

Prácticas de Diseño de Software

Laboratorio 1: **Pruebas Unitarias de Software**

Pruebas Unitarias de Software

Objetivos

- Introducción a las pruebas de software
- Familiarizarse con la herramienta JUnit

Índice

1. Pruebas de Software
 - 1.1. Definiciones
 - 1.2. Fallos en la Historia del Software
 - 1.3. Diseño de Casos de Prueba
 - 1.4. Tipos de Pruebas. Pruebas Unitarias (*Unit Testing*)
 - 1.5. Herramientas de Prueba de Unidad
 - 1.5.1. ¿Porqué utilizar herramientas de pruebas unitarias?
2. El framework JUnit
 - 2.1. Test JUnit
 - 2.1.1. Clase Assert
 - 2.1.2. Fixtures
 - 2.1.3. Mocks
 - 2.1.4. Test Suite
 - 2.1.5. Clases Parametrizadas
 - 2.2. Guía para escribir pruebas
 - 2.3. JUnit y Eclipse
 - 2.3.1. Ejecución de las pruebas en Eclipse
 - 2.4. Un Ejemplo Sencillo
3. Ejercicio a Desarrollar

1. Pruebas de Software

1.1 Definiciones

Una **prueba** (o *test*) es una actividad en la cual un sistema o uno de sus componentes se ejecuta en **circunstancias previamente especificadas**, los resultados se **observan** y registran y se realiza una **evaluación** de algún aspecto. Se dice que una prueba ha tenido éxito cuando la prueba ha permitido detectar algún error. Por el contrario la prueba no tiene éxito si no revela ningún error.

Hablamos de **pruebas de software** para referirnos al proceso de evaluación de un producto desde un punto de vista crítico. Las pruebas de software pueden perseguir validar o verificar el software. A menudo se confunden ambos términos, pero tienen significados diferentes, puesto que persiguen evaluar aspectos diferentes del software:

- *Validación*: El proceso de evaluación de un sistema o de uno de sus componentes durante o al final del proceso de desarrollo para determinar si satisface los requisitos marcados por el usuario.
- *Verificación*: El proceso de evaluación de un sistema o de uno de sus componentes para determinar si los productos de una fase dada satisfacen las condiciones impuestas al comienzo de dicha fase.

Las pruebas de software son un aspecto fundamental en el proceso de desarrollo de un producto software.

1.2 Fallos en la Historia del Software

A lo largo de la historia del software nos encontramos con fallos de software que han provocado situaciones catastróficas. Estos fallos históricos permiten que tomemos conciencia de la importancia de diseñar software de calidad, para lo que es necesario realizar las pruebas de software de forma correcta y completa. Algunos de los peores fallos de la historia del software son¹:

- Fallo en la sonda Mariner 1 (julio de 1972). Un "bug" en el software de vuelo de la sonda Mariner I provocó que ésta se desviara de su curso preestablecido. Los responsables de la misión se vieron obligados a destruir el cohete.
- Explosión en un gaseoducto soviético (1982). La mayor explosión registrada en la Tierra por causas no nucleares tuvo su origen en un fallo de programación.
- Acelerador médico Therac-25 (Década de los 80). El Therac-25 emitía radiación de alta energía sobre células cancerosas sin dañar el tejido circundante. Pero debido a un fallo de programación la máquina podía emitir 100 veces más energía de la requerida. A consecuencia de este "bug" murieron al menos cinco pacientes y varias decenas sufrieron los efectos de verse expuestos a una elevada radiación.
- El 'Gusano de Morris' (1988). Un estudiante estadounidense liberó un programa creado por él mismo que infectó entre 2.000 y 6.000 ordenadores sólo el primer día, antes ser rastreado y eliminado. Para que su 'gusano' tuviera efecto, Morris

¹ Fuente: www.wired.com

descubrió dos errores en el sistema operativo UNIX, que le permitieron tener acceso no autorizado a miles de ordenadores.

- Generador de números aleatorios de Kerberos. Los autores del sistema de generación de números aleatorios Kerberos —que se utiliza para hacer comunicaciones seguras a través de la Red— fallaron a la hora de conseguir que su programa realmente eligiera los números aleatoriamente. Debido a ese fallo, durante ocho años fue posible entrar en cualquier ordenador que utilizara el sistema Kerberos para autenticar el usuario.
- Caída de la red de AT&T (1990). Un "bug" en el software que controlaba los conmutadores de las llamadas de larga distancia del gigante telefónico AT&T dejó a 60.000 personas sin servicio durante nueve horas.
- División en coma flotante del Intel Pentium. Un problema con los microprocesadores provocó que éstos fallaran cuando tenían que dividir números con coma flotante. Aunque el fallo afectaba a pocos usuarios, resultó todo un problema para Intel, que terminó por cambiar entre tres y cinco millones de chips, en una operación que le costó más de 475 millones de dólares.
- El Ping de la Muerte. Debido a un problema que afectaba al código que maneja el protocolo IP, era posible "colgar" un ordenador con Windows enviándole un ping corrupto. El problema afectaba a varios sistemas operativos pero el peor parado era Windows, que se bloqueaba mostrando la famosa "pantalla azul de la muerte" al recibir esos paquetes.
- Desintegración del Ariane 5 (1996). Un "bug" en una rutina aritmética en la computadora de vuelo provocó que la computadora fallara segundos después del despegue del cohete; 0,5 segundos más tarde falló el ordenador principal de la misión. El Ariane 5 se desintegró 40 segundos después del lanzamiento.
- Sobredosis radiológica en el Instituto Nacional del Cáncer de la Ciudad de Panamá. Los ingenieros de la empresa Multidata Systems International calcularon erróneamente la dosis de radiación que un paciente podría recibir durante la terapia de radiología. El fallo estaba en el software de control de la máquina de rayos, que provocó que al menos ocho pacientes murieran por las altas dosis recibidas y otros 20 recibieran sobredosis que podrían causar graves daños a su salud.

1.3 Diseños de Casos de Prueba

Las pruebas deben ser diseñadas de forma que tengan la mayor probabilidad de encontrar el mayor número de errores con la mínima cantidad de esfuerzo y tiempo. Es por ello que las pruebas de software requieren el **diseño de casos de prueba**. Un caso de prueba es:

- un conjunto de entradas para ser ejecutadas,
- unas determinadas condiciones de ejecución y,
- unos resultados esperados.

Cuando nos enfrentamos al diseño de casos de prueba podemos adoptar dos enfoques:

- El enfoque estructural o de **caja blanca**. Se centra en la estructura interna del programa (analiza los caminos de ejecución). Con este enfoque se diseñan

pruebas de forma que se asegure que la operación interna se ajusta a las especificaciones, y que todos los componentes internos se han probado de forma adecuada.

- El enfoque funcional o de **caja negra**. Se centra en las funciones, entradas y salidas. En este caso, se diseñan pruebas de forma que se compruebe que cada función es operativa.

1.4 Tipos de Pruebas. Pruebas Unitarias (*Unit Testing*)

Existen diferentes tipos de pruebas de software que se pueden realizar durante el proceso de desarrollo de un producto software:

- Pruebas unitarias: se prueban los módulos de forma independiente
- Pruebas de integración: se prueban agrupaciones de módulos relacionadas
- Pruebas de sistema: se prueba el sistema como un todo
- Pruebas de validación (o de aceptación): prueba del sistema en el entorno real de trabajo con intervención del usuario final

Durante esta sesión de laboratorio nos centraremos en las pruebas unitarias (Unit Testing). Hoy en día este tipo de pruebas se ha convertido en una poderosa herramienta de pruebas al código, sobre todo en empresas dónde los proyectos de software frecuentemente alcanzan niveles de complejidad elevados.

Cuando escribimos código no lo hacemos de forma perfecta desde la primera vez, la compilación misma del proyecto es la primera prueba a nuestra aplicación. El compilador verifica la sintaxis e indica las líneas dónde ha encontrado errores. Es una gran ayuda pero termina ahí. El programa puede y ciertamente contendrá errores, muchos de los cuáles son únicamente visibles al ejecutarlo. Tenemos entonces que volver al código, buscar la línea del error y corregirlo, algo que incrementa el tiempo de desarrollo del software. No podemos eliminar todos los errores, pero sí es posible reducirlos a un nivel más aceptable. Aquí es donde entran las Unit Testing. Ayudándonos a buscar errores escribiendo pruebas. Estas pruebas son programas (métodos). Éstos prueban cada parte del proyecto y verifican si están trabajando correctamente.

Algunas veces no deseamos escribir pruebas, sobre todo al principio, pensamos que escribirlas es redundante, “¿para qué hacerlo si el código funciona bien?”. El problema es que no siempre es así. Aún escribiendo las clases más sencillas éstas son propensas a errores. Es verdad que en términos de escribir código es más el tiempo que se consume al escribir código y pruebas que sólo código, pero hacerlo tiene sus ventajas. No escribir pruebas es más rápido sólo cuándo se hace sin errores.

Unit Testing es una práctica muy importante sobre todo en el mundo empresarial, dónde los resultados esperados son bastante complejos. El dejar muchos errores en las aplicaciones finales conlleva desarrollos más lentos, lo que evidentemente no es nada favorable para las empresas.

1.5 Herramientas de Pruebas Unitarias

Actualmente existen diferentes herramientas de Unit Testing que permiten agilizar el proceso de pruebas unitarias. Uno de los objetivos de esta práctica es introducir este tipo de herramientas. En particular nos centramos en la herramienta de Unit Testing JUnit, un framework para agilizar las pruebas unitarias de código Java. Además de esta herramienta existen muchas otras para los más variados lenguajes tales como: PUnit para Python, NUnit para C#, etc.

1.5.1 ¿Porqué utilizar herramientas de pruebas unitarias?

- Las herramientas de “Unit Testing” nos ayudan a escribir código más rápidamente mientras incrementa su calidad.
- Son herramientas sencillas que permiten escribir pruebas sencillas, convirtiendo las pruebas de software en una tarea poco costosa.
- No es necesario aprender ningún lenguaje nuevo puesto que se pueden desarrollar utilizando el mismo lenguaje que utilizamos para desarrollar nuestros productos.
- Con estas herramientas, las pruebas comprueban automáticamente sus propios resultados y proporcionan *feedback* inmediato.

2. El framework JUnit

JUnit² es un framework de pruebas que sirve para agilizar las pruebas unitarias de código Java. JUnit utiliza anotaciones para identificar los métodos que especifican un test. Lo único necesario para utilizar las anotaciones y clases del framework JUnit es importar las librerías adicionales.

2.1 Test JUnit

Un *test JUnit* (o prueba JUnit) es un método escrito en Java por el desarrollador que ejecuta una pequeña pieza del código a probar y verifica si ha hecho lo que se esperaba. Un método de prueba normalmente tiene el nombre del método que prueba con el prefijo *test*. Estos métodos están contenidos en clases usadas únicamente para las pruebas. A este tipo de clase se les llama *clases de prueba* (Test class). Las clases de prueba suelen tener el mismo nombre que la clase a probar con el prefijo *test*. La implementación de las clases de prueba es igual que la de cualquier clase Java.

Cuando escribimos un test JUnit dentro de una clase de prueba simplemente tenemos que anotar ese método con la etiqueta `@Test`. Esta etiqueta identifica al método como un método de prueba. Los métodos de prueba son utilizados durante la ejecución de las pruebas. Estos métodos se implementan y compilan de la misma forma que un método normal. Tienen que ser públicos, sin parámetros y devolver `void`.

El código del método de prueba debe:

- establecer todas las condiciones necesarias para hacer la prueba (para esto JUnit proporciona las *fixtures* (ver sección 2.1.2)),
- llamar al método a probar,
- verificar que el método a probar ha funcionado como esperábamos (para esto JUnit proporciona la clase `Assert` con métodos útiles (ver sección 2.1.1)),
- dejar el estado del sistema como estaba (para esto JUnit proporciona las *fixtures* (ver sección 2.1.2)).

Habitualmente, las pruebas se crean en un proyecto diferente al proyecto normal a testear para evitar que los códigos se mezclen.

Generalmente por cada clase a probar (se pueden probar todas las clases o solo aquellas que su código tenga un mínimo de complejidad) existirá una clase de test que prueba sus métodos públicos no triviales.

El siguiente código muestra una clase de prueba con un método de prueba. Obsérvese que el método de prueba lleva la anotación `@Test`:

```
import org.junit.Test;
```

² <http://junit.org>

```
public class MiClaseDePrueba{

    @Test
    public void miMetodoDePrueba() {
        ...
    }
}
```

JUnit crea una nueva instancia de la `test class` antes de invocar a cada `@Test method`.

2.1.1. Clase Assert

El framework JUnit ofrece la clase `Assert` con una serie de métodos útiles para escribir pruebas. Estos métodos permiten comprobar que los resultados que evalúa la prueba son correctos, es decir, nos permiten verificar si el método a probar ha funcionado como esperábamos.

Algunos de los métodos más importantes de esta clase son los siguientes:

- `void assertEquals(String expected, String actual)`, comprueba que dos primitivas sean iguales
- `void assertFalse(boolean condition)`, comprueba que una condición sea falsa
- `void assertNotNull(Object object)`, comprueba que un objeto no es nulo
- `void assertTrue(boolean condition)`, comprueba que una condición sea cierta
- `void fail()`, falla un test sin mensaje

A continuación se muestra un ejemplo de código de un método de prueba utilizando estos métodos.

Supongamos que se quiere probar el método `largest` de la clase `Largest`, que devuelve el entero mayor de una lista:

```
public class Largest {

    public static int largest (int lista[]){
        ...
    }
}
```

Creemos la clase de prueba `TestLargest` y el método de prueba `testLargest`:

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

public class LargestTest {

    int []list = new int[]{8,7,9};
```



```

@Test
public void testLargest() {
    assertEquals(9, Largest.largest(list));
}
}

```

2.1.2. Fixtures

JUnit proporciona lo que se conoce como “fixtures”, elementos fijos que se crean antes de ejecutar cada prueba y que sirven como punto de referencia para la ejecución de las pruebas.

Para inicializar esos elementos se utilizan métodos marcados con las siguientes anotaciones:

- `@Before`, el método se invoca antes de la ejecución de **cada** método de prueba de la clase
- `@After`, el método se invoca después de la ejecución de **cada** método de prueba de la clase

También se pueden definir métodos para que sean invocados antes o después de la ejecución de **todos** los métodos de prueba de una clase de prueba. Estos métodos vienen anotados con las etiquetas:

- `@BeforeClass`, el método se invoca antes de la ejecución de **todas** las pruebas
- `@AfterClass`, el método se invoca después de la ejecución de **todas** las pruebas

A continuación se muestra un ejemplo de pruebas que utilizan este tipo de métodos. Las pruebas se definen en la clase `TestDB`. Suponer que necesitamos algún tipo de objeto de conexión a base de datos para poder hacer nuestras pruebas (`testAccountAccess` y `testEmployeeAccess`). En lugar de incluir código para conectarnos y desconectarnos a la base de datos en cada método de prueba podríamos utilizar los métodos `@BeforeClass` y `@AfterClass` para que lo hicieran. También podríamos tener métodos anotados con `@Before` y `@After` para que crearan un objeto `cuenta` antes de cada método de prueba y lo destruyeran después de cada método de prueba de la clase `TestDB`.

```

public class TestDB {

    private Connection dbConn;
    private Account acc;

    @BeforeClass
    protected void setUpBeforeClass() {
        dbConn = new Connection("oracle",15,"fred", "f");
        dbConn.connect();
    }
}

```

```

    @AfterClass
    protected void tearDownAfterClass() {
        dbConn.disconnect();
        dbConn = null;
    }

    @Before
    protected void setUp() {
        acc = new Account();
    }

    @After
    protected void tearDown() {
        acc = null;
    }

    public void testAccountAccess() {
        ... // Uses dbConn
    }

    public void testEmployeeAccess() {
        ... // Uses dbConn
    }
}

```

Al ejecutar las pruebas el orden de ejecución de los métodos será el siguiente:

```

1. setUpBeforeClass
2.     setUp
3.         testAccountAccess
4.     tearDown
5.     setUp
6.         testEmployeeAccess
7.     tearDown
8. tearDownAfterClass

```

2.1.3. Mocks

Un mock es un objeto simulado que sustituye a un objeto real utilizado por la clase o fragmento de código a probar. Los objetos mock son los que se usarán durante la ejecución de la prueba unitaria, lo que posibilitará que no necesitemos el objeto real y que no dependamos de él para poder probar correctamente y de manera completa el módulo.

El concepto de mock es sencillo; si tanto el objeto real como el objeto falso implementan la misma interfaz y el módulo que estamos probando trabaja contra interfaces en lugar de contra objetos, podremos hacer uso indistintamente de objeto falso o del objeto real sin que esto afecte al módulo, siempre y cuando ambos objetos tengan el mismo comportamiento.

El uso de Mocks debería ser un recurso constante dentro de nuestras pruebas, pero la generación de mocks puede resultar un poco pesada, lo que provoca que no siempre se empleen. Existen varios frameworks, como RhinoMock o NMock, que facilitan la generación de estos objetos falsos, aunque cualquier objeto falso que creemos puede considerarse un mock, aún sin hacer uso de un framework.

Ejemplo con Proxy.

Veamos un ejemplo. Imaginemos que un método `calculaSalarioNeto` que para calcular los tramos de las retenciones actuales se conecta a una aplicación de la Agencia Tributaria a través de Internet. En ese caso el resultado que devolverá dependerá de la información almacenada en un servidor remoto que no controlamos. Es más, imaginemos que la especificación del método nos dice que nos devolverá `BRException` si no puede conectar al servidor para obtener la información. Deberíamos implementar dicho caso de prueba, pero en principio nos es imposible especificar como entrada en JUnit que se produzca un fallo en la red. Tampoco nos vale cortar la red manualmente, ya que nos interesa tener una batería de pruebas automatizadas.

La solución para este problema es utilizar objetos mock. Supongamos que el método `calculaSalarioNeto` está accediendo al servidor remoto mediante un objeto `ProxyAeat` que es quien se encarga de conectarse al servidor remoto y obtener la información necesaria de él. Podríamos crearnos un objeto `MockProxyAeat`, que se hiciese pasar por el objeto original, pero que nos permitiese establecer el resultado que queremos que nos devuelva, e incluso si queremos que produzca alguna excepción.

A continuación mostramos el código que tendría el método a probar dentro de la clase `EmpleadoBR`:

```
public float calculaSalarioNeto(float salarioBruto) {
    float retencion = 0.0f;

    if(salarioBruto < 0) {
        throw new BRException("El salario bruto debe ser positivo");
    }

    ProxyAeat proxy = getProxyAeat();
    List<TramoRetencion> tramos;
    try {
        tramos = proxy.getTramosRetencion();
    } catch (IOException e) {
        throw new BRException(
            "Error al conectar al servidor de la AEAT", e);
    }

    for(TramoRetencion tr: tramos) {
        if(salarioBruto < tr.getLimiteSalario()) {
```

```

        retencion = tr.getRetencion();
        break;
    }
}

return salarioBruto * (1 - retencion);
}

ProxyAeat getProxyAeat() {
    ProxyAeat proxy = new ProxyAeat();
    return proxy;
}

```

Ahora necesitamos crear un objeto `MockProxyAeat` que pueda hacerse pasar por el objeto original. Para ello haremos que `MockProxyAeat` herede de `ProxyAeat`, sobrescribiendo los métodos para los que queramos cambiar el comportamiento, y añadiendo los constructores y métodos auxiliares que necesitemos. Debido al polimorfismo, este nuevo objeto podrá utilizarse en todos los lugares en los que se utilizaba el objeto original:

```

public class MockProxyAeat extends ProxyAeat {

    boolean lanzarExcepcion;

    public MockProxyAeat(boolean lanzarExcepcion) {
        this.lanzarExcepcion = lanzarExcepcion;
    }

    @Override
    public List<TramoRetencion> getTramosRetencion()
                                                throws IOException {

        if(lanzarExcepcion) {
            throw new IOException("Error al conectar al servidor");
        }

        List<TramoRetencion> tramos = new ArrayList<TramoRetencion>();
        tramos.add(new TramoRetencion(1000.0f, 0.0f));
        tramos.add(new TramoRetencion(1500.0f, 0.16f));
        tramos.add(new TramoRetencion(Float.POSITIVE_INFINITY, 0.18f));

        return tramos;
    }
}

```

Ahora debemos conseguir que dentro del método a probar se utilice el objeto mock en lugar del auténtico, pero deberíamos hacerlo sin modificar ni el método ni la clase a probar. Podremos hacer esto de forma sencilla si hemos utilizado métodos para crear y acceder a los objetos proxy. Podemos crear una subclase de `EmpleadoBR` en la que se sobrescriba el método que se encarga de obtener el objeto `ProxyAeat`, para que en su lugar nos instancie el mock:

```

class TestableEmpleadoBR extends EmpleadoBR {

    ProxyAeat proxy;

    public void setProxyAeat(ProxyAeat proxy) {
        this.proxy = proxy;
    }

    @Override
    ProxyAeat getProxyAeat() {
        return proxy;
    }

}

```

Si nuestra clase a probar no tuviese un método de `getProxyAeat` para crear el objeto `ProxyAeat`, siempre podríamos refactorizarla para extraer la creación del componente que queramos sustituir a un método independiente y así permitir introducir el mock de forma limpia.

El código de JUnit para probar nuestro método podría quedar como se muestra a continuación:

```

TestableEmpleadoBR ebr;
TestableEmpleadoBR ebrFail;

@Before
public void setUpClass() {
    ebr = new TestableEmpleadoBR();
    ebr.setProxyAeat(new MockProxyAeat(false));

    ebrFail = new TestableEmpleadoBR();
    ebrFail.setProxyAeat(new MockProxyAeat(true));
}

@Test
public void testCalculaSalarioNeto1() {
    float resultadoReal = ebr.calculaSalarioNeto(2000.0f);
    float resultadoEsperado = 1640.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
}

@Test(expected=BRException.class)
public void testCalculaSalarioNeto10() {
    ebrFail.calculaSalarioNeto(1000.0f);
}

```

Podremos utilizar mocks para cualquier otro tipo de componente del que dependa nuestro método. Por ejemplo, si en nuestro método se utiliza un generador de números aleatorios, y el comportamiento varía según el número obtenido, podríamos sustituir dicho generador por un mock, para así poder predecir en nuestro código el resultado que dará.

Ejemplo con Bases de Datos.

Otro ejemplo interesante en el que los objetos mock pueden resultar de ayuda son las pruebas de métodos que dependen de los datos almacenados en una base de datos.

Normalmente en nuestra aplicación tendremos una serie de objetos que se encargan del acceso a datos (Data Access Objects, DAO). El resto de componentes de la aplicación, como pueden ser nuestros componentes de negocio, utilizarán los DAO para acceder a los datos. Por lo tanto, dichos componentes de negocio podrían ser probados sustituyendo los DAO por objetos mock en los que podamos establecer de forma sencilla el estado de la base de datos que queremos simular en las pruebas.

Supongamos que tenemos una clase `EmpleadoBO` que contiene un método `getSalarioBruto(int idEmpleado)`. Dicho método toma como entrada el identificador de un empleado en la base de datos, y nos devolverá su salario bruto. En caso de que el empleado no exista en la base de datos, lanzará una excepción de tipo `BOException`. Lo mismo ocurrirá si no puede acceder a la base de datos. La implementación será como se muestra a continuación:

```
public class EmpleadoBO {
    public float getSalarioBruto(int idEmpleado) throws BOException {

        IEmpleadoDAO edao = getEmpleadoDAO();
        EmpleadoBR ebr = new EmpleadoBR();

        try {
            EmpleadoTO empleado = edao.getEmpleado(idEmpleado);
            if(empleado==null) {
                throw new BOException("El usuario no existe");
            }

            return ebr.calculaSalarioBruto(empleado.getTipo(),
                                           empleado.getVentasMes(), empleado.getHorasExtra());
        } catch (DAOException e) {
            throw new BOException("Error al obtener el salario bruto",
                                   e);
        } catch (BRException e) {
            throw new BOException("Error al calcular el salario bruto",
                                   e);
        }
    }

    IEmpleadoDAO getEmpleadoDAO() {
        IEmpleadoDAO edao = new JDBCEmpleadoDAO();
        return edao;
    }
}
```

En este caso será sencillo crear un mock de nuestro DAO de empleados y sustituir el DAO original por el impostor. Nos será de ayuda el haber definido una interfaz para el

DAO (IEmpleadoDAO), ya que de esta forma implementaremos el mock como otra versión del DAO que implemente la misma interfaz que la original:

```
public class MockEmpleadoDAO implements IEmpleadoDAO {

    List<EmpleadoTO> listaEmpleados;
    boolean falloConexion;

    public MockEmpleadoDAO(boolean falloConexion) {
        listaEmpleados = new ArrayList<EmpleadoTO>();
        this.falloConexion = falloConexion;
    }

    private void compruebaConexion() throws DAOException {
        if(falloConexion) {
            throw new DAOException("Fallo al conectar a la BD");
        }
    }

    public void addEmpleado(EmpleadoTO empleado)
                                                throws DAOException {
        this.compruebaConexion();
        if(this.getEmpleado(empleado.getId()) == null) {
            listaEmpleados.add(empleado);
        } else {
            throw new DAOException("El empleado la existe en la BD");
        }
    }

    public void delEmpleado(int idEmpleado) throws DAOException {
        this.compruebaConexion();
        EmpleadoTO empleado = this.getEmpleado(idEmpleado);
        if(empleado != null) {
            listaEmpleados.remove(empleado);
        } else {
            throw new DAOException("El empleado no existe en la BD");
        }
    }

    public EmpleadoTO getEmpleado(int idEmpleado)
                                                throws DAOException {
        this.compruebaConexion();
        for(EmpleadoTO empleado: listaEmpleados) {
            if(empleado.getId() == idEmpleado) {
                return empleado;
            }
        }

        return null;
    }

    public List<EmpleadoTO> getEmpleados() throws DAOException {
        this.compruebaConexion();
        return listaEmpleados;
    }
}
```

De esta forma podríamos implementar algunas de nuestras pruebas en JUnit como sigue a continuación:

```
public class EmpleadoBOTest {

    TestableEmpleadoBO ebo;

    @Before
    public void setUp() {
        EmpleadoDAO edao = new MockEmpleadoDAO(false);
        try {
            edao.addEmpleado(new EmpleadoTO(1, "12345678X",
                "Paco Garcia", TipoEmpleado.vendedor, 1250, 8));
            edao.addEmpleado(new EmpleadoTO(2, "65645342B",
                "Maria Gomez", TipoEmpleado.encargado, 1600, 2));
            edao.addEmpleado(new EmpleadoTO(3, "45452343F",
                "Manolo Gutierrez", TipoEmpleado.vendedor, 800, 0));
        } catch (DAOException e) {
            e.printStackTrace();
        }

        ebo = new TestableEmpleadoBO();
        ebo.setEmpleadoDAO(edao);
    }

    @Test
    public void testGetSalarioBruto() {
        try {
            float resultadoReal = ebo.getSalarioBruto(3);
            float resultadoEsperado = 1000.0f;

            assertEquals(resultadoEsperado, resultadoReal, 0.01);
        } catch (BOException e) {
            fail("Lanzada excepcion no esperada BOException");
        }
    }
}

class TestableEmpleadoBO extends EmpleadoBO {

    EmpleadoDAO edao;

    public void setEmpleadoDAO(EmpleadoDAO edao) {
        this.edao = edao;
    }

    @Override
    EmpleadoDAO getEmpleadoDAO() {
        return edao;
    }
}
```

Destacamos aquí que los fixtures han sido de gran ayuda para crear los objetos mock necesarios.

Sin embargo, si lo que nos interesa es probar nuestro DAO, ya no tiene sentido utilizar mocks, ya que estos estarían sustituyendo a las propias clases a probar. En estos casos será necesario probar la base de datos real. Para ello tenemos herramientas especializadas como DBUnit, que nos permiten establecer de forma sencilla el estado en el que queremos dejar la base de datos antes de cada prueba.

Otra opción sería utilizar los propios fixtures de JUnit para establecer el estado de la base de datos antes de cada prueba. Podríamos utilizar el método marcado con `@Before` para vaciar la tabla que estemos probando e insertar en ella los datos de prueba.

2.1.4. Test Suite

JUnit ofrece la posibilidad de definir un **Test suite**. Un **test suite** es una combinación de clases de prueba que permite ejecutar todas las pruebas contenidas en dichas clases a la vez en un orden establecido.

Para crear una **Test suit** se debe:

- Crear una clase Java
- Anotar la clase con la etiqueta `@RunWith(Suite.class)`
- Indicar las clases que forman parte de la suite con la etiqueta `@SuiteClasses`

A continuación se muestra un ejemplo de código de la **test suite** `AllTests` que agrupa las pruebas de las clases `LargestTest` y `MiClaseDePrueba`.

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses({ LargestTest.class, MiClaseDePrueba.class })
public class AllTests {

}
```

Cuando lancemos a ejecución la **test suit** `AllTests` se ejecutarán primero las pruebas de la clase `LargestTest`, y a continuación las pruebas de la clase `MiClaseDePrueba`.

2.1.5 Clases parametrizadas

Cuando nos encontramos con varios tests que solo difieren en los datos de entrada y en el resultado esperado, lo más lógico sería abstraer dichos tests en un único test

que pudiera ejecutarse utilizando un conjunto de datos diferente cada vez. JUnit permite hacer esto con los *tests parametrizados*.

Un test parametrizado se consigue parametrizando la clase de test. Para ello, simplemente basta con anotar la clase con la etiqueta: `@RunWith(Parameterized.class)`

Una clase parametrizada debe tener como mínimo:

1. Un método estático encargado de generar y devolver los datos de test.
2. Un único constructor que guarde los datos de test.
3. Un test.

Método generador de los datos de test.

Este método estará anotado con la etiqueta `@Parameters`, y debe devolver una colección de arrays. Cada array representa los datos que van a ser utilizados en una ejecución específica del test. El número de elementos de cada array debe coincidir con el número de parámetros del constructor de la clase (puesto que cada elemento del array se pasará al constructor cada vez que la clase sea instanciada).

Constructor de la clase.

El constructor debe almacenar cada conjunto de datos de test en los campos de la clase. Estos campos se accederán en los métodos de test. Solo es posible crear un único constructor. Esto significa que los arrays generados deben tener siempre el mismo tamaño.

Funcionamiento.

Cuando se invoca el test el método encargado de generar los datos de test se ejecuta y devuelve la colección de arrays, donde cada array es un conjunto de datos de test. A continuación se instancia la clase y se le pasa al constructor de ésta el **primer** conjunto de datos de test. El constructor almacena los datos en sus campos. Luego, se ejecutan los métodos de prueba, accediendo cada método al primer conjunto de datos de prueba. Una vez se han ejecutado todos los métodos de test, la clase se vuelve a instanciar, esta vez usando el **segundo** elemento de la colección de arrays. Y así hasta utilizar todos los elementos de la colección.

Ejemplo de test parametrizado.

En este ejemplo los datos de test consisten en un único string de entrada y un string como resultado esperado, pero se pueden utilizar tantos datos como se necesiten.

```
//Imports
...

@RunWith(Parameterized.class)
public class ParameterizedTestExample {

    // Fields
    private String datum;
    private String expectedResult;
```

```

//Constructor. Por cada elemento que haya en la colección
devuelta por el método etiquetado con @Parameters, se
instanciará esta clase.
public ParameterizedTestExample(String datum, String expected)
{
    this.datum = datum;
    this.expectedResult = expected;
}

//Método Generador de Datos. Devuelve una colección de arrays,
cada elementos del array se corresponderá con un parámetro del
constructor.
@Parameters
public static Collection<Object[]> generateData()
{
    return Arrays.asList(new Object[][] {
        //Datos de test generados. En este caso un conjunto
        de tres pares, en el que cada par es el dato de
        entrada y el resultado esperado.
        { "AGCCG", "AGTTA" },
        { "AGTTA", "GATCA" },
        { "GGGAT", "AGCCA" }
    })
}

//Test. Se ejecutará una vez por cada elemento de la colección
devuelta por el método generador de datos de test (es decir,
cada vez que la clase es instanciada).
@Test
public void testWhatever()
{
    Whatever w = new Whatever();
    String actualResult = w.doWhatever(this.datum);
    assertThat(actualResult, is(this.expectedResult));
}
}

```

2.2 Guía para escribir pruebas

A continuación se presenta una guía con los pasos recomendados para realizar pruebas unitarias utilizando el framework JUnit.

- Elegir qué métodos queremos probar. Se debe partir de la idea de que se deberían probar todos los métodos, es decir, el 100% del código debería testearse. En aquellos casos que se necesite reducir el esfuerzo en el diseño de las pruebas, algunos candidatos a quedarse fuera de las pruebas son aquellos métodos con código muy simple (como podrían ser setters y getters que no realizan ninguna comprobación adicional).
- Para cada método a probar decidir qué verificaciones queremos hacer sobre el método. En este punto tenemos que definir los casos de prueba. Cada caso de prueba tendrá: un contexto sobre el que se prueba el método, unos valores de entrada para el método y un resultado esperado.

- Construir una clase de prueba por cada clase en la que tenemos un método a probar. Las clases de prueba suelen definirse en un paquete diferente a los del código normal. De esta forma se consigue tener el código mejor organizado.
- Implementar los métodos de prueba en las clases de prueba. La implementación de los métodos se hará utilizando los casos de prueba. El método debe:
 - Establecer el estado inicial
 - Llamar al método de prueba
 - Verificar que el método hace lo que tiene que hacer
 - Dejar al sistema en el mismo estado
- Construir una **Test Suite** para organizar la ejecución de los métodos de prueba.
- Ejecutar la **Test Suite** y comprobar resultados.

2.3 JUnit y Eclipse

La versión actual (instalada en el lab) de Eclipse (Mars) incluye las librerías JUnit, por tanto no se requiere ninguna instalación adicional para hacer uso del framework JUnit.



Estas librerías permiten al diseñador crear automáticamente esquemas de clases de prueba para las clases Java. Para ello se debe utilizar la opción de menú `New->JUnit Test Case` del menú contextual que aparece al posicionar el ratón sobre clases Java. Al seleccionar esta opción el asistente nos da la opción de elegir los métodos `Before`, `After`, `BeforeClass` y `AfterClass` que queremos crear.

Se puede descargar JUnit desde <http://junit.org>

2.3.1 Ejecución de las pruebas en Eclipse

Para ejecutar las pruebas en Eclipse se debe seleccionar la opción `Run As->JUnit Test` en el menú contextual de las clases de Prueba.

JUnit para Eclipse ofrece una vista que muestra de forma visual si las pruebas han fallado o no. La Figura 1 muestra una captura de pantalla de la interfaz gráfica de ejecución de pruebas de JUnit.

La vista JUnit se divide en dos zonas. La zona superior muestra los resultados de las pruebas. El icono  indica una ejecución de pruebas exitosa. Mientras que el icono  indica una ejecución de pruebas con fallos. Al seleccionar un fallo, en la zona inferior se muestra la traza de ejecución del fallo en forma de árbol. Haciendo doble clic sobre la traza del fallo se accede al código correspondiente que ha provocado el fallo.

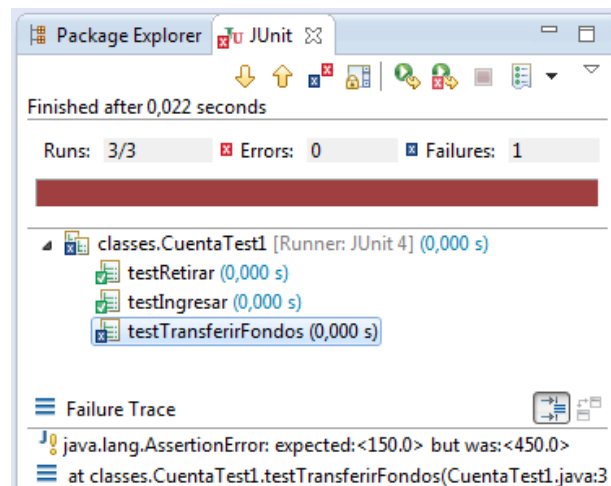


Figura 1. Vista JUnit

2.4 Un Ejemplo Sencillo

Supongamos que tenemos una clase `Lista` como la siguiente:

```
public class Lista extends Vector {
    public Lista() {...}
    public Lista(String[] elementos) {...}
    public Lista ordenar() {...}
    protected void ordenar(int iz, int dr) {...}
}
```

Se quiere verificar el método `ordenar`. Un posible caso de prueba sería:

```
String[] e3 = {"e", "d", "c", "b", "a"};
Lista reves = new Lista(e3);
Lista ordenada = reves.ordenar();
```

Y el resultado esperado:

"a","b","c","d","e"

Un posible método de prueba con JUnit 4.0 sería el siguiente:

```
public class ListaTest {
    @Test
    public void testOrdenarReves() {
        String[] ex = {"a", "b", "c", "d", "e"};
        Lista expected = new Lista(ex);
        String[] e3 = {"e", "d", "c", "b", "a"};
        Lista AlReves = new Lista(e3);
        assertEquals(expected, AlReves.ordenar());
    }
}
```

Una vez diseñado el método de prueba, se lanza a ejecución la clase `ListaTest` y se comprueba de forma visual si se ha detectado algún error.

3. Ejercicio a Desarrollar

- En PoliformaT->Recursos->Practicas->Laboratorio1 tenéis disponible el código de una clase `Cuenta`. La clase cuenta debe cumplir la siguiente especificación:
 - No se pueden hacer ingresos en metálico superiores a 3000 €.
 - Al retirar dinero no se puede dejar una cuenta con un balance negativo superior a 1500€.
 - Cuando se crea una cuenta con un balance negativo superior a 1500€, la cuenta se crea directamente congelada.
 - De las cuentas congeladas no se puede retirar dinero (ni directamente ni como origen de una transferencia de fondos).
- Se os pide que diseñéis las pruebas que consideréis oportunas para verificar los métodos de la clase: constructor, `Ingresar`, `Retirar` y `TransferirFondos`.

A continuación se muestran los pasos que tenéis que seguir para la realización del ejercicio:

1. Crear un nuevo proyecto Java
2. Añadir al proyecto la clase `Cuenta.java` cuyo código está disponible en PoliformaT
3. Crear una clase de prueba `TestCuenta` para la clase `Cuenta`, a través de la opción `New-> JUnit Test Case` del menú contextual de la clase `Cuenta`. Se mostrará la ventana de la Figura 2. Seleccionar `New JUnit 4 test` y los métodos que se desean generar y pulsar `Next`. En la siguiente ventana (ver Figura 3) seleccionar los métodos `constructor`, `Ingresar`, `Retirar` y `TransferirFondos` de la clase `Cuenta` que se quieren verificar. Finalmente hacer clic en el botón `Finish` para crear la clase de prueba. Como resultado tendréis una clase de prueba con cuatro métodos de prueba (uno para verificar cada método de la clase `Cuenta`).

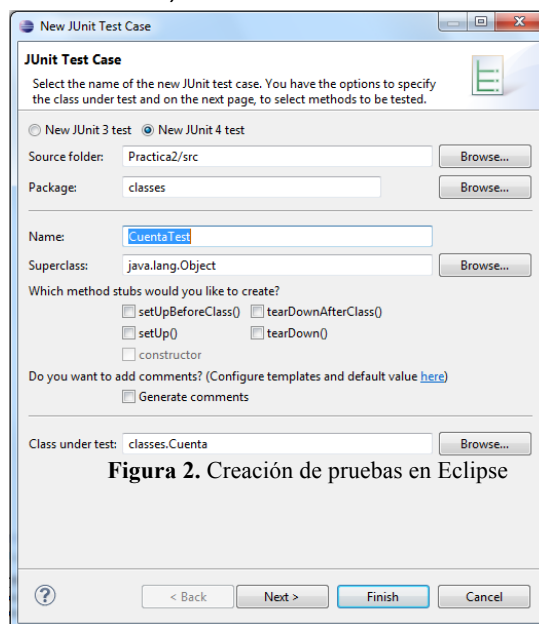


Figura 2. Creación de pruebas en Eclipse

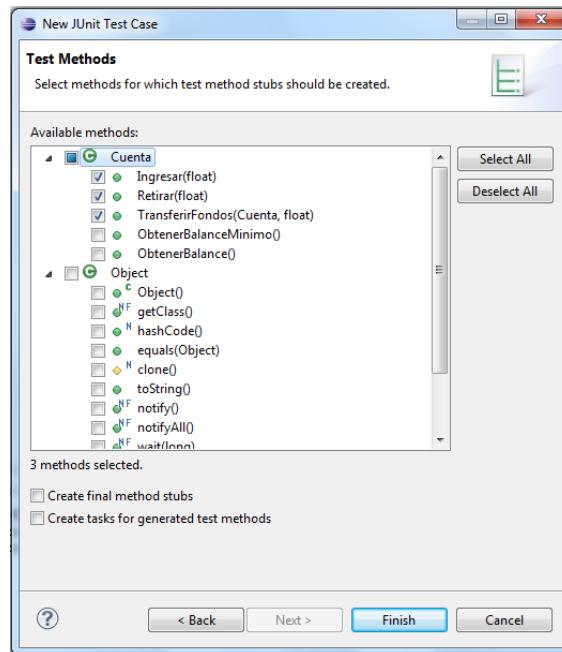




Figura 3. Selección de los métodos a probar

4. Diseñar los casos de prueba. En este paso debéis plantearos un conjunto de datos de prueba para cada uno de los métodos de prueba. Los datos de prueba deberán especificar: datos de entrada, resultado esperado, y el contexto sobre el que se hace la prueba (valores de la instancia Cuenta).
5. Completar el código de los métodos de prueba generados en la clase `TestCuenta` utilizando los casos de prueba diseñados en el paso anterior. Podéis crear más métodos de prueba además de los generados automáticamente.
6. Ejecutar los métodos de prueba. Para ejecutar las pruebas seleccionar la opción `Run as->JUnit Test` en el menú contextual de la clase `TestCuenta`.
7. Inspeccionar los resultados de las pruebas en la vista `JUnit` (ver Figura 4). La vista `JUnit` se muestra en la perspectiva actual cuando se ejecuta la prueba. Se divide en dos zonas. La zona superior muestra los resultados de las pruebas. El icono  indica una ejecución de pruebas exitosa. Mientras que el icono  indica una ejecución de pruebas con fallos. Al seleccionar un fallo, en la zona inferior se muestra la traza de ejecución del fallo en forma de árbol. Haciendo doble clic sobre la traza del fallo se accede al código correspondiente que ha provocado el fallo.

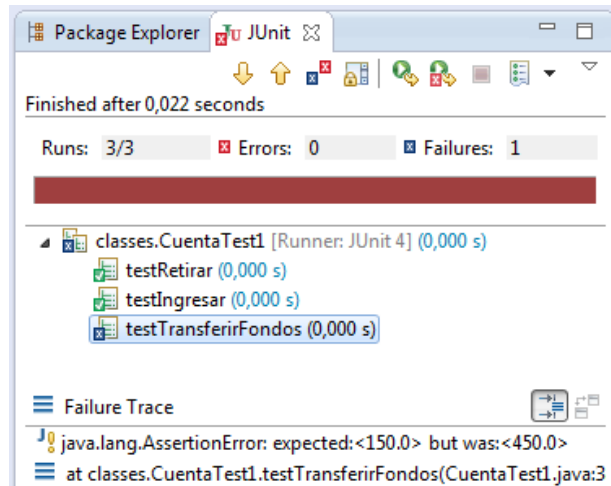


Figura 4. Vista Resultado Test *JUnit*

8. Corregir el código de los métodos de la clase `Cuenta` teniendo en cuenta los resultados de la prueba. Volver al paso 6 (o al paso 4 si pensáis que debéis replantear los casos de prueba) tantas veces como sea necesario.