



Herramientas de pruebas Java

=====

Contenidos

1. Pruebas de Software	1
1.1. Introducción	1
1.2. Pruebas Unitarias	3
1.3. JUnit	3
1.4. Hamcrest	6
1.5. Mockito	7
1.6. DBUnit	11
1.7. Pruebas de Integración	14
1.8. HttpUnit	14
1.9. Selenium	16
1.10. Pruebas aceptación	19
1.11. Concondion	20
1.12. Pruebas de seguridad	26
1.13. Usabilidad	27
1.14. Pruebas de Usabilidad	29
1.15. Pruebas Accesibilidad	30
2. Cobertura	32
2.1. Cobertura	33
2.2. Jacoco	33
3. JMeter	37
3.1. Que puede hacer	37
3.2. Que no puede hacer	37
3.3. Estructura	37
3.4. Instalación	37
3.5. Plan de Pruebas	38
3.6. Banco de trabajo	41
3.7. Jmeter Maven Plugin	42
3.8. Jenkins Performance Plugin	43
4. Calidad estática del código	44
4.1. Calidad del Código	44
4.2. Análisis Estático del Código	45
4.3. PMD	46
4.4. Checkstyle	48
4.5. Findbugs	48
5. Sonarqube	49
5.1. Introducción	49
5.2. Instalación	50

5.3. Conceptos	51
5.4. Organizacion	52
5.5. Carga de datos	53
5.6. Gestion de Usuarios y Seguridad	53
5.7. Cuadro de mando	53

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 validación visual de una persona que sepa que se está probando y con qué 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 también 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 módulo, 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 convencerán 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 prescindimos 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 transacciones 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 anotación @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
- assertEquals
- fail



Implementar una clase que obtenega los impuestos según los ingresos siguiendo las siguientes reglas:

- Ingreso \leq 8000 no paga impuestos.
- $8000 < \text{Ingreso} \leq 15000$ paga 8% de impuestos.
- $15000 < \text{Ingreso} \leq 20000$ paga 10% de impuestos.
- $20000 < \text{Ingreso} \leq 25000$ paga 15% de impuestos.
- $25000 < \text{Ingreso}$ 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 lo 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 unidad (normalmente será un metodo de una clase concreto) 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);
```

1.5.1. Stubing

Se persigue definir comportamientos del **Mock**, para ello se emplean los métodos estaticos de la clase **org.mockito.Mockito**, que son

- **atLeast**

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

Y de la clase **org.mockito.Matchers**, que son

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

Algunos ejemplos de definicion de comportamientos del Mock

```
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, salvo que se defina un comportamiento distinto.

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.

*En este ejemplo se esta verificando que el orden de ejecucion de los métodos **getUser** del mock **mockUserDao**, se ejecuta antes que el método **deleteDataSesion** del mock **mockDataSesionUserDao***

```
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

*En este ejemplo se verifica que el método **someMethod** del **mock** no se ejecuta nunca, que el método **someMethod(int)** se ejecuta 1 sola vez y que el método **someMethod(string)** se ejecuta 2 veces.*

```
verify(mock, never()).someMethod();  
verify(mock, only()).someMethod(2);  
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.

También 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 más habitual será el **FlatXmlDataSet**, que representa un XML.

Un ejemplo de implementación sería.

```

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 sobrescribir, 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 después 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 código 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 código HTTP retornado por la petición.
- `getResponseMessage()` que retorna el mensaje asociado al código 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 página.
- `getTitle()` que retorna el título de la página.
- `getElementsByTagName("div")` que retorna un `HTMLDivElement[]` con todas las etiquetas de la página 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**: Almacena 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 código obtenido de la macro grabada con Selenium IDE, tendrá las siguientes sentencias

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

```
FirefoxDriver driver = new FirefoxDriver();
```

- La petición

```
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 incluire 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 librerías
- 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 características 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 búsqueda por un string (arr), verificando que la lista retornada contiene los elementos de la segunda tabla.

1.11.7. Ejemplo

Definición del HTML sin instrumentar

```

<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>

```

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`

1.12. Pruebas de seguridad

Las pruebas de seguridad pretenden encontrar vulnerabilidades en las aplicaciones.

El referente en este aspecto es la organizacion [OWASP](https://www.owasp.org/images/5/5f/OWASP_Top_10_-2013_Final-_Español.pdf), a través de su https://www.owasp.org/images/5/5f/OWASP_Top_10_-2013_Final-_Español.pdf[TOP 10]

1.12.1. OWASP Top 10 2013

- A1 – Inyección

- A3 – Pérdida de Autenticación y Gestión de Sesiones
- A3 – Secuencia de Comandos en Sitios Cruzados (XSS)
- A4 – Referencia Directa Insegura a Objetos
- A5 – Configuración de Seguridad Incorrecta
- A6 – Exposición de datos sensibles
- A7 – Ausencia de Control de Acceso a las Funciones
- A8 – Falsificación de Peticiones en Sitios Cruzados (CSRF)
- A9 – Uso de Componentes con Vulnerabilidades Conocidas
- A10 – Redirecciones y reenvíos no validados

1.12.2. ZAPProxy

Herramienta de OWASP que permite realizar pruebas del Top 10, se puede descargar [aquí](#) y el código se puede encontrar [aquí](#)

La herramienta hace un mapa del sitio buscando las URL con el Spider.

Una vez tiene el mapa, comienza el escaneo activo, que prueba a construir todo tipo de ataques sin llegar a hacerlos efectivos.

Esta herramienta, también puede ser empleada como Proxy, pudiendo grabar todas las interacciones que se hacen con un navegador y evaluando las URLs.

1.12.3. ZAP Maven Plugin

Se proporciona un plugin para el manejo de ZAPProxy desde Maven, la documentación [aquí](#)

```
<plugin>
  <groupId>org.zaproxy.zapmavenplugin</groupId>
  <artifactId>zap-maven-plugin</artifactId>
  <version>1.6-SNAPSHOT</version>
</plugin>
```

1.13. Usabilidad

La usabilidad es la facilidad con que las personas pueden utilizar una herramienta

particular o cualquier otro objeto fabricado por humanos con el fin de alcanzar un objetivo concreto.

1.13.1. Recomendaciones

1. Propósito del sitio: debe quedar claro que servicio ofrece el sitio, se han de evitar elementos que confundan.
2. Equilibrio entre diseño e información: ni lo uno, ni lo otro, la web debe ser atractiva, pero debe tener contenidos. No conviene pasarse en cuanto a contenidos, ya que puede abrumar al visitante.
3. Cuida los enlaces: Todos los enlaces deben estar activos. Se pueden encontrar este tipo de fallos con herramientas como W3C Link Checker.
4. Destacar las acciones: los usuarios no leen, sino que escanean las paginas buscando lo que les interesa, conviene por tanto resaltar las acciones que puede realizar.
5. Diseño de calidad y coherente: estructura sencilla y lógica y diseño común en todas las paginas.
6. Legibilidad: las tipografías claras, textos en color que los resalten del fondo, tamaño de la letra adecuado.
7. Espacios en blanco: ayudan a estructurar la página, y diferenciar contenidos.
8. Uso de imágenes coherente: Contenido, calidad y tamaño.
9. Interfaz sencilla e intuitiva: LA usabilidad debe quedar clara "de un vistazo".
10. Navegación clara: Menús bien estructurados. Es aconsejable un menú en el pie con información básica de la web como la política de privacidad o el aviso legal.
11. Contenido accesible rapidamente: Evitar que el visitante realice muchos interacciones para llegar al objetivo.
12. Reducir los tiempos y tamaños: El sitio web no debería tardar más de 5 segundos en cargar.
13. Compatibilidad con navegadores y dispositivos: Aplicar responsive design.
14. Buscador: Situado en el lado superior derecho, debe ser funcional.
15. Respetar estándares de accesibilidad W3C.

1.13.2. 10 principios de La Heurística de Usabilidad de Nielsen

1. Visibilidad del estado del sistema: El sistema siempre debe mantener a los usuarios informados acerca de lo que está pasando, a través de la retroalimentación adecuada en un tiempo razonable.
2. Diferenciación entre el sistema y el mundo real: El vocabulario utilizado por el sistema hacia los usuarios deberá ser fácilmente comprensible para este, utilizando palabras, frases y conceptos que le resulten familiares.
3. Libertad y control para el usuario: Si los usuarios seleccionaran alguna función por error, se les ofrecerá en todo momento una opción de “deshacer” para salir del estado no deseado sin tener que pasar a través de un diálogo ampliado.
4. Coherencia y estándares: Los usuarios no deberían tener que preguntarse si diferentes palabras, situaciones o acciones significan lo mismo. En la plataforma se deberán seguir una coherencia terminológica.
5. Prevención de errores: Es importante cuidar el diseño para evitar futuros errores cuando interactúe el usuario con el sistema.
6. Reconocer en lugar de memorizar: Minimizar la carga de memoria del usuario haciendo visibles los objetos, acciones y opciones. El usuario no tiene por qué recordar información de una ventana a otra.
7. Flexibilidad y eficiencia de uso: Uso de aceleradores (no apreciables por los usuarios) para disminuir el tiempo de interacción con los usuarios expertos y permitirles personalizar las acciones más frecuentes.
8. Diseño estético y minimalista: Los diálogos no deben contener información que sea irrelevante o innecesaria.
9. Ayudar a los usuarios a reconocer, diagnosticar y corregir errores: Los mensajes de error deben expresarse en lenguaje sencillo, indicar el problema concreto y sugerir constructivamente una solución.
10. Ayuda y documentación: Aunque es mejor que el sistema pueda utilizarse sin documentación, puede ser necesario ofrecer ayuda. Dicha información debe ser fácil de buscar, centrada en la tarea del usuario y no ser demasiado extensa.

1.14. Pruebas de Usabilidad

Las pruebas de usabilidad son heurísticas, se basan en la observación y análisis de cómo un grupo de usuarios reales utiliza la aplicación.

En este tipo de pruebas se ha de dirigir al usuario hacia el cumplimiento de un objetivo, pero sin indicar el paso a paso, de tal forma que unicamente con la aplicación y la ayuda que esta ofrezca pueda llevarlo a cabo.

El usuario debera anotar todos los problemas con los que se encuentre, además del tiempo invertido en resolver cada una de las peticiones que se le hayan realizado.

Estos problemas serán los que se deban estudiar y si es preciso solventar posteriormente.

Al usuario se le podría entregar un formulario de seguimiento de las tareas indicando aspectos como

- Eficiencia
 - Tiempo total de realización de la tarea
 - Tiempos parciales de cada subtarea.
- Eficacia
 - Marcar hitos (subtareas) que se hayan cumplidos
 - Errores cometidos
- Anotaciones

1.15. Pruebas Accesibilidad

Las pruebas de accesibilidad, son un subconjunto de las pruebas de usabilidad, que pretenden corroborar que las paginas expuestas por una aplicación, pueden ser usadas obteniendo la misma funcionalidad con discapacidades o sin ellas.

El referente en cuanto a la accesibilidad lo marca el W3C con su directiva WCAG (directrices de accesibilidad del contenido web). De echo este organismo ofrece validadores online de HTML [aquí](#) y CSS [aquí](#).

El WCAG 1.0 tiene tres niveles de conformidad:

- Las personas con alguna discapacidad "no podrán acceder de ninguna manera a la información" de un documento que no supere el "nivel A".
- Las personas con alguna discapacidad "lo tendrán difícil para acceder a la información" de un documento que no supere el nivel "Doble A".
- Las personas con alguna discapacidad "encontrarán algunas dificultades para acceder a la información" de un documento que no supere el nivel "Triple A".

Hay una nueva version WCAG 2.0, que actualmente esta en Beta, esta version organiza los documentos en distintos niveles:

- **Principios fundamentales:** es el nivel más alto. Aquí se sitúan cuatro principios que proporcionan los fundamentos de accesibilidad Web: perceptibilidad, operabilidad, comprensibilidad y robustez.
- **Pautas generales:** situadas por debajo de los principios. Son doce y proporcionan los objetivos básicos que se deben lograr para crear un contenido accesible.
- **Criterios de éxito:** criterios de éxito verificables que permiten emplear las Pautas 2.0
- **Técnicas:** se pueden aplicar para cada una de las pautas y criterios de éxito.

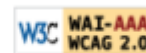
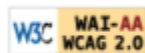
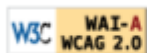
Todos estos documentos se aplican de forma conjunta para crear un sitio Web más accesible.

Los 4 principios básicos o fundamentales en los que se organizan las Pautas WCAG 2.0 son:

- **Perceptibilidad:** La información y los elementos de la interfaz de usuario deben ser presentados a los usuarios de forma que ellos puedan percibirlos. Dentro de este principio se recogen 4 Pautas:
 - Pauta 1.1: Alternativas textuales. Se deben proporcionar alternativas textuales para cualquier contenido no textual.
 - Pauta 1.2: Alternativa para multimedia tempo-dependientes. Se deben proporcionar alternativas para el contenido multimedia basado en el tiempo.
 - Pauta 1.3: Adaptable. El contenido se debe crear de varias formas pero sin perder información o estructura.
 - Pauta 1.4: Distinguible (vista y oído). Se debe facilitar a los usuarios el ver y escuchar el contenido.
- **Operabilidad:** Los componentes de la interfaz y la navegación deben ser operables. Dentro de este principio se recogen 4 Pautas:
 - Pauta 2.1: Acceso mediante teclado. Toda la funcionalidad debe estar disponible desde el teclado.
 - Pauta 2.2: Suficiente tiempo. La información debe permanecer durante suficiente tiempo para leer y usar el contenido.

- Pauta 2.3: Destellos No se debe diseñar con formas que puedan provocar ataques epilépticos.
- Pauta 2.4: Navegable Se debe proporcionar a los usuarios medios que ayuden a navegar, localizar el contenido y determinar dónde se encuentran.
- Comprensibilidad: La información y el manejo de la interfaz de usuario debe ser comprensible
 - Pauta 3.1: Legible y entendible. El contenido debe ser legible y comprensible.
 - Pauta 3.2: Predecible. La apariencia y la operabilidad de las páginas Web deben ser predecibles.
 - Pauta 3.3: Ayuda a la entrada de datos. Se debe ayudar a los usuarios a evitar y corregir los errores.
- Robustez: El contenido debe ser suficientemente robusto para que pueda ser interpretado por una amplia variedad de agentes de usuario, incluyendo los productos de apoyo.
 - Pauta 4.1: Compatible: La compatibilidad con los agentes de usuario debe ser máxima (tanto con los actuales como con los futuros).

Todas estas pautas, estaran divididas en otras mas concretas, que están catalogadas como de Nivel A, AA o AAA.



Existen multitud de herramientas que permiten realizar evaluaciones de la accesibilidad [aquí](#) un listado de las registradas en el W3C.

Una herramienta interesan a parte de las proporcionadas por el W3C, es [TAW Monitor](#), dado que realizará test de accesibilidad indicando problemas visibles en el test automática y aquellos puntos donde hará falta una revisión manual.

Existen otras herramientas interesantes por su rapidez de uso, que son los plugins para los navegadores, ya que ofrecen el resultado del analisis en la propia herramienta empleada para acceder a los contenidos (Por ejemplo Plugin para Chrome W3C Validator).

2. Cobertura

2.1. Cobertura

La Cobertura, representa la cantidad de código que cubren las pruebas realizadas sobre el código.

Existe un plugin de Maven que permite realizar la medición de la cobertura, presentando el resultado en informes **html** y **xml**.

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</artifactId>
      <version>2.5.2</version>
      <configuration>
        <formats>
          <format>xml</format>
          <format>html</format>
        </formats>
      </configuration>
    </plugin>
  </plugins>
</reporting>
```

2.2. Jacoco

Formado por las primeras sílabas de **Java Code Coverage**, es otro plugin de cobertura.

La página de referencia se encuentra [aquí](#)

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>0.7.5.201505241946</version>
      <executions>
        <execution>
          <id>pre-unit-test</id>
          <goals>
            <goal>prepare-agent</goal>
```

```

</goals>
<configuration>
  <!-- Establece la ubicacion del fichero con los
datos de la ejecucion. -->
  <destFile>${project.build.directory}/jacoco-
ut.exec</destFile>
  <!-- Establece la propiedad que contiene la
ruta del agente Jacoco para las pruebas unitarias-->
  <propertyName>surefireArgLine</propertyName>
</configuration>
</execution>
<execution>
  <id>post-unit-test</id>
  <phase>test</phase>
  <goals>
    <goal>report</goal>
  </goals>
  <configuration>
    <!-- Establece la ubicacion del fichero con los
datos de la ejecucion. -->
    <dataFile>${project.build.directory}/jacoco-
ut.exec</dataFile>
    <!-- Establece la ruta donde se genera el
reporte para pruebas unitarias -->
    <outputDirectory>${project.reporting.outputDirectory}/jacoco-
ut</outputDirectory>
  </configuration>
</execution>
<execution>
  <id>pre-integration-test</id>
  <phase>pre-integration-test</phase>
  <goals>
    <goal>prepare-agent</goal>
  </goals>
  <configuration>
    <!-- Establece la ubicacion del fichero con los
datos de la ejecucion. -->
    <destFile>${project.build.directory}/jacoco-
it.exec</destFile>
    <!-- Establece la propiedad que contiene la
ruta del agente Jacoco para las pruebas de integracion -->
    <propertyName>failsafeArgLine</propertyName>
  </configuration>
</execution>

```

```

        <execution>
          <id>post-integration-test</id>
          <phase>post-integration-test</phase>
          <goals>
            <goal>report</goal>
          </goals>
          <configuration>
            <!-- Establece la ubicacion del fichero con los
datos de la ejecucion. -->
            <dataFile>${project.build.directory}/jacoco-
it.exec</dataFile>
            <!-- Establece la ruta donde se genera el
reporte para pruebas de integracion -->
            <outputDirectory>${project.reporting.outputDirectory}/jacoco-
it</outputDirectory>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

```

Se ha de configurar igualmente que el agente sea ejecutado en las distintas fases de **test** e **integration-test**, indicando en el plugin con **argLine** la ubicacion del agente de **jacoco** que debera generar los datos del analisis.

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <version>2.12.4</version>
      <configuration>
        <argLine>${surefireArgLine}</argLine>
        <excludes>
          <exclude>**/integracion/*.java</exclude>
        </excludes>
        <includes>
          <include>**/unitarias/*.java</include>
        </includes>
      </configuration>
    </plugin>
  </plugins>
</build>

```

```

<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-failsafe-plugin</artifactId>
<version>2.8</version>
<configuration>
  <argLine>${failsafeArgLine}</argLine>
  <excludes>
    <exclude>**/unitarias/*.java</exclude>
  </excludes>
  <includes>
    <include>**/integracion/*.java</include>
  </includes>
</configuration>
<executions>
  <execution>
    <id>pasar test integracion</id>
    <phase>integration-test</phase>
    <goals>
      <goal>integration-test</goal>
    </goals>
  </execution>
  <execution>
    <id>validar pruebas integracion</id>
    <phase>verify</phase>
    <goals>
      <goal>verify</goal>
    </goals>
  </execution>
</executions>
</plugin>
</plugins>
</build>

```

Finalmente se pueden añadir los resultados del analisis al sitio añadiendo

```

<reporting>
  <plugins>
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>0.7.5.201505241946</version>
    </plugin>
  </plugins>
</reporting>

```

3. JMeter

Herramienta independiente para realizar pruebas funcionales y de stress sobre un servidor Web.

Se pagina oficial con la documentación esta [aquí](#)

3.1. Que puede hacer

- Cuellos de botella en el sistema.
- Fallos en aplicaciones.
- Peticiones que es capaz de absorber el servidor.
- Simulación de un uso cotidiano de una aplicación.
- Simulación de estrés de una aplicación.
- ...

3.2. Que no puede hacer

- JMeter simplifica la generación de los planes de Test, pero no puede generarlos por si mismo.
- Se precisa de tiempo para generar planes de Test eficientes.

3.3. Estructura

Al acceder a JMeter se ve que la aplicación esta dividida en dos partes.

- **Plan de Pruebas.** Donde desarrollaremos nuestros planes de pruebas.
- **Banco de Trabajo.** Donde tendremos las herramientas y operaciones a utilizar en nuestro plan de pruebas.

Podremos mover contenidos del Banco de Trabajo a los Planes de Prueba para su uso.

3.4. Instalación

La descarga se realiza desde [aquí](#)

Necesaria JVM. Compatibilidad 1.5x en adelante.

Para asignar memoria a JMeter, emplearemos la variable de entorno JVM_ARGS.

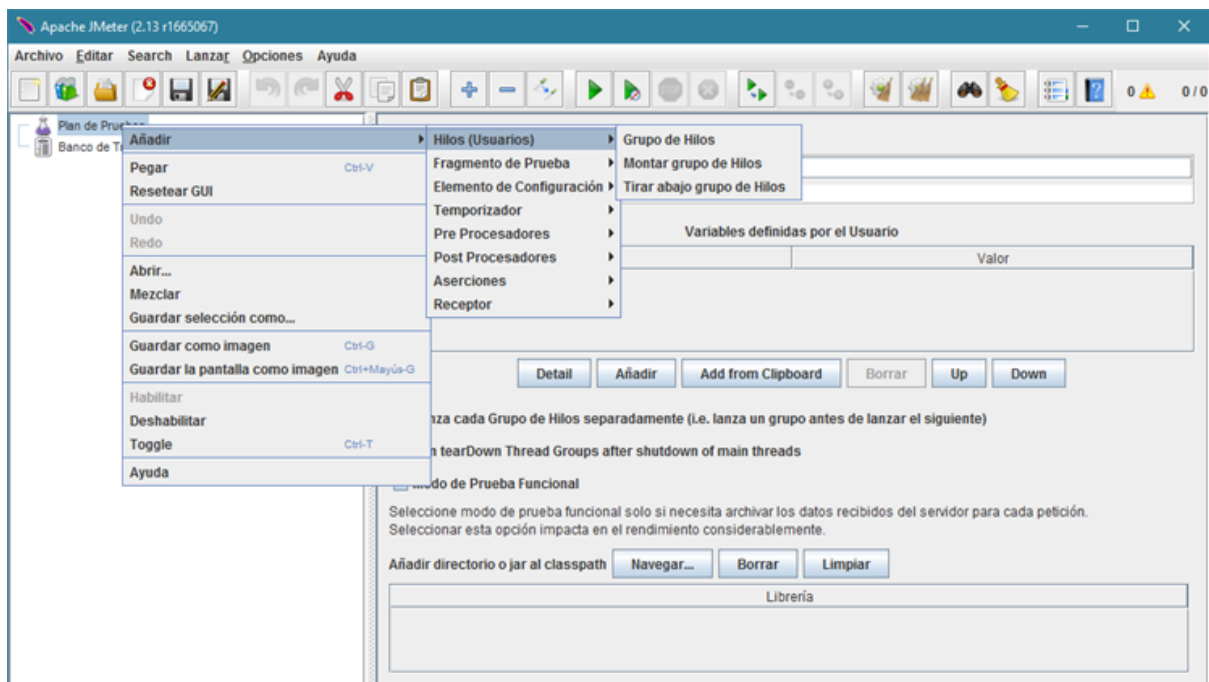
```
JVM_ARGS=-Xms256m -Xmx512m
```

Para incluir un jar en nuestros test, por ejemplo el driver de una base de datos, se incluirá en la carpeta lib.

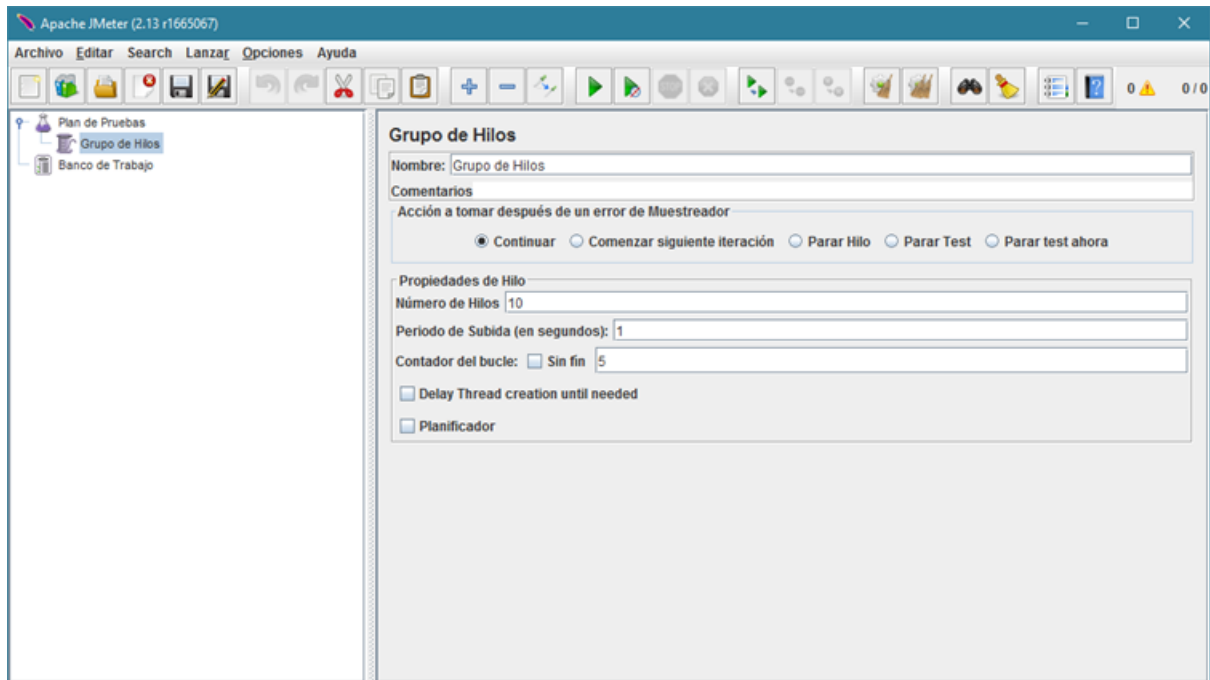
3.5. Plan de Pruebas

Los Planes de Pruebas estan compuestos por elementos de las siguientes tipologías.

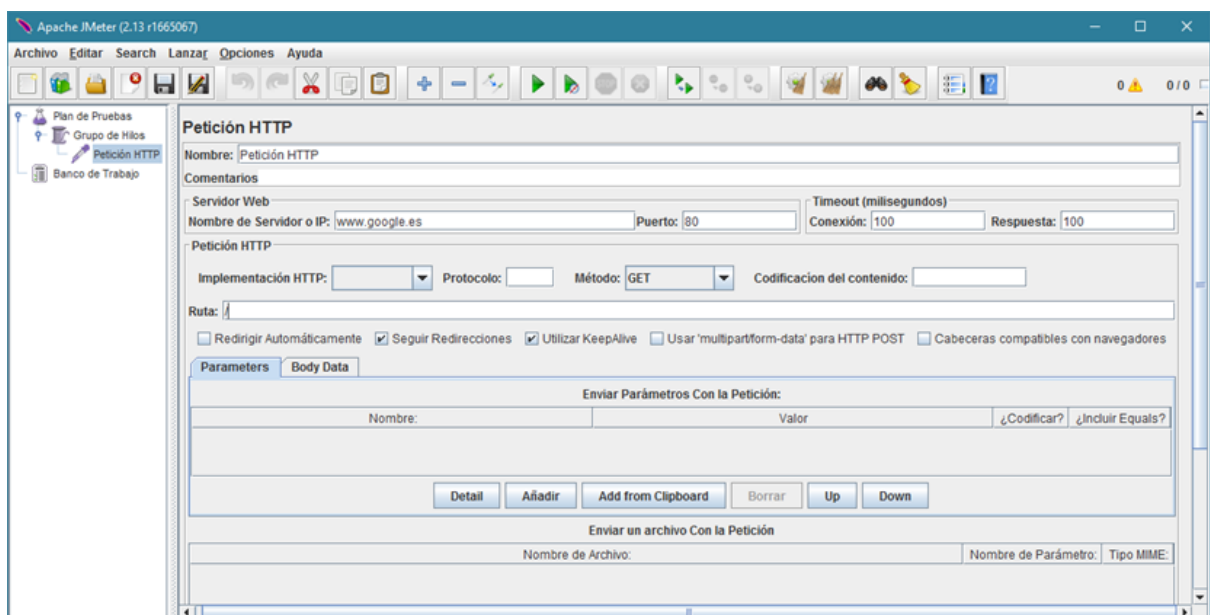
- Grupo de Hilos. Simulara al numero de usuarios.



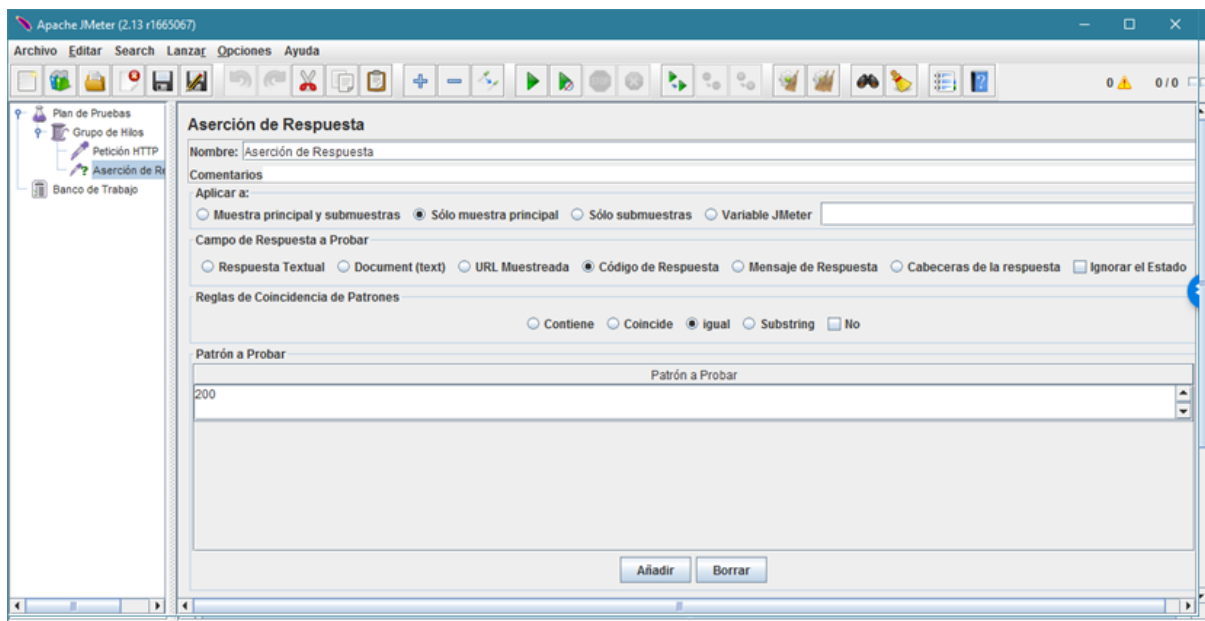
Permite definir usuarios simultáneos y peticiones de cada usuario.



- Procesadores. Permiten realizar modificaciones en la petición original. Se dividen en
 - Pre-Procesadores. Modifican la petición antes de ejecutarse.
 - Post-Procesadores. Modifican la petición después de ejecutarse.
- Controladores. Existen de diversos tipos
 - Muestreadores. Indican las acciones que JMeter puede hacer. Tenemos entre otros los siguientes tipos:
 - Petición HTTP. Nos permite realizar una petición HTTP, indicando protocolo, método, parámetros, ...

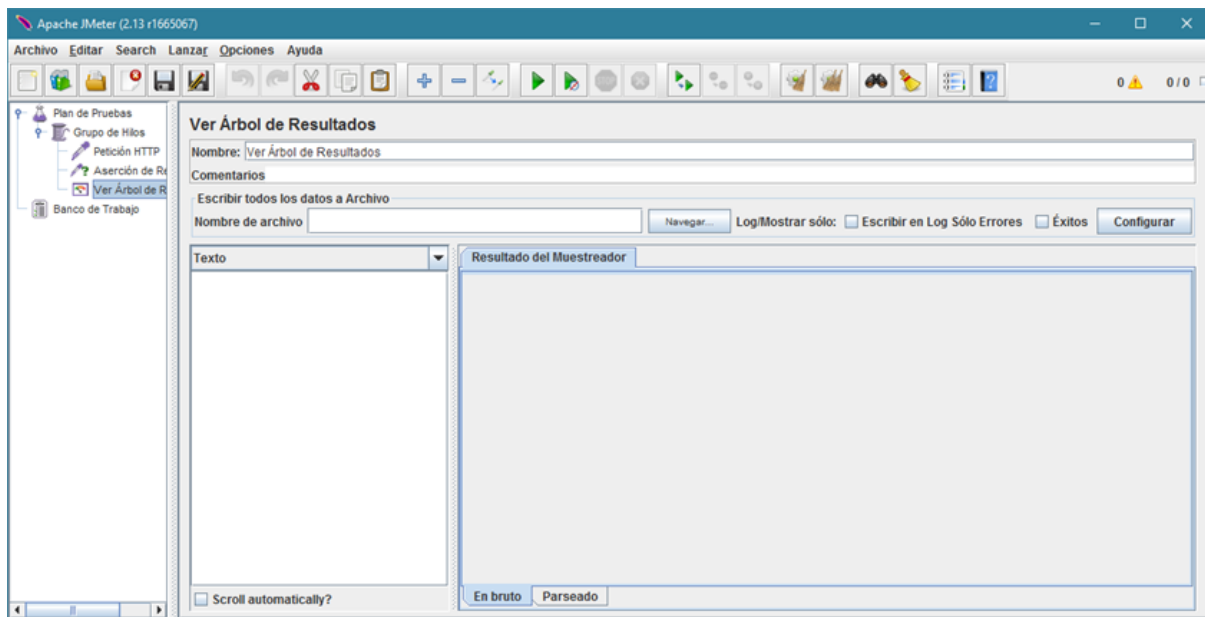


- Petición JDBC. Nos permite ejecutar una consulta sobre la Base de Datos.
- Petición WebServices (SOAP). Permite realizar peticiones SOAP a un servicio web, empleando el WSDL.
 - Aserciones. Comprobación de resultados esperados. Algunas de las aserciones disponibles son
- Aserción de respuesta. Nos permite comprobar si alguno de los campos de la respuesta coincide con un determinado patrón, OJO!! es la respuesta no el HTML.



- Aserción de esquema XML. Valida que la respuesta cumple el esquema indicado mediante un fichero XSD.
- Aserción XML. Comprueba que el resultado es un XML bien construido.
- Aserción de Tamaño. Nos permite comprobar si alguno de los campos de la respuesta tiene un tamaño determinado.
 - Elementos de configuración. Trabaja en conjunto con los Muestreadores para modificarlos.
 - Controladores lógicos. Modifican la lógica de lo que se debe hacer (Toma de decisiones).
 - Temporizadores. Incluye pausas en el test, para simular la realidad.
 - Receptores o Listeners. Muestran los resultados de las peticiones en distintos formatos. Para mostrar los resultados se debe añadir un Listener al plan de pruebas. Tenemos entre otros los siguientes tipos:

- Árbol de resultados. Para cada petición muestra la respuesta HTTP, la petición y los datos HTML devueltos.



- Informe agregado. Muestra un resumen de los resultados
- Gráfico de resultados. Muestra un gráfico de rendimiento

3.6. Banco de trabajo

El Banco de Trabajo es el lugar donde tengamos nuestras herramientas que nos ayudaran a configurar nuestro Plan de Pruebas, una de las herramientas mas interesantes que tendremos será el **Servidor Proxy HTTP**, esta herramienta nos permitirá obtener los pasos seguidos en una navegación, es decir es capaz de traducirnos a acciones de JMeter, los pasos que hacemos en una navegación, para posteriormente poder grabarlos como macro de JMeter.

Los que se genera en el **banco de trabajo**, solo esta disponible mientras JMeter esta arrancado.

Para utilizar el **Servidor Proxy HTTP**, debemos seguir los siguientes pasos:

- Creamos en el Banco de Trabajo un elemento Servidor Proxy HTTP.
- Configuramos el controlador objetivo, es decir donde queremos que vaya poniendo los pasos que se vayan a seguir en la navegación.
- Establecemos los patrones a incluir o excluir en la navegación, es posible que no interese que se almacenen imágenes, javascript, ...

- Arrancamos el Proxy.
- Configuramos el navegador a utilizar, para que pase a través de este proxy.
- Realizamos la navegación.
- Paramos el Proxy.

3.7. Jmeter Maven Plugin

Este plugin permite ejecutar los test generados con JMeter en una fase de Maven, la documentación se puede encontrar [aquí](#)

Se ha de añadir el plugin, seleccionando la fase en la que se quiere ejecutar, las habituales serán **integration-test** y **verify**.

```
<plugin>
  <groupId>com.lazerycode.jmeter</groupId>
  <artifactId>jmeter-maven-plugin</artifactId>
  <version>1.10.1</version>
  <executions>
    <execution>
      <id>jmeter-tests</id>
      <phase>verify</phase>
      <goals>
        <goal>jmeter</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

Además se han de incluir los ficheros **jmx** generados con JMeter en el directorio **<Project Dir>/src/test/jmeter**

Se pueden añadir plugins a la ejecución

```

<plugin>
  <groupId>com.lazerycode.jmeter</groupId>
  <artifactId>jmeter-maven-plugin</artifactId>
  <version>1.10.1</version>
  <executions>
    <execution>
      <id>jmeter-tests</id>
      <phase>verify</phase>
      <goals>
        <goal>jmeter</goal>
      </goals>
      <configuration>
        <jmeterPlugins>
          <plugin>
            <groupId>kg.apc</groupId>
            <artifactId>jmeter-plugins</artifactId>
          </plugin>
        </jmeterPlugins>
      </configuration>
    </execution>
  </executions>
  <dependencies>
    <dependency>
      <groupId>kg.apc</groupId>
      <artifactId>jmeter-plugins</artifactId>
      <version>1.0.0</version>
    </dependency>
  </dependencies>
</plugin>

```

Se puede encontrar una aplicación de ejemplo [aquí](#)

3.8. Jenkins Performance Plugin

Es un plugin de Jenkins, que permite incorporar los resultados obtenidos con JMeter en la UI de Jenkins.

Una vez añadido, habrá que definir un nuevo **Post Procesor** a la tarea de tipo **Publicar Informes de Test de rendimiento**, donde se han de establecer **patrón de búsquedas** a `*/.jtl`

Y posteriormente otro nuevo **Post Procesor** de tipo **Guardar los archivos generados**, donde se han de establecer **Ficheros para guardar** a `**/*jtl-report.html`

Herramientas de pruebas Java

Jenkins > JMeter Demo > Configuración

Añadir un nuevo paso ▾

Acciones para ejecutar después.

Publicar informes de tests de rendimiento

Informes de Rendimiento

JMeter

Patrón de búsqueda: */*.jtl

Borrar

Añadir un nuevo informe ▾

Select mode:

Use Error thresholds on single build:

Relative Threshold ☒ Error Threshold

Inestable 0

Fallido 0

Avanzado...

Use Relative thresholds for build comparison:

Unstable % Range 0.0 0.0

Failed % Range 0.0 0.0

Compare with previous Build ☒ Compare with Build number 0

Compare based on Average Response Time ▾

Performance display

☒ Performance Per Test Case Mode

☐ Show Throughput Chart

Borrar

Guardar los archivos generados

Ficheros para guardar */jtl-report.html

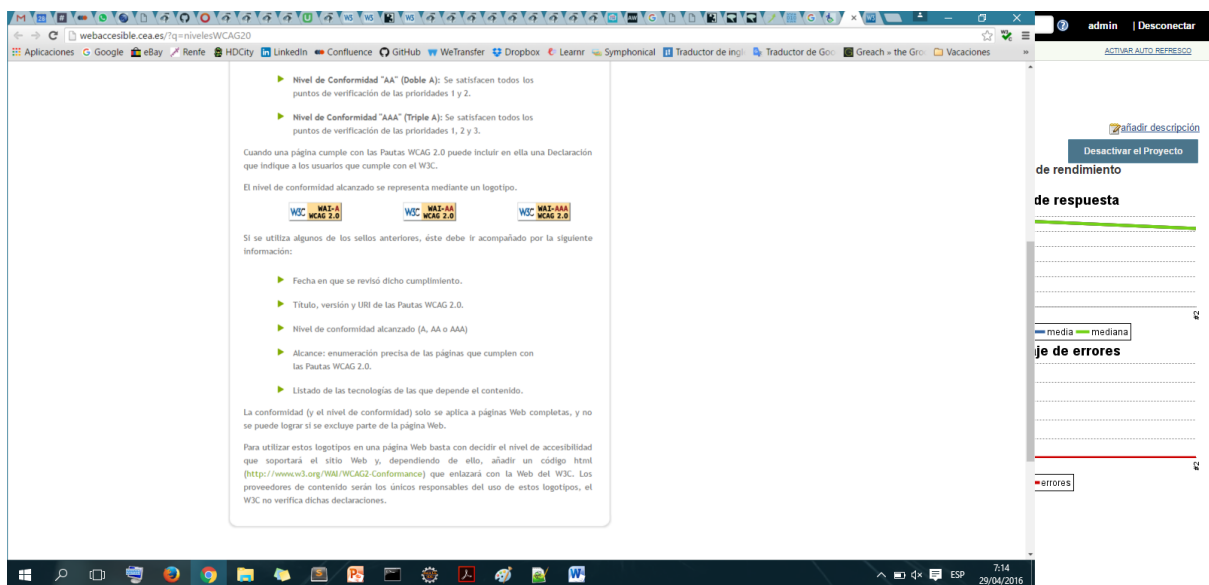
Avanzado...

Borrar

Añadir una acción ▾

Guardar Aplicar los cambios

Una vez se lanza la tarea, se obtienen nuevos datos en la vista de la tarea como **Tendencia de rendimiento**



4. Calidad estática del código

4.1. 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? 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.

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.

4.2. 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.

Emplearemos herramientas que incluyen

- Analizadores léxicos y sintácticos que procesan el código fuente.
- 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.

Se deberian realizar análisis estáticos del código cada vez que se crea una nueva funcionalidad, así como cuando el desarrollo se complica, nos cuesta implementar algo que supuestamente debe ser sencillo.

4.3. PMD

Detecta patrones de posibles errores que pueden aparecer en tiempo de ejecución, por ejemplo

- 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
- También incluye detección de CopyPaste (CPD)

La pagina de referencia [aquí](#)

Los patrones que se emplean se encuentran catalogados en distintas categorías, se pueden consultar [aquí](#)

Se pueden añadir nuevas reglas o configurar las que ya se incluyen en caso de que esto fuera necesario.

Se dispone de un plugin de Maven para la generación de reportes

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-pmd-plugin</artifactId>
      <version>3.6</version>
      <configuration>
        <linkXref>true</linkXref>
      </configuration>
    </plugin>
  </plugins>
</reporting>
```

Este plugin también puede ser configurado como plugin de la fase de Test

```

<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-pmd-plugin</artifactId>
      <version>3.6</version>
      <configuration>
        <failOnViolation>true</failOnViolation>
        <failurePriority>2</failurePriority>
        <minimumPriority>5</minimumPriority>
      </configuration>
      <executions>
        <execution>
          <phase>test</phase>
          <goals>
            <goal>pmd</goal>
            <goal>cpd</goal>
            <goal>cpd-check</goal>
            <goal>check</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>

```

Con los parametros

- **failOnViolation** → le indicamos que si hay fallos, haga que la fase de Test falle, supeditamos el exito de los Test al analisis de PMD
- **failurePriority** → le indicamos a partir de que prioridad se considera fallo, las prioridades van de 1 a 5, siendo 1 la maxima y 5 la menor, si se define por ejemplo 2, solo se consideran las reglas con prioridad 1 y 2.
- **minimumPriority** → Minima prioridad de las reglas a evaluar.

Y los goal

- **pmd** → Ejecuta las reglas de pmd
- **cpd** → Ejecuta las reglas de cpd
- **cpd-check** → Chequea los resultados de cpd

- **check** → Chequea los resultados de pmd

4.4. Checkstyle

Inicialmente se desarrolló con el objetivo de crear una herramienta que permitiese comprobar que el código de las aplicaciones se ajustase a los estándares dictados por **Sun Microsystems**.

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.

La página de referencia [aquí](#)

Dispone de un plugin de Maven

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <version>2.9.1</version>
    </plugin>
  </plugins>
</reporting>
```

4.5. Findbugs

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

Tiene una serie de categorías que catalogan los errores

- malas prácticas
- mal uso del lenguaje
- internacionalización
- posibles vulnerabilidades
- mal uso de multihilo
- rendimiento

- seguridad, ...

La página de referencia [aquí](#) y la descripción de los bugs [aquí](#)

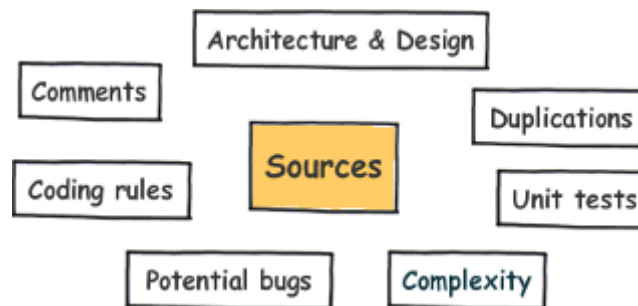
Hay un plugin de maven

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>findbugs-maven-plugin</artifactId>
      <version>2.5.2</version>
    </plugin>
  </plugins>
</reporting>
```

5. Sonarqube

5.1. Introducción

Herramienta que centraliza otras herramientas que analizan la calidad estática del código de un proyecto. Cubre 7 ejes principales de la calidad del software



Ofrece información sobre

- Cobertura
- Complejidad ciclomatica.
- Buenas practicas.

A traves de herramientas como

- Checkstyle
- PMD

- FindBugs

Hay disponible una demo con APIs conocidas [aquí](#)

También hay un grupo español, que ofrece información [aquí](#)

Algunos de los plugins mas interesantes de Sonar

- PDF Export. Plugin que permite generar pdf con la info de Sonar
- Motion Chart. Plugin que permite mostrar graficos en movimiento con la evolucion de las metricas
- Timeline. Plugin que visualiza el historico de las metricas
- Sonargraph. Plugin enables you to check and measure the overall coupling and the level of cyclic dependencies
- Taglist. Plugin handles Checkstyle ToDoComment rule and Squid NoSonar rule and generates a report.

5.2. Instalación

Descargar la distribución de [aquí](#)

Configurar la base de datos en el fichero

```
SONAR_HOME/conf/sonar.properties
```

Estos son los posibles valores para mysql, de no configurarse se empleará una base de datos Derby, no recomendable para entornos de producción.

```
sonar.jdbc.url:
jdbc:mysql://localhost:3306/sonar?useUnicode=true&characterEncoding=utf8
sonar.jdbc.driverClassName: com.mysql.jdbc.Driver
sonar.jdbc.validationQuery: select 1
sonar.jdbc.username=sonar
sonar.jdbc.password=sonar
```

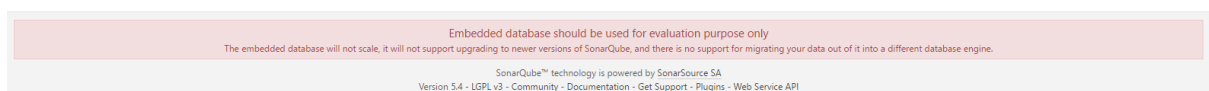
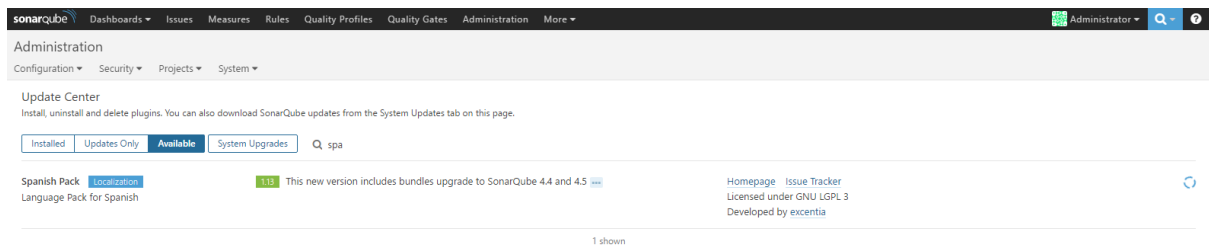
Arrancar el servidor con el comando para el sistema operativo correspondiente que se encuentra en

```
SONAR_HOME/bin/<sistema operativo>/<comando>
```

Esta opción arranca Sonar en el puerto **9000**, el puerto también se puede configurar en el anterior fichero de configuración.

Existe un usuario administrador creado por defecto con **admin/admin**

Se puede instalar un plugin para el idioma español, para ello acceder a **Administration/System/Update Center/Available Plugins** y buscar el **Spanish Pack**



5.3. Conceptos

Deuda técnica: Es un cálculo basado únicamente en **reglas** y **evidencias**. La deuda técnica en Sonar, se calcula con la metodología SQALE (Software Quality Assessment based on Lifecycle Expectations). Se mide en días.

Reglas: Representan aquellos puntos que se desean vigilar en los proyectos. Se pueden definir con un perfil de calidad. Se pueden obtener más reglas a partir de nuevos Plugins. Se pueden definir nuevas reglas basadas en plantillas.

Evidencias: Son los incumplimientos de las Reglas de calidad que se presentan en el código. Tendrán asociada una severidad. Con las evidencias se puede hacer:

Comentar, Asignar, Planificar, Confirmar, Cambiar Severidad, Resolver y Falso Positivo. Todas estas tareas se pueden realizar de forma individual o conjunta. Se pueden definir evidencias manuales.

5.4. Organización

La interface web de sonar, se divide en tres partes

- Menu superior
- Menu lateral
- Zona de visualizacion de datos

En el menu superior aparecen los siguientes items

- Cuadros de mando para volver en cualquier momento a la página de inicio
- Proyectos para acceder al listado completo de proyectos, vistas, desarrolladores, etc. o para acceder de forma rápida a proyectos recientemente accedidos
- Medidas, permite definir consultas sobre las medidas, se pueden guardar para visualizarlas en un cuadro de mando.
- Evidencias para acceder al servicio de evidencias
- Reglas para acceder a la página de reglas
- Perfiles navegar y gestionar perfiles de calidad
- Configuración para acceder a la configuración del sistema (acceso restringido a administradores de sistema)
- Conectarse / <Nombre> para conectarse con tu usuario. Dependiendo de tus permisos de usuario, tendrás acceso a diferentes servicios y cuadros de mando. Autenticarte en el sistema te permitirá tener acceso a tu propia interfaz web personalizada. Desde aquí puedes modificar tu perfil y desconectarte.
- Buscar un componente: proyecto, fichero, vista, desarrollador, etc. para acceder rápidamente a él. Pulsa 's' para acceso directo a la caja de búsqueda.

El menú lateral ira cambiando sus opciones dependiendo del área en la que nos encontremos, de los permisos de usuario y de las extensiones que se hayan incorporado en la instalación. Proporciona acceso a diferentes cuadros de mando y servicios.

5.5. Carga de datos

Para cargar los datos de un proyecto, se puede hacer de varias formas, la mas habituales son

- A través de un plugin de Maven.

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>sonar-maven-plugin</artifactId>
  <version>2.7</version>
</plugin>
```

Y su goal

```
mvn sonar:sonar
```

Para la seleccion de un perfil de Sonar a emplear, se ha de indicar el parametro **sonar.profile**

```
-Dsonar.profile="Mi Perfil"
```

Si se desea publicar la cobertura en Sonar, se ha de seguir los pasos que se pueden encontrar en <https://github.com/SonarSource/sonar-examples/blob/master/projects/languages/java/code-coverage>

- A través del plugin de jenkins sonarqube.

5.6. Gestion de Usuarios y Seguridad

Se pueden añadir nuevos usuarios, grupos, definir permisos a nivel global o de proyecto, desde **Administration/Security/Users**

5.7. Cuadro de mando

Los cuadros de mando, son los componentes principales de Sonar, ya que son los que nos permiten configurar que información de que proyecto queremos visualizar y como visualizarlo.

Se organizan en columnas, pudiendo seleccionar entre 5 distribuciones distintas.

Se dividen en **Widget**, habiendo **Widget** orientados a distintos propositos, si se busca un Widget concreto se pueden filtrar los Widget mostrados por categorias.

Los Widget a parte de mostrar información, permiten acceder a vistas mas avanzadas, ya que en general los Widget ofrecen resúmenes de la información.

Existen Widget que ofrecen información sobre

- Tamaño de los ficheros
- Bloques duplicados
- Mala distribución de la complejidad
- Código Spaguetti
- Falta de pruebas unitarias
- Cumplimiento de estándares y defectos potenciales
- Contabilización de comentarios
- Eventos en cuanto a la calidad.
- Treemap
- Evidencias y deuda técnica.
- Pirámide de deuda técnica. Muestra la deuda técnica organizada de abajo arriba por prioridad en su resolución.
- Resumen de deuda técnica. Ofrece un Ratio entre lo que se necesita invertir para solventar la deuda técnica y lo que se necesita invertir para crear el proyecto desde cero.

Herramientas de pruebas Java

sonarqube

Cuadros de mandoEvidenciasMedidasReglasPerfilesUmbralConfiguraciónMás

Administrator

Mi Cuadro de mando

Category: **Cualquier** Filters History Hotspots Issues Technical Debt Tests

Alertas

Mostrar alertas del proyecto.

Añade un widget

Descripción

Muestra información general relativa a un proyecto.

Añade un widget

Bienvenido

Mensaje de bienvenida para proporcionar enlaces a los recursos más valiosos como documentación y soporte.

Añade un widget

Documentación y Comentarios

Informa sobre comentarios y documentación del código.

Añade un widget

Cobertura de código

Muestra los resultados de la ejecución de tests y de su cobertura.

Añade un widget

Duplicados

Informa sobre copiar/pegar y duplicados en el código.

Añade un widget

Cobertura tests de integración

Informa de la cobertura de código de los tests de integración.

Añade un widget

Eventos

Muestra los eventos ocurridos en la vida de un proyecto como versiones o alertas.

Añade un widget

Complejidad

Muestra la complejidad total, media y su distribución.

Añade un widget

Evidencias y deuda técnica.

Muestra información de las evidencias y la deuda técnica.

Añade un widget


Search:

Embedded database should be used for evaluation purpose only

The embedded database will not scale, it will not support upgrading to newer versions of SonarQube, and there is no support for migrating your data out of it into a different database engine.

SonarQube™ technology is powered by SonarSource SA

Version 5.4 - LGPL v3 - Community - Documentation - Get Support - Plugins - Web Service API



Preface | 55

texto fijo con lo que quieras =====