Herramientas de Testing Java.

Víctor Herrero Cazurro

Contenidos

1. Pruebas unitarias y de integración.
2. Pruebas funcionales.
3. Prueba de aceptación.
4. Pruebas de regresión.
5. Calidad del código.

Pruebas unitarias

- Una prueba unitaria es una forma de probar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.
- En las pruebas unitarias se prueban clases concretas, no conjuntos de clases, es decir una prueba unitaria prueba la funcionalidad de un método de una clase suponiendo que los objetos que emplea realizan sus tareas de forma correcta.

Pruebas de integración

- Las pruebas de integración son aquellas que se refieren a la prueba o pruebas de todos los elementos unitarios que componen un proceso, hecha en conjunto, de una sola vez.
- Es decir, se prueba una clase, empleando la implementación real del resto de clases que necesite para ejecutarse.

Junit

- Framework para pruebas unitarias
  - Pruebas de funcionamiento de una clase
- Paquetes junit.* (JUnit 3) y org.junit.* (JUnit 4)
- Disponible en Eclipse (JUnit 3 y 4)
  - Eclipse nos proporciona la creación de Junit Test Case (caso de prueba) y Junit Test Suite (conjunto de casos de pueba).
- JUnit 4 admite timeout, excepciones esperadas, tests ignorables, …

Junit

- Paquetes de pruebas (Suites)
  - Permite ejecutar pruebas de varias clases en conjunto desde una lanzadora
  - En JUnit 3 se añaden clases a TestSuite
    - **suite.addTest(ClaseTest.suite**());
    - **suite.addTestSuite(ClaseTest.class**);
  - En JUnit 4 se usan anotaciones de clase
    - @**RunWith(Suite.class**)
    - @**SuiteClasses**({C1Test.class, C2Test.class})

Junit 4

- Antes de Java 5
  - Los métodos debían llamarse **testXXX**()
  - Se admitían los métodos **setUp**() y **tearDown**() para inicializar y liberar objetos (pre-post condiciones) antes de cada Test.

Junit 4

- Después de Java 5
  - Se utilizan anotaciones @**Before**, @**Test**, @**After**, @**AfterClass** y @**BeforeClass**.
  - Aparecen @**AfterClass** y @**BeforeClass**, que son dos métodos que se ejecutarán una única vez antes de empezar las pruebas y justo después de terminarlas.
  - No importan los nombres de los métodos ni hay que heredar **TestCase**.

Junit 4

Anotación	Descripción
BeforeClass	Sólo puede haber un método con este marcador. Este método será invocado una vez al principio del lanzamiento de todas las pruebas. Se suele utilizar para inicializar los atributos comunes a todas las pruebas.
AfterClass	Sólo puede haber un método con este marcador. Este método será invocado un sola vez cuando finalicen todas las pruebas.
Before	Los métodos marcados con está anotación, serán invocados antes de iniciarse cada prueba (antes de que se ejecute cada método).
After	Los métodos marcados con está anotación, serán invocados después de ejecutarse cada prueba (después de que se ejecute cada método).
Test	Representa un test que se debe ejecutar. Puede tener dos tipos de modificadores: Test(timeout=X) que significa que el test será válido si se ejecuta en un tiempo máximo de X milisegundos. Test(expected = java.util.NoSuchElementException.class). Que significa que el test será válido si se lanza una determinada excepción.
Ignore	Los métodos marcados con esta anotación no serán ejecutados. Se suelen poner para desactivar las pruebas que no pueden ser realizadas por algún motivo.

Testeando excepciones

- En JUnit 4, se proporciona un parámetro **expected**, junto con la anotación **@Test**, que permite indicar que en la ejecución de un test, se espera, es decir, el resultado bueno, es que se lance una excepción.

```
@Test(expected=InvalidIngresoException.class)
public void comprobamosQueLanzamosException()
                                    throws InvalidIngresoException{
}
```

Restricciones Temporales

- También JUnit 4, proporciona otro parámetro **timeout**, junto con la anotación **@Test**, que permite indicar el tiempo máximo que debe tardar la ejecución de un test.

```
@Test(timeout=12000)
public void testDeRendimiento() {
}
```

Test paramétricos

- JUnit 4 proporciona un mecanismo, para poder ejecutar un Test Case, con juegos de datos distintos, sin tener que codificar la prueba varias veces.
- Para emplear este mecanismo hay que
  - Anotar la clase de Test con

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

Test paramétricos

- Método estático publico, con la siguiente firma, donde cada array de objetos, representa un juego de datos, y cada objeto del array es un dato de prueba, ya sea de entrada o esperado

```
@Parameters
public static Collection<Object[]> data() {}
```

- Atributos de clase de la misma tipología que los objetos de los array retornados por el anterior método.
- Constructor con tantos parámetros como atributos de clase.

Asertos

- Tanto en JUnit 3 como en 4, se proporciona una clase con métodos estáticos, que permiten realizar comprobaciones (asertos) del estado actual de los datos que participan en un test.
- En JUnit 3

```
junit.framework.Assert
```

- En JUnit 4

```
org.junit.Assert
```

Asertos

- Entre los métodos que se encuentran en la clase están
  - **assertEquals**
  - **assertFalse**
  - **assertTrue**
  - **assertNotNull**
  - **assertNull**
  - **assertNotSame**
  - **assertSame**
  - **fail**

Hamcrest

- Permite realizar los Test, con un lenguaje mas cercano, mas legible.
- La librería **hamcrest-library**, proporciona una clase con métodos estáticos.

```
org.hamcrest.Matchers
```

Hamcrest

- Estos métodos estáticos, se emplean en formar un predicado, que forma el aserto, para que la forma de leer el código sea mas natural.
- Este predicado se puede emplear tanto en
  - El método **assertThat** de **Junit**.

```
org.junit.Assert.assertThat
```

  - El método **assertThat** de **Hamcrest**.

```
org.hamcrest.MatcherAssert.assertThat
```

Hamcrest

- Algunos de los métodos estáticos de Hamcrest.
  - **is**
  - **not**
  - **nullValue**
  - **empty**
  - **endsWith**
  - **startsWith**
  - **hasItem**
  - **hasItems**
  - **hasProperty**

Hamcrest

- Un ejemplo de aserto con Hamcrest

```
assertThat(calculadora, is(not(nullValue())));
```

Junit 4: Ejercicio

- Implementar una clase que cumpla con los siguientes requisitos
  - Obtener los impuestos según los ingresos siguiendo las siguientes reglas:
    - Ingreso <= 8000 no paga impuestos.
    - 8000 < Ingreso <= 15000 paga 8% de impuestos.
    - 15000 < Ingreso <= 20000 paga 10% de impuestos.
    - 20000 < Ingreso <= 25000 paga 15% de impuestos.
    - 25000 < Ingreso paga 19.5% de impuestos.
  - Creamos una clase con un método calcularImpuestosPorIngresos, que recibiendo una cantidad double como parámetro, que son los ingresos de una persona, retornará los impuestos que ha de pagar dicha persona (double).

DbUnit

- Herramienta para simplificar las pruebas unitarias de operaciones sobre base de datos.
- El framework DbUnit extiende JUnit 3.

http://www.dbunit.org/

- Nos permitirá realizar nuestras pruebas sobre un estado de la base de datos conocido.
- Podremos cargar los datos de test de un XML que tengamos generado y del cual conozcamos el estado de los datos.

DbUnit

- Nos permitirá comparar los datos que hay en la base de datos, con un XML con los datos esperados.
- En lugar de extender **TestCase** se extiende **DatabaseTestCase**, que es una clase abstracta.
- Requiere implementar dos métodos
  - protected IDatabaseConnection getConnection() throws Exception.
  - protected IDataSet getDataSet() throws Exception.

DbUnit

- En **getConnection**(), habrá que especificar la conexión con la base de datos a partir de un **java.sql.Connection**.
- En **getDataSet**(), habrá que especificar el origen de los datos que se van a emplear como conjunto de datos conocidos de partida, existen varios tipos, pero el mas habitual será el **FlatXmlDataSet**.

DbUnit

- Un ejemplo de implementación de estos métodos seria

```
private IDataSet loadedDataSet;

protected IDatabaseConnection getConnection() throws Exception {
        Class.forName("com.mysql.jdbc.Driver");
        Connection jdbcConnection =  DriverManager.getConnection(
                "jdbc:mysql://localhost/test", "root", "root");
        return new DatabaseConnection(jdbcConnection);
}

protected IDataSet getDataSet() throws Exception {
        loadedDataSet = new FlatXmlDataSet(
                                        new InputSource("db/input.xml"));
        return loadedDataSet;
}
```

DbUnit

- Existen otro dos métodos que quizás sea interesante sobreescribir, aunque ya tienen una implementación.
  - getSetUpOperation
  - getTearDownOperation
- La implementación de estos métodos es la siguiente.

```
protected DatabaseOperation getSetUpOperation() throws Exception {
        return DatabaseOperation.CLEAN_INSERT;
}
protected DatabaseOperation getTearDownOperation() throws Exception {
        return DatabaseOperation.NONE;
}
```

DbUnit

- Para lo que sirven, es para preparar la base de datos antes de empezar las pruebas, y para dejarla en un estado al finalizar.
- Lo que hacen es realizar una acción sobre la base de datos, con el conjunto de datos que representa el **DataSet**.

DbUnit

- En el caso de la implementación por defecto, lo que se hace es borrar las tablas que aparecen en el **DataSet** e insertar los registros que aparecen, antes de iniciar las pruebas, y no se realiza nada al acabar, pero hay otras acciones a llevar a cabo, como se muestra en la siguiente tabla.

DbUnit

DatabaseOperation.UPDATE	Se actualiza la base de datos con el contenido del **DataSet**. Esta operación supone que los datos ya existen ya que realizará UPDATE y falla si no están
DatabaseOperation.INSERT	Se inserta el contenido del **DataSet** en la base de datos. Habrá que tener cuidado con las claves.
DatabaseOperation.DELETE	Se elimina el contenido del **DataSet** de la base de datos.
DatabaseOperation.DELETE_ALL	Elimina todos los registros de las tablas que estén en el **DataSet**. Si una tabla no figura en el **DataSet** no se borra.
DatabaseOperation.TRUNCATE	Trunca las tablas que estén en el **DataSet**. Si una tabla no figura en el **DataSet** no se borra.
DatabaseOperation.REFRESH	Se actualizan los registros que existan y se insertan los que no existan de los que figuran en el **DataSet.** Los que no figuran en el **DataSet** permanecen inalterados.
DatabaseOperation.CLEAN_INSERT	Se realiza un DELETE_ALL y un INSERT de los datos del **DataSet.**
DatabaseOperation.NONE	No se realiza ninguna operación.

DbUnit

- Esto se produce por que **DatabaseTestCase** tiene esta implementación de **setUp**() y **tearDown**().

```
protected void setUp() throws Exception {
        super.setUp();
        executeOperation(getSetUpOperation());
}
protected void tearDown() throws Exception {
        super.tearDown();
        executeOperation(getTearDownOperation());
}
```

- Para cada Test, se cargaran los datos necesarios, lo cual nos asegura que el estado de la Base de datos es el deseado.

DbUnit

- Realización de pruebas
  - **org.dbunit.Assertion** permite establecer aserciones entre **DataSets**.
- La librería de **DbUnit**, tiene una serie de herramientas, que permiten generar los **DataSet** que se emplearán en las pruebas.
- Por ejemplo, para crear un XML, se tiene la clase **FlatXmlWriter**.

DbUnit

- Ejemplo de obtención de un XML a partir de una Base de datos.

```
Class.forName(driverName);
conn = DriverManager.getConnection(urlDB, userDB, passwordDB);
IDatabaseConnection connection = new DatabaseConnection(conn, schemaBD);

QueryDataSet partialDataSet = new QueryDataSet(connection);

// Especificar que tablas formaran parte del Dataset
partialDataSet.addTable("LIBROS");

// Especificar la ubicación del fichero a generar
FlatXmlWriter datasetWriter = new FlatXmlWriter(
                              new FileOutputStream("db/" + nameXML + ".xml"));

// Generar el fichero
datasetWriter.write(partialDataSet);
```

DbUnit. Ejemplo

- Partiendo de una pequeña aplicación que es capaz de insertar, modificar, borrar y consultar datos de una BD MySQL, habrá que crear un test con DBUnit, que cargando una BD controlada, compruebe que realizando una serie de operaciones sobre la BD, la BD resultante, es como una BD esperada.

HttpUnit

- Framework para probar sitios Web basado en JUnit4.
- Realmente lo que nos proporciona **HTTPUnit**, es la capacidad de interactuar programáticamente con una aplicación web de forma amigable, y por tanto seguiremos empleando **Junit** para los Test en si.

http://www.httpunit.org/

HttpUnit

- Tendremos la clase **com.meterware.httpunit.WebConversation** que simulara un navegador accediendo al servidor.
- **WebResponse** es el objeto retornado por la petición.
- La respuesta será HTML
  - **getText**() la devuelve como texto
  - **getDOM**() la devuelve como objeto XML DOM

HttpUnit

- Podemos obtener los enlaces y interactuar con ellos.

```
WebLink link = response.getLinkWith("texto");
link.click();
```

- También existe **getLinkWithImageText**() que busca en el texto ALT de una imagen.
- Tablas

```
WebTable table = resp.getTables()[0];
Método asText() la devuelve como String[][]
```

- Marcos
  - getFrameContents("marco") devuelve un WebResponse

HttpUnit

- Formularios
  - WebForm form = resp.getForms()[0];
  - form.getParameterValue("parámetro") ;
  - getParameterNames() devuelve un String[]
  - setParameter("parámetro", "valor");
  - toggleCheckbox("parámetro"); para checkbox
  - submit(); envía los datos
  - reset(); resetea a los valores por defecto

HttpUnit. Ejercicio

- Partiendo de una aplicación web con un formulario sencillo, con un campo nombre y otro mail, que al dar a submit del formulario, se envían dichos datos a otra pagina donde se pintan en una tabla.
- Establecer una conversación con dicha aplicación, siguiendo los siguientes pasos
  - Acceder a la pagina de inicio que contiene el formulario.
  - Comprobar que el código de respuesta es 200 y el mensaje OK.
  - Comprobar que existe al menos un formulario y posteriormente que es el único.
  - Comprobar que tiene los campos ”nombre” y “mail”.
  - Rellenar dichos campos y enviar el formulario.
  - Comprobar que la respuesta del formulario, es una pagina con un titulo “Datos enviados”.
  - Comprobar que hay algún tag DIV.
  - Comprobar que únicamente hay dos DIV, que tienen como name “nombre” y “mail”, y que su contenido es el enviado por el formulario.
  - Comprobar que tenemos una única tabla.
  - Comprobar que el contenido de la segunda columna, son los valores enviados por el formulario.
    - [0][1] -> Nombre
    - [1][1] -> Mail
  - Comprobar que también existe algún link, y que existe uno con el texto inicio.
  - Comprobar que si realizamos click sobre el link, volvemos a la pagina inicial con el formulario.

HttpUnit. Ejercicio

- WebConversation webConversation = new WebConversation();
- WebResponse loginResponse = webConversation.getResponse("http://localhost:8080/Servidor");
- String nombre = “nombre";
- String mail = “correo@electronico.es";
- loginResponse.getForms()[0].setParameter("nombre",nombre);
- loginResponse.getForms()[0].setParameter("mail", mail);
- WebResponse homePage = loginResponse.getForms()[0].submit();
- assertEquals("Datos enviados", homePage.getTitle());
- HTMLElement[] list = homePage.getElementsByTagName(“div");
- for (HTMLElement htmlElement : list) {
- if (htmlElement.getName().equals("nombre")){
- assertEquals(nombre,htmlElement.getText());
- }
- if (htmlElement.getName().equals("mail")){
- assertEquals(mail,htmlElement.getText());
- }
- }

Mocks con mockito

- Para poder crear un buen conjunto de pruebas unitarias, es necesario centrarse exclusivamente en la clase a testear
- Para ello se pueden simular el resto de clases involucradas.
- De esta manera se crean test unitarios potentes que permiten detectar los errores allí donde se producen.
- Para simular las clases involucradas, se emplea el concepto de mock objects.

Mocks con mockito

- Mockito es una herramienta que permite generar mock objects dinámicos. Estos pueden ser de clases concretas o de interfaces.
- Esta parte de la generación de las pruebas, no se centra en la validación de los resultados, sino en las interacciones de la clase a probar con otras.

Mocks con mockito

- La creación de pruebas con Mockito se divide en tres fases
  - **Stubbing**: Definición del comportamiento de los Mock ante unos datos concretos.
  - **Invocación**: Utilización de los Mock, al interaccionar la clase que se esta probando con ellos.
  - **Validación**: Validación del uso de los Mock.

Mocks con mockito

- Se pueden definir los Mock, con anotaciones o con el método mock. Las anotaciones siempre dejan el código mas limpio.

```
@Mock
private IUserDAO mockUserDao;

private IDataSesionUserDAO mockDataSesionUserDao =
                                                mock(IDataSesionUserDAO.class);
```

- De emplearse las anotaciones, o bien se invoca la siguiente línea

```
MockitoAnnotations.initMocks(testClass);
```

Mocks con mockito

- O bien, se emplea como runner, la clase **MockitoJUnitRunner**,

```
@RunWith(MockitoJUnitRunner.class)
```

- Stubbing

```
when(mockUserDao.getUser(validUser.getId())).thenReturn(validUser);

when(mockUserDao.getUser(invalidUser.getId())).thenReturn(null);
```

- Por defecto todos los métodos que devuelven valores de un mock devuelven null, una colección vacía o el tipo de dato primitivo apropiado.

Mocks con mockito

- Verificación

```
Mockito.verify(mockUserDao).getUser(validUser.getId());
Mockito.verify(mockDataSesionUserDao).deleteDataSesion(validUser, validId);
ordered.verify(mockUserDao).getUser(validUser.getId());
ordered.verify(mockDataSesionUserDao).deleteDataSesion(validUser, validId);
```

- Una vez creado, el mock recuerda todas las interacciones. Se puede elegir indiferentemente que interacción verificar.
- También se puede verificar el orden en el que se han realizado las verificaciones, para lo cual previamente hay que definir un objeto InOrder.
-

```
ordered = inOrder(mockUserDao, mockDataSesionUserDao);
```

Mocks con mockito

- Argument matchers permiten definir el comportamiento de los Mock, empleando 'comodines', de forma que los parámetros a los mismos no se tengan que definir explícitamente.

```
when(mockDataSesionUserDao.deleteDataSesion((User) eq(null), anyString()))
        .thenThrow(new OperationNotSupportedException());

when(mockDataSesionUserDao.updateDataSesion(eq(validUser), eq(validId),
        anyObject())).thenReturn(true);

when(mockDataSesionUserDao.updateDataSesion(eq(validUser), eq(invalidId),
        anyObject())).thenThrow(new OperationNotSupportedException());

when(mockDataSesionUserDao.updateDataSesion((User) eq(null), anyString(),
        anyObject())).thenThrow(new OperationNotSupportedException());
```

Cobertura de pruebas

- La cobertura de pruebas será la medida que nos dice la cantidad de código que estamos probando con nuestros Test.
- Es decir, cuantos de los caminos que tenemos en nuestro código, estamos recorriendo con nuestras pruebas.
- Para medirla, eclipse dispone de un plugin llamado EclEmma, que nos mostrará de forma grafica las partes de código cubiertas por nuestras pruebas y las que no lo están, mostrándonos un tanto porciento de cobertura.

Cobertura de pruebas: EclEmma

- EclEmma, es un plugin de eclipse, que nos permitirá visualizar la cobertura de los tests unitarios.
- Para instalarlo, lo hacemos como cualquier plugin de eclipse, partiendo de esta url.
  - http://update.eclemma.org/
  - Help -> Install New Software -> Available Software -> Add Site
- Este plugin lo que nos añade es un botón en la barra de herramientas de Eclipse, que nos permitirá lanzar los test de JUNIT en conjunción con el calculo de la cobertura, y por tanto al finalizar los test, nos aparecerá a parte del resultado del Test, la cobertura.

Cobertura de pruebas: EclEmma

- Para probarlo podemos escribir una clase con dos métodos sencillos

```
public class OperadorAritmetico {
		public static int suma(int a, int b) {
			return a + b;
		}

		public static int division(int a, int b) throws Exception {
			if(b==0) {
				throw new Exception();
			}
			return a / b;
		}
}
```

Cobertura de pruebas: EclEmma

- Crearemos también una prueba unitaria pero únicamente de uno de los métodos.

```
@Test
public void suma() {

		int a = 5;
		int b = 3;

		int suma = OperadorAritmetico.suma(a, b);

		Assert.assertEquals(8, suma);
}
```

- Al pasar este Test, el plugin EclEmma, nos mostrará las partes del código no probado, que en este caso será todo el método de división.

Selenium

- Paquete de herramientas para automatizar pruebas de aplicaciones Web en distintas plataformas

http://seleniumhq.org/

- La documentación la podremos obtener en

http://seleniumhq.org/docs/index.html

- Las herramientas que componen el paquete son
  - Selenium IDE.
  - Selenium Remote Control (RC) o selenium 1.
  - Selenium WebDriver o selenium 2.

Selenium IDE

- Se trata de un plugin de Firefox, que nos permitirá grabar y reproducir una macro con una prueba funcional, la cual podremos repetir las veces que deseemos.

https://addons.mozilla.org/es-es/firefox/addon/favorites-selenium-ide/

- Las acciones que se realizan en la navegación mientras se graba la macro, se traducen en comandos.
- La macro por defecto se guarda en HTML, aunque también se puede obtener como java, c#, Python, ...
- También se podrán insertar validaciones y no solo acciones sobre la pagina.

Selenium IDE

- El HTML que se genera tiene una tabla con 3 columnas:
  - Comando de Selenium.
  - Primer parámetro requerido
  - Segundo parámetro opcional
- Los comandos de selenium se dividen en tres tipos:
  - **Acciones**– Acciones sobre el navegador.
  - **Almacenamiento**– Almacenamiento en variables de valores intermedios.
  - **Aserciones**– Verificaciones del estado esperado del navegador.

Selenium IDE

- Los comandos de navegación mas habituales son:
  - **open**: abre una página empleando la URL.
  - **click/clickAndWait**: simula la acción de click, y opcionalmente espera a que una nueva pagina se cargue.
  - **waitForPageToLoad**: para la ejecución hasta que la pagina esperada es cargada. Es llamada por defecto automáticamente cuando se invoca **clickAndWait**.

Selenium IDE

- Los comandos de navegación mas habituales son:
  - **waitForElementPresent**: para la ejecución hasta que el UIElement esperado, esta definido por un tag HTML presente en la pagina.
  - **chooseCancelOnNextConfirmation**: Predispone a seleccionar en la próxima ventana de confirmación el botón de Cancel.

Selenium IDE

- Los comandos de almacenamiento mas habituales son:
  - **store**: Almacena en la variable el valor.
  - **storeElementPresent**: Almacenara True o False, dependiendo de si encuentra el UI Element.
  - **storeText**: Almacena el texto encontrado. Es usado para localizar un texto en un lugar de la pagina especifico.

Selenium IDE

- Los comandos de verificación mas habituales son:
  - **verifyTitle/assertTitle**: verifica que el titulo de la pagina es el esperado.
  - **verifyTextPresent**: verifica que el texto esperado esta en alguna parte de la pagina.
  - **verifyElementPresent**: verifica que un UI element esperado, esta definido como tag HTML en la presente pagina.

Selenium IDE

- Los comandos de verificación mas habituales son:
  - **verifyText**: verifica si el texto esperado y su tag HTML estan presentes en la pagina.
  - **assertAlert**: verifica si sale un alert con el texto esperado.
  - **assertConfirmation**: verifica si sale una ventana de confirmacion con el texto esperado.

Selenium IDE Ejercicio

- Basándonos en la aplicación web empleada anteriormente para HTTPUnit, generar la macro que realice el mismo flujo y añadir las comprobaciones siguientes.
  - Comprobar que tiene los campos ”nombre” y “mail”.
  - Comprobar que la respuesta del formulario, es una pagina con un titulo “Datos enviados”.
  - Comprobar que tiene unos div con el atributo “name” igual a “nombre” y “mail”, y que su contenido es el enviado por el formulario.
  - Comprobar que existe un link con el texto “inicio”.

Selenium Web Driver

- Es el motor de pruebas automatizadas de Selenium, desde el plugin de eclipse, se obtiene una clase de Test de Junit 4, esta clase de Test, se debe incluir en un proyecto, con las dependencias de Selenium y Junit.
- Con Maven, las dependencias necesarias son:

```
<dependency>
		<groupId>org.seleniumhq.selenium</groupId>
		<artifactId>selenium-java</artifactId>
		<version>2.19.0</version>
</dependency>
<dependency>
		<groupId>junit</groupId>
		<artifactId>junit</artifactId>
		<version>4.10</version>
</dependency>
```

Selenium Web Driver

- De no usar Maven las librerías se pueden descargar de.

http://selenium.googlecode.com/files/selenium-java-2.9.0.zip

Selenium WebDriver Ejercicio

- Recoger la anterior macro y exportarla en formato **WebDriver**, para ejecutar el Test desde Eclipse.

Pruebas de aceptación

- El objetivo de las pruebas de aceptación es validar que un sistema cumple con el funcionamiento esperado y permitir al usuario final de dicho sistema determinar su aceptación desde el punto de vista de su funcionalidad y rendimiento.
- Las pruebas de aceptación son definidas por el usuario final y preparadas por el equipo de desarrollo, aunque la ejecución y aprobación corresponden al usuario final.

Pruebas de aceptación

- La validación del sistema se consigue mediante la realización de pruebas de caja negra que demuestran la conformidad con los requisitos y que se recogen en el plan de pruebas
- Dicho plan está diseñado para asegurar que se satisfacen todos los requisitos funcionales especificados por el usuario teniendo en cuenta también los requisitos no funcionales relacionados con el rendimiento, seguridad de acceso al sistema, a los datos y procesos, así como a los distintos recursos del sistema.

Pruebas de aceptación

- La mayoría de los desarrolladores de productos de software llevan a cabo un proceso denominado pruebas alfa y beta para descubrir errores que parezca que sólo el usuario final puede descubrir.
  - **Prueba alfa**: se lleva a cabo, por un cliente, en el lugar de desarrollo. Se usa el software de forma natural con el desarrollador como observador, registrando los errores y problemas de uso.
  - **Prueba beta**: se llevan a cabo por los usuarios finales del software en los lugares de trabajo de los clientes. A diferencia de la prueba alfa, el desarrollador no está presente normalmente. El cliente registra todos los problemas e informa al desarrollador.

Pruebas de aceptación: ATDD

- Si llevamos las pruebas de aceptación al marco de TDD, estaríamos hablando de ATDD (Aceptance Test Driven Development) la idea es que a cada requisito se le asocia un test o una suite de tests de junit y entonces si el test falla es porque no está implementada la funcionalidad.
- De aquí se sacan dos mejoras respecto TDD
  - El cliente o el supervisor va viendo día a día como sus requisitos se van implementando (cada vez hay más verdes), eliminando el riesgo de que no se cumpla al final con las entregas.
  - Se ve que cosas que funcionan van dejando de funcionar por otras cosas más complicadas (efectos laterales, regresión) permitiendo priorizar funcionalidades.

Concordion

- Es una herramienta que nos permite realizar las pruebas de aceptación.

http://www.concordion.org/

- En Concordion, las especificaciones (o pruebas de aceptación) se escriben en archivos XHTML, usando los elementos comunes para darle formato. De esta manera se logran especificaciones fáciles de leer y que todos pueden comprender.
- Y las pruebas a realizar a partir de los requisitos HTML se materializan realizando asociaciones entre el texto y las pruebas (instrumentación del HTML), extrayendo la información valiosa para la prueba automatizada.

Concordion

- Las pruebas en Concordion son pruebas Junit.
- La instrumentación del HTML, consiste en añadir comandos de Concordion como parámetros en los elementos HTML. Los navegadores web ignoran los atributos que no entienden, de modo que estos comandos son invisibles a efectos prácticos.
- Los comandos usan el espacio de nombres "concordion" definido al principio de cada documento como sigue:

```
<html xmlns:concordion="http://www.concordion.org/2007/concordion">
```

Concordion

- El resultado de una prueba con Concordion, será un HTML, generado a partir de la ruta que especifica la propiedad del sistema “java.io.tmpdir”. Para que se sitúe en un lugar conocido, tendremos que añadir al arranque de la JVM.

```
-Djava.io.tmpdir="D:/Proyectos/Test Driven Development/workspace/Concordion/resultTest/"
```

Concordion

- Concordion tendrá una serie de comandos que nos permitirán la instrumentalización, son los siguientes.
  - **concordion:assertEquals**
  - **concordion:assertTrue**
  - **concordion:assertFalse**
  - **concordion:set**
  - **concordion:execute**
  - **concordion:verifyRows**

Concordion

- **concordion:assertEquals**
  - Este comando nos sirve para comparar el resultado de un método de Java, con un texto incluido en el HTML.
  - Para este comando, tendremos que cuando el método Java devuelve un tipo de dato Integer, Double, … se emplea el resultado del método toString() del objeto para la comparación. Y si el metodo Java devuelve void, se comparara con (null)

Concordion

- **concordion:assertEquals**
  - Con el siguiente código

```
<span concordion:assertEquals=“getNombre()">Victor</span>
```

  - Se podrían tener los siguientes resultados

Resultado del método	Resultado de la prueba
Victor	SUCCESS
Juan	FAILURE
victor	FAILURE

Concordion

- **concordion:assertTrue**
  - Este comando nos sirve para comparar si el resultado de un método de Java en True.

Concordion

- **concordion:assertTrue**
  - Con el siguiente código

```
<p>
Mientras el siguiente texto "<span concordion:set="#texto">12</span>"
represente un numero entero sera
<span concordion:assertTrue="isNumberInteger(#texto)">valido</span>.
</p>
```

  - Podríamos tener los siguientes resultados

Resultado del método	Resultado de la prueba
True	SUCCESS
False	FAILURE

Concordion

- **concordion:execute**
  - Este comando nos servirá para:
    - Ejecutar instrucciones cuyo resultado es void.
    - Ejecutar instrucciones cuyo resultado es un objeto.
    - Manejar frases con estructuras poco habituales.
  - Las instrucciones con resultado void que se ejecutaran en un test, por regla general serán del tipo “**set**” o “**setUp**”, con cualquier otro uso, será una mala señal, no estaremos escribiendo bien la especificación.

```
<span concordion:execute="setCurrentTime(#time)" />
```

Concordion

- **concordion:set**
  - Este comando nos sirve para definir variables temporales en nuestro Test, que pueden emplearse como parámetros de los métodos Java.

```
<p>
El saludo para el usuario <span concordion:set="#nombre">Pepe</span>sera:
<span concordion:assertEquals="saludaA(#nombre)">Hola Pepe!
</p>
```

  - Este código emplea en el Test, una variable temporal definida en la línea anterior.

Concordion

- **concordion:execute**
  - Cuando la instrucción nos retorna un objeto, este se almacenara en una variable temporal, accediendo posteriormente a los atributos o métodos del objeto a través de la variable.

```
<span concordion:execute="#resultado = split(#TEXT)">Victor Herrero</span>
<span concordion:assertEquals="#resultado.nombre"> Victor</span>
 y apellido <span concordion:assertEquals="#resultado.apellido"> Herrero
</span>.
```

  - Vemos en este ejemplo el uso de (#TEXT), esta expresión es equivalente a

```
<span concordion:set="#nombre">Victor Herrero</span>
<span concordion:execute ="#resultado = split(#nombre)" />
```

Concordion

- **concordion:execute**
  - Para manejar frases con una estructura poco habitual, por ejemplo una en la que se emplee un parámetro antes de definirlo.

```
<p> Se debería mostrar el saludo "<span>¡Hola Pepe!</span>" al usuario
<span>Pepe</span> cuando éste acceda al sistema. </p>
```

  - Este caso se instrumentaría de la siguiente forma, de tal forma que primero se procesan los set, luego el comando del execute y por ultimo los assertEquals.

```
<p concordion:execute="#greeting = greetingFor(#firstName)">
Se debería mostrar el saludo
"<span concordion:assertEquals="#greeting">¡Hola Pepe!</span>"
al usuario <span concordion:set="#firstName">Pepe</span> cuando éste acceda
al sistema. </p>
```

Concordion

- **concordion:execute**
  - En una tabla lo podríamos usar de la siguiente forma.

```
<table concordion:execute="#resultado = split(#nombreCompleto)">
<tr>
            <th concordion:set="# nombreCompleto ">Nombre Completo</th>
            <th concordion:assertEquals="#resultado.nombre">Nombre</th>
            <th concordion:assertEquals="#resultado.apellido">Apellido</th>
</tr>
 <tr>
            <td>Pau Gasol</td> <td >Juan</td> <td >Pérez</td>
</tr>
<tr>
            <td>Felipe Reyes</td> <td>Felipe</td> <td>Reyes</td>
 </tr>
</table>
```

Concordion

- **concordion:verifyRows**
  - Este comando nos permite comprobar los contenidos de una colección de resultados que devuelve el sistema.

```
<table concordion:execute="setUpUser(#username)">
                <tr><th concordion:set="#username">Nombre de usuario</th></tr>
                <tr><td>john.lennon</td></tr>
                <tr><td>ringo.starr</td></tr>
                <tr><td>george.harrison</td></tr>
                <tr><td>paul.mccartney</td></tr>
</table>
<p>La búsqueda por "<b concordion:set="#searchString">arr</b>" devolverá:</p>
<table concordion:verifyRows="#username : getSearchResultsFor(#searchString)">
                <tr><th concordion:assertEquals="#username">
                                Nombres de usuario con correspondencia</th></tr>
                <tr><td>george.harrison</td></tr>
                <tr><td>ringo.starr</td></tr>
</table>
```

Concordion

- El procedimiento para trabajar con concordion, será.
  - Incluir los siguientes jar en el classpath.
    - concordion-1.4.2.jar
    - junit-3.8.2.jar o junit-4.8.2.jar
    - ognl-2.6.9.jar
    - xom-1.2.5.jar
  - Se puede hacer con Maven.

```
<dependency>
                        <groupId>org.concordion</groupId>
                        <artifactId>maven-concordion-plugin</artifactId>
                        <version>1.0.0</version>
</dependency>
```

Concordion

- Construir el fichero HTML con los requisitos.
- Instrumentalizarlo añadiendo el espacio de nombres y las expresiones.

```
<html xmlns:concordion="http://www.concordion.org/2007/concordion">
<span concordion:assertEquals="getGreeting()“>
```

- Añadir en el mismo paquete la clase de Test con los métodos que serán invocados desde el HTML, que extienda de “**ConcordionTestCase**”
- Configurar el directorio de salida del Test.
- Ejecutar el Test con Junit.
- Comprobar el estado del fichero generado.

Concordion: Ejemplo

- Añadir a un proyecto, el fichero HelloWorld.html.

```
<html>
   <body>
      <p>Hello World!</p>
   </body>
</html>
```

- Instrumentar el fichero.

```
<html xmlns:concordion="http://www.concordion.org/2007/concordion">
   <body>
      <p concordion:assertEquals="getGreeting()">Hello World!</p>
   </body>
</html>
```

Concordion: Ejemplo

- Añadir en el mismo paquete, un fichero Java **HelloWorldTest.java**

```
import org.concordion.integration.junit3.ConcordionTestCase;
public class HelloWorldTest extends ConcordionTestCase {
    public String getGreeting() {
        return "Hello World!";
    }
}
```

- Ejecutar el Test con Junit, obteniendo como resultado

```
C:\temp\concordion-output\example\HelloWorld.html
Successes: 1  Failures: 0
```

Concordion: Ejercicio

- Dada la siguiente historia de usuario (requisitos), generar las pruebas de aceptación necesarias.

- Mientras el password "Password" sea el correcto para el usuario "Juan", este estará validado y su DNI deberá ser "11111111-C".

Concordion: Ejercicio

- Dada los siguientes requisitos, generar las pruebas de aceptación necesarias.
  - Si se invoca con un id existente, se devuelve la provincia correspondiente
  - Si se invoca con un id inexistente, se devuelve null
  - Si se invoca con un null, se tira una java.lang.IllegalArgumentException

Pruebas de regresión.

- Se denominan Pruebas de regresión a cualquier tipo de pruebas de software que intentan descubrir las causas de nuevos errores (bugs), carencias de funcionalidad, o divergencias funcionales con respecto al comportamiento esperado del software.
- Estos nuevos errores, se producirán por cambios recientes realizados en partes de la aplicación que anteriormente al citado cambio no eran propensas a este tipo de error.

Pruebas de regresión.

- Este tipo de cambio puede ser debido a
  - Prácticas no adecuadas en el control de versiones.
  - Falta de consideración acerca del ámbito o contexto en el que se produce el error (fragilidad de la corrección).
  - Consecuencia del rediseño de la aplicación.
- Por lo tanto, en la mayoría de las situaciones del desarrollo de software se considera una buena práctica que cuando se localiza y corrige un bug, se grabe una prueba que exponga el bug y se vuelvan a probar regularmente después de los cambios subsiguientes que experimente el programa.

Pruebas de regresión.

- La herramienta que usaremos para nuestras pruebas de regresión, será JUnit, ya que nos permitirá tanto generar el Test que compruebe nuestra nueva funcionalidad desarrollada, como que se siguen cumpliendo los requerimientos anteriores y que no se ha alterado su funcionalidad después de la nueva modificación.
- Por lo tanto, estaremos haciendo pruebas de regresión, siempre que lancemos pruebas sobre una funcionalidad ya desarrollada con anterioridad, para comprobar que los últimos desarrollos, no han hecho fallar esas funcionalidades.

Pruebas de regresión: Ejercicio

- Dada la entidad user, con los atributos:
  - Nombre.
  - Password.
  - Edad.
- Crear la lógica que comunique a dos usuarios siempre que no sean iguales.
- Dos usuarios son iguales si tienen el mismo Nombre, Password y la edad.

Calidad del código

- Decimos que un código tiene calidad, cuando tenemos facilidad de mantenimiento y de desarrollo.
- ¿Como podemos hacer que nuestro código tenga mas calidad? Pues consiguiendo que nuestro código no tenga partes que hagan que:
  - Se reduzca el rendimiento.
  - Se provoquen errores en el software.
  - Se compliquen los flujos de datos.
  - Lo hagan mas complejo.
  - Supongan un problema en la seguridad.

Calidad del código

- Tendremos dos técnicas para mejorar el código fuente de nuestra aplicación y, con ello, el software que utilizan los usuarios como producto final:
  - **Test**. Son una serie de procesos que permiten verificar y comprobar que el software cumple con los objetivos y con las exigencias para las que fue creado.
  - **Análisis estático del código**. Proceso de evaluar el software sin ejecutarlo.

Análisis estático del código

- Es una técnica que se aplica directamente sobre el código fuente tal cual, sin transformaciones previas ni cambios de ningún tipo. La idea es que, en base a ese código fuente, podamos obtener información que nos permita mejorar la base de código manteniendo la semántica original. Esta información nos vendrá dada en forma de sugerencias para mejorar el código.

Análisis estático del código

- Emplearemos herramientas que incluyen, por un lado, analizadores léxicos y sintácticos que procesan el código fuente y, por otro, un conjunto de reglas que aplicar sobre determinadas estructuras. Si nuestro código fuente posee una estructura concreta que el analizador considere como "mejorable" en base a sus reglas nos lo indicará y nos sugerirá una mejora.

Análisis estático del código

- Tendremos dos tipos de análisis estáticos del código, el manual y el automático.
  - El análisis automático que realiza un programa de ordenador sobre nuestro código, reduce la complejidad que supone detectar problemas en la base del código ya que los busca utilizando a unas reglas que tiene predefinidas.
  - El análisis manual que realiza una persona, se centra en apartados propios de nuestra aplicación en concreto como, por ejemplo, determinar si las librerías que está utilizando nuestro programa se están utilizando debidamente o si la arquitectura de nuestro software es la correcta.

Análisis estático del código

- Deberíamos tratar de realizar el análisis estático del código cada vez que hayamos creado una nueva funcionalidad en nuestro software, para comprobar que la implementación realizada es de calidad. También cuando notamos que el desarrollo se complica, nos cuesta implementar algo que supuestamente debe ser sencillo.
- El análisis estático del código, nos permitirá comprobar la calidad del software y nos sugerirá modificaciones para mejorarlo, pero no nos dirá si el software hace lo que tiene que hacer.

Herramientas: PMD

- Su funcionamiento se basa en detectar patrones, los cuales son posibles errores que pueden aparecer en tiempo de ejecución,
  - Código que no se puede ejecutar nunca porque no hay manera de llegar a él,
  - Código que puede ser optimizado,
  - Expresiones lógicas que puedan ser simplificadas,
  - Malos usos del lenguaje, etc
- La pagina de referencia es

http://pmd.sourceforge.net/

Herramientas: PMD

- Estos patrones se encuentran catalogados en distintas categorías, que podemos consultar en

http://pmd.sourceforge.net/rules/index.html

- Es una herramienta independiente pero tiene plugin para los mas conocidos entornos de desarrollo, para eclipse.

http://pmd.sf.net/eclipse

Herramientas: PMD

- Los pasos a seguir para emplear este plugin, serán.
  - Configurar las reglas que queremos que se apliquen.
  - Checkear el código con PMD.
  - Generar los reports.
  - Se pueden visualizar los problemas encontrados en la view Problems de eclipse.

Herramientas: PMD

- Esta herramienta es extensible y configurable ya que se pueden añadir nuevas reglas o configurar las que ya se incluyen en caso de que esto fuera necesario.
- Además también podemos configurar el formato en el que deseamos que nos devuelva los resultados para que elijamos entre aquello que nos resulte más cómodo: xml, html, txt, etc.

Herramientas: CPD

- Forma parte de PMD, aunque funciona de una manera autónoma, y es sin duda uno de los analizadores más útiles que podamos encontrar.
- CPD son las siglas de “Copy Paste Detector” y con ese nombre queda clara cuál es su misión: buscar duplicidades en el código. Todo aquello que esté escrito más de una vez en nuestro código será detectado por CPD.

http://pmd.sourceforge.net/cpd.html

- Viene una opción en el menú de PMD del plugin de Eclipse para lanzar esta funcionalidad.

Herramientas: CPD

- Cuando hay código duplicado en nuestra aplicación y tenemos que modificarlo, tenemos un problema, tendremos que recorrer todo el código para repercutir ese mismo cambio en todas las copias.
- Tendremos dos casos claros en los que encontraremos partes de código duplicadas.
  - Cuando no apliquemos herencia.
  - Cuando empleemos un algoritmo de tratamiento de datos en varios lugares, escribiéndolo en cada lugar, en vez de extraerlo a un método.

Herramientas: CheckStyle

- Inicialmente se desarrolló con el objetivo de crear una herramienta que permitiese comprobar que el código de nuestras aplicaciones se ajustase a los estándares dictados por Sun Microsystems, empresa creadora del lenguaje Java.
- Sin embargo, posteriormente se añadieron nuevas capacidades que han hecho que sea un producto muy similar a PMD. Es por ello que también busca patrones en el código que se ajustan a categorías muy similares a las de este analizador.
- http://checkstyle.sourceforge.net/

Herramientas: CheckStyle

- Disponemos de un plugin de eclipse

```
http://eclipse-cs.sf.net/update/
```

- Una ventaja que tiene con respecto a PMD es que además encuentra errores en la documentación que podamos haber escrito en el javadoc. PMD no realiza ningún tipo de validación mientras que Checkstyle es muy riguroso en ese sentido.
- Gracias a ello, si tenemos una función que recibe dos parámetros y documentamos uno o ninguno, Checkstyle nos avisará de que estamos dejando parte de la función sin documentar.

Herramientas: FindBugs

- Es un producto de la Universidad de Maryland que, como su nombre indica, está especializado en encontrar errores.

http://findbugs.sourceforge.net/

- Al igual que los demás tiene una serie de categorías donde poder catalogar dichos errores, y es que según findbugs estos pueden ser malas prácticas, mal uso del lenguaje, internacionalización, posibles vulnerabilidades, mal uso de multihilo, rendimiento, seguridad...

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

Herramientas: FindBugs

- También se dispone de un plugin para eclipse.

  http://findbugs.cs.umd.edu/eclipse/

- Los resultados de las pruebas, se pueden ver o bien en la view Problems de eclipse o bien en una perspectiva propia de FindBugs.

Fin