TALENTO DIGITAL
INTELIGENCIA HUMANA

Talento Digital para Chile:

MÓDULO 1
PROGRAMACIÓN BÁSICA EN JAVA

UN PROYECTO DE:

DESARROLLADO POR:

MÓDULO 1 - PROGRAMACIÓN BÁSICA EN JAVA

1.1 PRUEBAS UNITARIAS EN JAVA

Semana 4 - Día 20

Objetivo de la jornada

- Conocer las características de las pruebas unitarias, ventajas y limitaciones.
- Conocer la herramienta JUnit para realizar pruebas unitarias en Java.
- Integrar JUnit en Eclipse para resolver pruebas a casos de ejemplo planteados.

Características de las pruebas unitarias, ventajas y limitaciones

Una Prueba Unitaria, es una forma de comprobar que el código realizado, hace lo que se supone debe hacer, es decir, se asegura que el código no presente fallos, errores, o cálculos inesperados, y que siempre arroja como resultado el valor correcto. Esto se realiza realizando pruebas a clases aisladas de su interacción con otras clases. Además de verificar que el código hace lo que tiene que hacer, también se debe verificar que sea correcto el nombre, los nombres y tipos de los parámetros, el tipo de lo que se devuelve, que si el estado inicial es válido, entonces el estado final es válido también.

Para que una prueba unitaria tenga la calidad suficiente se deben cumplir los siguientes requisitos:

- Automatizable: No debería requerirse una intervención manual. Esto es especialmente útil para integración continua.

- Completas: Deben cubrir la mayor cantidad de código.

- Repetibles o Reutilizables: No se deben crear pruebas que sólo puedan ser ejecutadas una sola vez.

- Independientes: La ejecución de una prueba no debe afectar a la ejecución de otra.

● Profesionales: Las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc. Es recomendable seguir estos requisitos para que las pruebas no pierdan parte de su función.

Ventajas Las pruebas unitarias tienen por objetivo aislar cada parte del programa y mostrar que las partes individuales son correctas. Estas pruebas aisladas proporcionan cinco ventajas básicas:

● Fomentan el cambio: Las pruebas unitarias facilitan que el programador cambie el código para mejorar su estructura, puesto que permiten hacer pruebas sobre los cambios y así asegurarse de que los nuevos cambios no han introducido defectos.

● Simplifica la integración: Puesto que permiten llegar a la fase de integración con un grado alto de seguridad de que el código está funcionando correctamente.
● Documenta el código: Las propias pruebas son documentación del código, puesto que ahí se puede ver cómo se utiliza.

● Separación de la interfaz y la implementación: Dado que la única interacción entre los casos de prueba y las unidades bajo prueba son las interfaces de estas últimas, se puede cambiar cualquiera de los dos sin afectar al otro, a veces usando objetos maquetados que habilitan de forma aislada el comportamiento de objetos complejos.

● Los errores están más acotados y son más fáciles de localizar: Dado que se tienen pruebas unitarias que pueden desenmascararlos.

Limitaciones

Lo primero que se debe tener en cuenta es que las pruebas unitarias no descubrirán todos los errores del código. Algunos enfoques se basan en la generación aleatoria de objetos para amplificar el alcance de las pruebas de unidad. Esta técnica se conoce como testing aleatorio. Por definición, sólo prueban las unidades por sí solas. Por lo tanto, no descubrirán errores de integración, problemas de rendimiento y otros problemas que afectan a todo el sistema en su conjunto. Además, puede no ser trivial anticipar todos los casos especiales de entradas que puede recibir en realidad la unidad de programa bajo estudio.

Las pruebas unitarias sólo son efectivas si se usan en conjunto con otras pruebas de software.

Introducción a JUnit

JUnit es una librería opensource desarrollada para poder probar el funcionamiento de las clases y métodos que componen una aplicación o Software, asegurando de que se comportan como deben ante distintas situaciones de entrada. El concepto fundamental de esta herramienta es el caso de prueba y la suite de prueba. Los casos de prueba son clases o módulos que disponen de métodos para probar los métodos de una clase o módulo concreta/o. Así, para cada clase que se quisiera probar se definiría u correspondiente clase de caso de prueba. Mediante las suites se pueden organizar los casos de prueba, de forma que cada suite agrupa los casos de prueba de módulos que están funcionalmente relacionados. De esta forma, se construyen programas que sirven para probar los módulos de un sistema, y que se pueden ejecutar de forma automática. A medida que la aplicación vaya avanzando, se dispondrá de un conjunto importante de casos de prueba. Es importante tener en cuenta que cuando se cambia un módulo que ya ha sido probado, el cambio puede haber afectado a otros módulos, y sería necesario volver a ejecutar las pruebas para verificar que todo sigue funcionando.

Integración de JUnit en Eclipse

Eclipse es una plataforma de software compuesto por un conjunto de herramientas de programación de código abierto multiplataforma para desarrollar aplicaciones con una interfaz gráfica, escrita con una sintaxis basada en XML, que proporciona funcionalidades similares a las del cliente pesado como: arrastrar y soltar, pestañas, ventanas múltiples, menús desplegables.

Eclipse incorpora opciones para poder trabajar con JUnit desde él. Primero se debe tener un proyecto Java ya creado, o bien crearlo nuevo. Una vez hecho esto, se debe añadir la librería de JUnit al build path del proyecto pulsando con el botón derecho sobre el proyecto y seleccionar **Java Build Path > Add Libraries**. Luego aparecerá una ventana en la que se podrá elegir la librería a añadir, siendo una de ellas JUnit.

Una vez añadida la librería, ya se pueden crear los casos de prueba de un proyecto. Si se crea un caso de prueba sin tener todavía añadida la librería JUnit, Eclipse preguntará si se desea añadir la librería.

Creando un caso de prueba

Utilizando JUnit para probar los métodos de un sistema. Para ello se deben crear una serie de clases en las que se implementarán las pruebas diseñadas. Esta implementación consistirá básicamente en invocar el método que está siendo probado pasando los parámetros de entrada establecidos para cada caso de prueba, y comprobar si la salida real coincide con la salida esperada. Esto en principio se podría hacer sin necesidad de utilizar JUnit, pero el utilizar esta herramienta será de gran utilidad ya que proporciona un framework que obliga a implementar las pruebas en un formato estándar que podrá ser reutilizable y entendible por cualquiera que conozca la librería. El aplicar este framework también ayudará a tener una batería de pruebas ordenada, que pueda ser ejecutada fácilmente y que muestre los resultados de forma clara mediante una interfaz gráfica que proporciona la herramienta.

Para implementar las pruebas en JUnit se utilizarán dos elementos básicos: Por un lado, se marcará con la anotación @Test los métodos que se quieran ejecutar con JUnit. Estos serán los métodos en los que se implementarán las pruebas. En estos métodos se llamará al método probado y se comprobará si el resultado obtenido es igual al esperado.

Para comprobar si el resultado obtenido coincide con el esperado se utilizarán los métodos assert de la librería JUnit. Estos son una serie de métodos estáticos de la clase Assert (para simplificar el código se puede hacer un import estático de dicha clase), todos ellos con el prefijo assert-. Existen multitud de variantes de estos métodos, según el tipo de datos que se estén comprobando (assertTrue, assertFalse, assertEquals, assertNull, etc). Las llamadas a estos métodos servirán para que JUnit sepa qué pruebas han tenido éxito y cuáles no. Cuando se ejecuten las pruebas con JUnit, se mostrará un informe con el número de pruebas exitosas y fallidas, y un detalle desglosado por casos de prueba. Para los casos de prueba que hayan fallado, se indicará además el valor que se ha obtenido y el que se esperaba.

Además de estos elementos básicos anteriores, a la hora de implementar las pruebas con JUnit deberemos seguir una serie de buenas prácticas que se detallan a continuación:

● La clase de pruebas se llamará igual que la clase a probar, pero con el sufijo -Test.

Por ejemplo, si se quiere probar la clase MiClase, la clase de pruebas se llamará MiClaseTest.

● La clase de pruebas se ubicará en el mismo paquete en el que estaba la clase probada.
Si MiClase está en el paquete PaquetePrueba, MiClaseTest pertenecerá en ese mismo paquete. De esta forma se asegura tener acceso a todos los miembros de tipo protegido y paquete de la clase a probar.

● Mezclar clases reales de la aplicación con clases que sólo servirán para realizar las pruebas durante el desarrollo no es nada recomendable, pero no se quiere renunciar a poner la clase de pruebas en el mismo paquete que la clase probada. Para solucionar este problema lo que se hará es crear las clases de prueba en un directorio de fuentes diferente.

Si los fuentes de la aplicación se encuentran normalmente en un directorio llamado src, los fuentes de pruebas irían en un directorio llamado test.

● Los métodos de prueba (los que están anotados con @Test), tendrán como nombre el mismo nombre que el del método probado, pero con prefijo test-. Por ejemplo, para probar miMetodo se tendría un método de prueba llamado testMiMetodo.

● Aunque dentro de un método de prueba se pueden poner tantos assert como se desee, es recomendable crear un método de prueba diferente por cada caso de prueba que se tenga.

Por ejemplo, si para miMetodo se han diseñado tres casos de prueba, se podrían tener tres métodos de prueba distintos: testMiMetodo1, testMiMetodo2, y testMiMetodo3. De esta forma, cuando se presenten los resultados de las pruebas se podrán ver exactamente qué caso de prueba es el que ha fallado.

En general la construcción de pruebas sigue siempre estos mismos patrones: llamar al método probado y comprobar si la salida real coincide con la esperada utilizando los métodos assert,

Entorno

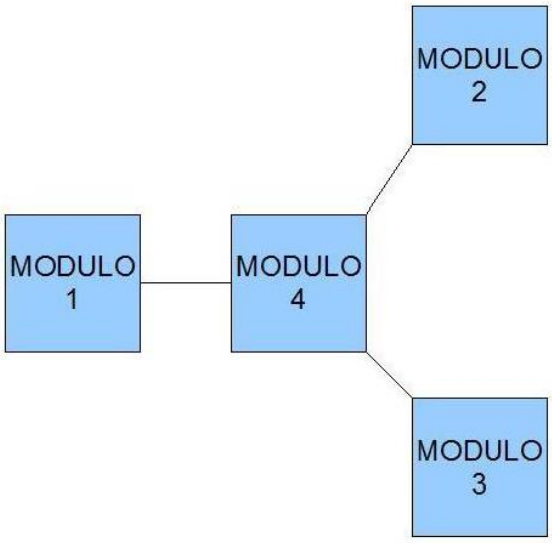
Las características del equipo y recursos utilizados para las pruebas y la elaboración de este tutorial son las siguientes:

- Equipo: Portátil «ASUS Notebook G1 Series» (Core 2 Duo T7500 2.20GHz, 2GB RAM, 100 GB HD).
- Sistema operativo: Windows Vista Ultimate.
- Eclipse Ganymede 3.4.1
- Junit 4.5

¿Por qué usar Test?

Vamos a explicar de manera sencilla que son conceptualmente los test y por qué son necesarios en nuestro proyecto Software.

Pongamos que nos encontramos en una situación típica de cualquier sistema en desarrollo con varios desarrolladores trabajando en paralelo, cada uno en un módulo independiente pero todos estos suelen estar relacionados.



En primer lugar tenemos que decir que existen principalmente 2 tipos de pruebas con las que vamos a trabajar, estas son:

- Pruebas unitarias: Consisten en probar la correcta funcionalidad del módulo en cuestión como si actuara independiente de los demás.

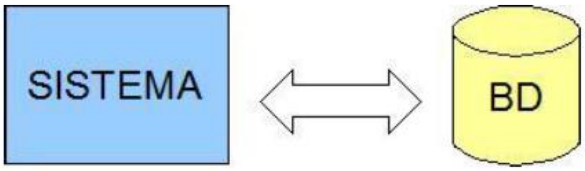
- Pruebas de integración: Como su nombre indica, se prueba la correcta integración de cada módulo (ya probado con pruebas unitarias) con los demás.

Supongamos que el sistema está implementado, probado y ya en funcionamiento, pero se desea añadir una nueva funcionalidad al módulo 4.

Tras el desarrollo de éste, ¿estamos seguros de que los demás módulos siguen funcionando correctamente y no se ha alterado su funcionalidad?

La manera más rudimentaria y primitiva es probar a mano todos los módulos relacionados. ¿No sería más facil automatizar este proceso?

Automatizando la tarea de realizar las pruebas conseguimos, además de ganar tiempo cada vez que tengamos que hacer las pruebas, asegurarnos que se realizan todas las pruebas necesarias. Mediante pruebas de integración no solo se nos permite probar la interacción de un módulo con otro dentro del mismo sistema, sino que podemos probar las conexiones con otros sistemas externos como por ejemplo un gestor de Base de datos.



Por otro lado, se ha demostrado que una forma muy eficaz, limpia y profesional de trabajar es realizar una programación basada en pruebas. Esto consiste en invertir el proceso natural de desarrollo, cogiendo los requisitos, realizando una batería de pruebas a partir de éstos para finalmente desarrollar el sistema que los cumpla. En resumen, un proyecto Software no puede considerarse como tal sin una buena batería de pruebas que comprueben su correcto funcionamiento frente a cambios

Características de JUnit 4

En primer lugar e importante es que el nombre de la clase de Test debe tener la siguiente estructura: «test»

Métodos

A grandes rasgos, una clase de Test realizada para ser tratada por JUnit 4 tiene una estructura con 4 tipos de métodos:

- Método setUp: Asignamos valores iniciales a variables antes de la ejecución de cada test. Si solo queremos que se inicialicen alprincipio una vez, el método se debe llamar «setUpClass»
- Método tearDown: Es llamado después de cada test y puede servir para liberar recursos o similar. Igual que antes, si queremos que solo se llame al final de la ejecución de todos los test, se debe llamar «tearDownClass»
- Métodos Test: Contienen las pruebas concretas que vamos a realizar.
- Métodos auxiliares.

Anotaciones

En versiones anteriores de Junit no existían caracteres especiales, que llamamos anotaciones, y que se han incluído en su versión 4 para intentar simplificar más la labor del programador. Se trata de palabras clave que se colocan delante de los definidos antes y que indican a las librerías JUnit

Instrucciones concretas.

A continuación pasamos a ver las más relevantes:

• @RunWith: Se le asigna una clase a la que JUnit invocará en lugar del ejecutor por defecto de JUnit

• @Before: Indicamos que el siguiente método se debe ejecutar antes de cada test (precede al método setUp). Si tiene que preceder al método setUpClass, la notación será «@BeforeClass»

• @After: Indicamos que el siguiente método se debe ejecutar después de cada test (precede al método tearDown). Si tiene que preceder al método tearDownClass, la notación será «@AfterClass»

• @Test: Indicamos a Junit que se trata de un método de Test.

En versiones anteriores de JUnit los métodos tenían que tener un nombre con la siguiente estructura: «test». Con esta notación colocada delante de los métodos podemos elegir el nombre libremente.

Funciones de aceptación/rechazo

Una vez hemos creado las condiciones para probar que una funcionalidad concreta funciona es necesario que un validador nos diga si estamos obteniendo el resultado esperado o no. Para esta labor se definen una lista de funciones (incluidas en la clase Assert) que se pueden ver detalladas en el javadoc de JUnit: Javadoc.

Pasamos a detallar las más comunes para así ahorraros trabajo:

• assertArrayEquals: Recibe como parámetro 2 arrays y comprueba si son iguales. Devuelve assertionError si no se produce el resultado esperado

• assertEquals: Realiza la comprobación entre 2 valores de tipo numérico. Devuelve assertionError si no se produce el resultado esperado.

• assertTrue: Comprueba si una condición se cumple. Devuelve assertionError si no se produce el resultado esperado

• fail: devuelve una alerta informando del fallo en el test

4.4. Ejemplo

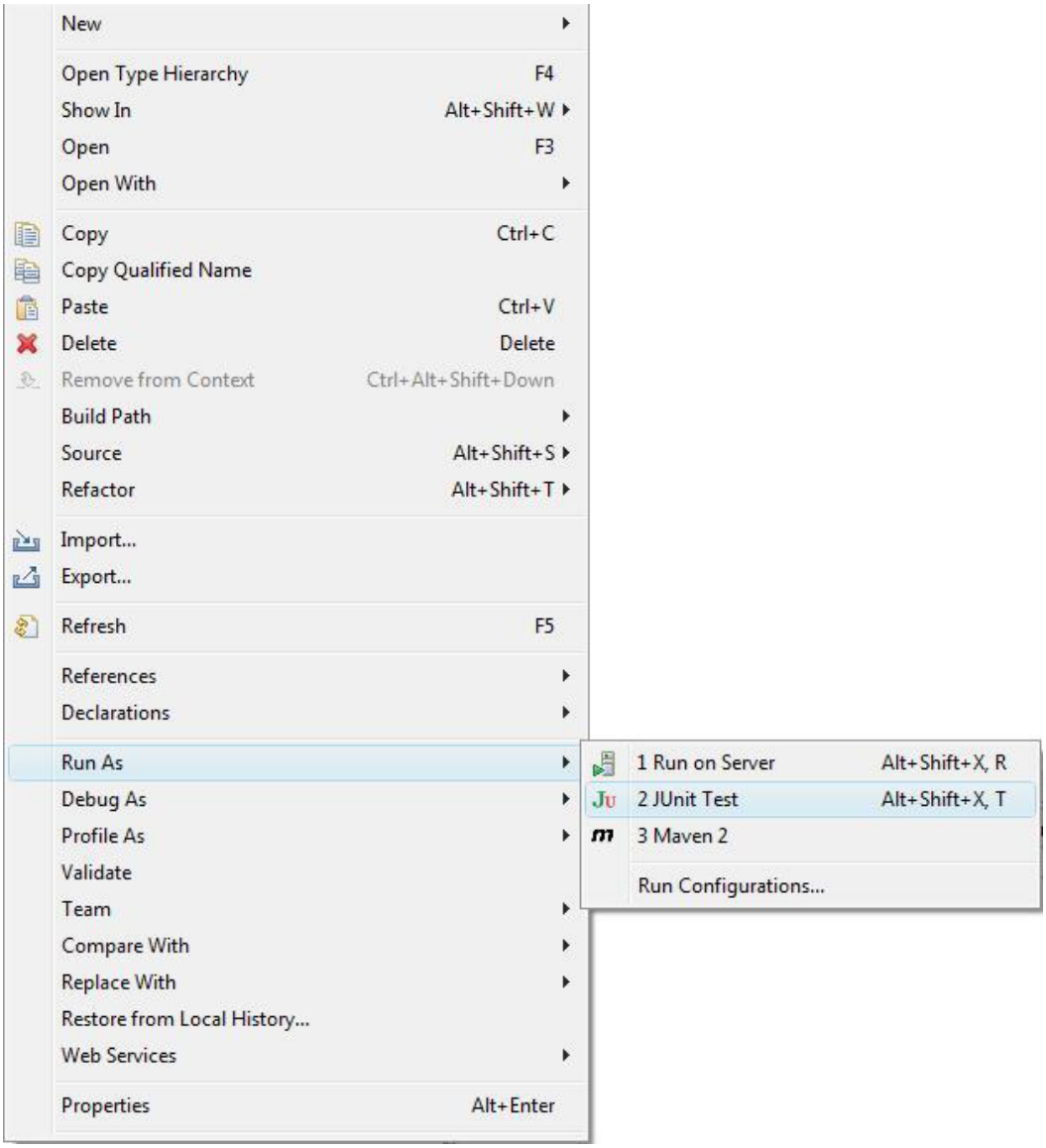
```
1  package com.autentia.training.tutorialJUnit;
2
3  import org.junit.AfterClass;
4  import org.junit.Assert;
5  import org.junit.BeforeClass;
6  import org.junit.Test;
7
8  public class pruebaTest {
9
```

```
10      @BeforeClass
11      public static void setUpClass() throws Exception {
12      //Inicialización general de variables, escritura del log...
13      }
14
15      @AfterClass
16      public static void tearDownClass() throws Exception {
17      //Liberación de recursos, escritura en el log...
18      }
19
20      @Before
21  public void setUp() {
22  //Inicialización de variables antes de cada Test
23  }
24
25      @After
26  public void tearDown() {
27  //Tareas a realizar después de cada test
28  }
29
30      @Test
31  public void comprobarAccion() {
32  //creamos el entorno necesario para la prueba
33      //Usamos alguna de las funciones arriba descritas
34      //para realizar la comprobación
35  }
36
37  public void funcionAuxiliar() {
38  //tareas auxiliares
39  }
40
41 }
```

JUnit en Eclipse

La versión actual de Eclipse (Ganymede, 3.4) ya incluye la integración de éste con las librerias JUnit por lo que es sencillísimo ejecutar una clase de Test.

Unicamente tenemos que abrirnos la vista de explorador de pproyectos, aquí hacer clic con el botón derecho en la clase Test en concreto, en el menú que aparece elegir «Run as» -> «JUnit Test»



En la consola nos aparecerá el resultado de la ejecución.