Sesión S01. Pruebas-Diseño de caja blanca

1-. Definición del proceso de pruebas

Las pruebas son un conjunto de actividades conducentes a conseguir algunos de estos objetivos:

- **Encontrar defectos**
- Evaluar el nivel de calidad del software
- Obtener información para la toma de decisiones
- Prevenir defectos

Las pruebas muestran la **PRESENCIA de defectos** ⇒ No pueden demostrar la ausencia de estos

2-. Actividades del proceso de pruebas

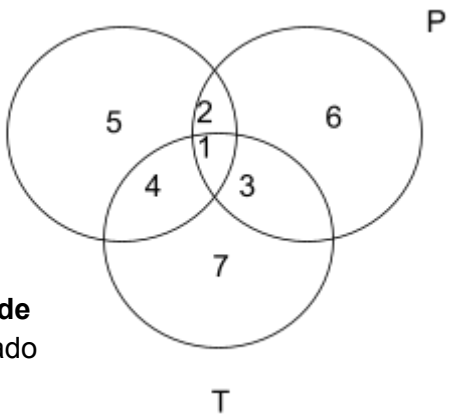
1. Planificación y control de las pruebas ⇒ Definimos los objetivos de las pruebas, asegurándonos de cumplir esos objetivos
2. Diseño de las pruebas ⇒ Consiste en decidir con qué datos de entrada concretos vamos a probar el código
3. Implementación y ejecución de las pruebas ⇒ Poder ejecutar las pruebas de forma automática
4. Evaluación del proceso de pruebas y emitir un informe ⇒ Aplicamos las métricas adecuadas para comprobar si hemos alcanzado los objetivos de prueba planificados

3-. Diseño de pruebas

Trabajaremos con técnicas de pruebas **DINÁMICAS** ⇒ requieren ejecutar el código para detectar la presencia de errores.

Dado un programa y sus especificaciones, consideraremos:

- El conjunto S de comportamientos especificados (resultado esperado) para dicho programa
- El conjunto P de comportamientos (entradas + salida esperadas) programados
- El conjunto T que refleja los casos de prueba (datos concretos de entrada + resultado esperado)




Nuestro objetivo es seleccionar el subconjunto **mínimo** de entradas para evidenciar el **máximo** número de errores posibles

El resultado del proceso de diseño es una **tabla de casos de prueba.** Cada fila contiene los datos de entrada y el resultado esperado de un comportamiento del programa.

4-. Diseño de pruebas de caja blanca

Los métodos de diseño de casos de prueba basados en el código:

1. **Analizan** el código y obtienen una representación en forma de grafo
2. **Seleccionan** un subconjunto de **caminos** en el grafo según algún criterio
3. **Obtienen** un conjunto de **casos de prueba** que ejercitan dichos caminos

Observaciones:

- Podemos detectar comportamientos no especificados, nunca podremos detectar comportamientos no implementados
- Dependiendo del método concreto utilizado, obtendremos conjuntos DIFERENTES de casos de prueba. Pero el conjunto obtenido será **EFECTIVO** y **EFICIENTE**.
- Las técnicas o métodos estructurales son costosos de aplicar, por lo que suelen usarse sólo a nivel de **UNIDADES** de programa.

4.1-. Grafo de flujo de control (CFG)

Un CFG es una **representación gráfica** de una unidad de programa. Usaremos un grafo dirigido, donde:

- Cada nodo representa una o más sentencias secuenciales y/o una ÚNICA CONDICIÓN
    - Cada nodo estará etiquetado con un entero cuyo valor será único
    - Si un nodo contiene una condición anotaremos a su derecha dicha condición
- Las **aristas** representarán el flujo de ejecución entre dos conjuntos de sentencias.
    - Si un nodo contiene una condición etiquetamos las aristas que salen del nodo con “T” o “F”.

4.2-. Criterios de selección de caminos

- **Estructuralmente** un camino es una secuencia de instrucciones en una unidad de programa
- **Semánticamente** un camino es una instancia de una ejecución de una unidad de programa
- Es necesario seleccionar un conjunto de caminos con algún criterio de selección.

4.3-. McCabe’s basis path method

Es un método de DISEÑO de pruebas de caja blanca que permite ejecutar cada **camino independiente** del programa

El objetivo del programa será ejecutar todos los caminos independientes( ⇒ Es un camino en un grafo de flujo (CFG) que difiere de otros caminos en al menos un nuevo conjunto de sentencias y/o una nueva condición ):

- Estaremos ejecutando todas las sentencias del programa, al menos una vez.

- Además estaremos garantizando que todas las condiciones se ejecuten en sus vertientes verdadero/falso

Descripción del método:

1. Construir el grafo del programa (CFG) a partir del código a probar
2. Calcular la complejidad ciclomática (CC) del grafo de flujo
3. Obtener los caminos independientes del grafo
4. Determinar los datos concretos de entrada (y salida esperada) de la unidad a probar, de forma que se estén ejecutando todos los caminos independientes.

4.3.1-. Complejidad Ciclomática (CC)

Es una métrica que proporciona una medida de la complejidad lógica de un componente software
Se puede calcular de las siguientes maneras:

- CC = número de arcos - número de nodos + 2
- CC = número de regiones
- CC = número de condiciones + 1

Estas dos últimas formas solo son aplicables si el código no tiene saltos incondicionales (break)

Observaciones:
- El valor de CC indica el MÁXIMO número de caminos independientes en el grafo ⇒ Un camino independiente aporta un nodo o una arista nueva al conjunto de caminos
- A mayor CC, mayor complejidad lógica, por lo tanto, mayor esfuerzo de mantenimiento, y también mayor esfuerzo de pruebas
- El valor máximo de CC conjuntamente aceptado como “tolerable” es 10

4.3.2-. Caminos independientes

Buscamos (como máximo) tantos caminos independientes como valor obtenido de CC
- Cada camino independiente contiene un nodo, o bien una arista, que no aparece en los caminos independientes anteriores
- Con ello recorreremos TODOS los nodos y TODAS las aristas del grafo

Es posible que con un número inferior a CC recorramos todos los nodos y todas las aristas

Sesión S02. Drivers

1-. Automatización de las pruebas

El objetivo es implementar código (drivers) para ejecutar los tests de forma automática
Para eso nuestro código de pruebas tendrá que:

- Una vez establecida las precondiciones sobre los valores de entrada + resultado esperado (1) al código a probar (SUT = System Under Test)
- Obtener el resultado real (2)
- Comparar el resultado esperado con el real (3)
- Emitir un informe que contesta a nuestra pregunta inicial (4)

Implementaremos tantos drivers como casos de prueba ⇒ Cada driver invocará a nuestra SUT y proporcionará un informe

2-. Pruebas unitarias

Una unidad de programa es una “pieza” de código, que puede ser invocada desde fuera de la unidad y puede invocar a otras unidades de programa.
El objetivo es encontrar DEFECTOS en el código de las UNIDADES probadas ⇒ Queremos probar cada unidad por separado

3-. Pruebas de unidad dinámicas: Drivers

DRIVER: conductor de la prueba. Contiene el código necesario para EJECUTAR el caso de prueba sobre SUT ( ⇒ Es el código que queremos probar)

Requieren ejecutar una unidad de forma **AISLADA** para poder detectar defectos en el código de dicha unidad
El “tamaño” de las unidades dependerá de lo que consideramos como unidad.
Cada unidad será invocada desde un driver durante las pruebas, con los datos de entrada diseñados previamente.

4-. Junit 5

Junit es una API java que permite implementar los drivers y ejecutar los casos de prueba sobre componentes (SUT) de forma automática

Junit 5 = Junit Platform + Junit Jupiter + Junit Vintage

- Junit Platform ⇒ “detección” y ejecución de los tests
- Junit Jupiter ⇒ **implementar** y ejecutar los tests
- Junit Vintage ⇒ ejecuta tests Junit 3 y Junit 4

Junit se puede utilizar para implementar drivers de pruebas **unitarias** y también de **integración:**

- En una prueba unitaria estamos interesados en probar una única unidad
- En una prueba de integración estamos probando varias unidades

Junit denomina los test a un método sin parámetros, que devuelve void, y está anotado con @Test

4.1-. Sentencias Assert (org.junit.api.assertions)

Junit proporciona sentencias (aserciones) para determinar el resultado de las pruebas y poder emitir el informe correspondiente
Son métodos estáticos, cuyas principales características son :

- Se utilizan para comparar el resultado esperado con el real
- El ordeń de los parámetros para los métodos assert… es:
    - resultado ESPERADO, resultado REAL [, mensaje opcional]
    -

Todos los métodos “assert” generan una excepción de tipo **AssertionFailedError** si la aserción no se cumple.

Si queremos ejecutar varias sentencias Assert las agrupamos con el método **assertAll** ( ⇒ Ejecuta todas las asserts que contengan fallen o no ). En el caso de no añadir el método es posible que no se ejecuten todos los asserts.

Si el resultado esperado es que nuestro SUT lance una excepción, usaremos la aserción **assertThrows()**. Si al ejecutar sut() NO se lanza la excepción de tipo ExpectedException, la ejecución del test fallará.
Si solamente queremos comprobar que se lanza la excepción esta sentencia NO es necesaria.

Si el resultado esperado es que nuestro SUT no lance ninguna excepción, usaremos la aserción assertDoesNotThrow(). No necesitamos capturar la excepción.

4.2-. Anotaciones

En el caso de que todos los tests requieran las MISMAS acciones para preparar los datos de entrada (**antes** de ejecutar el elemento a probar) ⇒ @BeforeEach
En el caso de que todos los tests requieran las MISMAS acciones para finalizar las pruebas (**después** de ejecutar el elemento a probar) ⇒ @AfterEach

Si es necesario realizar acciones previas a la ejecución de todos los tests una única vez, implementaremos dichas acciones en un método anotado con @BeforeAll o @AfterAll respectivamente.

4.2.1-. Etiquetas de los tests: @Tag

- Tanto las clases como los tests pueden anotarse con @Tag. Esta anotación permite “etiquetar” nuestros tests para **FILTRAR** su “**descubrimiento**” y “**ejecución**”
    - Si anotamos la clase, automáticamente estaremos etiquetando todos sus tests

    - Podemos usar varias “etiquetas” para la clase y/o las tests
- Las anotaciones @Tag nos permitirán discriminar la ejecución de los tests según sus etiquetas

4.2.2-. Tests parametrizados

Si el código de varios tests es idéntico a excepción de los valores concretos del caso de prueba que cada uno implementa, podemos sustituirlos por un test parametrizado.

Se trata de implementar un único test ⇒ @ParaterizedTest

Si el test parametrizado solamente necesita **un parámetro,** de tipo primitivo o String ⇒ @ValueSource

Si el tests requiere **más de un parámetro** ⇒ @MethodSource (Esta anotación devuelve una tupla de tipo Stream<Arguments>)

5-. Junit y Maven

**Para poder implementar** los tests con Junit5 (y compilarlos) necesitamos incluir la librería “junit-jupiter-engine”. De esta forma tendremos acceso a las clases del paquete org.junit.jupiter.api.

El valor “**test**” de la etiqueta <**scope**> indica que la librería sólo se utiliza para compilar y ejecutar los tests.
Si usamos tests **parametrizados** ⇒ librería “junit-jupiter-params”

Ejecutaremos los tests mediante la goal **surefire:test** ⇒ La goal surefire:test está asociada por defecto a la dase test de maven
Para “filtrar” la ejecución de los tests en función de sus anotaciones @Tag, tendremos que configurar el plugin usando las propiedades “**groups**” y “**excludedGroups**” del plugin

5.1-. Configuración del plugin surefire

Podemos alternar el valor de ciertas propiedades de los plugins usando la etiqueta <configuration>. Cada plugin tiene su conjunto de propiedades.

De forma alternativa podemos configurar cualquier plugin desde la línea de comandos. Pero la configuración del pom **PREVALECE** sobre la línea de comandos.

5.2-. Informes Junit

Cuando ejecutamos cada test, podemos obtener 3 resultados posibles:
- Pass: cuando el resultado esperado coincide con el real
- Failure: el método Assert lanza una excepción de tipo AssertionFailedError
- Error: se genera cualquier otra excepción durante la ejecución del test

El **informe** de la ejecución de todos los tests se guarda en target/surefire-reports, en formato **txt** y **xml**

5.3-. Maven y pruebas unitarias

Fase **compile**:

- Se compilan los fuentes del proyectos (/src/main/java)
- GOAL por defecto: **compiler:compile**
- artefactos generados en /target/classes
- Si hay errores de compilación se detiene la construcción

Fase **test-compile**:

- Se compilan los tests unitarios (/src/test/java)
- GOAL por defecto: **compiler:testCompile**
- artefactos generados en /target/test-classes
- Si hay errores de compilación se detiene la construcción

Fase **test**:

- Se ejecutan los tests unitarios (/target/test-classes)
- Se ejecutan los métodos anotados con @Test de las clases
- GOAL por defecto: **surefire:test**
- artefactos generados en /target/surefire-reports
- Se detiene el proceso de construcción si algún tests falla

Sesión S03. Pruebas de caja negra

1-. Diseño de casos de prueba

Seleccionamos de forma sistemática un conjunto de casos de prueba efectivo y eficiente ⇒ Independientemente del método de diseño elegido, SIEMPRE obtenemos un conjunto **EFICIENTE** y **EFECTIVO**

Formas de identificar los casos de prueba:

- Los casos de prueba obtenidos son independientes de la implementación
- El diseño de los casos de prueba puede realizarse en paralelo o antes de la implementación
- Podemos detectar comportamientos NO IMPLEMENTADOS
- Nunca podremos detectar comportamientos implementados, pero no especificados

Los métodos de diseño basados en la ESPECIFICACIÓN:

1. **Analizan** la especificación y PARTICIONAN el conjunto S
2. **Seleccionan** un conjunto de **comportamientos** según algún criterio
3. **Obtienen** un conjunto de **casos de prueba** que ejercitan dichos comportamientos

2-. Método de diseño de caja negra

Se trata de un proceso SISTEMÁTICO que identifica, a partir de la ESPECIFICACIÓN disponible, un conjunto de CLASES de equivalencia para **cada** una de las **entradas** y **salidas** del “elemento” a probar.

Cada clase de equivalencia de entrada representa un subconjunto del total de datos posibles de entrada que tienen un mismo comportamiento.

Objetivo ⇒ Minimizar el número de casos de prueba requeridos para cubrir TODAS las particiones al menos una vez.

2.1-. Identificar una partición

Las particiones se identifican en base a condiciones de entrada/salida de la unidad a probar. Una condición de entrada/salida, puede aplicarse a una única variable de entrada/salida en una especificación o con un subconjunto de ellas.

Las “variables” de entrada/salida no necesariamente se corresponden con “parámetros” de entrada/salida de la unidad a probar.

- Las particiones deben ser DISJUNTAS
- Todos los miembros de una partición de entrada deben tener su “imagen” en la misma partición de salida

2.2-. Identificación de las clases de equivalencia

Paso 1. Identificar las clases de equivalencia para cada entrada/salida, siguiendo las siguientes heurísticas:

- #1 Si la E/S especifica un rango de valores válidos ⇒ 1 válida / 2 inválidas
- #2 Si la E/S especifica un número N de valores válidos ⇒ 1 válida / 2 inválidas
- #3 Si la E/S especifica un conjunto de valores válidas ⇒ 1 válida / 1 inválida
- #4 Si lo valores de entrada se van a tratar de forma diferente por el programa ⇒ 1 válida para cada valor de entrada
- #5 Si la E/S especifica una situación debe ser ⇒ 1 válida / 1 inválida
- #6 Si los elementos van a ser tratados de forma distinta ⇒ Subdividir la partición en particiones más pequeñas.

Paso 2. Identificar los casos de prueba de la siguiente forma:

- Hasta que todas las clases válidas estén cubiertas, escribir un nuevo caso de prueba que cubra el máximo número de clases válidas todavía no cubiertas
- Hasta que todas las clases inválidas estén cubiertas, escribir un nuevo caso de prueba que cubra una y sólo una clase inválida todavía no cubierta
- Elegir un valor concreto para cada partición

Paso 3. El resultado de este proceso será una TABLA con tantas FILAS como CASOS de PRUEBA hayamos obtenido

3-. Observaciones

- Etiquetar las particiones de una misma E/S con la misma letra
- Hay que tener en cuenta las precondiciones de entrada/salida al realizar las particiones
- Si las E/S son objetos, tendrás que considerar cada atributo del objeto como un parámetro diferente
- Las E/S NO son ÚNICAMENTE parámetros de la unidad a probar
- Los objetos en java siempre son referencias por las variables. Por lo tanto tendremos que considerar el valor NULL como partición no válida al usar objetos.

Sesión 04. Dependencias externas (I)

1-. Pruebas unitarias y dependencias externas

Las pruebas de unidad dinámicas requieren ejecutar cada unidad (SUT: System Under Test) de forma aislada para poder detectar defectos (bugs) en dicha unidad.

- Objetivo : encontrar DEFECTOS en el código de las UNIDADES probadas

El código de la unidad a probar (SUT) tiene que ser exactamente el mismo código que se utilizará en producción. ⇒ No está permitido **alterar circunstancialmente/temporalmente**

Podemos definir un **código testable** como aquél que permite que un componente sea fácilmente probada de forma AISLADA

- Para poder probar un componente de forma aislada debemos ser capaces de **CONTROLAR** sus DEPENDENCIAS externas, también denominadas COLABORADORES, o DOCs
- Una **dependencia externa** es un objeto con quién interactúa nuestro código a probar y sobre el que no tenemos ningún control.

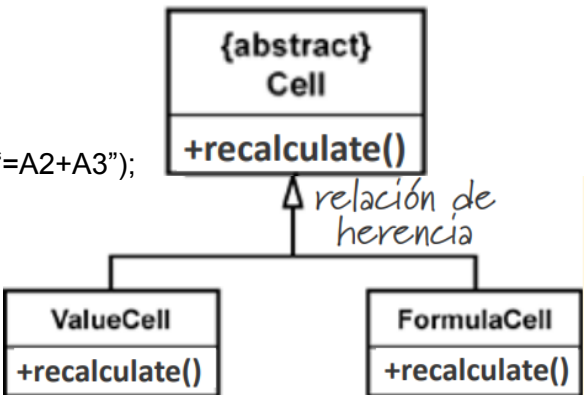
2-. Seam

Un seam es una línea de código desde la cual se puede alterar el comportamiento del programa sin tocarla ⇒ Para poder conseguir un seam en nuestro código puede que necesitemos **REFACTORIZAR** nuestro SUT

Cómo identificar un seam ⇒ Si **podemos cambiar el método** que se invocará desde esta línea SIN alterar el código que la unidad que la contiene.

<u>Ejemplo.</u>

```
public class CustomSpreadsheet extends Spreadsheet {
        public Spreadsheet buidMartSheet () {
                ...
                Cell myCell = new FormulaCell (this, “A1”, “=A2+A3”);
                ...
                myCell.recalculate()
                ...
        }
        ...
}
```



```
public class CustomSpreadsheet extends Spreadsheet {
        public Spreadsheet buildMartSheet(Cell cell) {
                ...
                cell.recalculate()
                ...
        }
        ...
}
```

3-. Pasos para automatizar las pruebas

1. Identificar las **dependencias** externas de nuestro SUT
2. Asegurarnos de que nuestro código es TESTABLE. **Puede que** necesitemos **REFACTORIZAR** nuestro SUT para poder realizar dichos reemplazos controlados durante las pruebas
3. Proporcionar una implementación ficticia (**DOBLE**), que reemplazará al código real
4. Implementar los DRIVERS correspondientes.
    a. Verificación basada en el **ESTADO**:
        - Solo estamos interesados en comprobar el estado resultante de la invocación de nuestro SUT
    b. Verificación basada en el **COMPORTAMIENTO**:
        - Además de comprobar el resultado verifica que las interacciones entre nuestro SUT y las dependencias externas se realizan correctamente

3.1-. Dependencias externas

Una **dependencia externa** es otra **UNIDAD** en nuestro sistema con la cual interactúa nuestro código a probar (SUT), y sobre la que no tenemos ningún control.

- Nuestro test no puede controlar lo que dicha dependencia devuelve a nuestro código a probar ni cómo se comporta
- Utilizaremos dobles para controlar el resultado de nuestra dependencia externa y así probar nuestra unidad de forma AISLADA

3.2-. Nuestro SUT debe ser testable

Necesitamos poder cambiar la dependencia real por su doble. Esto no será posible si no tenemos un seam para CADA dependencia externa.

En java:

- Nuestro **DOBLE** debe IMPLEMENTAR la misma INTERFAZ que el colaborador (DOC), o debe EXTENDER la misma CLASE que el colaborador.

Nuestro SUT será **TESTABLE** si podemos “inyectar” dicho doble en nuestra SUT durante las pruebas de alguna de las siguientes formas:

- (1) Como un parámetro de nuestra SUT
- (2) A través del constructor de la clase que contiene nuestra SUT
- (3) A través de un método setter de la clase que contiene nuestra SUT
- (4) A través de un método de factoría local de la clase que contiene nuestra SUT, o una clase factoría

**Si nuestro SUT NO es testable**, entonces tendremos que **REFACTORIZAR** el código de nuestra SUT para que podamos inyectar el doble de alguna de las formas anteriores, teniendo en cuenta que:

- (1) Si añadimos un parámetro a nuestro SUT, estamos OBLIGANDO a que cualquier código cliente de nuestra SUT tenga que CONOCER dicha dependencia antes de invocar a nuestra SUT
- (2-3) Si añadimos un parámetro al constructor de nuestra SUT, estamos OBLIGADOS a declarar la dependencia como atributo de la clase que contiene nuestro SUT.
- (3) No podremos añadir un método setter si el constructor realizase alguna acción significativa sobre nuestra dependencia.
- (4) Si usamos un método factoría local, no se ven afectados, ni los clientes de nuestro SUT, ni la estructura de la clase que contiene nuestro SUT, aunque alteramos el comportamiento de la clase que contiene nuestro SUT al añadir el nuevo método.

3.3-. Implementar el doble

Test Stub ⇒ Es un objeto que actúa como un punto de **CONTROL** para entregar ENTRADAS INDIRECTAS al SUT, cuando se invoca a alguno de los métodos de dicho stub. Un stub utiliza verificación basada en el **estado**.

Usos de un Test Double:

- Aislar el código a probar
- Acelerar la velocidad de ejecución
- Conseguir ejecuciones deterministas cuando el comportamiento depende de situaciones aleatorias o dependientes del tiempo
- Simular condiciones especiales
- Conseguir acceder a la información oculta.
- Controlar entradas indirectas (stub) o salidas indirectas (mock) del SUT

3.4-. Implementación del driver

El DRIVER se encarga, durante la fase de preparación de los datos, de crear el doble (stub) y de inyectar dicho doble para que sea invocado por nuestro SUT, en lugar del colaborador real. Usaremos tantos stubs como dependencias externas necesitemos controlar

Cuando utilizamos un stub, la verificación del resultado de las pruebas: se realiza sobre la clase a probar (SUT). Verificamos que el estado resultante de nuestro SUR es el esperado.

Sesión 05. Dependencias externas (II)

1-. Verificación y pruebas unitarias

Detectar problemas ⇒ Este proceso se denomina **VERIFICACIÓN**: se trata de buscar defectos en el programa que provocarán que éste no funcione correctamente.

El objetivo es DEMOSTRAR la presencia de DEFECTOS en el código de las UNIDADES. Lo haremos de forma dinámica, y podemos implementar los drivers de varias formas para verificarlo

2-. Frameworks para implementar dobles

Podemos utilizar algún framework para evitarnos la implementación “manual” de los dobles de nuestras pruebas.

- En general, cualquier framework nos generará una implementación configurable de los dobles a nivel de ejecución, generando el bytecode necesario.
- Primero configuraremos los dobles para controlar las entradas indirectas a nuestro SUT, o para registrar o verificar las salidas indirectas de nuestro SUT.
- Inyectamos los dobles en la unidad a probar antes de invocarla

Ventajas y desventajas:

- Los frameworks suelen “tergiversar” la terminología que hemos visto sobre dobles
- La verificación basada en el **comportamiento** genera un riesgo de que los tests tengan un alto nivel de acoplamiento con los detalles de implementación. Un framework contribuye a crear tests **potencialmente frágiles y difíciles de mantener**

Algunos ejemplos ⇒ **EasyMock**, Mockito, JMockit, PowerMock

3-. EasyMock y tipos de dobles

EasyMock crea de forma automática la clase que contiene el doble de nuestra dependencia externa.

EasyMock.niceMock(Clase.class) ⇒ Devuelve un doble para la clase o interfaz “clase.class”

- El orden en el que se realizan las invocaciones a los métodos del doble NO se chequean
- Por defecto se permiten las invocaciones a todos los métodos del objeto. Si no hemos programado las expectativas de algún método, éste devolverá valores “vacíos” adecuados.
- Todas las llamadas esperadas a métodos realizadas por el doble deben realizarse con los argumentos específicos

3.1-. Implementación de stubs

1. Creamos el stub ⇒ Podemos crear un stub a partir de una clase o de una interfaz
2. Programamos las expectativas del stub ⇒ Para programar las expectativas usaremos el método estático EasyMock.expect()
3. Indicamos al framework que el stub ya está listo para ser invocado por nuestro SUT ⇒ Después de programas las expectativas SIEMPRE tendremos que ACTIVAR el stub usando el método replay()

Si no cambiamos el estado del doble a “replay” las expectativas NO se tendrán en cuenta, por lo que si se invoca al doble desde nuestro SUT se devolverán los valores por defecto.

3.2-. Programación de expectativas

Un stub puede devolver resultados diferentes dependiendo de los valores de los parámetros de las invocaciones
O podemos “relajar” los valores de los parámetros, de forma que devolvamos un determinado resultado independientemente de los valores de entrada del stub

- Usaremos métodos anyObject(), anyString(), any… en esos casos

3.3-. Tipos de dobles

Creación de STUBS:

- El orden en el que se realizan las invocaciones al doble no se chequean
- Por defecto se permiten las invocaciones a todos los métodos del objeto. Si no hemos programado las expectativas de algún método, éste devolverá valores “vacíos”

Creación de MOCKS (1 invocación):

- El orden en el que se realizan las invocaciones al doble no se chequean
- El comportamiento por defecto para todos los métodos del objeto es lanzar un AssertionError para cualquier invocación no “esperada”

Creación de MOCKS (cualquier nº de invocaciones):

- Se comprueba el orden en el que se realizan las invocaciones al doble
- Si se invoca un método no “esperado” se lanza un AssertionError

Todas las llamadas esperadas a métodos realizados por el doble deben realizarse con argumentos especificados.

3.4-. Implementación de mocks

1. Creamos el mock ⇒ Podemos crear un mock a partir de una clase o una interfaz
2. Programamos las expectativas del mock: determinamos cuál será el valor de las salidas indirectas del SUT, también las entradas indirectas al SUT ⇒ Programamos las expectativas con el método EasyMock.expect(), para indicar cómo se invocará al doble
3. Indicamos al framework que el mock ya está listo ⇒ replay()
4. Verificamos las expectativas del mock ⇒ verify()

3.5-. Programación de expectativas

Número de invocaciones:

- Debemos indicar cómo interaccionará nuestro SUT con el objeto mock que hemos creado
    - Cuando ejecutemos nuestro SUT durante las pruebas, el mock registrará TODAS las interacciones desde el SUT
    - Si es un StrictMock y las invocaciones del SUT no coinciden con las expectativas programadas ⇒ El doble provocará un fallo (AssertionError)
    - Si es un Mock y las invocaciones del SUT no coinciden con las expectativas programadas ⇒ El doble provocará un fallo

- Para especificar las expectativas podemos indicar:
    - que se esperan un determinado número de invocaciones
    - Las expectativas pueden expresarse de forma “encadenada”

Valores de los parámetros:

- Debemos de tener en cuenta que:
    - EasyMock, para comparar argumentos de tipo Object, utiliza por defecto el método equals()
    - Si estamos interesados en que el parámetro de la expectativa sea exactamente la misma instancia, usaremos el método same()
    - Los Arrays son comparados por defecto con Arrays.equals()

Orden de invocación de expectativas

- Con respecto al orden de ejecución de UN MOCK:
    - Si usamos un Mock, el orden de las invocaciones a dicho objeto, NO se chequea.
    - Si usamos un StrictMock, el orden de las invocaciones a dicho objeto, debe coincidir exactamente con el orden establecido en las expectativas

- Con respecto al orden de ejecución de las expectativas entre VARIOS MOCKS:
    - Para poder establecer un orden de invocaciones entre objetos StrictMock, usaremos un ImocksControl

4-. Partial Mocking

La librería EasyMock nos permite realizar un mocking parcial de una clase, utilizando el método partialMockBuilder(), de la siguiente forma:

```
ToMock mock = partialMockBuilder(ToMock.class)
.addMockedMethod(“mockedMethod”).createMock();
```

Sesión S06. Pruebas de integración

1-. El proceso de integración

El objetivo de la integración del sistema es construir una versión “operativa” del sistema mediante:

- La agregación, de forma incremental
- La adición de nuevos componentes no deben “perturbar” el funcionamiento de los componentes que ya existen (**pruebas de regresión**)

Las pruebas de integración del sistema constituyen un proceso sistemático para “ensamblar” un sistema software, durante el cual se ejecutan pruebas conducentes a descubrir errores asociados con las **interfaces** de dichos componentes

- Se trata de garantizar que los componentes, sobre lo que previamente se han realizado pruebas unitarias, funcionan correctamente cuando son “combinados” de acuerdo con lo indicado por el diseño

2-. Tipos de interfaces y errores comunes

La interfaz entre componentes, puede ser de varios tipos:

- Interfaz a través de **parámetros** ⇒ los datos se pasan de un componente a otro en forma de parámetros
- **Memoria compartida** ⇒ Se comparte un bloque de memoria entre los componentes
- **Interfaz procedural** ⇒ Un componente encapsula un conjunto de procedimientos que pueden llamarse desde otros componentes
- **Paso de mensajes** ⇒ Un componente A prepara un mensaje y lo envía al componente B.

Errores más comunes derivados de la interacción de los componentes a través de sus interfaces:

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

Las pruebas para detectar defectos en las interfaces son difíciles, ya que algunos de ellos solo pueden manifestarse bajo condiciones inusuales.

3-. Estrategias de integración

- Big Bang ⇒ Una vez realizadas las pruebas unitarias, se integran TODAS las unidades
    - Sólo si el tamaño es muy pequeño o tiene pocas unidades
- Top-down ⇒ Integramos primero los componentes con mayor nivel de abstracción, y vamos añadiendo componentes cada vez con menor nivel de abstracción
    - Necesitamos implementar dobles
    - Adecuado para sistemas con una interfaz de usuario “compleja”

- Bottom-up ⇒ Integramos primero los componentes de infraestructura que proporciona servicios comunes y posteriormente añadimos los componentes funcionales, cada vez con mayor nivel de abstracción
    - Necesitamos implementar muchos menos dobles
    - Adecuado para sistemas con una infraestructura y/o lógica de negocio “compleja”
- Sandwich ⇒ Es una mezcla de las dos estrategias anteriores
- Dirigida por los riesgos ⇒ se eligen primero aquellos componentes que tengan un mayor riesgo
- Dirigida por las funcionalidades/hilos de ejecución ⇒ Se ordenan las funcionalidades con algún criterio y se integra de acuerdo con este orden

4-. Integración con una base de datos

Dbunit es un framework de código abierto basado en JUnit ⇒ DBUnit NO sustituye a la Base de Datos, sólo nos permite **CONTROLAR** su estado previo y verificar su estado después de la invocación de nuestro SUT

El escenario típico de ejecución de pruebas con base de datos utilizando DBUnit es el siguiente:

1. Eliminar cualquier estado previo de BD resultante de pruebas anteriores
    - No restauramos el estado de la BD después de cada test
2. Cargar los datos necesarios para las pruebas en la BD
    - Sólo cargaremos los datos NECESARIOS para cada test
3. Ejecutar las pruebas utilizando métodos de la librería DBUnit para las aserciones

Componentes principales del API:

- IDataBaseTester ⇒ JdbcDatabaseTester
    - Es la interfaz que permite el acceso a la BD, devuelve conexiones con una BD de tipo IDatabaseConnection
    - Implementaremos ⇒ JdbcDatabaseTester - usa un DriverManager para obtener conexiones con la BD
    - Métodos que se utilizan:
        - setDataSet( IDataSet dataSet ) ⇒ Inyecta los datos de prueba para iniciar la BD para las prueba.Utilizaremos getDataSet() para recuperar los datos
        - onSetUp() ⇒ realiza una operación CLEAN_INSERT en la BD, insertando en la BD el valor del “dataset” inyectado con el método setDataSet
        - getConnection() ⇒ Devuelve la conexión con la BD
- IDatabaseConnection
    - Representa una CONEXIÓN con una BD
    - Métodos que se utilizan:
        - createDataSet() ⇒ crea un dataset con TODOS los datos existentes actualmente en la base de datos
- DatabaseOperation

    - Clase abstracta que define el contrato de la interfaz para OPERACIONES realizadas sobre la BD
    - Utilizaremos un dataset como entrada para una DatabaseOperation
- IDataSet ⇒ FlatXmlDataSet
    - Representa una colección de tablas
    - Se utiliza para situar la BD en un estado determinado y para comparar el estado actual de la BD con el estado esperado.
    - Implementaremos ⇒ FlatDataSet - lee/escribe datos en formato xml
- ITable
    - Representa una colección de datos tabulares (de una tabla de la BD)
    - Se puede usar para preparar los datos iniciales de la BD
    - También se utiliza en aserciones, para comparar tablas de bases de datos reales con las esperadas.
    - Implementaremos ⇒ DefaultTable - ordenación por clave primaria
- Assertion
    - Clase que define métodos estáticos para realizar aserciones
    - Métodos que se utilizan:
        - assertEquales(... , ...)
    - El API DBUnit usa el método:
        - org.dbunit.Assertion.assertEquals

5-. DBUnit y pruebas de integración con maven

Para utilizar DBUnit en nuestros drivers en proyectos Maven tendremos que incluir en el fichero de configuración las siguientes dependencias:

- Librería DbUnit ⇒ versión 2.7.0
- Librería para acceder a una BD MySql ⇒ versión 8.0.23
- Plugin failsafe para ejecutar los tests de integración ⇒ versión 2.22.2
    - La goal “failsafe: integration-test” ⇒ Se ejecutan todos los métodos de integración
    - La goal “failsafe: verify” ⇒ Detiene la ejecución si algún test de integración “falla”

6-. Proceso completo para automatizar las pruebas de integración con maven

- 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 para poder copiar el artefacto generado en nuestro repositorio local. Si algún test falla, se detiene la construcción

Sesión S07. Pruebas de sistema y aceptación (I)

1-. Pruebas de sistema vs Pruebas de aceptación

1.1-. Pruebas de sistema

Objetivo ⇒ Encontrar defectos derivados del COMPORTAMIENTO del sw como de un todo REQUIERE los REQUISITOS del sistema.

Los casos de prueba se desarrollan desde el punto de vista del ⇒ DESARROLLADOR

1.2-. Pruebas de aceptación

Objetivo ⇒ Valorar en qué grado el software desarrollado satisface las expectativas del cliente REQUIERE los criterios de ACEPTACIÓN

Los casos de prueba se desarrollan desde el punto de vista del ⇒ USUARIO (CLIENTE)

2-. Pruebas del sistema

Son pruebas dinámicas y obtienen los casos de prueba a partir de las especificaciones del sistema (caja negra)
Se realizan durante el proceso de desarrollo, y usan todos los componentes del sistema, una vez que estos han sido integrados.
Las diferencias con las pruebas de integración son:

- Se incluyen componentes reutilizables desarrollados por “terceros”
- Los comportamientos a probar son los especificados para el sistema en su conjunto.

<u>Ejemplo.</u>

- Método de diseño basado en **casos de uso**
    - Permite probar todas las interacciones entre componentes
    - Cada caso de uso se implementa utilizando varios componentes del sistema
    - La prueba de un caso de uso “fuerza” el que se lleven a cabo las diferentes interacciones entre los componentes implicados
    - Se deben fijar políticas de prueba para elegir de forma adecuada un subconjunto de pruebas efectivo:
        - Todas las funcionalidades del sistema que sean accedidas desde menús deben ser probadas
        - Combinaciones de funciones que sean accedidas a través del mismo menú
        - Siempre que se proporcionan entradas de usuario, se deben probar entradas correctas e incorrectas
        - Las funciones más utilizadas en un uso normal, deben ser las más probadas
- Método de diseño de **transición de estados**

- Se usa en sistemas con ESTADO. Un estado viene dado por un conjunto de valores de variables del sistema
- A partir de la especificación se obtiene una representación en forma de grafo. Dicho grafo modela los estados de una entidad del sistema, representados en sus nodos
- A partir del grafo, se selecciona un conjunto de caminos mínimos para conseguir un objetivo concreto
- Finalmente se determinan los datos de entrada y salida esperados para los comportamientos que ejercitan dicho conjunto de caminos

3-. Pruebas de aceptación

Un producto está listo para ser entregado al cliente después de que se hayan realizado las pruebas del sistema .
Las pruebas de aceptación son pruebas formales orientadas a determinar si el sistema satisface los criterios de aceptación (**acceptance criteria**)

- Los criterios de aceptación de un sistema deben satisfacer para ser aceptados por el cliente

Dos categorías de pruebas de aceptación:

- User acceptance testing (UAT) ⇒ Son **dirigidas por el cliente** para asegurar que el sistema satisface los criterios de aceptación contractuales
    - Pruebas ⍺ ⇒ En el lugar de desarrollo para usuarios “conocidos”
    - Pruebas β ⇒ para usuarios “anónimos”
- Business acceptance testing (BAT) ⇒ Son **dirigidas por la organización** que desarrolla el producto para asegurar que el sistema eventualmente “pasará” las UAT. Son un ensayo de las UAT en el lugar de desarrollo.

3.1-. Criterios de aceptación

Los criterios de aceptación son aquellas características pactadas con el cliente, constituyen el “núcleo” de cualquier acuerdo contractual entre el proveedor y el cliente

Criterios:

- El principio básico para diseñar los criterios de aceptación es asegurar que la calidad del sistema es aceptable
- Los criterios de aceptación deben ser medibles, y por lo tanto, cuantificables

Algunos atributos de calidad que pueden formar parte de los criterios de aceptación son:

- Corrección funcional y completitud ( ⇒ QUE? )
- Exactitud, integridad de datos, rendimiento, fiabilidad y disponibilidad,mantenibilidad … ( ⇒ COMO? )

3.2-. Propiedades emergentes

Cualquier atributo incluido en los criterios de aceptación es una **propiedad emergente**.

Las propiedades “emergentes” son aquellas que no pueden atribuirse a una parte específica del sistema, sino que “emergen” solamente cuando los componentes han sido integrados, ya que son el resultado de las complejas interacciones entre sus componentes.

Hay dos tipos de propiedades emergentes:
- **Funcionales** ⇒ Ponen de manifiesto el propósito del sistema después de integrar sus componentes
- **No funcionales** ⇒ relacionadas con el comportamiento del sistema en su entorno habitual de producción

3.3-. Diseño de pruebas de aceptación

Normalmente se trata de un proceso black-box basado en la especificación del sistema.

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 sido implementado de forma adecuada a 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

4-. Pruebas de propiedades emergentes funcionales

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

Selenium es un conjunto de herramientas de pruebas open-source para automatizar pruebas o propiedades emergentes funcionales sobre aplicaciones Web
- Selenium WebDriver (API) ⇒ Permite crear tests robustos, que pueden escalarse y distribuirse en diferentes entornos
- Selenium IDE (extensión de un navegador) ⇒ Permite crear scripts de pruebas utilizando la aplicación web tal y como un usuario haría normalmente
- Selenium Grid ⇒ Es un servidor proxy que permite ejecutar tests en paralelo usando múltiples máquinas y diferentes navegadores

5-. Comandos Selenese

Un comando Selenese puede tener uno o dos parámetros:
- **target** ⇒ hace referencia a un elemento HTML de la página a la que se accede
- **value** ⇒ contiene el 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”. Una secuencia de tests scripts forman una test suite

Hay cuatro tipos de comandos:
- Actions ⇒ Interactúan directamente con los elementos de la página
- Accessors ⇒ Permiten almacenar valores en variables

- Assertions ⇒ Verifican que se cumple una determinada condición
    - Assert ⇒ Detiene la ejecución si no se cumple
    - Verify ⇒ Se registra el fallo y continúa la ejecución
    - WaitFor ⇒ espera a que se cumpla una condición antes de fallar
    - Control flow ⇒ Alteran el flujo secuencial de ejecución de los comandos. Pueden crear ramas condicionales o bucles.

5.1-. Locators

Se utilizan en el campo target e identifican un elemento en el código de una página web. Podemos utilizar los siguientes tipos de “locators”:

- id ⇒ Hace referencia al atributo id del elemento html
- name ⇒ Atributo name del elemento html
- xpath ⇒ Es el lenguaje utilizado para localizar nodos en un documento HTML

Sesión S08. Pruebas de aceptación (II)

1-. Características de 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.

WebDriver es una interfaz cuya implementación concreta la realizan dos clases :

- RemoteWebDriver
- HtmlUnitDriver

Una página web está formada por elementos HTML, que son objetos de tipo **WebElement** en el contexto de WebDriver
Una vez localizados los WebElements, podremos realizar acciones sobre ellos

2-. Mecanismos de localización

Utilizamos el método:

WebElement findElement (By by) throws NoSuchElementException
java.util.List<WebElement> findElements (By by)

- Como parámetros de entrada se requiere una instancia de “By”, que nos permite localizar elementos en una página web
- El inspector de elementos de Chrome puede ayudarnos a localizar los elementos HTML de las páginas web cargadas por el navegador

Hay 8 formas de localizar un WebElement en una página web:

- By.name(), By.id(), By.tagName(), By.className(), By.linkText(), By.partialLinkText(), By.xpath(), By.cssSelector()
- Locator **css**: los “css selectors” son patrones de caracteres utilizados para identificar un elemento HTML basado en una combinación de etiquetas HTML, id , class, y otros atributos

3-. Acciones sobre los WebElements

Una vez que hemos localizado el elemento que nos interesa, podemos ejecutar **ACCIONES** sobre ellos.

- Cada tipo de elemento tiene asociado un conjunto diferente de posibles acciones

Ejemplo de acciones:

- sendKeys (secuencia de caracteres) ⇒ Se utilizan para introducir texto en elementos textbox o textarea
- clear() ⇒ Se utiliza para borrar texto en elementos textbox o textarea

- submit() ⇒ Puede aplicarse sobre un elemento form, o sobre un elemento que esté dentro de form

3.1-. Tiempos de espera

Podemos establecer tiempos de espera en nuestros tests para evitar errores debidos a que no se localiza un elemento en la página porque todavía se esté cargando

- Tiempo de espera implícito ⇒ es común a todos los WebElements y tiene asociado un timeout global para todas las operaciones del diver
- Tiempo de espera explícito ⇒ se establece de forma individual para cada WebElement

3.2-. Múltiples acciones

Podemos indicar a WebDriver que realice múltiples acciones agrupandolas en una acción compuesta, siguiendo estos tres pasos:

- Invocar la clase **Actions** para agrupar las acciones
    - La clase Actions se utiliza para emular eventos complejos de usuarios
- Construir la acción ( **Action** ) compuesta por el conjunto de acciones anteriores
- Realizar ( ejecutar ) la acción compuesta

4-. WebDriver y maven

Necesitamos incluir la dependencia con la librería de WebDriver en nuestro proyecto Maven ⇒ org.seleniumhq.selenium

Los test de aceptación pueden implementarse en:

- En src/test/java junto con el resto de drivers del proyecto
- En src/test/java de un proyecto maven independiente

4.1-. Mantenibilidad de nuestros tests

Los tests implementados para nuestra aplicación web, funcionará siempre y cuando no se produzcan cambios en la vista web

- Si una o más páginas cambian de nuestra aplicación web sufren cambios, tendremos que cambiar el código de nuestros tests

Para facilitar la mantenibilidad, y reducir la duplicación de código de nuestros tests es útil el patrón de diseño “Page Object Model”

- La idea es independizar los tests de las páginas html
- Básicamente consiste en crear una clase para cada página web
    - Sus miembros serán los elementos de la página web correspondiente
    - Sus métodos serán todos los servicios que nos proporciona la página

Sesión S09. Pruebas de aceptación (III)

1-. Propiedades emergentes no funcionales

Muchas de las propiedades emergentes NO FUNCIONALES se categorizan como "-ilidades":

- fiabilidad ⇒ probabilidad de funcionamiento sin fallos durante un tiempo determinado en un entorno específico
- disponibilidad ⇒ tiempo durante el cual un sistema proporciona servicio al usuario
- mantenibilidad ⇒ capacidad de un sistema para soportar cambios. Hay tres tipos de cambios:
    - correctivos ⇒ son provocados por errores detectados en la aplicación
    - adaptativos ⇒ son provocados por cambios en el hardware y/o software sobre los que se ejecuta nuestra aplicación
    - perfectivos ⇒ debido a que se quiere añadir/modificar las funcionalidades existentes para ampliar/mejorar el "negocio" que sustenta nuestra aplicación
- Escalabilidad ⇒ Hace referencia a la capacidad de mantener el tiempo de respuesta ante cambios en el número de usuarios que utilizan el sistema

2-. Métricas

Los criterios de aceptación deben incluir propiedades emergentes "cuantificables"

- Hay que tener mucho cuidado con criterios de aceptación ambiguos, como "las peticiones se tienen que servir en un tiempo razonable"

Para juzgar en qué grado se satisfacen los criterios de aceptación se utilizan diferentes métricas:

- Para estimar la **fiabilidad** ⇒ Métricas MTTF (Mean Time To Failure)
  ⇒ MTTR (Mean Time To Repair)
  ⇒ MTBF = MTTF + MTTR (Mean time between failures)
- Para estimar la **disponibilidad** ⇒ Se utiliza la métrica MTTR para medir el "downtime" del sistema
- Para estimar la **mantenibilidad** ⇒ métrica MTTR que refleja el tiempo consumido en analizar un defecto correctivo
- La **escalabilidad** del sistema ⇒ Utiliza el número de transacciones por unidad de tiempo

3-. Ejemplos de pruebas

- Las **pruebas de carga** validan el **rendimiento** de un sistema en términos de "tratar un número específico de usuarios manteniendo un ratio de transacciones"
- Las **pruebas de stress** consisten en "forzar" peticiones al sistema por encima del límite del diseño del software.

    - Las pruebas comprueban la **fiabilidad** y **robustez** del sistema cuando se supera la carga normal
- Para evaluar la **fiabilidad** de un sistema podemos utilizar lo que utilizar lo que se denominan **pruebas estadísticas**, que consisten en:
    - Construir un “perfil operacional”, que refleje el uso real del sistema. Como resultado se identifican las “clases” de entradas y la probabilidad de ocurrencia para cada clase, asumiendo un uso “normal”
    - Generar un conjunto de datos de prueba que reflejen dicho perfil operacional
    - Probar dichos datos midiendo el número de fallos y el tiempo entre fallos, calculando la fiabilidad del sistema después de observar un número de fallos estadísticamente significativo

4-. Consideraciones sobre el rendimiento

- No hay que dejar las pruebas de rendimiento para el final del proyecto
- Posibles fallos relacionados con el rendimiento pueden conllevar el retocar mucho código
    - Las propiedades emergentes NO funcionales están condicionadas fundamentalmente por la ARQUITECTURA del sistema
    - Normalmente la arquitectura del sistema se determina en las primeras fases del desarrollo
    - El coste de reparar un error siempre es proporcional al intervalo de tiempo transcurrido desde que se produjo el error, hasta que éste es detectado
- Minimizaremos los problemas derivados de la validación de las propiedades emergentes no funcionales mediante una buena estrategia de pruebas combinada con una arquitectura software que considere el rendimiento desde el inicio del desarrollo

5-. JMeter

Apache JMeter es una herramienta de escritorio 100% Java diseñada para medir el rendimiento mediante pruebas de carga.
Permite múltiples hilos de ejecución concurrente, procesando diversos y diferentes patrones de petición.

- JMeter permite trabajar con muchos tipos distintos de aplicaciones. Para cada una de ellas proporciona un **Sampler** o Muestreador, para hacer las correspondientes peticiones

5.1-. Plan de pruebas

Un plan de pruebas JMeter describe una serie de “pasos” que JMeter realizará cuando se ejecute el plan.
Un plan de pruebas está formado por:

- Uno o más grupos de hilos (Thread Groups)
    - Un hilo de ejecución es el punto de partida de cualquier plan de pruebas. ⇒ Cada hilo representa a un usuario que se ejecuta independientemente de otros hilos.
- Controladores lógicos (Logic Controllers)

        - Determinan la lógica que JMeter utiliza para decidir cuándo enviar las peticiones
        - Ejemplos :
            - Simple controller ⇒ Sirve para “agrupar” los elementos
            - Loop controller ⇒ Itera sobre sus elementos hijos un cierto número de veces
            - Only once controller ⇒ Indica a JMeter que sus elementos hijos deben ser procesados UNA ÚNICA vez en el plan de pruebas
            - Interleave controller ⇒ Ejecutará uno de sus sub controladores o samplers en cada iteración del bucle de pruebas
- Samplers, Listeners, Timers, Assertions, Pre-Processors
    - Los Samplers envían peticiones a un servidor
- Elementos de Configuración (Configuration Elements)
    - “Trabajan” conjuntamente con un sampler
    - Un elemento de configuración es accesible sólo dentro de la rama del árbol en la que se sitúa el elemento
    - Un elemento de configuración dentro de una rama del árbol tiene mayor preferencia que el mismo elemento (mismo tipo de elemento) en una rama padre

5.2-. Timers

Los temporizadores permiten introducir pausas antes de realizar cada una de las peticiones de cada hilo.
Podemos utilizar varios tipos de temporizadores:

- Constan timer ⇒ Retrasa cada petición de usuario la misma cantidad de tiempo
- Uniform random timer ⇒ Introduce pausas aleatorias (con la misma probabilidad)
- Gaussian random timer ⇒ Introduce pausas según una determinada distribución (gausiana)

5.3-. Aserciones

Las aserciones permiten hacer afirmaciones sobre las respuestas recibidas del servidor que se está probando. Podemos añadir aserciones a cualquier sampler. Se trata de probar que la aplicación devuelve el resultado esperado.

5.4-. Listeners

Los **listeners** se utilizan para ver y/o almacenar en el disco los resultados de las peticiones realizadas. Proporcionan acceso a la información que JMeter va acumulando sobre los casos de prueba a medida que se ejecutan los test

- TODOS los listeners guardan los MISMOS datos, la única diferencia es la forma en que presentan dichos datos en pantalla

Sesión S10. Análisis de pruebas

1-. Métricas y análisis de pruebas

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 estamos alcanzando nuestros objetivos, no pudiendo controlar el proceso de software ni podremos mejorarlo
- Tiene que haber una relación entre lo que podemos medir, y lo que queremos "conocer"

Independientemente del objetivo concreto de nuestras pruebas, siempre buscamos **efectividad.** Hay 2 causas fundamentales que pueden provocar que nuestras pruebas no sean efectivas.

- Podemos ejecutar el código, pero con un mal diseño de pruebas, de forma que los casos de prueba sean tales que nuestro código esté "pobremente" poblado o no probado de ninguna manera
- Podemos, de forma "deliberada", dejar de probar partes de nuestro código

2-. Cobertura de código

La cobertura de código es la característica que hace referencia a cuánto código estamos probando con nuestros tests. ⇒ No proporciona un indicador la calidad del código, ni la calidad de nuestras pruebas, sino la **extensión** de nuestros tests

Hay 7 formas principales para cuantificar la cobertura de código:

- Cobertura de líneas ⇒ Un 100% significa que hemos ejecutado cada sentencia (línea)
    - Un 100% de cobertura de líneas no suele ser un nivel aceptable de caso de prueba
    - A pesar de constituir el nivel más bajo de cobertura, en la práctica puede ser difícil de conseguir
- Cobertura de decisiones ⇒ Un 100% significa que hemos ejercitado cada rama/decisión en su vertiente V o F
    - Para conseguir la cobertura de ramas, necesitamos diseñar casos de prueba de forma que cada DECISIÓN que tenga como resultado true/false sea evaluado al menos 1 vez
- Cobertura de condiciones ⇒ Un 100% significa que hemos ejercitado cada condición
    - La cobertura de condiciones es mejor que la cobertura de decisiones debido a que cada CONDICIÓN INDIVIDUALES es probada al menos 1 vez
- Cobertura de condiciones múltiples ⇒ Un 100% significa que hemos ejercitado todas las posibles combinaciones de condiciones

    - Si conseguimos un 100% de cobertura de condiciones múltiples, también conseguimos cobertura de decisiones, cobertura de condiciones, cobertura de condiciones + decisiones
    - No garantiza una cobertura de caminos
- Cobertura de decisiones y condiciones ⇒ Ejercitaremos cada condición en el programa y cada decisión en el programa
- Cobertura de bucle ⇒ Un 100% implica probar el bucle con 0 iteraciones, 1 iteración y múltiples iteraciones
- Cobertura de caminos ⇒ Un 100% de cobertura implica que se han probado todos los posibles caminos de control. Es muy difícil de conseguir cuando hay bucles. Un 100% de cobertura de caminos implica un 100% de cobertura de ramas.
    - No garantiza una cobertura de condiciones múltiples

3-. JaCoCo

JaCoCo genera un informe con las siguientes métricas:

- Instructions
    - Número de instrucciones bytecode de Java
- Branches
    - JaCoCo llama brach tanto a una decisión como a una condición
- Complejidad ciclomática
- Líneas
    - Una línea se considera ejecutada cuando se ejecuta al menos una de las instrucciones bytecode asociada a esa línea
- Métodos
    - Cada método no abstracto contiene una instrucción como mínimo. Un método se considera ejecutado, cuando se ejecuta al menos una de sus instrucciones
- Clases
    - Una clase se considera ejecutada, cuando se ejecuta al menos uno de sus métodos

4-. Plugin JaCoCo para maven

JaCoCo instrumenta los ficheros .class para obtener los datos de cobertura. La instrumentación tiene lugar en tiempo de ejecución usando un mecanismo denominado “Java Agent”

- jacoco:prepare-agent
    - Prepara el valor de la propiedad “argLine” para pasarla como argumento a la JVM durante la ejecución de las pruebas unitarias
    - Por defecto se asocia a la fase “initialize”
    - Indica qué clases deben ser instrumentadas, y cuál será el fichero con los datos resultantes del análisis de cobertura
- jacoco:prepare-agent-integration
    - Por defecto se asocia a la fase “pre-integration-test”
- jacoco:report

    - A partir de los datos obtenidos genera un informe en formato html, xml y cvs. Está asociada por defecto a la fase “verify”.
    - test unitarios
- jacoco:report-integration
    - test de integración
- jacoco:check
    - Comprueba si se ha alcanzado un determinado valor de cobertura, y detiene el proceso de construcción si no se alcanza dicho valor
    - Por defecto se asocia a la fase “verify”, y usa los datos del fichero jacoco.exe

Sesión 11. Planificación de pruebas

1-. Planificación predictiva vs planificación adaptativa

**Planificación predictiva** ⇒ El sistema ya está definido solo hay que repartir la tarea de forma equitativa. Soporta muy mal los cambios

**Planificación adaptativa** ⇒ Intenta encontrar un equilibrio entre el esfuerzo invertido en realizar el plan, frente a la información disponible en cada momento. Facilidad a la hora de incluir cambios

2-. Iteraciones y time-boxing

Las iteraciones SIEMPRE deben tener el mismo tamaño

Las iteraciones deben ser SIEMPRE **time-boxed**: nunca se puede retrasar el tiempo de entrega, es preferible y es mucho más sencillo cambiar el “scope”

3-. Alcance y efectividad de las pruebas

Para conseguir un resultado aceptable hay que:
- Priorizar ⇒ decidir qué tests son más importantes
- Fijar criterios para conseguir unos objetivos de pruebas de forma que sepamos cuándo parar

El objetivo final es asegurar que las pruebas son EFECTIVAS
- Un buen test es aquel que encuentra un defecto. Si encontramos un defecto estamos creando una oportunidad de mejorar la calidad del producto

El proceso debe ser EFICIENTE:
- Encontrar el mayor número de defectos con el menor número de pruebas posibles
- Para ello se deben utilizar buenas técnicas de DISEÑO de caso de prueba

4-. El proceso de pruebas

- Test planning and control
    - Se determina ¿QUÉ? se va a probar, ¿CÓMO?, ¿QUIÉN? y ¿CUÁNDO?
    - Se determina QUÉ se va hacer si los planes no se ajustan a la realidad
- Test analysis and design
    - Se determina que CASOS DE PRUEBA se van a utilizar
    - Un caso de prueba es un DATO CONCRETO + RESULTADO ESPERADO
- Test implementation and execution
    - Se implementan y ejecutan los tests

- Evaluating exit criteria and reporting
    - Se verifica que se alcanzan los completion criteria
- Test closure activities
    - Se trata de verificar que los informes ya están disponibles, …

4.1-. Pruebas en modelos secuenciales

Se trata de incluir en el plan de desarrollo las actividades de pruebas. Se debe contemplar un equilibrio entre las pruebas estáticas y las dinámicas

4.2-. Pruebas en modelos iterativos y ágiles

En modelos iterativos, las pruebas se planifican a nivel de ITERACIÓN y RELEASE ⇒ Cada RELEASE se divide en 4 fases. Cada fase contiene una o más ITERACIONES

En modelos ágiles, las pruebas se planifican a nivel de DÍA, ITERACIÓN y RELEASE

5-. Pruebas y diseño: TDD

TDD es una práctica de pruebas utilizadas en desarrollos ágiles, como por ejemplo XP
Consiste en desarrollar aplicando estos 3 pasos:

- Paso 1. Escribir una prueba para el nuevo código
- 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 aplicación “evoluciona” a medida que vamos desarrollando la aplicación
El diseño de la aplicación se realiza de forma incremental ajustando la estructura del código en pequeños incrementos a medida que se añade más comportamiento

5.1-. Classical vs Mockits TDD

Classical ⇒

- Consiste en utilizar objetos reales siempre que sea posible y utilizar stubs si es “complicado” utilizar el objeto real
- Esta aproximación soporta un estilo de diseño **INSIDE-OUT**: comenzamos por los componentes de bajo nivel
- Minimizamos el problema de las dependencias

Mockits TDD ⇒

- Utilizará un mock para cualquier objeto con un comportamiento interesante
- Soporta un estilo de diseño **OUTSIDE-IN**: comenzamos por los componentes de alto nivel
- El diseño OUTSIDE-IN ⇒ Utiliza verificación basada en el comportamiento

6-. Integraciones continuas

Las integraciones continuas consisten en **INTEGRAR** el código del proyecto de forma ininterrumpida en una máquina aparte de la de cada desarrollador
Las integraciones continuas realizan **CONSTRUCCIONES PLANIFICADAS** del sistema.

6.1-. Componentes y funcionamiento de un sistema de CI

1. Un desarrollador “sube” el código en el que ha estado trabajando al repositorio SCM, y sólo si las pruebas unitarias han “pasado”
2. El CI recupera la última versión del SCM y ejecuta un script de construcción del proyecto
3. El CI genera un feedback. En el momento en el que se “rompe” el sistema, hay que reparar el error