Herramientas de pruebas Java

1. Pruebas de Software

1.1. Introducción

Son la parte del desarrollo, que persigue corroborar que el Software generado cumple con los requisitos planteados, son un extra en el desarrollo, no forman parte de la aplicación desarrollada.

Existen dos grandes grupos de pruebas

- Pruebas de caja blanca
- Pruebas de caja negra

Hay muchas formas de realizar las pruebas, algunas de ellas exigen la validacion visual de una persona que sepa que se esta probando y con que información, y que pueda estimar el resultado esperado, esta aproximación no es muy conveniente, dado que no permite automatizar el proceso de las pruebas.

Lo ideal será que la prueba contenga no solo el algoritmo de prueba, sino tambien las comprobaciones (asertos) del resultado, para ello existen en el mercado numerosos frameworks que permiten realizar una aproximación *Validación Rojo-Verde.

1.1.1. Pruebas de caja blanca

Tipo de pruebas de software que se realiza sobre las funciones internas de un módulo, buscan recorrer todos los caminos posibles del modulo, cerciorándose de que no fallen.

Dado que probar todo es inviable en la mayor parte de los casos, se define **Cobertura** como una medida porcentual de cuánto código se ha cubierto.

Las pruebas de caja blanca nos convencen de que un programa hace bien lo que hace, pero no de que haga lo que necesitamos.

1.1.2. Pruebas de caja negra

Se dice que una prueba es de caja negra cuando prescinde de los detalles del código y se limita a lo que se ve desde el exterior. Intenta descubrir casos y circunstancias en los que el módulo no hace lo que se espera de él, ejercitan los requisitos funcionales.

Las pruebas de caja negra están especialmente indicadas en aquellos módulos que van a ser interfaz con el usuario.

1.1.3. Cualidades deseables del Software

Existen un conjunto de cualidades, que se pueden evaluar de forma automatizada

- **Correcto**. Se comporta según los requisitos.
- **Robusto**. Se comporta de forma razonable, aun en situaciones inesperadas.
- **Confiable**. Se comporta según las expectativas del usuario (Correcto + Robusto).
- **Eficiente**. Emplea los recursos justos.
- **Verificable**. Sus características pueden ser comprobadas.

Y otras que no, que a lo maximo que se puede aspirar es a acotarlas, con arquitectura, buenas practicas, ...

- **Amigable**. Fácil de utilizar.
- **Mantenible**. Se puede modificar fácilmente.
- **Reusable**. La misma pieza de software se puede usar sin cambios en otro proyecto.
- **Portable**. Es ejecutable en distintos ambientes (SO, ...).
- **Legible**. El código es fácilmente interpretable.
- **Interoperable**. Puede interaccionar con otros software.

1.1.4. Most Important Test

- **Unitario**. Prueba un modulo lógico.
- **Integración**. Prueba un conjunto de módulos trabajando juntos.
- **Regresión**. Determina si los cambios recientes en un módulo afectan a otros módulos.
- **Humo**. Pruebas de integración completa, ejecutadas de forma periódica, que buscan encontrar errores en releases de forma temprana.
- **Sistema**. Verificación de que el ingreso, procesado y recuperación de datos se hace de forma correcta.
- **Aceptación**. Determinación por parte del cliente de si acepta el modulo.
- **Stress**. Comprobación del funcionamiento del sistema ates condiciones adversas, como memoria baja, gran cantidad de accesos y concurrencia en transacciones.
- **Carga**. Tiempo de respuesta para las transaciones del sistema, para diferentes supuestos de carga.

1.1.5. Most Important Metrics

- **Complejidad ciclomática**. Numero de caminos independientes en el código.
- **Cobertura**. Cantidad de código fuente cubierto por test.
- **Código duplicado**. Mide las veces que aparecen un numero de líneas repetidas (> 10 líneas).
- **Comentarios**. Cantidad de código que tiene documentación.
- **Diseño del software**. Indica el grado de acoplamiento de los módulos entre si.

- **Líneas**. Cantidad de líneas del código.
- **Malas practicas de codificación**. Aparición de numero mágicos, bloques de try sin procesar, ...

1.2. 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.

1.3. JUnit

Framework para pruebas.

Paquetes junit.* (JUnit 3) y org.junit.* (JUnit 4) Embedido en Eclipse (JUnit 3 y 4), eclipse 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, test parametrizados, ...

1.3.1. Test Suites

Conjunto de pruebas a ejecutar de forma conjunta.

```
@RunWith(Suite.class)
@SuiteClasses({C1Test.class, C2Test.class})
public class TestSuite{

}
```

1.3.2. Ciclo de vida de los Test

Se proporcionan anotaciones que permiten actuar en las distintas fases del ciclo de vida

- @Before: El método de instancia anotado con esta anotación, se ejecutara antes de cada Test de la clase, por tanto tantas veces como métodos de instancia anotados con @Test existan.
- @After: El método de instancia anotado con esta anotación, se ejecutara despues de cada Test de la clase, por tanto tantas veces como métodos de instancia anotados con @Test existan.
- @BeforeClass: El método estatico anotado con esta anotación, se ejecutara antes de cualquier otro de la clase y solo una vez.
- @AfterClass: El método estatico anotado con esta anotación, se ejecutara despues de todos los otros métodos de instancia de la clase y solo una vez.

- @Test: El método anotado con esta anotación, representa un Test.
- @Ignore: Permite ignorar un método de Test.

1.3.3. Probando el lanzamiento de excepciones

La anotacion @Test, permite realizar pruebas, donde el resultado esperado sea una **Excepcion**.

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

1.3.4. Probando restricciones temporales

La anotacion @Test, permite realizar pruebas, donde el tiempo transcurrido en la ejecución esta limitado por el requisito.

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

1.3.5. Test parametrizados

El API incorpora un **Runner**, que permite la ejecución repetida de Test, cada vez con unos datos de prueba, para lo cual hay que

- Anotar la clase con **@RunWith**

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

- Crear un método publico estatico que retorne un **Array de Arrays**, anotado con **@Parameters**

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

Puede ser interesante emplear la clase de utilidad **Arrays**

```
Arrays.asList(new Object[][] {{},{},{}});
```

- Crear Atributos de clase de la misma tipologia que los elementos de los Arrays.
- Constructor que establezca los atributos.

1.3.6. Asertos

Los asertos, son métodos estaticos del API, que permiten realizar validaciones, para poder comprobar que los datos obtenidos estan dentro del rango esperado.

Algunos de los asertos que se proporcionan son:

- assertEquals
- assertFalse
- assertTrue
- assertNotNull
- assertNull
- assertNotSame
- assertSame
- fail

Implementar una clase que obtenega 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).

1.4. Hamcrest

Framework especializado en proporcionar asertos mas semanticos, permite realizar los Test, con un lenguaje mas cercano, mas legible.

La librería hamcrest-library, proporciona una clase con métodos estáticos **org.hamcrest.Matchers**.

Estos métodos estáticos, se emplean en formar un predicado, que forma el aserto, para que la forma de leer el código sea mas natural.

Algunos de los métodos estáticos de Hamcrest.

- is

- not
- nullValue
- empty
- endsWith
- startsWith
- hasItem
- hasItems
- hasProperty

Un ejemplo de predicado con **Hamcrest**, podria ser el siguiente, donde se **valida que** un objeto **calculadora es no nulo**. La lectura comprensiva de la sentencia, coincide con los escrito.

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

Otro ejemplo, que **valida que** el objeto **persona tiene la propiedad** "nombre".

```
assertThat(persona, hasProperty("nombre"));
```

1.5. Mockito

Para poder crear un buen conjunto de pruebas unitarias, es necesario centrarse exclusivamente en la clase a testear, para ello se pueden simular, con **Mocks** el resto de clases involucradas, de esta manera se crean test unitarios potentes que permiten detectar los errores allí donde se producen y no en dependencias del supuesto código probado.

Mockito es una herramienta que permite generar **Mocks** 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 los que han de retornar aquellos componentes de los que depende la clase probada.

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.

Se pueden definir los **Mock** con

- La anotacion @Mock aplicada sobre un atributo de clase.

```
@Mock
private IUserDAO mockUserDao;
```

De emplearse las anotaciones, se ha de ejecutar la siguiente sentencia para que se procesen dichas anotaciones y se generen los objetos **Mock**

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

O bien emplear un **Runner** especifico de Mockito en la clase de Test que emplee Mockito, el **MockitoJUnitRunner**

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

- O con el método estatico **mock**.

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

Algunos de los métodos estáticos que ofrece la clase **org.mockito.Mockito** son

- atLeast
- atMost
- atLeastOnce
- doNothing
- doReturn
- doThrow
- when
- inOrder
- never
- only
- verify
- mock

Algunos de los métodos estaticos que ofrece la clase **org.mockito.Matchers**, para establecer datos genericos son

- any
- anyString

- anyObject
- contains
- endsWith
- startsWith
- eq
- isA
- isNull
- isNotNull

1.5.1. Stubing

Se persigue definir comportamientos del **Mock**, para ello se emplea el método estatico **when** y **thenReturn**

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

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

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());
```

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

1.5.2. Verificación

Se puede verificar el orden en el que se han ejecutado los métodos del **Mock**, pudiendo llegar a diferenciar el orden de invocación de un mismo método por los parametros enviados.

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

Tambien se puede verificar el numero de veces que se ha invocado una funcionalidad

```
verify(mock, never()).someMethod();
verify(mock, only()).someMethod();
verify(mock, times(2)).someMethod("some arg");
```

1.6. DBUnit

Herramienta para simplificar las pruebas unitarias de operaciones sobre base de datos.

Permite establecer un estado de la base de datos conocido, para que el entorno en el que se producen las pruebas sea conocido.

Podremos cargar los datos de test de un XML que tengamos generado y del cual conozcamos el estado de los datos.

Tambien proporciona un API, para comparar los datos que hay en la base de datos, con un XML con los datos esperados.

1.6.1. Procedimiento

En lugar de extender **TestCase** se extiende **DatabaseTestCase**, que es una clase abstracta, que requiere implementar dos métodos

- 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**, que representa un XML.

Un ejemplo de implementación 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, schema);
}

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

En la clase **DatabaseTestCase**, existen otros 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;
}
```

Estos métodos permiten definir que hacer con la Base de Datos antes y despues de ejecutar las pruebas, realizando una acción sobre la base de datos, con el conjunto de datos que representa el DataSet, 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.

Las acciones permitidas son:

- DatabaseOperation.UPDATE
- DatabaseOperation.INSERT
- DatabaseOperation.DELETE
- DatabaseOperation.DELETE_ALL
- DatabaseOperation.TRUNCATE
- DatabaseOperation.REFRESH

- DatabaseOperation.CLEAN_INSERT
- DatabaseOperation.NONE

Conviene automatizar la generacion de los XML que representan el estado esperado de la base de datos, para ello, se puede emplear el siguiente codigo

```
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);
```

Un ejemplo de XML seria

```
<?xml version='1.0' encoding='UTF-8'?>
<dataset>
    <LIBROS ISBN="2" TITULO="La Catedral del Mar"
                AUTOR="Ildefonso Falcones" />
    <LIBROS ISBN="3" TITULO="Las Legiones Malditas"
                AUTOR="Santiago Posteguillo"/>
</dataset>
```

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.

1.7. 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 el codigo empleando la implementación real de todas las clases que necesite para ejecutarse.

1.8. HttpUnit

Nos proporciona la capacidad de interactuar programáticamente con una aplicación web de forma amigable.

Proporciona la clase **com.meterware.httpunit.WebConversation** que simulará a un navegador accediendo al servidor.

```
WebConversation webConversation = new WebConversation();
WebResponse formResponse = webConversation.getResponse("http://localhost:8080/5-
Servidor/");
```

Una vez realizada una petición, se obtendrá un objeto **WebResponse**, que representa un HTML, por lo que se podrán realizar tareas como

- getResponseCode() que retorna el codigo HTTP retornado por la petición.
- getResponseMessage() que retorna el mensaje asociado al codigo HTTP (por ejemplo para HTTP 200, el mensaje es OK)
- getText() que convierte el HTML en texto
- getDOM() que convierte el HTML en un objeto XML DOM
- getForms() que retorna un array con los formularios de la pagina.
- getTitle() que retorna el titulo de la pagina.
- getElementsByTagName("div") que retorna un HTMLElement[] con todas las etiquetas de la pagina del tipo indicado.
- getLinkWith() que retorna los enlaces de la pagina.

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

- getLinkWithImageText() que busca en el texto ALT de una imagen.
- getTables() que retorna las tablas HTML en la pagina

```
WebTable table = resp.getTables()[0];
```

- getFrameContents("marco") que retorna un frame (marco) de la pagina como un nuevo **WebResponse**

1.8.1. Formularios

El API permite interaccionar con los formularios a traves de la clase **WebForm**, para poder establecer datos que se haran llegar al servidor.

```
WebForm form = resp.getForms()[0];
```

- getParameterValue() que retorna el valor de uno de los parametros

```
form.getParameterValue("parámetro") ;
```

- getParameterNames() que retorna un String[] con los nombres de todos los parametros.
- setParameter() que permite establecer el valor de un parametro.

```
setParameter("parámetro", "valor");
```

- toggleCheckbox("parámetro") que permite cambiar el valor de un checkbox.
- submit() que envía los datos del formulario
- reset() que resetea a los valores por defecto los parametros del formulario

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.

1.9. Selenium

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

La documentación la podremos obtener aquí

Las herramientas que componen el paquete son

- Selenium IDE.
- Selenium Remote Control (RC) o selenium 1.
- Selenium WebDriver o selenium 2.

1.9.1. 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. Se puede descargar aquí

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, aunque las validaciones son mas faciles de escribir en el código generado (java, c#, …)

Una vez grabada la macro, 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.

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.
- **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.

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.

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.
- **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.

1.9.2. Selenium WebDriver

Es el motor de pruebas automatizadas de Selenium, se encarga de arrancar un navegador que responde a las ordenes del Test, provocando la ejecución de la macro grabada.

No todas las versiones de Firefox son compatibles con **Selenium WebDriver**, se puede encontrar mas información aquí o aquí

Para descargar versiones antiguas de Firefox, se puede hacer desde aquí

Se puede probar con la version de selenium 2.52.0 y Firefox 45.

Para la dependencia del proyecto con selenium webdriver, añadir

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

El codigo obtenido de la macro grabada con Selenium IDE, tendra las siguientes sentencias

- La creación del objeto que representa la interaccion con el navegador

FirefoxDriver driver = new FirefoxDriver();

- La peticion

driver.get(baseUrl + "/05-Servidor/");

1.10. Pruebas 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.

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.

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.

1.11. Concordion

Es una herramienta que nos permite realizar las pruebas de aceptación. Su pagina principal aquí

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.

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">
```

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="<directorio donde almacenar los resultados>/resultTest/"
```

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**

Para poder trabajar con Concordion, se incluira la siguiente dependencia Maven

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

1.11.1. Procedimiento

- Añadir librerias
- Crear los XHTML instrumentalizados con las etiquetas de concordion, en el mismo paquete donde se definen las clases de test.
- Definir las clases de Test, que heredaran de **ConcordionTestCase**, no tienen porque tener aserciones, solo la logica de invocación al SUT.

1.11.2. 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)

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

1.11.3. concordion:assertTrue

Este comando nos sirve para comparar si el resultado de un método de Java en True.

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

1.11.4. 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 Aserto (concordion:assertEquals), una variable temporal definida en la línea anterior (#nombre).

1.11.5. 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)" />
```

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), que es una variable especial que hace referencia al texto dentro del elemento HTML, esta expresión es equivalente a

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

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>
```

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>
```

Consiguiendo en este caso, verificar distintas caracteristicas del objeto obtenido como resultado de la operación.

1.11.6. 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="#resultado : getSearchResultsFor(#searchString)">
    <tr><th concordion:assertEquals="#resultado">Nombres de usuario con
correspondencia</th></tr>
    <tr><td>george.harrison</td></tr>
    <tr><td>ringo.starr</td></tr>
</table>
```

En este ejemplo, primero se establecen los datos de prueba, ejecutando el método setUpUserName, tantas veces como elementos hay en la tabla y posteriormente se ejecuta la busqueda por un string (arr), verificando que la lista retornada contiene los elementos de la segunda tabla.

1.11.7. Ejemplo

Definicion del HTML sin instrumentalizar

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

Instrumentalizar el fichero

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

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!";
    }
}
```

Obteniendo como resultado

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

1.11.8. 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".

1.11.9. 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