

S06 - Pruebas de Integración

Objetivo: encontrar **DEFECTOS** en la **INTERACCIÓN** de las unidades probadas.

Cuestión fundamental: ¿cuál es el “orden” en el que vamos a realizar dicha integración?

Tipos de interfaces y errores comunes

- Interfaz a través de parámetros
- Memoria compartida
- Interfaz procedural
- Paso de Mensajes

Errores más comunes:

- Mal uso de la interfaz
- Malentendido sobre la interfaz
- Errores temporales

Estrategias de Integración

- **Big Bang:** integrar TODAS las unidades (a la vez) sólo cuando el tamaño de nuestra aplicación es muy “pequeño” o tiene pocas unidades.
- **Top-down:** integramos de mayor a menor abstracción (interfaz de usuario (compleja) → backend → BD), necesario implementar dobles.
- **Bottom-up:** integramos primero de menor a mayor nivel de abstracción (BD → backend → lógica de negocio (compleja) → interfaz)
- **Sándwich:** es una mezcla de las dos estrategias anteriores.
- **Dirigida por los riesgos:** elegir primero los componentes con mayor riesgo (con más probabilidad de errores por su complejidad).
- **Dirigida por las funcionalidades / hilos de ejecución:** Se ordenan las funcionalidades y se integran según ese orden.

Hola,

el módulo común NO tiene pruebas unitarias porque lo único que contiene son objetos TO (con getters-setters).
Por otro lado, cuando hacemos una estrategia de integración Top-Down, vamos a necesitar dobles, que habrá que indicar.

Por lo tanto:

BO + DAO --> te sobra, este paso

Paso 1: BO + DAO + COMUN + doble de Proxy --> no necesitamos el doble de común ya que común no tiene pruebas unitarias.

Paso 2: BO + DAO + PROXY + COMUN

También se podría haber hecho en orden inverso: añadiendo primero proxy (en cuyo caso necesitaríamos el doble de dao).

saludos,

eli

Pruebas de regresión

Las pruebas de integración se realizan de forma **incremental**, añadiendo cada vez un determinado número de unidades al conjunto, hasta que al final tengamos probadas TODAS las unidades integradas. En cada una de las "**fases**" tendremos que **REPETIR** TODAS las **pruebas anteriores**. A este proceso de "repetición" se le denomina PRUEBAS DE REGRESIÓN. (Los **últimos componentes** integrados son siempre los **MENOS PROBADOS**).

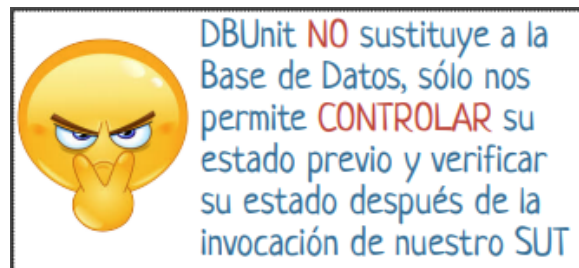
DbUnit

Proporciona una solución "elegante" para controlar la dependencia de las aplicaciones con una base de datos.

- Permite gestionar el estado de una base de datos durante las pruebas.
- Permite utilizar con JUnit.

Escenario típico:

1. Eliminar estado previo de la BD resultante de pruebas anteriores (ANTES de ejecutar cada test).
2. Cargar los datos necesarios para las pruebas en la BD.
3. Ejecutar las pruebas utilizando métodos de la librería DbUnit para las aserciones.



Interfaz Itable

Itable representa a una sola tabla se usa para comparar entre tablas de la BD.

Implementación:

- DefaultTable - ordenación por clave primaria
- SortedTable - proporciona una vista ordenada de la tabla
- ColumnFilterTable - permite filtrar columnas de la tabla original

Interfaz IDataset

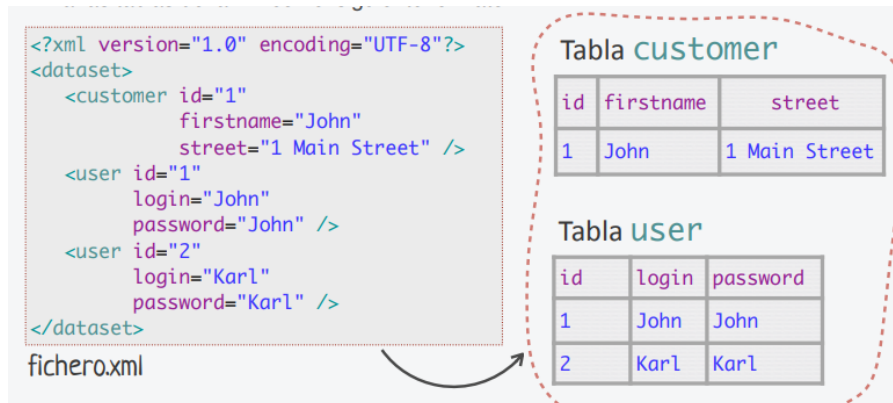
Representa a un conjunto de tablas, se usa para simular una BD y comprobar su estado.

Implementación:

- FlatXmlDataSet - lee/escribe datos en formato xml
- QueryDataSet - guarda colecciones de datos resultantes de una query

Clase FlatXmlDataSet

FlatXmlDataSet permite crear datasets a partir de un XML



Ejemplos

```
public class ClienteDAO_IT {

    private ClienteDAO clienteDAO; //SUT
    private IDatabaseTester databaseTester;
    private IDatabaseConnection connection;

    @BeforeEach
    public void setUp() throws Exception {

        String cadena_conexionDB = "jdbc:mysql://oldbox.cloud:32769/DBUNIT?useSSL=false";
        databaseTester = new MiJdbcDatabaseTester("com.mysql.cj.jdbc.Driver",
            cadena_conexionDB, "root", "ppss");
        //obtenemos la conexión con la BD
        connection = databaseTester.getConnection();

        clienteDAO = new ClienteDAO();
    }

    @Test
    public void testInsert() throws Exception {
        Cliente cliente = new Cliente(1, "John", "Smith");
        cliente.setDireccion("1 Main Street");
        cliente.setCiudad("Anycity");

        //Inicializamos el dataSet con los datos iniciales de la tabla cliente
        IDataset dataSet = new FlatXmlDataFileLoader().load("/cliente-init.xml");
        //Inyectamos el dataset en el objeto databaseTester
        databaseTester.setDataSet(dataSet);
        //inicializamos la base de datos con los contenidos del dataset
        databaseTester.onSetup();
        //invocamos a nuestro SUT
        Assertions.assertDoesNotThrow(() -> clienteDAO.insert(cliente));

        //recuperamos los datos de la BD después de invocar al SUT
        IDataset databaseDataSet = connection.createDataSet();
        //Recuperamos los datos de la tabla cliente
        ITable actualTable = databaseDataSet.getTable("cliente");

        //creamos el dataset con el resultado esperado
        IDataset expectedDataSet = new FlatXmlDataFileLoader().load("/cliente-esperado.xml");
        ITable expectedTable = expectedDataSet.getTable("cliente");

        Assertion.assertEquals(expectedTable, actualTable);
    }
}
```

```

@Test
public void testRetrieve() throws Exception {

    Cliente cliente = new Cliente(1, "John", "Smith");
    cliente.setDireccion("1 Main Street");
    cliente.setCiudad("Anycity");

    // Inicializamos la BD
    // Inicializamos el dataSet con los datos iniciales de la tabla cliente
    IDataset dataSet = new FlatXmlDataFileLoader().load("/cliente-init_ej1d.xml");
    // Inyectamos el dataset en el objeto databaseTester
    databaseTester.setDataSet(dataSet);
    // Inicializamos la base de datos con los contenidos del dataset
    databaseTester.onSetup();

    // invocamos a nuestra SUT --> Assertions = junit
    Cliente clienteReal = Assertions.assertDoesNotThrow(() -> clienteDAO.retrieve(1));

    // comprobar cliente
    Assertions.assertAll(
        () -> Assertions.assertEquals(cliente.getId(), clienteReal.getId()),
        () -> Assertions.assertEquals(cliente.getNombre(), clienteReal.getNombre()),
        () -> Assertions.assertEquals(cliente.getApellido(), clienteReal.getApellido()),
        () -> Assertions.assertEquals(cliente.getDireccion(), clienteReal.getDireccion()),
        () -> Assertions.assertEquals(cliente.getCiudad(), clienteReal.getCiudad())
    );

    // recuperamos los datos de la BD después de invocar al SUT
    IDataset databaseDataSet = connection.createDataSet();
    // recuperamos los datos de la tabla cliente
    ITable actualTable = databaseDataSet.getTable("cliente");

    // creamos el dataset con el resultado esperado
    IDataset expectedDataSet = new FlatXmlDataFileLoader().load("/cliente-init_ej1d.xml");
    ITable expectedTable = expectedDataSet.getTable("cliente");

    // comparamos tablas y generamos informe --> Assertion = dbunit
    Assertion.assertEquals(expectedTable, actualTable);
}
}

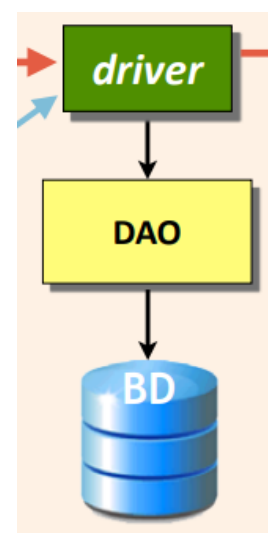
```

Capa de acceso a datos DAOs

Las clases DAO son aquellas que se encargan de acceder a la BD

- **Interfaces**

Permite que los cambios en la implementación de la fuente de datos subyacente NO afecten a los componentes de negocio y especifica las operaciones que se pueden realizar sobre la BD. Operaciones CRUD



- **JDBC (Los DataAccessObject)**

Cada JDBC implementa los DAO de la interfaz asociada. Cada DAO implementa una interfaz (operaciones CRUD sobre la BD)

- **FactoriaDAO**

Usamos una clase factoría que proporciona los DAOs utilizados por los objetos de la capa de negocio

pre-integration-test: se ejecutan acciones previas a la ejecución de los tests

integration-test: se ejecutan los tests de integración. Deben tener el prefijo o sufijo IT. Si algún test falla NO se detiene la construcción

post-integration-test: en esta fase “detendremos” todos los servicios o realizaremos las acciones que sean necesarias para volver a restaurar el entorno de pruebas

verify: en esta fase se comprueba que todo está listo (no hay ningún error) para poder copiar el artefacto generado en nuestro repositorio local (fase install)

Si algún test ha fallado, se detiene la construcción

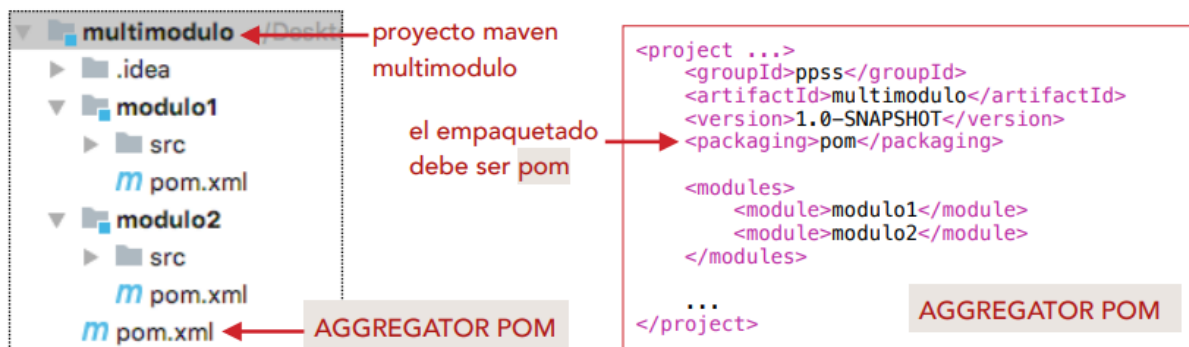


Las fases pre-integration-test ... verify, por defecto NO tienen asociada ninguna goal cuando el empaquetado del proyecto es jar, por lo que tendremos que incluir y configurar los plugins necesarios en el pom del proyecto

Proyecto Multimódulo

Un proyecto multimódulo es aquel que posee diferentes partes (módulos) que se encargan de hacer cosas. La ventaja es la reducción de duplicación.

Un proyecto maven multimódulo contiene una lista de módulos “agregados” denominado “aggregator pom”



En cada módulo “hijo” añadiremos las coordenadas del proyecto “padre”, del cual heredarán su configuración

```

<project ...>
  <parent>
    <artifactId>multimodulo</artifactId>
    <groupId>ppss</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <artifactId>modulo1</artifactId>
  ...
</project>

```

modulo1/pom.xml

```

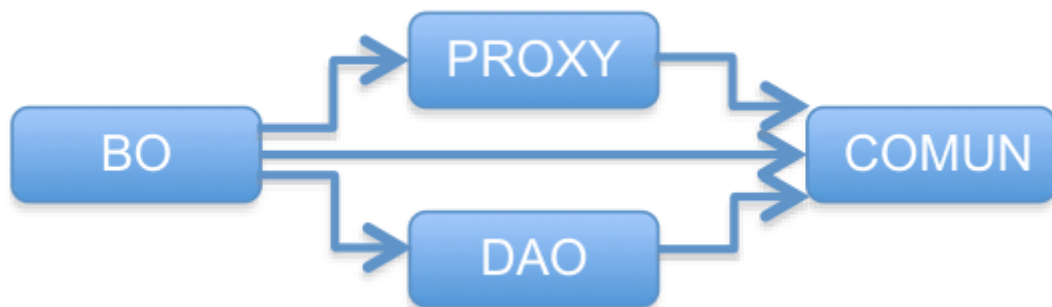
<project ...>
  <parent>
    <artifactId>multimodulo</artifactId>
    <groupId>ppss</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>

  <artifactId>modulo2</artifactId>
  ...
</project>

```

modulo2/pom.xml

En este proyecto multimódulo tenemos las siguientes dependencias:



Por ejemplo $BO \rightarrow PROXY$, se “lee” como el módulo matriculación-bo depende del módulo matriculación-proxy.

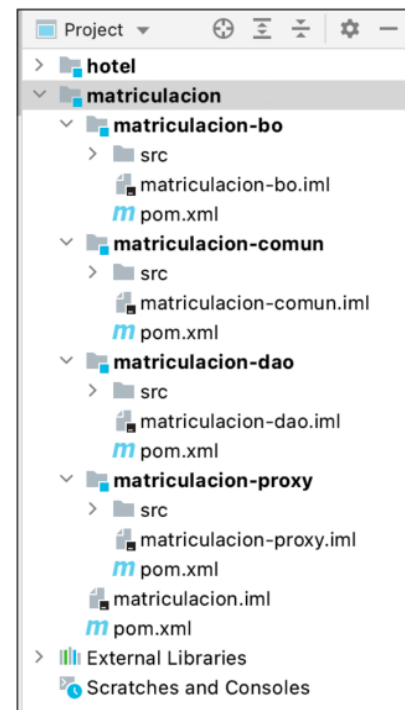
En cada módulo hijo, introduciremos en su pom las siguientes líneas, indicando cual es su padre:

```

<parent>
  <artifactId>matriculacion</artifactId>
  <groupId>ppss</groupId>
  <version>1.0-SNAPSHOT</version>
</parent>
<modelVersion>4.0.0</modelVersion>

```

En el POM quedaria asi:



- Matriculación

```
<modelVersion>4.0.0</modelVersion>
<groupId>ppss</groupId>
<artifactId>matriculacion</artifactId>
<version>1.0-SNAPSHOT</version>
  <modules>
    <module>matriculacion-comun</module>
    <module>matriculacion-dao</module>
    <module>matriculacion-proxy</module>
    <module>matriculacion-bo</module>
  </modules>
  <packaging>pom</packaging>
</name>matriculacion</name>
```

- BO

```
<dependencies>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>matriculacion-comun</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>matriculacion-dao</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>matriculacion-proxy</artifactId>
    <version>${project.version}</version>
  </dependency>
</dependencies>
```

- DAO

```
<dependencies>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>matriculacion-comun</artifactId>
    <version>${project.version}</version>
  </dependency>
</dependencies>
```

- PROXY

```
<artifactId>matriculacion-proxy</artifactId>
<dependencies>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>matriculacion-comun</artifactId>
    <version>${project.version}</version>
  </dependency>
</dependencies>
```

- COMÚN no tendría ninguna dependencia en el POM

S07 - Pruebas del sistema y aceptación 01

ACEPTACIÓN: su objetivo es valorar en qué grado el software desarrollado satisface las expectativas del cliente. REQUIERE los CRITERIOS DE ACEPTACIÓN.

- **VALIDACIÓN**
- Los casos de prueba se diseñan desde el punto de vista del **USUARIO** (CLIENTE)
- **Criterios de aceptación:** deben satisfacerse para ser aceptados por el cliente.
 - Pruebas de aceptación:
 - **User acceptance testing (UAT):** dirigidas por el cliente para asegurar que el sistema satisface los criterios de aceptación contractuales (pruebas α y pruebas β).
 - $\alpha \rightarrow$ en el lugar de desarrollo y para usuarios “conocidos”
 - $\beta \rightarrow$ para usuarios “anónimos”
 - **Business acceptance testing (BAT):** dirigidas por la organización que desarrolla el producto. Ensayo de las UAT en el lugar de desarrollo.

Diseño de pruebas de aceptación

- Deberían ser responsabilidad de un grupo separado de pruebas que no esté implicado en el proceso de desarrollo.
- Ejemplos de métodos de diseño de pruebas emergentes funcionales:
 - Diseño de pruebas **basado en requerimientos:** son pruebas de validación (se trata de demostrar que el sistema ha implementado de forma adecuada los requerimientos y que está preparado para ser usado por el usuario)
 - **Diseño de pruebas de escenarios:** los escenarios describen la forma en la que el sistema debería usarse.

SISTEMA: su objetivo es encontrar DEFECTOS derivados del COMPORTAMIENTO del sw como un todo. REQUIERE los REQUISITOS (funcionalidades) del sistema.

- **VERIFICACIÓN**
- Son pruebas **dinámicas** y obtienen los casos de prueba a partir de **caja negra**
- Se realizan durante el proceso de desarrollo y se prueba:
 - Si los componentes son **compatibles**
 - Si **interaccionan** correctamente
- Las diferencias con las pruebas de integración son:
 - Se incluyen componentes desarrollados por “terceros”
 - Los comportamientos a probar son los especificados para el sistema en su conjunto. (sistema de compra on-line, se prueba el proceso "completo")
- Ejemplo de métodos de diseño del sistema:
 - Método de diseño **basado en casos de uso**
 - Probar las interacciones entre componentes
 - muestran qué entradas se requieren y qué salidas deben producirse entre los componentes
 - Es necesario fijar políticas de pruebas para elegir el proceso de pruebas más eficiente

- Método de diseño de **transición de estados**
 - Se usa en sistemas con ESTADO
 - A partir de la especificación se obtiene una representación en forma de grafo. Dicho grafo modela los estados de una entidad del sistema (Los nodos son estados y las aristas las transiciones)
 - A partir del grafo, se selecciona un **conjunto de caminos mínimo** para conseguir un objetivo concreto

Propiedades Emergentes Funcionales: Selenium

Las pruebas de propiedades emergentes funcionales tienen como objetivo el comprobar que el sistema ofrece la FUNCIONALIDAD esperada por el cliente.

- ✦ **Selenium WebDriver** (API): Permiten crear tests robustos, que pueden escalarse y distribuirse en diferentes entornos
- ✦ **Selenium IDE** (extensión de un navegador: Firefox, Chrome). Permite crear scripts de pruebas utilizando la aplicación web tal y como un usuario haría normalmente: a través del navegador.
- ✦ **Selenium Grid**: es un servidor proxy que permite ejecutar tests en paralelo usando múltiples máquinas y diferentes navegadores.

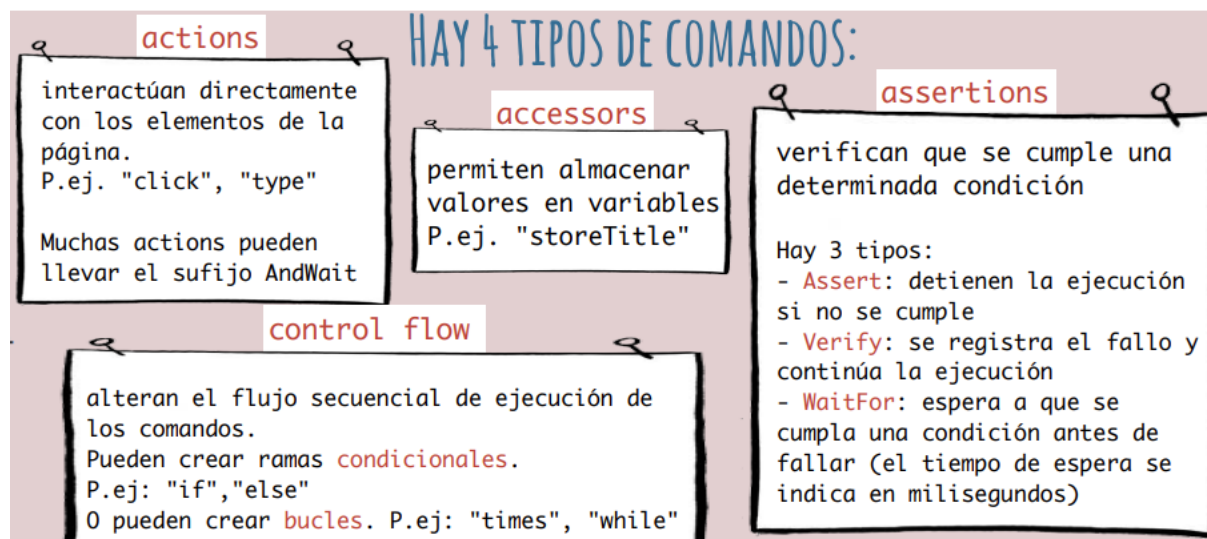


Comandos Selenese

COMMAND + [TARGET] + [VALUE]

- **target**: hace referencia a un elemento HTML de la página a la que se accede. (p.ej "link" indica un hiperenlace)
- **value**: contiene un texto, patrón, o variable a introducir en un campo de texto o para seleccionar una opción de una lista de opciones

Una secuencia de comandos Selenese forman un **"test script"** (caso de prueba). Una secuencia de tests scripts forman una **test suite**.



Locators

Utilizados en el campo **target** e identifican un elemento en el código de una página web (un botón, cuadro de texto ...) locatorType=location.

Tipos de locators:

- **id**: el id del elemento html
- **name**: el atributo nombre del elemento html
- **xpath**: lenguaje utilizado para localizar nodos en un documento HTML
 - rutas absolutas: xpath=/html/body/form[1]
 - rutas relativas: xpath=//form[1]

Command	Target	Value
open	http://demo.guru99.com/test/newtours/	
verifyText	//tr[4]/td/table/tbody/tr[2]/td/font	User*
verifyElementPresent	name=password	
open	http://demo.guru99.com/test/facebook.html	
type	id=email	hola

Control Flow

Son comandos de control de flujo

Command	Target	Value
execute script	return "a"	myVar
if	\${myVar} === "a"	
execute script	return "a"	output
else if	\${myVar} === "b"	
execute script	return "b"	output
else		
execute script	return "c"	output
end		
assert	output	a

Existe un comando assert comprobar valores

P07-Selenium

AUTOMATIZACIÓN DE LAS PRUEBAS DE ACEPTACIÓN CON SELENIUM IDE.

- Generamos los casos de prueba a partir de la selección de diferentes escenarios y/o requerimientos de nuestra aplicación. Selenium IDE NO es un lenguaje de programación, es una **herramienta que nos permite generar scripts** (de forma automática o manual), que podemos ejecutar desde un navegador, por lo tanto sólo podremos usar esta herramienta si nuestra aplicación tiene una interfaz web.
- Aunque Selenium IDE no es un lenguaje de programación, en las últimas versiones se han incorporado **comandos para controlar el flujo de ejecución de nuestros scripts** (incluyendo por ejemplo acciones condicionales y bucles.
- **No es posible integrar Selenium IDE en una herramienta de construcción de proyectos**, lo que constituye una limitación frente a otras aproximaciones.

S08 - Pruebas de aceptación 02

Selenium WebDriver

- Proporciona un buen control del navegador a través de implementaciones específicas para cada uno de ellos
- Permite realizar una programación más **flexible** de los tests

WebElements

Los webs elements son los elementos html de la web. Para localizar un elemento lo haremos con “**WebElement driver.findElement(By by)**” podemos localizar por multitud de parametros (By.name(), By.id(), By.tagName(), By.className(), By.linkText(), By.partialLinkText(), By.xpath(), By.cssSelector())

Acciones sobre los WebElement

- **sendKeys**(secuencia de caracteres)
 - Se utiliza para introducir texto en elementos textbox o textarea
- **clear()**
 - se utiliza para borrar texto en elementos textbox o textarea
- **submit()**
 - Envía el formulario de la página web
- **click()**
 - botón izquierdo del ratón
- **click**(WebElement onElement)
 - Pulsación del botón izquierdo del ratón sobre un WebElement

- `doubleClick()`
 - doble click con botón izquierdo
- `contextClick()`
 - botón derecho del ratón
- `getAttribute()`
- `getLocation()`
- `getText()`
- `isDisplayed()`
- `isEnabled()`
- `isSelected()`

```
driver.findElement(By.name("firstname")).sendKeys("Ejercicio1");
driver.findElement(By.cssSelector("input[name='lastname']")).sendKeys("Ejercicio1");
driver.findElement(By.xpath("//*[@id='email_address']")).sendKeys("Ejercicio1@ppss.com");
driver.findElement(By.id("password")).sendKeys("Ejercicio1");
driver.findElement(By.cssSelector("button[title='Register']")).click();
```

Tiempos de espera

- Tiempo de espera **implícito**: es **común a todos los WebElements** y tiene asociado un timeout global para todas las operaciones del driver

```
@BeforeEach
public void start(){
    driver=new ChromeDriver();
    driver.manage().timeouts().implicitlyWait(Duration.ofSeconds(10));
    driver.get("http://demo-store.seleniumacademy.com");
}
```

- Tiempo de espera **explícito**: se establece de forma **individual** para cada WebElement

```
...
//Create Wait using WebDriverWait.
//This will wait for 10 seconds for timeout before
//title is updated with search term
WebDriverWait wait = new WebDriverWait(driver, 10);
wait.until(ExpectedConditions.titleContains("selenium"));
...
```

timeout explícito

Si el elemento se carga antes del límite especificado se cancela el timeout

Múltiples acciones

Podemos indicar a WebDriver que realice varias acciones

1. Invocar la clase Actions para agrupar las acciones
2. Construir la acción (Action)
3. Realizar (ejecutar) la acción compuesta

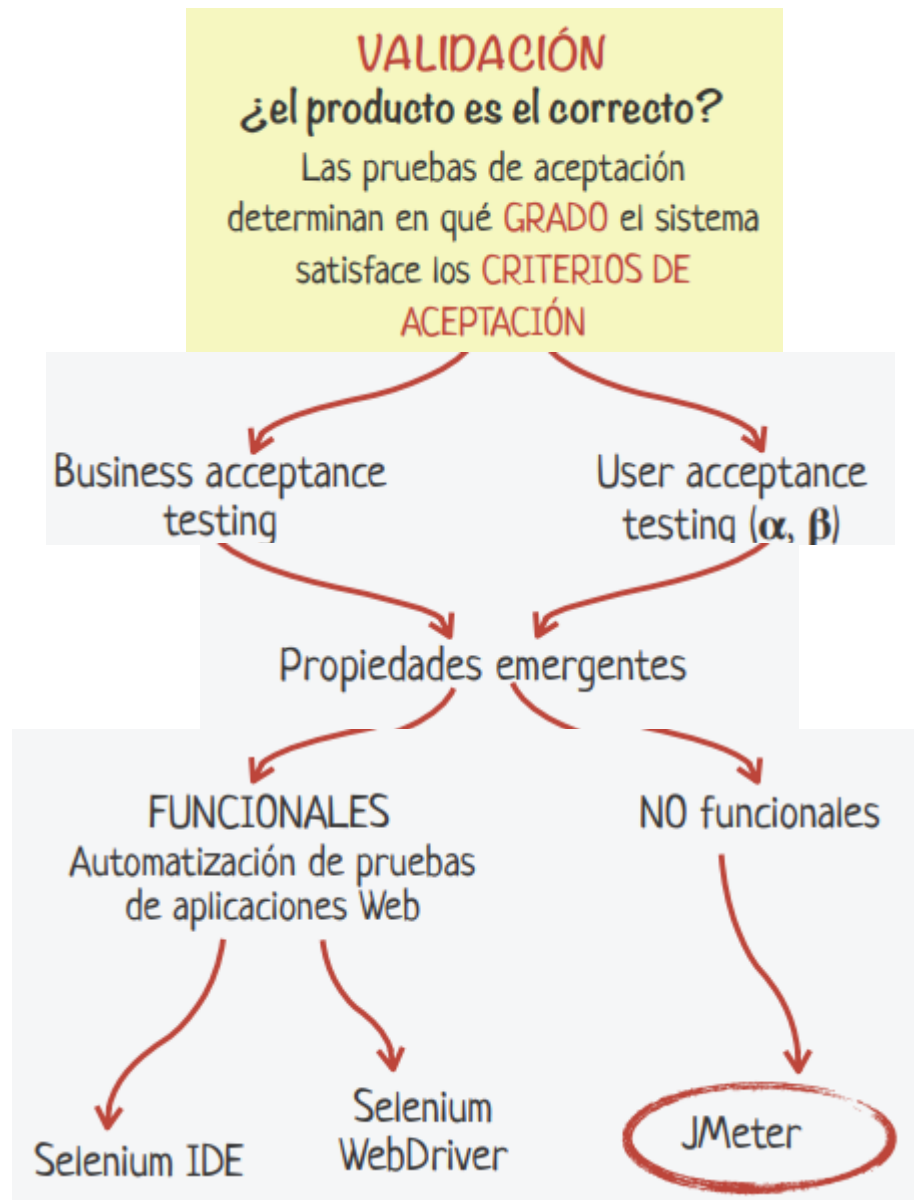
```
WebDriver driver = new ChromeDriver();
driver.get("http://www.example.com");
WebElement one = driver.findElement(By.name("one"));
WebElement three = driver.findElement(By.name("three"));
WebElement five = driver.findElement(By.name("five"));
// Add all the actions into the Actions builder
Actions builder = new Actions(driver); (1)
// Generate the composite action
Action compositeAction = builder.keyDown(Keys.CONTROL)
    .click(one)
    .click(three) (2)
    .click(five)
    .keyUp(Keys.CONTROL)
    .build();
// Perform the composite action.
compositeAction.perform(); (3)
```

Page Object Model (POM)

Para facilitar la **mantenibilidad** de los tests y **no duplicar código** es útil el patrón de diseño "Page Object Model" (POM) que consiste en crear un test para cada **página web** es una **clase** de tal forma que sus **atributos** serán los **WebElements** de la página correspondiente, y sus **métodos** serán todos los **SERVICIOS** que nos proporciona la **página**

VER EJERCICIO 2 PRÁCTICA 8

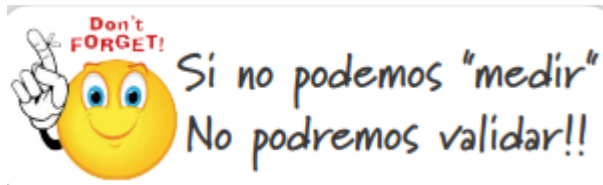
S09 - Pruebas de aceptación 03



Propiedades emergentes no funcionales hacen referencia a cómo de bien cumplen con sus requerimientos funcionales:

- **Fiabilidad:**
 - Probabilidad de funcionar sin fallos durante un periodo de tiempo.
- **Disponibilidad:**
 - Tiempo en el que da servicio
- **Mantenibilidad:**
 - Capacidad de soportar cambios
- **Escalabilidad:**
 - Capacidad de albergar nuevos usuarios sin afectar al rendimiento
- **Robustez:**
 - Capacidad de seguir funcionando frente a fallos

Las pruebas de aceptación determinan en qué GRADO el sistema satisface los CRITERIOS DE ACEPTACIÓN, éstos deben incluir propiedades emergentes “cuantificables”.



Para estimar la **fiabilidad** se utilizan pruebas aleatorias. Utilizan métricas:

- MTTF (Mean Time To Failure)
- MTTR (Mean Time To Repair)
- MTBF = MTTF + MTTR (MTBF: Mean Time Between Failures)

Para estimar la **disponibilidad** se utiliza la métrica MTTR para medir el “downtime” del sistema.

Para estimar la **mantenibilidad** se utiliza la métrica MTTR (que refleja el tiempo consumido en analizar un defecto correctivo, diseñar la modificación, implementar el cambio, probarlo y distribuirlo).

La **escalabilidad** del sistema utiliza el número de transacciones (operaciones) por unidad de tiempo.

Ejemplos de pruebas

- **Pruebas de carga:** validan el **rendimiento** de un sistema.
- **Pruebas de stress:** forzar peticiones al sistema por encima de su límite.
 - Las pruebas de stress comprueban la **fiabilidad** y **robustez** del sistema.
- Para evaluar la **fiabilidad** de un sistema podemos usar las pruebas estadísticas.
 - Construir un “perfil operacional”
 - Generar un conjunto de datos de prueba que reflejen lo anterior.
 - Probar dichos datos midiendo el número de fallos y el tiempo entre fallos.
- Para evaluar el **rendimiento**, necesitamos:
 1. **Identificar los criterios de aceptación no funcionales** (tiempos de respuesta, fiabilidad, utilización de recursos...)
 2. **Diseñar los tests** basados en **escenarios reales** de nuestra aplicación
 3. **Preparar el entorno de pruebas** para que sea lo **más realista posible**
 4. **Automatizar las pruebas** grabando escenarios de prueba
 5. **Analizaremos los resultados** y realizaremos los cambios oportunos para conseguir nuestro objetivo

Consideraciones

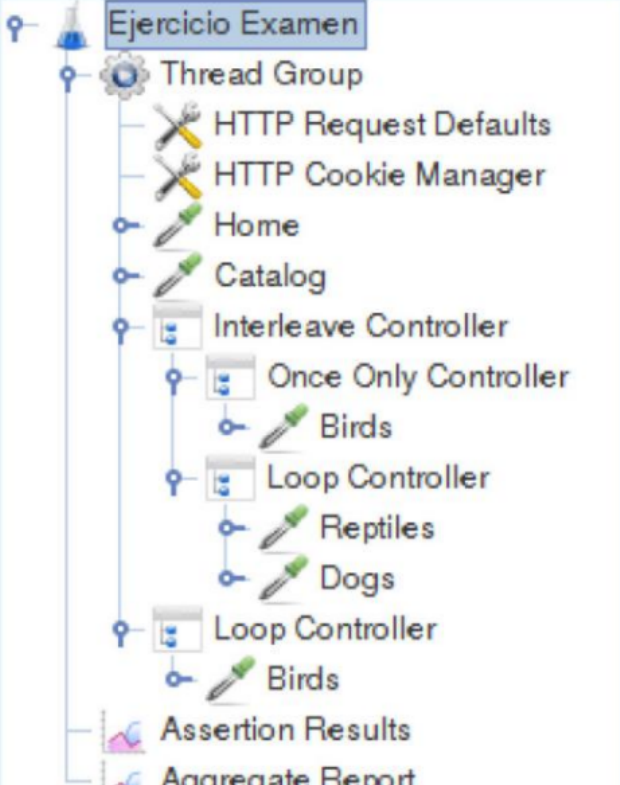
- No hay que dejar las pruebas de rendimiento para el final del proyecto.
- Las propiedades emergentes NO funcionales están condicionadas fundamentalmente por la ARQUITECTURA del sistema.
- La arquitectura del sistema se determina en las primeras fases del desarrollo.
- Con una buena estrategia de pruebas combinada con una arquitectura software que considere el rendimiento desde el inicio del desarrollo se minimizan problemas.

Ejercicios

Dado el siguiente plan JMeter que es ejecutado por un sólo usuario con las siguientes configuraciones:

Thread Properties
Loop Count : 3

En ambos Loop Controller
Loop Count : 2



Handwritten analysis of the JMeter plan:

H, C, B, B, B
H, C, R, D, R, D, B, B
H, C, R, D, R, D, B, B

Indica cuántas veces se ejecuta la petición Birds:

Seleccione una:

☐ Dejo la pregunta en blanco

☐ 1

☐ 3

☐ 5

☒ 7 ✓

A partir de este informe de JMeter obtenido con la siguiente configuración del elemento Thread Group:

Number of Threads (users) : 1

Loop Count : 3

Indica en qué instante (expresado en segundos) termina su ejecución el usuario:

Label	# Samp...	Average	Median	90% Li...	95% Li...	99% Li...	Min	Maxim...	Error %	Throug...	Receiv...	Sent K...
Home	1	5	5	5	5	5	5	5	0.00%	200.0/...	263.67	25.20
Catalog	1	12	12	12	12	12	12	12	0.00%	83.3/sec	478.68	12.29
Reptiles	35	13	10	32	33	57	1	57	0.00%	42.7/sec	146.32	10.93
Dogs	35	8	7	14	18	37	1	37	0.00%	43.5/sec	173.66	10.95
Birds	28	12	6	34	53	55	1	55	0.00%	75.1/sec	256.87	18.99
TOTAL	100	11	7	32	34	55	1	57	0.00%	82.5/sec	299.10	20.73

Seleccione una:

- ☐ 2,2
- ☐ Dejo la pregunta en blanco
- ☒ 3,3
- ☐ 1,1
- ☐ no se puede saber porque falta en la configuración el valor del Ramp-up

Se realiza $100 * 11 = 1100$ ms, 1,1 s, loop counts 3 por lo tanto $1,1 * 3 = 3,3$. El Ramp-up no nos importa en este caso porque es solo 1 usuario

S10 - Análisis de pruebas

¿Qué es una métrica?

Se define como una medida cuantitativa del grado en el que un sistema, componente o proceso, posee un determinado atributo

- Si **no** podemos medir → **no** podemos saber si alcanzamos nuestros **objetivos**.
 - **no** podremos **controlar** el proceso software.
 - **no** podremos **mejorarlo**.

Se puede medir cualquier cosa: líneas de código probadas, número de errores, número de pruebas realizadas,...

SIEMPRE buscamos la **efectividad**.

Dos causas que pueden provocar que nuestras pruebas **no sean efectivas**:

- Ejecutar el **código** y que esté **pobrementemente probado o no probado** de ninguna manera
 - Difícil de detectar de forma automática.
- De forma “deliberada”, **dejamos de probar partes de nuestro código**.
 - Si no se ejecuta el código → **no** se está probando.
 - En el problema de la **COBERTURA*** de código, es decir en analizar cuál es la EXTENSIÓN de nuestras pruebas

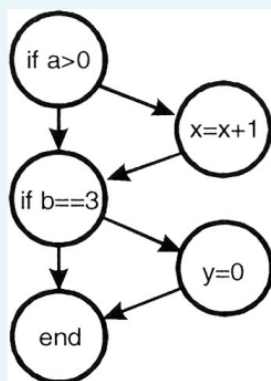
* (Cuánto código estamos probando, **NO** proporciona un indicador la **calidad** del **código**, ni la **calidad** de nuestras **pruebas**, sino de la **extensión** de nuestros **tests**)

7 formas de cuantificar la cobertura del código

[NIVEL 1] Cobertura de líneas

Un 100% significa que hemos **ejecutado cada línea de código**

```
if (a>0) {  
    x=x+1;  
}  
if (b==3) {  
    y=0;  
}
```



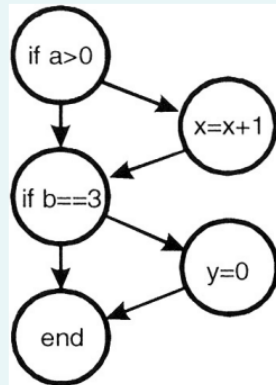
- ❑ Para este código podemos conseguir un **100% de cobertura de líneas con un único caso de prueba** (por ejemplo a=6, y b=3). Sin embargo estamos **dejando de probar 3 de los cuatro caminos posibles**.
- ❑ Un 100% de cobertura de líneas **no suele ser un nivel aceptable de pruebas**. Podríamos clasificarlo como de NIVEL 1
- ❑ A pesar de constituir el nivel más bajo de cobertura, en la práctica **puede ser difícil de conseguir**

[NIVEL 2] Cobertura de ramas

Un 100% significa que hemos ejecutado cada rama o DECISIÓN en sus vertientes verdadera y falsa.

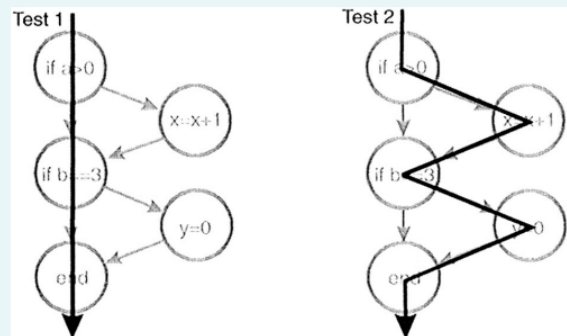
- En caso de un switch hay que ejecutar cada case.
- 100% cobertura de ramas IMPLICA 100% de cobertura de líneas.

```
if (a>0) {  
    x=x+1;  
}  
if (b==3) {  
    y=0;  
}
```



Un 100% de cobertura de ramas (decisiones) implica un 100% de cobertura de líneas

□ Para este código podemos conseguir un 100% de cobertura de ramas con dos casos de prueba (por ejemplo a=0, b=2; y a=4, b=3). Sin embargo estamos dejando de probar 2 de los cuatro caminos posibles.



[NIVEL 3] Cobertura de condiciones

Un 100% de cobertura de condiciones NO garantiza un 100% de cobertura de líneas

DIFERENCIA ENTRE condiciones y ramas.

Con la cobertura de **condiciones** se prueba **CADA CONDICIÓN INDIVIDUAL** en su vertiente verdadera y falsa; en cambio, con la cobertura de **ramas** se prueba cada bifurcación, si una bifurcación tiene varias condiciones se prueba el **CONJUNTO** (el if al completo), no individualmente.

```
if (a>0 & c==1) {  
    x=x+1;  
}  
if (b==3 | d<0) {  
    y=0;  
}
```

Nivel 3 (CONDICIÓN)

- ♣ (a=7, c=1, b=3, d = -3)
[V, V, V, F]
- ♣ (a=-4, c=2, b=5, d = 6)
[F, F, F, V]

Nivel 2 (DECISIÓN)

- ♣ [(a=7, c=1); (b=3, d = -3)]
[V; V]
- ♣ [(a=-4, c=1), (b=5, d = -3)]
[F; F]

(No se prueba la vertiente falsa de: "c==1")

[NIVEL 4] Cobertura de condiciones y decisiones (Condition decision coverage)

Un 100% de cobertura de condiciones múltiples (nivel 5) puede no tener sentido, por ello ejercitaremos **cada condición** en el programa y cada **decisión** en el programa

```
if(x & y) {  
    sentenciaCond;  
}
```

❖ En este ejemplo podemos conseguir el nivel 3 (cobertura de condiciones), con dos casos: (x =TRUE, y = FALSE) y (x=FALSE, y =TRUE). Pero observa que en este caso "sentenciaCond" nunca será ejecutada. Podemos ser más completos si buscamos también una cobertura de decisiones

❑ Podemos conseguir el nivel 4 creando casos de prueba para cada condición y cada decisión:

- ❖ (x=TRUE, y=FALSE),
- ❖ (x=FALSE, y=TRUE),
- ❖ (x=TRUE, y=TRUE)

W2
W3
W4
F
V

Ej.: if((A > 0) & (B > 0))

- **Nivel 2:** true; false (A nivel de if)
- **Nivel 3:** true & true; false & false (A nivel de condición individual)
- **Nivel 4:** true & false; false & true; **true & true** (condiciones individuales + decisiones)

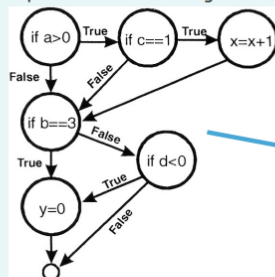
[NIVEL 5] Cobertura de condiciones múltiples

Un 100% de cobertura significa que hemos ejercitado **todas las posibles combinaciones de condiciones**.

Un 100% de cobertura de condiciones múltiples **implica** un 100% de cobertura de condiciones, decisiones, y condiciones/decisiones (Todos los inferiores excepto nivel 1), **NO IMPLICA** un 100% de cobertura de caminos.

```
if (a>0 & c==1) {  
    x=x+1;  
}  
if (b==3 | d<0) {  
    y=0;  
}
```

operador && de java



Un 100% de cobertura de condiciones múltiples implica un 100% de cobertura de condiciones, decisiones, y condiciones/decisiones

❖ En este ejemplo con código Java (sin cortocircuito), este nivel de cobertura podemos conseguirlo con cuatro casos de prueba:

- ▶ a=5, c=1, b=3, d=-3
- ▶ a=-4, c=1, b=3, d=7
- ▶ a=5, c=2, b=5, d=-4
- ▶ a=-1, c=2, b=5, d=0

❖ Si el lenguaje evalúa las condiciones en "cortocircuito" (&&, ||) podemos conseguir un 100% de cobertura de condiciones múltiples, considerando cómo el compilador evalúa realmente las condiciones múltiples de una decisión.

- ▶ a=5, c=1, b=5, d=0
- ▶ a=5, c=2, b=5, d=-4
- ▶ a=-1, c=2, b=3, d=-3

❖ Si conseguimos un 100% de cobertura de condiciones múltiples, también conseguimos cobertura de decisiones, condiciones y condiciones+decisiones

❖ Una cobertura de condiciones múltiples no garantiza una cobertura de caminos

En el de cortocircuito son 3 porque no hace falta probar el caso de F && V de la primera condición debido al cortocircuito, lo mismo con el V || F.

[NIVEL 6] Cobertura de bucles

Un 100% de cobertura implica probar el bucle con 0 iteraciones, una iteración y múltiples iteraciones

[NIVEL 7] Cobertura de caminos

Un 100% de cobertura implica que se han probado todos los posibles **caminos de control** (difícil si hay bucles)

Un 100% de cobertura de caminos **implica** un 100% de cobertura de bucles, ramas/decisiones y sentencias/líneas

Un 100% de cobertura de caminos **NO garantiza** una cobertura de condiciones múltiples (ni al contrario)

Diferencia entre decisión y condición, según ELI

Hola Raul,

en una cobertura de RAMAS se prueba cada DECISIÓN (se llama cobertura de ramas o decisiones).
Una decisión puede estar formada por varias condiciones.

Ejemplo de nivel 2 (DECISIONES) --> DECISIONES Y RAMAS ES LO MISMO!!!:

(100% de decisiones)

(a=7, c=1, b=3, d = -3)

[V, V, V, V]

(a=-4, c=2, b=5, d = 6)

[F, F, F, F]

Ejemplo de nivel 3 (CONDICIONES)

(100% condiciones)

(a=7, c=11, b=3, d = -31)

[V, F, V, V]

(a=-4, c=1, b=5, d = -3)

[F, V, F, V]

Estos dos casos de prueba NO consiguen un 100% de cobertura de condiciones, ya que la condición ($d < 0$) sólo la pruebas en su vertiente Verdadera.

saludos,

eli

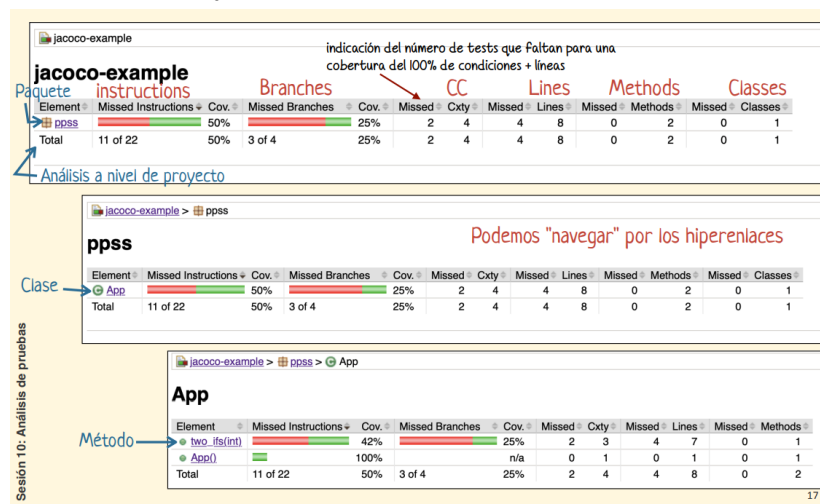


JACOCO

Sirve para recopilar **información sobre la cobertura** conseguida al ejecutar un cierto número de tests.

Jacoco genera un **informe** con las siguientes **métricas**:

- **Instructions** → Número de instrucciones de Java
- **Branches** → Para cada if y switch, **IMPORTANTE** Jacoco llama BRANCHES tanto a DECISIONES como a CONDICIONES, junta todo en una celda (try y catch NO se considera como condiciones)
- **Complejidad Ciclomática (CC)** → Para cada método no abstracto, número máximo de casos de prueba para cubrir todos los caminos independientes.
- **Líneas** → Número de líneas ejecutadas según bytecode de Java
- **Métodos** → Número de métodos no abstractos que contienen una instrucción como mínimo.
- **Clases** → Número de clases ejecutadas, se considera ejecutada si se llama al menos a un método suyo.



¿Do you remember?

(1) $CC = \text{arcos} - \text{nodos} + 2$

(2) $CC = \text{número de condiciones} + 1$

(3) $CC = \text{número de regiones cerradas en el grafo, incluyendo la externa}$

Como JACOCO calcula el CC

(4) $CC = B - D + 1$

B → Número de **branches**: Las aristas que corresponden a las vertientes V y F de una condición.

D → Número de **decisions points**: El nodo que contiene un if/switch y genera que el programa se "ramifique".

¿Para qué sirve el CC?

- Mide la complejidad lógica del código
 - A mayor valor, el código resulta más difícil de mantener y probar → Mayor riesgo de introducir nuevos errores.
 - Si $CC > 15$ → Puede que toque refactorizar para mejorar la mantenibilidad del código.
- Es el número máximo de tests, garantiza la ejecución de TODAS las líneas y TODAS las condiciones en su vertiente verdadera y falsa

PLUGIN DE JACOCO

La instrumentación tiene lugar **on-the-fly** usando un mecanismo denominado **Java Agent**.

- **jacoco:prepare-agent**

- Se asocia a la fase “**initialize**”, por defecto
- Indica qué clases van a ser instrumentadas y cuál será el fichero en el que se guardan los resultados: **target/jacoco.exec**

- **jacoco:prepare-agent-integration**

- Se asocia a la fase “**pre-integration-test**”, por defecto
- Los resultados de análisis se guardan en: **target/jacoco-it.exec**

- **jacoco:report**

- Se asocia a la fase “**verify**”, por defecto
- Genera un informe en formato html, xml y cvs en el directorio: **target/site/jacoco** en base a la información de **target/jacoco.exec**

- **jacoco:report-integration** (igual pero sustituyendo **jacoco** por **jacoco-it**)

- Se asocia a la fase “**verify**”, por defecto
- Genera un informe en formato html, xml y cvs en el directorio: **target/site/jacoco-it** en base a la información de **target/jacoco-it.exec**

- **jacoco:check**

- Comprueba si se ha alcanzado un determinado **valor de cobertura**, si no se alcanza dicho valor → Detiene la construcción
 - Por defecto se asocia la fase “**verify**” y se usan los datos de **target/jacoco.exec**
 - **DIFERENTES NIVELES<element>**: BUNDLE, PACKAGE, CLASS, SOURCEFILE o METHOD
 - **CONTADOR** para cada nivel (**<counter>**): INSTRUCTION, LINE, BRANCH, COMPLEXITY, METHOD, CLASS
 - **Establecer valores máximos y mínimos**, para cada contador(**<value>**): TOTALCOUNT, COVEREDCOUNT, MISSEDCOUNT, COVEREDRATIO, MISSEDRATIO
- (Hay que usar **mvn verify** para llamar a estas ejecuciones)

```
<plugin>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.8</version>

  <executions>
    <execution>
      <id>default-prepare-agent</id>
      <goals>
        <goal>prepare-agent</goal>
      </goals>
    </execution>
    <execution>
      <id>default-report</id>
      <goals>
        <goal>report</goal>
      </goals>
    </execution>
    <execution>
      <id>default-check</id>
      <goals>
        <goal>check</goal>
      </goals>
      <configuration>
        <rules>las reglas se añaden a jacoco:check
        <rule>
          <element>BUNDLE</element> A nivel de paquete
          <limits>
            <limit> En el contador de COMPLEXITY (CC)
              <counter>COMPLEXITY</counter>
              <value>COVEREDRATIO</value>
              <minimum>0.60</minimum>
            </limit>
          </limits>
        </rule>
      </rules>
    </configuration>
  </executions>
</plugin>
```

Preparamos la instrumentación y el análisis de cobertura

Generamos el informe en formato html (a partir del fichero jacoco.exec)

Establecemos unos niveles de cobertura

podemos incluir todas las "reglas" que consideremos necesarias

La construcción se detendrá si, a nivel de proyecto, la complejidad ciclomática no alcanza el 60%

S11 - Análisis de pruebas

Planificación **predictiva** vs **adaptativa**

- Los modelos iterativos y ágiles realizan una planificación adaptativa.
- No es posible tener claras todas las “variables” que intervienen en el desarrollo de un proyecto al comienzo del mismo (**cono de incertidumbre**).
- Un **plan ADAPTATIVO** intenta encontrar un equilibrio entre el esfuerzo invertido en realizar el plan, frente a la información (conocimiento) disponible en cada momento. Sencillo incluir cambios.
- **Predictiva** soporta muy mal los cambios!!!

Diferentes **niveles de planificación**: cada modelo de proceso considera determinados **horizontes**:

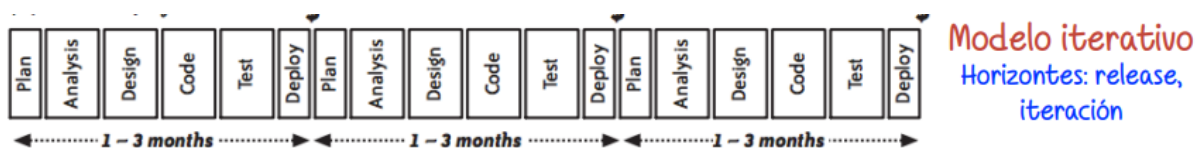
- Los modelos ágiles planifican como mínimo a nivel de día, iteración y release.

Un plan con muy poca exactitud es incontrolable. No podemos “ver más allá del horizonte”. (no te flipes haciendo predicciones a futuro que no sabes si va a cumplir)



- Modelo secuencial (cascada)
 - Horizonte: release
- Modelo iterativo
 - Horizonte: release, iteración
- Modelo ágil (**XP**)
 - Horizontes: release, iteración, día

Con una aproximación iterativa al equipo le cuesta menos **REPARAR LOS FALLOS** ya que hay una retroalimentación mucho más frecuente. (Este no necesariamente implica una mayor productividad).



En un modelo iterativo es **FUNDAMENTAL** que las iteraciones sean del **mismo “tamaño”** y que sean **time-boxed**. Este modelo, generalmente, está conducido por el cliente, de esta forma puede validar sus expectativas cuanto antes. Las iteraciones deben ser **SIEMPRE** time-boxed: nunca se retrasa el tiempo de entrega, es preferible cambiar el “scope”.

Planificar con una iteración como una mini-cascada implica que no podremos cambiar el “scope” sin repercutir negativamente en el éxito del proyecto.

Alcance y efectividad de las pruebas

Es importante tener en cuenta:

- Cuántas pruebas son suficientes
- Para conseguir resultados hay que:
 - priorizar tests
 - Saber cuando parar
- Asegurar las pruebas efectivas
- y tiene que ser efectivo y eficiente



Proceso de pruebas

1. ¿QUÉ?, ¿CÓMO?, ¿QUIÉN? y ¿CUÁNDO? (planning) y a QUÉ se va a hacer si los planes no se ajustan a la realidad (control)
2. Determinar casos de prueba
3. Implementar casos de prueba

NIVELES DE PRUEBAS *Tenemos que integrar las actividades de pruebas con el resto de actividades de desarrollo en el plan del proyecto*

Secuencia temporal de niveles de pruebas (DINÁMICAS)

	VERIFICACIÓN. Objetivo: encontrar defectos Realizada por los desarrolladores			VALIDACIÓN. Objetivo: ver en qué grado se satisfacen las expectativas del cliente. Requieren al usuario
	UNIDAD	INTEGRACIÓN	SISTEMA	ACEPTACIÓN
OBJETIVO (cuantificable)	Encontrar defectos en las unidades. Deben de probarse de forma AISLADA	Encontrar defectos en la interacción de las unidades. Debe establecerse un ORDEN de integración	Encontrar defectos derivados del comportamiento del sistema como un todo.	Valorar en qué GRADO se satisfacen las expectativas del cliente. Basadas en criterios de ACEPTACIÓN (prop. emergentes) cuantificables
DISEÑO	Camino básico (CB) Particiones equivalentes (CN)	Guías (consejos) en función de los tipos de interfaces (CN)	Basado en casos de uso (CN) Transición de estados (CN)	Basado en requerimientos (CN) Basado en escenarios (CN) Pruebas de rendimiento (CN) Pruebas α y β (CN)
AUTOMATIZACIÓN (implement. + ejecución)	Java, Maven, JUnit, EasyMock	Java, Maven, JUnit, DbUnit, EasyMock	Java, Maven, JUnit, DbUnit, Webdriver	Java, Maven, JUnit, Webdriver: JMeter Usuario (α y β)

en un plan de pruebas hay que decidir QUÉ, CÓMO, CUÁNDO Y QUIÉN, para cada celda de la tabla (actividades de diseño + implementación)

Aspecto de un plan XP

Story cards: contienen historias de usuario. Características requeridas por el usuario.

entrega = release

Cada TASK tiene asociado un conjunto de TESTS (en el reverso de la tarjeta)

La Metodología XP

XP define cuatro variables para cualquier proyecto de software: **costo, tiempo, calidad y alcance** de las cuales solo 3 de ellas pueden ser fijas. Por ejemplo, si el cliente establece el alcance y la calidad, y el jefe de proyecto el precio, el grupo de desarrollo tendrá libertad para determinar el tiempo que durará el proyecto. **Se trata de establecer un equilibrio** entre las cuatro variables del proyecto.

Su ciclo de vida es el siguiente:

- Entender lo que el cliente necesita > Fase de **Exploración**
- Estimar el esfuerzo > Fase de **Planificación**
- Crear la solución > Fase de **Iteraciones**
- Entregar el producto final al cliente > Fase de puesta en **producción**

Lo que lo distingue de otras metodologías (al igual que otras ágiles) son sus ciclos de desarrollo cortos, en cada iteración se realiza un ciclo completo de análisis, diseño, desarrollo y pruebas

TDD vs BDD

TDD es una práctica de pruebas utilizada en desarrollos ágiles, p.e XP.

- Paso 1. Escribir una prueba para el nuevo código y ver cómo falla
- Paso 2. Implementar el nuevo código, haciendo “la solución más simple que pueda funcionar”
- Paso 3. Comprobar que la prueba es exitosa y refactorizar el código.

Con TDD el **diseño de la aplicación** “evoluciona” a medida que vamos desarrollando la aplicación.

Classical vs Mockist TDD (BDD)

- Classical:
 - diseño INSIDE-OUT (comenzar por los componentes de bajo nivel, NO stubs).
- Mockist TDD:
 - diseño OUTSIDE-IN (comenzar por los componentes de alto nivel, SI mocks).
 - El diseño del sistema evoluciona a través de iteraciones a medida que se escriben los tests.
 - Utiliza verificación basada en el comportamiento (necesario especificar estado inicial y final del objeto a probar y la interacción con sus dependencias).
 - BDD nos permite centrarnos en el valor de negocio de nuestra aplicación.
 - BDD utiliza conversaciones basadas en ejemplos, y expresadas de forma que sean fácilmente automatizables, reduciendo así la pérdida de información y los malentendidos

Integraciones Continuas (CI)

Consiste en **INTEGRAR** el código del proyecto de forma ininterrumpida en una máquina aparte de la de cada desarrollador, la cual debe estar funcionando 24/7.

Es habitual utilizar un **Servidor de Integraciones Continuas** para automatizar todo el proceso.

Funcionamiento de un sistema de CI

CI construye el sistema lo prueba e informa

1. Desarrollador “sube” el código en el que ha estado trabajando al repositorio SCM (después de hacer un build local) y sólo si las pruebas unitarias han pasado. El servidor CI consulta el SCM para detectar cambios.
2. El CI recupera la última versión del SCM y ejecuta un script de construcción del proyecto (construcción planificada).
3. El CI genera un feedback (informe). En el momento en el que se “rompe” el sistema, hay que reparar el error.

