundos que debe esperar por la ejecución. P.e.:

```
@Test(timeout=1000)
public void getHTTP(…){…}
```

  - El método debe ejecutarse en menos de un segundo.

- *Expected* – se espera una excepción. P.e.:

```
@Test(expected=DBConnectException.class)
public void accessDB(…) {…}
```

  - Se quiere probar que la connexión a la BD falla

Tests parametrizados

- Múltiples entradas de prueba: **`@Parameters`**
- Una clase estática devuelve la serie de parámetros a ejecutar como instancia de la clase **`Collection`**
- Condiciones:
  - (i) el método a probar no debe tener parámetros
  - (ii) hay que pasarle al constructor de la clase los parámetros devueltos por la clase **`Collection`** almacenándolos en atributos privados de la clase
  - (iii) La clase debe anotarse así:

```
@RunWith(value=Parameterized.class)
```

Ejemplo JUnit 4

```
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class RegistroTest {


  @Test
  public void personaEnRegistro() {
    Registro registro = new Registro();
    boolean resultado = registro.checkByTitle("Armando Camorra");
    assertEquals("La persona debería de estar.", true, result);
  }
}
```

Ejemplo JUnit4 (II)

```
import org.junit.Test;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import java.util.Arrays;
import java.util.Collection;
import org.junit.runner.RunWith;
import static org.junit.Assert.*;

@RunWith(value = Parameterized.class)
public class ComplejosTest {

    private Complejo par1, par2, res;

    public ComplejosTest(Complejo par1, Complejo par2, Complejo res){
          this.par1 = par1;
          this.par2 = par2;
          this.res = res;
    }

    @Parameters
    public static Collection valores () {
         Object[][] data = new Object [][] {
         {new Complejo (0,0), new Complejo (1,2), new Complejo(1,2)},
         {new Complejo (2,3), new Complejo (4,0), new Complejo(6,3)}
         };
         return Arrays.asList (data);
    }

    @Test
    public void testSumar() {
         Complejo resultado = par1.sumar(par2);
         assertEquals(res, resultado);
    }
}
```

Pruebas del Software
Extensiones a Junit

Daniel Rodríguez

Contenidos

- *Frameworks* que extienden JUnit
  - Pruebas de Bases de datos: DBUnit
  - Aplicaciones Web: JWebUnit, HTTPUnit, HTMLUnit, JMeter (Pruebas de carga en general)
- Pruebas de Cobertura
  - Con caja blanca: Cobertura, Emma
- Generadores de casos de prueba:
  - Clasificación métodos de prueba
    - Criterios basados en defectos
  - Herramientas de captura y reproducción
  - Automatización de pruebas: JTestCase

Pruebas software con Bases de Datos

D. Rodríguez, G. López

Pruebas de software con bases de datos

- Clasificación de Bases de Datos (BD):
  - BD de producción
    - Base de datos real sobre la que trabaja la aplicación
  - BD datos de prueba
    - BD con las mismas características que la BD de producción, pero con muchos menos datos
      - Mayor rapidez en los accesos.
  - BD de integración:
    - BD idéntica a la de producción, donde se realizan las pruebas antes de ejecutarlas sobre la BD real.

Pruebas de software con bases de datos

- Si se quiere aislar la Bases de Datos (BD)
  - Objetos Simulados (Mock Objects)
    - Datos creados artificialmente
    - Es mucho más rápido que acceder a la BD
    - No se prueba realmente la BD
    - Ver transparencias correspondientes
- Para proyectos dirigidos por BD
  - **DBUnit**
    - Extensión de JUnit, aunque puede ser usado independientemente (por ejemplo con Ant)

  - Su principal utilidad es poner la base de datos en un estado consistente entre las distintas ejecuciones de las pruebas
  - Proporciona los métodos para comparar datos entre ficheros, preguntas a la BD y tablas de la base de datos.
  - Projecto de Código abierto

    `http://dbunit.sourceforge.net/`

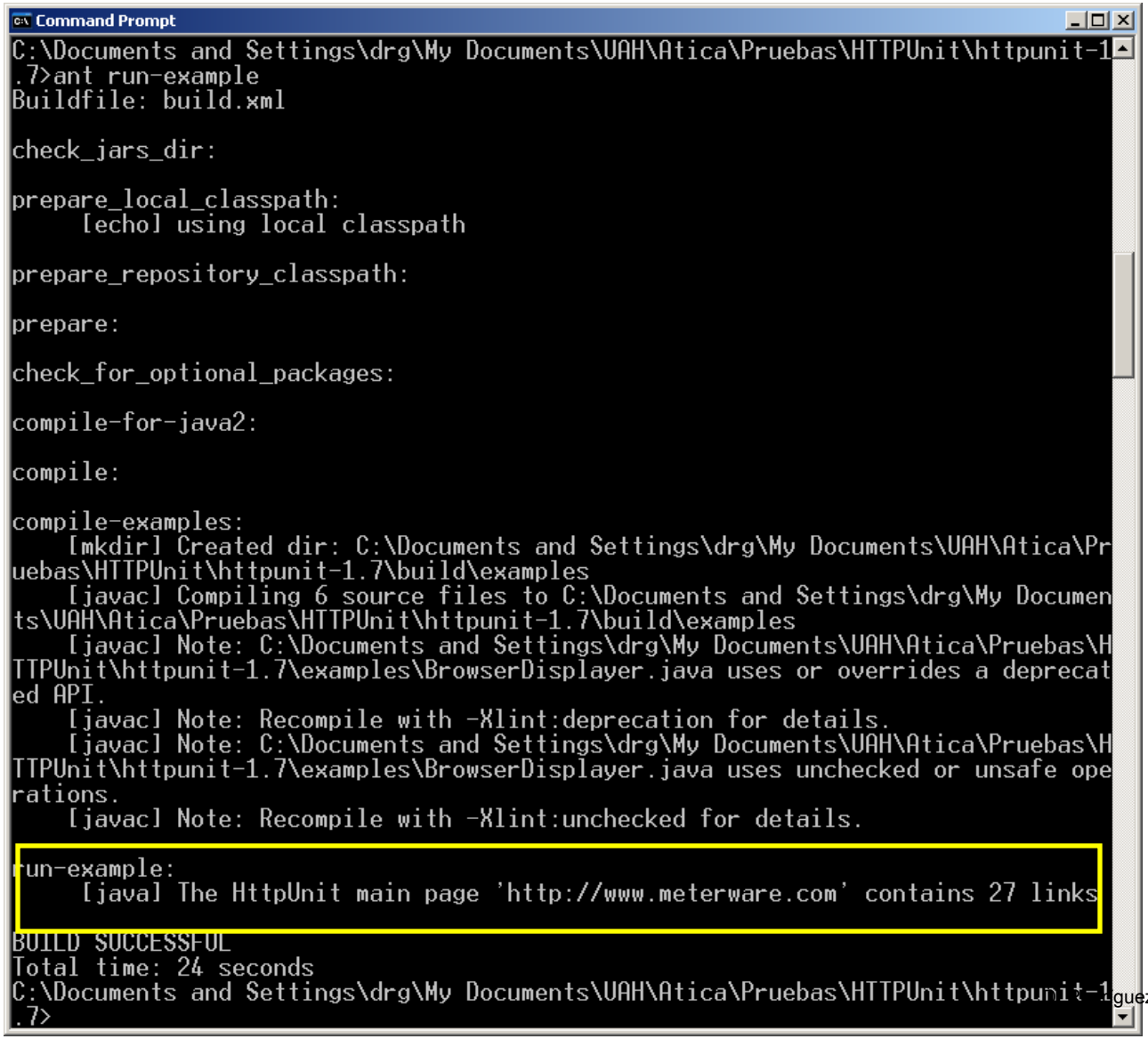
Pruebas Web

HTTPUnit, HTMLUnit, JWebUnit, JMeter

HTTPUnit

- HTTPUnit
  - Extensión de JUnit para obtener páginas Web, navegar enlaces, obtener la estructura de una página, formularios, etc.
  - Código abierto:
    `http://httpunit.sourceforge.net/`

HTTPUnit. Ejemplo de ejecución.



HTMLUnit

- HTMLUnit
  - Modela documentos HTML proporcionando una API para rellenar formularios, comprobar enlaces, etc.
  - Simula un navegador Web
  - Código abierto:
    `http://htmlunit.sourceforge.net/`

JWebUnit

- jWebUnit extiende JUnit facilitando la creación de *test* de aceptación de aplicaciones Web
  - Navegación de los enlaces
  - Entrada y envío de formularios
  - Validación de índices de contenidos, etc.
  - Extiende HTTPUnit
- Código abierto:

  `http://jwebunit.sourceforge.net/`

JMeter

- Permite la ejecución de pruebas de cargas basadas en diferentes protocolos:
  - Web - HTTP, HTTPS
  - SOAP
  - Database via JDBC
  - LDAP
  - JMS
  - Mail - POP3
- Posibilidades gráficas de medidas de rendimiento
- Código abierto:
  - http://jakarta.apache.org/jmeter/
  - Integrado con Netbeans

Pruebas de cobertura

Pruebas de Cobertura

- Monitorizan la ejecución de los casos de prueba para medir el grado de cobertura del código alcanzado por los casos de prueba
- Se añaden trazas al código para registrar las instrucciones que se han ejecutado.
- La idea es fijar un grado de cobertura, e ir incrementando los casos de prueba hasta alcanzarlo.

Herramientas de cobertura

- Cobertura

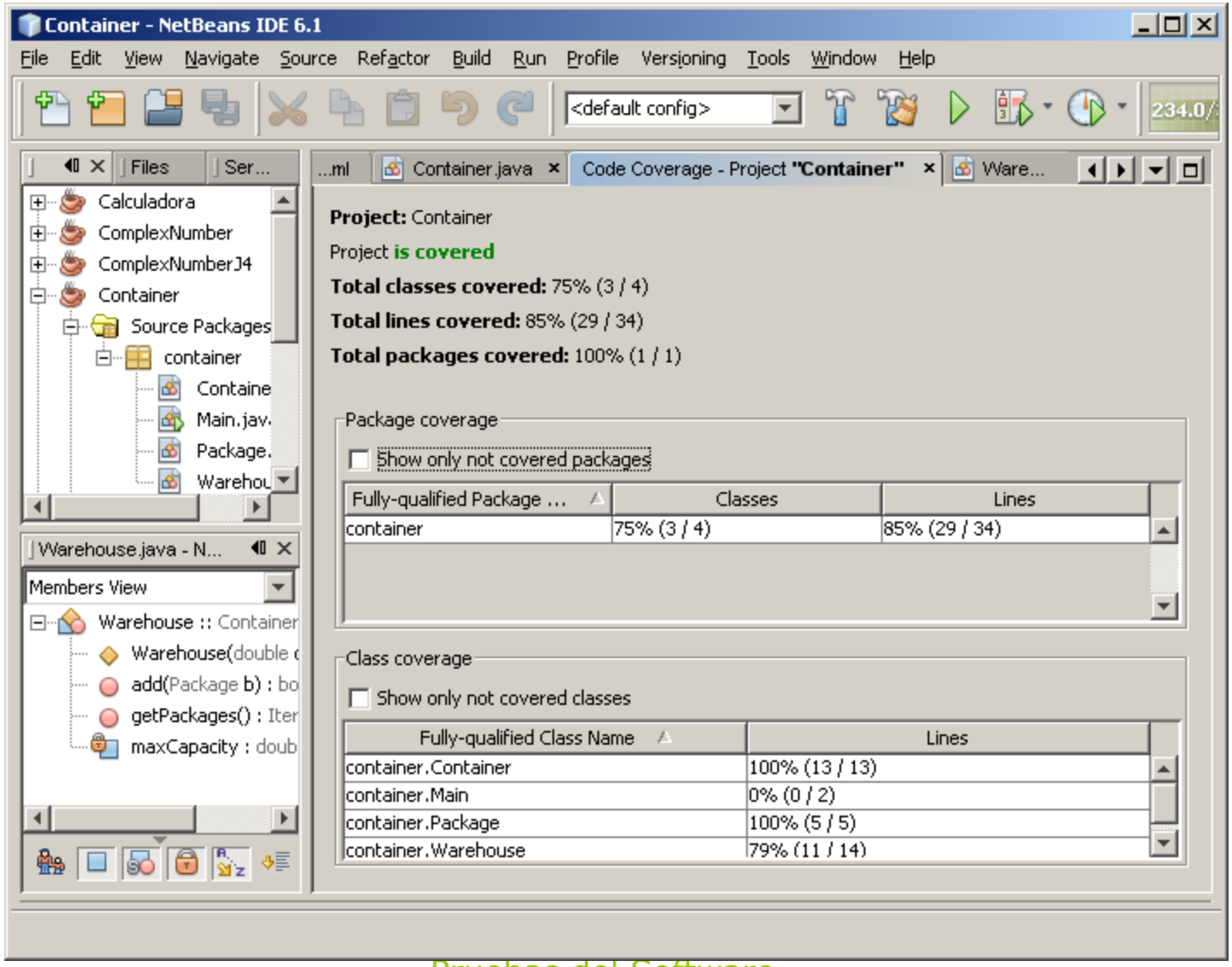
  `http://cobertura.sourceforge.net/`

  - Cobertura de sentencias y ramas para el código
  - Medidas de complejidad ciclomática y la media por paquetes y proyecto completo.
  - Código abierto
- Emma

  `http:emma.sourceforge.net/`

  - Similar a *Cobertura*.
  - Ejecución desde línea de comando o integrada en Netbeans y Eclipse (comercial)

Cobertura con Emma. Netbeans



Generadores de casos de prueba

Clasificación de los métodos de prueba

- **Fuente de información** para su generación
  - Métodos basados en la *especificación*
    - Sin conocer la estructura del código (caja negra)
  - Métodos basados en el *código* (caja blanca)
  - Métodos basados en *ambos*, especificación y código (caja gris)
- **Criterio de suficiencia**
  - *Criterio de parada* indicando si es necesario seguir o no probando
    - Métodos estructurales. Deben cubrir un conjunto de elementos de estructura basados en el grafo del programa según su *flujo de control* o *flujo de datos*.
  - Métodos basados en encontrar defectos
    - Miden la capacidad de encontrar defectos que tiene el conjunto de casos de prueba
  - Métodos basados en encontrar errores
    - Generar casos que comprueben ciertos puntos específicos
- **Técnica/s utilizada/s**:
  - Ejecución simbólica, algoritmos genéticos, etc.

Criterios basados en defectos

- Pruebas de mutación (*mutation testing*)
  - Crear programas con defectos (mutantes) a partir del original.
  - El objetivo de *mutation testing* es generar casos de prueba que distingan a los mutantes de los programas originales.
    - Si un mutante es distingido por un caso de prueba, se destruye (*killed*).
    - En caso contrario el mutante sigue activo (cuando la salida del programa original y el mutante, es la misma.
  - La suficiencia de un conjunto de casos se mide con:
    - Mutation Score = (D / (M – E)) * 100, donde
      - D es el nro de mutantes destruidos, M es el total de mutantes y E es el nro de mutantes equivalentes.
  - Jester – the Junit Test Tester
    - Extiende JUnit con la técnica de pruebas de mutación

      `http://jester.sourceforge.net/`

- Introducción de fallos artificiales (*error seeding*)
  - Se introducen errores deliberadamente, y se mide el porcentaje de errores encontrados.

Herramientas para captura y reproducción

- Herramientas capaces de funcionar en 2 modos:
  - **Captura**: eventos de entrada y salida durante la ejecución de un programa:
    - Entradas de información
    - Teclas pulsadas
    - Salidas
  - **Reproducción**: las capturas previas para ejecutar casos de pruebas
- Útiles en pruebas de regresión
- Inconvenientes:
  - Alto coste de implantación
- Ejemplo de herramienta:

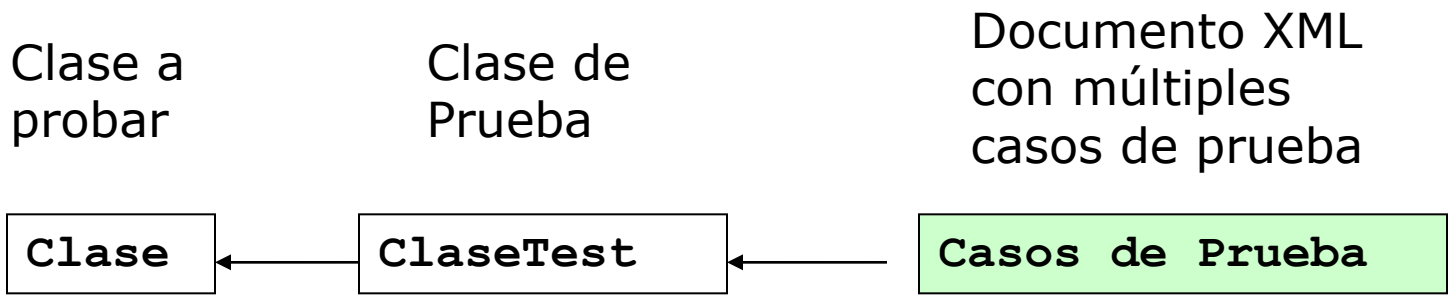
  `http://marathonman.sourceforge.net/`

Selenium

- Herramienta de reproducción y captura para Web
  - **`http://selenium.openqa.org/`**
  - Extensión para Firefox.

JTestCase

- Automatizar casos de test donde sólo cambian los valores de los parámetros.



- Código libre:

  `http://jtestcase.sourceforge.net/`

JTestCase Ejemplo

```
public class Calculator {
  public Calculator() {super();}
  public int calculate(int var1, int var2, String opt) {
   if (opt.equals("+"))   return var1 + var2;
   if (opt.equals("-"))   return var1 - var2;
   return 0;
  }
}
```

```
<test-case name="positive-add">
  <params>
    <param name="var1" type="int">10</param>
    <param name="var2" type="int">20</param>
    <param name="opt" type="java.lang.String">+</param>
  </params>
 <asserts>
   <assert name="result" type="int" action="EQUALS">30</assert>
 </asserts>
</test-case>
<test-case name="positive-minus">
  <params>
    <param name="var1" type="int">10</param>
    <param name="var2" type="int">20</param>
    <param name="opt" type="java.lang.String">-</param>
  </params>
  <asserts>
     <assert name="result" type="int" action="EQUALS">-10</assert>
  </asserts>
</test-case>
```