

Tema 7 ~ Pruebas de Integración

Nivel de integración vs Nivel de pruebas unitarias

Nivel de pruebas unitarias: encontrar defectos en el código de las unidades. La cuestión fundamental se basa en saber aislar cada unidad del resto de código.

Nivel de pruebas de integración: encontrar defectos derivados de la interacción entre las unidades que conforman el proyecto. La cuestión fundamental es el orden de integración.

Pruebas de regresión: Cuando estamos integrando es conveniente y prácticamente mandatorio realizar y ejecutar los tests de integración cada vez que añadimos un nuevo componente al sistema. A éste tipo de pruebas se les llama pruebas de regresión.

Distintos tipos de interfaz entre componentes: [PIMI]

- **Paso de mensajes:** Componente A prepara un mensaje y lo envía al componente B. El mensaje de respuesta del componente B incluye los resultados de ejecución (Web)
- **Interfaz parametrizada:** Los datos se pasan entre componentes a través de parámetros
- **Memoria compartida:** Se comparte un bloque de memoria entre los componentes
- **Interfaz procedural:** Un componente encapsula un conjunto de procedimientos que pueden ser llamados desde otros componentes (Objetos)

Estrategias de integración:

- **Big Bang:** Una vez se han acabado de probar las unidades (pruebas unitarias) se integra todo a la vez
- **Top-Down:** Se integran primero los componentes de mayor abstracción
- **Bottom-Up:** Se integran primero los componentes de bajo nivel
- **Sandwich:** Mezcla el Bottom-Up y el Top-Down
- **Por riesgos:** Se integra primero los componentes con más probabilidad de errores debido a su complejidad.
- **Por funcionalidades:** Se ordenan las funcionalidades por algún criterio y se integra de la más relevante a la menos relevante.

Librerías necesarias para los tests de integración: Mysql, DbUnit y JUnit

DbUnit: Se trata de un framework que nos permite controlar la dependencia de las aplicaciones con una base de datos:

- Permite gestionar el estado de la BD durante las pruebas
- Permite ser utilizado junto JUnit

Escenario típico de ejecución:

1. Eliminar estado previo de la BD (Siempre ANTES de las pruebas, NUNCA después)
2. Cargar los datos necesarios para las pruebas en la BD
3. Ejecutar las pruebas utilizando DbUnit para las aserciones

Componentes principales de DbUnit:

- **IDataBaseTester:** Objeto para poder obtener una conexión con la DB
- **IDataBaseConnection:** Objeto para poder interactuar con la BD para realizar las pruebas.
- **DataBaseOperation:** Define el contrato de la interfaz para operaciones realizadas sobre la BD. Utilizaremos un **dataset** como entrada para una DataBaseOperation.
- **ITable, IDataset y Aserciones (de dataset e ITable).**

Fases del ciclo de vida de Maven relacionadas con los tests de integración:

- **pre-integration-test:** iniciar servicios (bd, servidor web...)
- **integration-test:** Se ejecutan los tests de integración, si algún test falla no se detiene la construcción
- **post-integration-test:** Se detienen todos los servicios o se realizan las acciones necesarias para restaurar el entorno de pruebas.
- **verify:** Se comprueba que no hay errores para copiar el artefacto generado en nuestro repositorio local. Si algún tests falla se detiene la construcción.

No hay ninguna goal asociada a estas fases si el empaquetado es jar, por lo tanto hay que configurarlas en el POM.

Pruebas de Integración - POM:

Agregar las dependencias principales: Necesitaremos agregar las dependencias principales de **MySQL**, **DbUnit** y **JUnit** dentro del apartado `<dependencies>`. Pondremos `<scope>test</scope>` en las dependencias ya que sólo se emplearan en `src/test`.

Pre-Integration: Para inicializar las tablas de nuestra BD podemos configurar el POM dentro del apartado de construcción de la siguiente manera:

- Añadimos el plugin sql de Maven para ejecutar sentencias SQL
- Añadimos la dependencia con el driver JDBC para acceder a la BD **MySQL** en nuestro caso.
- Configuramos el driver para que conecte con nuestra BD usando la contraseña y el usuario que se pida.
- En el apartado `<executions>` crear tantas `<execution>` como operaciones inicializadoras queramos crear, estas operaciones inicializadores han de identificarse con un `<id>` para después poder invocarlas utilizando comandos Maven:

mvn sql:execute@nombre-puesto-en-id

Indicaremos la fase también en `<phase>` para poder ejecutarlos al hacer:

mvn pre-integration-test

Ejecución de los tests de integración: Se lleva a cabo mediante el plugin **failsafe**, por lo tanto tendremos que añadirlo en el POM en el apartado de construcción (`<build><plugins>`) del proyecto. Además tendremos que especificar las goals en el apartado executions (integration-test y verify).

Pruebas de Integración - Comandos:

Para ejecutar las fases

- mvn pre-integration-test
- mvn integration-test
- mvn verify

Para ejecutar goals

- mvn sql:execute@nombre-id
- mvn failsafe:integration-test
- mvn failsafe:verify

Pruebas de Integración - Ficheros:

- **Extensiones:** IT*.java, *IT.java, *ITCase.java
- **Artefactos generados en:** /target/failsafe-reports. Informes en *.xml
- **La capeta src/test/resources** almacena ficheros adicionales no java que se necesiten para ejecutar el código de pruebas.

Tema 8 y 9 ~ Pruebas funcionales

Validación VS Verificación:

Verificación: Se basa en responder a la pregunta de si el sistema está implementado correctamente.

Validación: El objetivo de este nivel es valorar en qué grado el software desarrollado satisface las expectativas del cliente. Un término clave en las pruebas de aceptación es el **Acceptance Criteria**, el cual define los criterios bajos los cuales el sistema será aceptado por el usuario.

Categorías pruebas aceptación:

User acceptance testing (UAT): Dirigidas por el cliente para asegurar que el sistema satisface los criterios de aceptación.

- Pruebas alpha: Se trata de pruebas internas en las que el cliente prueba y un técnico anota las posible anomalías
- Pruebas beta: Se lanzan un número de copias a los clientes para que éstos reporten los errores.

Bussiness Acceptance testing (BAT): Dirigidas por la entidad desarrolladora para asegurar que el sistema pase las UAT. Son un ensayo a las mismas.

Propiedad emergente: Es cualquier atributo incluido en los criterios de aceptación. Esas propiedades emergen al integrar el sistema, por lo tanto no pueden calcularse directamente a partir de las unidades:

- **Funcionales:** QUÉ tiene que hacer el sistema, es decir, su propósito (Ej.: Sacar dinero)
- **No funcionales:** CÓMO de bien el sistema performa sus requerimientos funcionales

Métodos de diseño de pruebas emergentes funcionales:

- **Diseño de pruebas basado en requerimientos:** Pruebas de validación en las que cada requerimiento debe ser testable
- **Diseño de pruebas de escenarios:** Los escenarios describen la forma en la que el sistema debería usarse. Un escenario está formado normalmente por varios requerimientos.

Selenium IDE: Se trata de un plugin que se instala en el navegador que permite crear scripts de control sobre páginas web.

- Un conjunto de tests scripts (casos de prueba) test scripts en Selenium IDE se denomina SUITE.
- El código fuente de los comandos Selenese es en formato HTML

Comandos Selenese:

Parámetros:

- **target:** elemento de la página a la que se accede. Suele ser un **locator**, es decir, el elemento HTML al cual se refiere un determinado comando.
- **value:** texto, patrón o variable a introducir

Tipos de comando:

- **actions:** Manipulan el estado de la aplicación (click) (...andWait)
- **accessors:** Examinan el estado de la aplicación y almacenan el resultado en variables
- **assertions:** Verifican que el que el estado de la aplicación de la aplicación es el esperado.
 - **verify:** no aborta ejecución. Negativa: `verifyNot`
 - **assert:** aborta la ejecución. Negativa: `assertNot`

Comandos:

open
click/clickAndWait
verifyTitle/assertTitle
verifyText/assertText
verifyElementPresent/assertElementPresent
waitForPageToLoad
store, storeText, storeEval

WebDriver vs. Selenium IDE:

Ventajas de Selenium IDE:

- Implementar tests más rápidamente
- No se requiere experiencia previa para con lenguajes de programación
- La búsqueda de elementos en la página es fácil

Inconvenientes de Selenium IDE:

- La herramienta a veces registra la localización absoluta de los elementos en la página, si cambian de sitio el test fallará.
- Tests inflexibles
- Duplicación de código.

Selenium IDE no es un lenguaje de programación, por lo tanto tiene limitaciones que con WebDriver no encontraremos. WebDriver nos permite utilizar java entre otros lenguajes.

WebDriver:

Una página web está formada por WebElements (elementos HTML). Una vez localizados, podemos realizar acciones sobre ellos. Aquí una lista con diversos WebElements:

- `<select><option... -> DropDown`
- `<input type="radio"... -> RadioButton`
- Los "type" pueden ser: checkbox, text, password, button...

Ejemplos de acciones:

```
driver.findElement(By.name("username")).sendKeys("hello");
driver.findElement(By.linkText("Register here")).click();
```

Podemos establecer dos tipos de tiempo de espera mientras se carga la página:

- **Tiempo de espera implícito:** Común a todos los WebElements.
`driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);`
- **Tiempo de espera explícito:** Se establece de forma individual para cada WebElement.
`WebDriverWait wait = new WebDriverWait(driver, 10);`
`wait.until(ExpectedConditions.titleContains("selenium"));`

Realización de múltiples acciones agrupándolas en una acción compuesta

```
Actions builder = new Actions(driver);
// Generate the composite action
Action compositeAction = builder.keyDown(Keys.CONTROL)
.click(one)
.click(three)
.click(five)
.keyUp(Keys.CONTROL)
.build();
// Perform the composite action.
compositeAction.perform();
```

Tema 10 ~ Pruebas no funcionales

Propiedades emergentes no funcionales: [DIMA FIES(TA)]

- **Disponibilidad:** Tiempo durante el cual el sistema proporciona servicio.
- **Mantenibilidad:** Capacidad del sistema para soportar cambios: correctivos, adaptativos y perfectivos.
- **Fiabilidad:** Probabilidad de funcionamiento sin fallos durante un tiempo determinado en un entorno específico.
- **Escalabilidad:** Capacidad de mantener el tiempo de respuesta ante cambios en el número de usuarios usando el sistema.

Estas propiedades han de ser medibles, mediante las siguientes métricas:

- **MTTF:** Mean time to failure
- **MTTR:** Mean time to repair
- **MTBF:** Mean time between failure

Fiabilidad:	MTBF = MTTR + MTTF
Disponibilidad:	MTTR
Mantenibilidad:	MTTR, se puede medir con la CC
Escalabilidad:	Tiempo de respuesta según usuarios

Pruebas de carga/stress: Forman parte de las pruebas para comprobar que el sistema cumple con los requisitos no funcionales que contribuyen al rendimiento del sistema

- **Pruebas de carga:** Son las que se llevan a cabo con **JMeter**. Validan el **rendimiento** de un sistema en términos de transacciones por número de usuarios (Ejemplo: Una petición del sistema debe tardar menos de 2 segundos cuando hay 10k usuarios)
- **Pruebas de stress:** Comprueban la **fiabilidad** y **robustez** del sistema cuando se supera la carga normal. (Robustez: Capacidad del sistema ante entradas erróneas y fallos)

Pruebas estadísticas [PDM]: Permiten evaluar la **fiabilidad** de un sistema y consisten en:

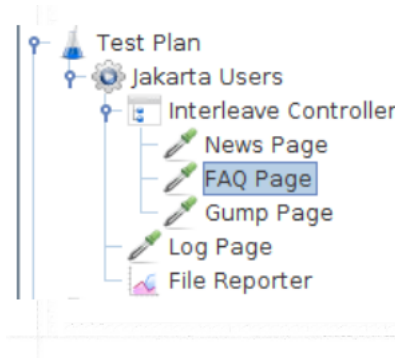
1. Crear un perfil operacional que refleje el uso real del sistema
2. Generar un conjunto de datos de prueba que reflejen dicho perfil
3. Medir fallos y tiempo entre fallos

Pasos para evaluar el rendimiento [CRITENAU]

1. Identificar los criterios de aceptación (propiedades emergentes no funcionales)
2. Diseñar los tests (perfil operacional)
3. Preparar el entorno de pruebas (real)
4. Automatizar las pruebas (utilizando alguna herramienta)
5. Resultados: Analizar y realizar cambios en el sistema para conseguir nuestro objetivo

JMeter – Plan de pruebas: Serie de pasos que JMeter realizará cuando se ejecute el plan. Formando por:

- **Thread groups:** Cada hilo representa a un usuario. Es el punto de partida de cualquier plan de pruebas. Los hilos son independientes. El Ramp-Up o periodo de subida es para que los hilos se generen de forma gradual.
- **Samplers:** Enviar peticiones al servidor, se ejecutan en el orden en el que aparecen en el árbol (POST, GET...)
- **Controladores lógicos:** Lógica que JMeter utiliza para decidir cuándo enviar peticiones (es decir, sus hijos). Hay de varios tipos:
 - **Simple Controller:** No tiene efecto, sirve como agrupador
 - **Loop Controller:** Itera sobre sus hijos un nº de veces
 - **Only once Controller:** Sus hijos deben ser ejecutados una única vez en el plan de pruebas.
 - **Interleave Controller:** Ejecuta uno de sus hijos en cada iteración del bucle de pruebas.



Loop Iteration	Each JMeter Thread Sends These HTTP Requests
1	News Page
1	Log Page
2	FAQ Page
2	Log Page
3	Gump Page
3	Log Page
4	Because there are no more requests in the controller, JMeter starts over and sends the first HTTP Request, which is the News
4	Log Page
5	FAQ Page
5	Log Page

- **Elementos de configuración:** No realizan peticiones (excepto el HTTP Proxy server, el cual está solo disponible para el banco de trabajo (workbench)), pero pueden modificarlas a través de varios atributos. Un elemento de configuración dentro de una rama del árbol tiene mayor preferencia que el mismo tipo de elemento en una rama padre.
- **Timers:** Por defecto JMeter envía las peticiones sin realizar pausas entre las mismas. Esto podría saturar el servidor, los timers nos permiten introducir pausas antes de realizar cada una de las peticiones de cada hilo. Hay de tres tipos: **Constant timer** (retrasa cada petición la misma cantidad de tiempo), **Uniform random timer** (introduce pausas aleatorias) y **Gaussian random timer** (introduce pausas según una determinada distribución).
- **Assertions:** Se trata de probar que la aplicación devuelve el resultado esperado (cualquier driver debe incluirla).
- **Listeners o receptores:** Ver y/o almacenar en disco los resultados de las peticiones. Podrás visualizar el rendimiento en ellos.

Tema 11 ~ Análisis de pruebas

Métrica: Medida cuantitativa del grado en que un sistema, componente o proceso, posee un determinado atributo.

Causas fundamentales por las que nuestras pruebas no son efectivas:

- Mal diseño o diseño pobre
- Podemos, de forma deliberada, dejar de probar algunas cosas. (Por ejemplo, por falta de tiempo)

Probar con un mal diseño es mejor que no probar.

Analizar de forma automática lo bien o mal que hemos diseñado es **imposible** a día de hoy, lo que sí se puede medir es la **cobertura**, es decir, la cantidad de código probado, a más código probado NO implica un mejor diseño.

Niveles de cobertura [7 niveles]:

1. **Statement coverage:** Todas las sentencias (líneas)
2. **Branch coverage:** Todas las ramas en su vertiente V y F (Implica 1)
3. **Condition coverage:** Todas las condiciones
`if((A>0) && (B>0)) - T && T - F && F`
4. **Multicondition coverage:** Todas las combinaciones de condiciones
`if((A>0) && (B>0)) - T && T - F && F - F && T - T && F`
5. **Condition decision coverage:** Las condiciones siguientes en algunos lenguajes de programación se evalúan dependiendo de las condiciones que las preceden. Este punto es relativo a 4, pero en java y C++ con sólo hacer `T && T - T && F - F && F`.
6. **Loop Coverage:** Implica probar el bucle con 0, 1 y múltiples iteraciones.
7. **Path coverage:** Se han probado todos los posibles caminos de control. Implica un 100% de cobertura de ramas (2).

Categorías de analizadores automáticos de código (en Java)

- Inserción de código de instrumentación en el código fuente.
- Inserción de código de instrumentación en el byte-code de Java (Cobertura)
- Inserción de código en una máquina virtual de java modificada

Métricas generadas por Cobertura (3)

- **Line coverage:** ¿Cuántas líneas se ejecutan?
- **Branch coverage:** Porcentaje de condiciones cubiertas frente al total de condiciones del programa
- **Complejidad ciclomática:** ¿Casos de prueba necesarios para cubrir todos los caminos linealmente independientes?

¿Para qué sirve la complejidad ciclomática?

- Mide la complejidad lógica del código:
 - A mayor valor, el código será más difícil de mantener
 - Si el valor es muy alto (>15) puede llevarnos a tomar la decisión de refactorizar.
- Nos da una cota superior del nº máximo de tests a realizar para cubrir todos los caminos linealmente independientes.

Cobertura – Comandos principales:

- **cobertura:instrument** -> Modifica el bytecode para incluir contador (número de líneas)
- **Informe de cobertura (html) podemos visualizarlo haciendo:**
 - **mvn cobertura:cobertura**
 - Incluyendo plugin de cobertura dentro de **<reporting>** y ejecutando **mvn site**

Cobertura - Directorios:

- Información instrumentación – target/cobertura/cobertura.ser
- Clases instrumentadas generadas – target/generated-classes/cobertura
- Informe de cobertura – target/site/cobertura

Cobertura – Más comandos:

Goal	Report?	Description
<code>cobertura:check</code>	No	Check the coverage percentages for <u>unit tests</u> from the last instrumentation, and optionally fail the build if the targets are not met. To fail the build you need to set <code>configuration/check/haltOnFailure=true</code> in the plugin's configuration.
<code>cobertura:check-integration-test</code>	No	Check the coverage percentages for <u>unit tests and integration tests</u> from the last instrumentation, and optionally fail the build if the targets are not met. To fail the build you need to set <code>configuration/check/haltOnFailure=true</code> in the plugin's configuration.
<code>cobertura:clean</code>	No	Clean up the files that Cobertura Maven Plugin has created during instrumentation.
<code>cobertura:cobertura</code>	Yes	Instrument the compiled classes, run the <u>unit tests</u> and generate a Cobertura report.
<code>cobertura:cobertura-integration-test</code>	Yes	Instrument the compiled classes, run the <u>unit tests and integration tests</u> and generate a Cobertura report.
<code>cobertura:dump-datafile</code>	No	Output the contents of Cobertura's data file to the command line.
<code>cobertura:help</code>	No	Display help information on cobertura-maven-plugin. Call <code>mvn cobertura:help -Ddetail=true -Dgoal=<goal-name></code> to display parameter details.
<code>cobertura:instrument</code>	No	Instrument the compiled classes.

Con **mvn clean** se borra todo el directorio target.

```

<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</artifactId>
      <version>2.7</version>
      <configuration>
        <instrumentation>
          <ignores>
            <ignore>com.example.boringcode.*</ignore>
          </ignores>
          <excludes>
            <exclude>com/example/dullcode/**/*.class</exclude>
            <exclude>com/example/**/*.Test.class</exclude>
          </excludes>
        </instrumentation>
      </configuration>
    </plugin>
  </plugins>
</build>

```

Se IGNORAN las llamadas a estos métodos

Estas clases NO serán instrumentadas

```

<configuration>
  <check>
    <branchRate>85</branchRate>
    <lineRate>85</lineRate>
    <haltOnFailure>true</haltOnFailure>
    <totalBranchRate>85</totalBranchRate>
    <totalLineRate>85</totalLineRate>
    <packageLineRate>85</packageLineRate>
    <packageBranchRate>85</packageBranchRate>
  </check>
</configuration>

```

Si pongo la etiqueta check, siempre tendré que poner algo, si quiero que todo sea al **50%** escribiré `<haltOnFailure>true</haltOnFailure>`.

Por lo tanto hasta que no tengamos suficiente cubierto nuestro código, Maven no nos dejará continuar, es decir se interrumpirá la construcción y por lo tanto el artefacto generado NO se copia en el repositorio local.

Si por un casual la complejidad ciclomática nos devuelve un valor con decimales, esto significará que ha calculado el CC haciendo media del CC de todos los métodos del entorno establecido.

Tema 12 ~ Planificación de pruebas

Qué, Cómo, Cuándo, Quién: Preguntas a contestar por el modelo de proceso de desarrollo elegido

Planificación predictiva vs Planificación adaptativa

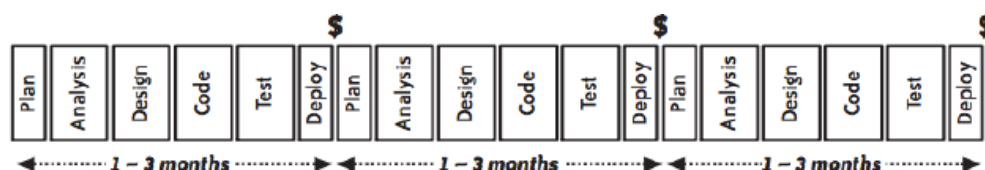
- **Planificación predictiva:** Se realiza la planificación global al principio del proyecto
- **Planificación adaptativa:** Se realiza la planificación conforme va avanzando el proyecto.

Niveles de planificación

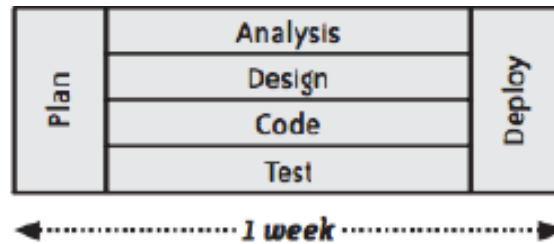
- **Modelo secuencial:** Modelo plano sin iteraciones en el cual sólo hay un planning y una release.



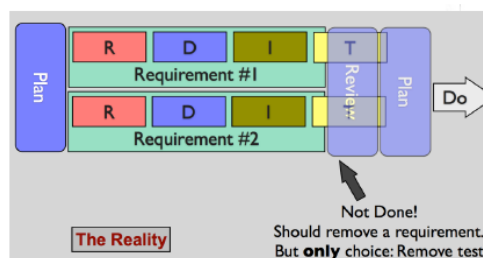
- **Modelo iterativo:** Modelo dividido en iteraciones en las cuales se van produciendo deploys del proyecto, en éste modelo hasta que no se acaba una fase no se va a la siguiente (mini cascada).



- **Modelo ágil (XP):** Se producen en paralelo los procesos de análisis, diseño, código y test. Esto implica que las iteraciones duren menos. **Esto no necesariamente implica una mayor productividad.**



Con este modelo el equipo reaccionará mejor a los fallos ya que la retroalimentación será más frecuente. **Es un error planificar una iteración como una mini cascada:**



El proceso de pruebas:

- Se determina qué se va a probar, cómo, quién y cuándo. Además se determinará que se hará si los planes no se ajustan a la realidad
- Se determinan los casos de prueba
- Se implementan y ejecutan los tests
- Se verifica que se alcanzan los **completion criteria** y se genera un informe
- Se trata de asegurar que los informes están disponibles

TDD – Test Driven Development

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

TDD - Aproximaciones

- **Clásica:** Usar objetos reales, stubs y dobles. Soporta **INSIDE-OUT**, donde se comienza por bajo nivel y NO se necesitan stubs.
- **Mockist:** Usa mocks. Por eso surgió BDD. Soporta **OUTSIDE-IN**, donde se empieza por el nivel más alto. Se utilizan mocks para sustituir las capas inferiores.

TDD – Factores a considerar/Costes asociados

- Cada prueba debe operar de forma aislada
- Se debe evitar hinchar las pruebas unitarias cuya cobertura se solapa (estrategia de pruebas)
- Es difícil de mantener a medida que crece el proyecto (mas pruebas unitarias)
- El proceso de construcción tiene que ejecutar todas las pruebas cada vez.

BDD – Behaviour Driven Development

Es un conjunto de prácticas de ingeniería software diseñadas para ayudar a los equipos a construir y entregar software más valioso y de mayor calidad más rápido.

BDD no implementa tests unitarios, implementa especificaciones de bajo nivel. Cambia la forma en la cual escribiremos los tests centrándonos más en qué debería hacer el sistema, es decir, podremos escribir tests más focalizados en los objetivos concretos de dichos tests.

BDD – Ventajas e inconvenientes

- **Ventajas (4):**
 - Reducen el trabajo inútil. El desarrollo se centra solamente en features que proporcionan valor al negocio.
 - Reducen el coste de desarrollo.
 - Fácil de mantener. El código actúa como una documentación
 - Entregas más rápidas.
- **Desventajas (3):**
 - Se requiere un elevado compromiso entre los stakeholders del proyecto
 - Requiere planificaciones ágiles o iterativas
 - Requiere personal capacitado para escribir los tests correctamente