



BDD con Cucumber



© JMA 2020. All rights reserved

INTRODUCCIÓN

© JMA 2020. All rights reserved

Desarrollo Guiado por Pruebas (TDD)

- El Desarrollo Guiado por Pruebas, es una técnica de programación (definida por KentBeck); consistente en desarrollar primero el código que pruebe una característica o funcionalidad deseada antes que el código que implementa dicha funcionalidad.
- El objetivo a lograr es que no exista ninguna funcionalidad que no esté avalada por una prueba.
- Lo primero que hay que aprender de TDD son sus reglas básicas:
 - No añadir código sin escribir antes una prueba que falle
 - Eliminar el Código Duplicado empleando Refactorización

© JMA 2020. All rights reserved

Ritmo TDD

- TDD invita a seguir una serie de tareas ordenadas, que a menudo se denomina ritmo TDD, y que se basa en los siguientes pasos:
 1. Escribir una prueba que demuestre la necesidad de escribir código.
 2. Escribir el mínimo código para que el código de pruebas compile
 3. Implementar exclusivamente la funcionalidad demandada por las pruebas
 4. Mejorar el código (Refactoring) sin añadir funcionalidad
 5. Volver al primer paso
- Este ritmo permite formalizar las tareas que se han de realizar para conseguir un código fácil de mantener, bien diseñado y que se puede probar automáticamente.

© JMA 2020. All rights reserved

Beneficios de TDD

- Reducen el número de errores y bugs ya que éstos, aplicando TDD, se detectan antes incluso de crearlos.
- Facilitan entender el código y que, eligiendo una buena nomenclatura, sirven de documentación.
- Facilitan mantener el código:
 - Protege ante cambios, los errores que surgen al aplicar un cambio se detectan (y corrigen) antes de subir ese cambio.
 - Protegen ante errores de regresión (rollbacks a versiones anteriores).
 - Dan confianza.
- Facilitan desarrollar ciñéndose a los requisitos.
- Ayudan a encontrar inconsistencias en los requisitos
- Ayudan a especificar comportamientos
- Ayudan a refactorizar para mejorar la calidad del código (Clean code)
- A medio/largo plazo aumenta (y mucho) la productividad.

© JMA 2020. All rights reserved

Estrategia RED – GREEN

- Se recomienda una estrategia de test unitarios conocida como **RED** (fallo) – **GREEN** (éxito), es especialmente útil en equipos de desarrollo ágil.
- Una vez que entendamos la lógica y la intención de un test unitario, hay que seguir estos pasos:
 - Escribe el código del test (**Stub**) para que compile (pase de **RED** a **GREEN**)
 - Inicialmente la compilación fallará **RED** debido a que falta código
 - Implementa sólo el código necesario para que compile **GREEN** (aún no hay implementación real).
 - Escribe el código del test para que se **ejecute** (pase de **RED** a **GREEN**)
 - Inicialmente el test fallará **RED** ya que no existe funcionalidad.
 - Implementa la funcionalidad que va a probar el test hasta que se ejecute adecuadamente **GREEN**.
 - **Refactoriza** el test y el código una vez que este todo **GREEN** y la solución vaya evolucionando.

© JMA 2020. All rights reserved

Ritmo TDD



© JMA 2020. All rights reserved

Refactorizar el código en pruebas

- Una refactorización es un cambio que está pensado para que el código se ejecute mejor o para que sea más fácil de comprender.
- No está pensado para alterar el comportamiento del código y, por tanto, no se cambian las pruebas.
- Se recomienda realizar los pasos de refactorización independientemente de los pasos que amplían la funcionalidad.
- Mantener las pruebas sin cambios aporta la confianza de no haber introducido errores accidentalmente durante la refactorización.

© JMA 2020. All rights reserved

Desarrollo Dirigido por Comportamiento (BDD)

- El Desarrollo Dirigido por Comportamiento (Behaviour Driver Development) es una evolución de TDD (Test Driven Development o Desarrollo Dirigido por Pruebas), el concepto de BDD fue inicialmente introducido por Dan North como respuesta a los problemas que surgían al enseñar TDD.
- En BDD también vamos a escribir las pruebas antes de escribir el código fuente, pero en lugar de pruebas unitarias, lo que haremos será escribir pruebas que verifiquen que el comportamiento del código es correcto desde el punto de vista de negocio. Tras escribir las pruebas escribimos el código fuente de la funcionalidad que haga que estas pruebas pasen correctamente. Después refactorizamos el código fuente.
- Partiremos de historias de usuario, siguiendo el modelo “Como [rol] quiero [característica] para [los beneficios]”. A partir de aquí, en lugar de describir en 'lenguaje natural' lo que tiene que hacer esa nueva funcionalidad, vamos a usar un lenguaje ubicuo (un lenguaje semiformal que es compartido tanto por desarrolladores como personal no técnico) que nos va a permitir describir todas nuestras funcionalidades de una única forma.

© JMA 2020. All rights reserved

BDD

- Para empezar a hacer BDD sólo nos hace falta conocer 5 palabras, con las que construiremos sentencias con las que vamos a describir las funcionalidades:
 - Feature (característica): Indica el nombre de la funcionalidad que vamos a probar. Debe ser un título claro y explícito. Incluimos aquí una descripción en forma de historia de usuario: “Como [rol] quiero [característica] para [los beneficios]”. Sobre esta descripción empezaremos a construir nuestros escenarios de prueba.
 - Scenario: Describe cada escenario que vamos a probar.
 - Given (dado): Provee el contexto para el escenario en que se va a ejecutar el test, tales como el punto donde se ejecuta el test, o prerequisites en los datos. Incluye los pasos necesarios para poner al sistema en el estado que se desea probar.
 - When (cuando): Especifica el conjunto de acciones que lanzan el test. La interacción del usuario que acciona la funcionalidad que deseamos testear.
 - Then (entonces): Especifica el resultado esperado en el test. Observamos los cambios en el sistema y vemos si son los deseados.

© JMA 2020. All rights reserved

Desarrollo Dirigido por Tests de Aceptación (ATDD)

- El Desarrollo Dirigido por Test de Aceptación (ATDD), técnica conocida también como Story Test-Driven Development (STDD), es una variación del TDD pero a un nivel diferente.
- Las pruebas de aceptación o de cliente son el criterio escrito de que un sistema cumple con el funcionamiento esperado y los requisitos de negocio que el cliente demanda. Son ejemplos escritos por los dueños de producto. Es el punto de partida del desarrollo en cada iteración.
- ATDD/STDD es una forma de afrontar la implementación de una manera totalmente distinta a las metodologías tradicionales. Cambia el punto de partida, la forma de recoger y formalizar las especificaciones, sustituye los requisitos escritos en lenguaje natural (nuestro idioma) por historias de usuario con ejemplos concretos ejecutables de como el usuario utilizara el sistema, que en realidad son casos de prueba. Los ejemplos ejecutables surgen del consenso entre los distintos miembros del equipo y el usuario final.
- La lista de ejemplos (pruebas) de cada historia, se escribe en una reunión que incluye a dueños de producto, usuarios finales, desarrolladores y responsables de calidad. Todo el equipo debe entender qué es lo que hay que hacer y por qué, para concretar el modo en que se certifica que el software lo hace.

© JMA 2020. All rights reserved

ATDD

- El algoritmo o ritmo es el mismo de tres pasos que en el TDD practicado exclusivamente por desarrolladores pero a un nivel superior.
- En ATDD hay dos prácticas claves:
 - Antes de implementar (fundamental lo de antes de implementar) una necesidad, requisito, historia de usuario, etc., los miembros del equipo colaboran para crear escenarios, ejemplos, de cómo se comportará dicha necesidad.
 - Después, el equipo convierte esos escenarios en pruebas de aceptación automatizadas. Estas pruebas de aceptación típicamente se automatizan usando Selenium o similares, “frameworks” como Cucumber, etc.

© JMA 2020. All rights reserved

Data Driven Testing (DDT)

- Se basa en la creación de tests para ejecutarse en simultáneo con sus conjuntos de datos relacionados en un framework. El framework provee una lógica de test reusable para reducir el mantenimiento y mejorar la cobertura de test. La entrada y salida (del criterio de test) pueden ser resguardados en uno o más lugares del almacenamiento central o bases de datos, el formato real y la organización de los datos serán específicos para cada caso.
- Todo lo que tiene potencial de cambiar (también llamado "variabilidad," e incluye elementos como el entorno, puntos de salida, datos de test, ubicaciones, etc) está separado de la lógica del test (scripts) y movido a un 'recurso externo'. Esto puede ser configuración o conjunto de datos de test. La lógica ejecutada en el script está dictada por los valores.
- Los datos incluyen variables usadas tanto para la entrada como la verificación de la salida. En casos avanzados (y maduros) los entornos de automatización pueden ser obtenidos desde algún sistema usando los datos reales o un "sniffer", el framework DDT por lo tanto ejecuta pruebas sobre la base de lo obtenido produciendo una herramienta de test automáticos para regresión.

© JMA 2020. All rights reserved

BEHAVIOUR DRIVER DEVELOPMENT

© JMA 2020. All rights reserved

Desarrollo Dirigido por Comportamiento (BDD)

- El Desarrollo Dirigido por Comportamiento (Behaviour Driver Development) es una evolución de TDD (Test Driven Development o Desarrollo Dirigido por Pruebas), el concepto de BDD fue inicialmente introducido por Dan North como respuesta a los problemas que surgían al enseñar TDD.
- Además del enfoque TDD también se guía por el enfoque Specification by Example (SBE). Dicho método define los requisitos y pruebas funcionales de la aplicación por medio de ejemplos realistas y no abstractos.
- En BDD también vamos a escribir las pruebas antes de escribir el código fuente, pero en lugar de pruebas unitarias, lo que haremos será escribir pruebas que verifiquen que el comportamiento del código es correcto desde el punto de vista de negocio. Tras escribir las pruebas escribimos el código fuente de la funcionalidad que haga que estas pruebas pasen correctamente. Una vez superadas, refactorizamos el código fuente.
- BDD se define en un idioma ubicuo, común entre todos los participantes, lo que mejora la comunicación y cierra la brecha entre la gente de negocios y la gente técnica al:
 - Fomentar la colaboración entre roles para construir una comprensión compartida del problema a resolver.
 - Trabajar en iteraciones pequeñas y rápidas para aumentar la retroalimentación y el flujo de valor
 - Producir documentación del sistema que se verifica automáticamente contra el comportamiento del sistema.

© JMA 2020. All rights reserved

Proceso ágil

- BDD enfoca el trabajo colaborativo en ejemplos concretos del mundo real que ilustran cómo queremos que se comporte el sistema. Usamos esos ejemplos para guiarnos desde el concepto hasta la implementación, en un proceso de colaboración continua.
- BDD no reemplaza el proceso ágil existente, lo mejora con un conjunto de complementos para el proceso existente que hará que el equipo sea más capaz de cumplir las promesas de agilidad: lanzamientos oportunos y confiables de software funcional que satisfaga las necesidades cambiantes de la organización, lo que requiere algo de disciplina y esfuerzo de mantenimiento.
- BDD fomenta el trabajo en iteraciones rápidas, desglosando continuamente los problemas de su usuario en partes pequeñas que pueden fluir a través de su proceso de desarrollo lo más rápido posible.
- Hay que tener en cuenta antes de implementar BDD:
 - Cada requisitos debe convertirse en historias de usuario, definiendo ejemplos concretos.
 - Cada ejemplo debe ser un escenario de un usuario en el sistema.
 - Ser consciente de la necesidad de definir "la especificación del comportamiento de un usuario" en lugar de "la prueba unitaria de una clase".
 - Comprender la fórmula 'Given-When-Then' u otras como las historias de usuario 'Role-Feature-Reason'.

© JMA 2020. All rights reserved

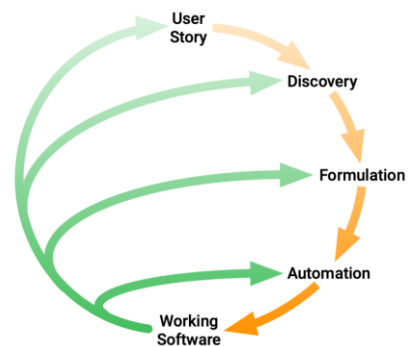
Beneficios

- Ya no estás definiendo 'pruebas', sino que estás definiendo 'comportamientos'.
- Mejora la comunicación entre desarrolladores, testers, usuarios y la dirección.
- Debido a que BDD se especifica utilizando un lenguaje simplificado y común, la curva de aprendizaje es mucho más corta que TDD.
- Como su naturaleza no es técnica, puede llegar a un público más amplio.
- El enfoque de definición ayuda a una aceptación común de las funcionalidades previamente al desarrollo.
- Esta estrategia encaja bien en las metodologías ágiles, ya que en ellas se especifican los requisitos como historias de usuario y de aceptación.

© JMA 2020. All rights reserved

Prácticas

- Esencialmente, la actividad diaria de BDD es un proceso iterativo de tres pasos:
 - Descubrimiento: Primero, tome el próximo pequeño cambio al sistema, una historia de usuario, y consulte sobre ejemplos concretos de la nueva funcionalidad para explorar, descubrir y acordar los detalles de lo que se espera que se haga.
 - Formulación: Luego, documente esos ejemplos de una manera que pueda automatizarse y verifique si están de acuerdo.
 - Automatización: Finalmente, implemente el comportamiento descrito por cada ejemplo documentado, comenzando con una prueba automatizada para guiar el desarrollo del código.
- La idea es hacer que cada cambio sea pequeño e iterar rápidamente, volviendo a subir un nivel cada vez que necesite más información. Cada vez que automatiza e implementa un nuevo ejemplo, agrega algo valioso al sistema y está listo para responder a los comentarios.



© JMA 2020. All rights reserved

Descubrimiento: lo que podría hacer

- La parte más difícil de construir un sistema de software es decidir con precisión qué construir.
- Aunque la documentación y las pruebas automatizadas son producidas por un equipo de BDD, se puede pensar en ellas como buenos efectos secundarios. El objetivo real es un software valioso y que funcione, y la forma más rápida de lograrlo es a través de conversaciones entre las personas que están involucradas en imaginar y entregar ese software.
- BDD ayuda a los equipos a tener las conversaciones correctas en el momento adecuado para minimizar la cantidad de tiempo dedicado a las reuniones y maximizar la cantidad de código valioso que produce.
- Utilizamos conversaciones estructuradas, llamadas talleres de descubrimiento, que se centran en ejemplos del mundo real del sistema desde la perspectiva de los usuarios. Estas conversaciones aumentan la comprensión compartida de nuestro equipo sobre las necesidades de nuestros usuarios, las reglas que rigen cómo debe funcionar el sistema y el alcance de lo que se debe hacer.
- También puede revelar brechas en nuestra comprensión, donde necesitamos más información antes de saber qué hacer.
- El escrutinio de una sesión de descubrimiento a menudo revela una funcionalidad de baja prioridad que se puede diferir del alcance de una historia de usuario, lo que ayuda al equipo a trabajar en incrementos más pequeños y mejora su flujo.
- El descubrimiento es el lugar adecuado para comenzar en BDD. No se disfrutará mucho de las otras dos prácticas hasta que se haya dominado el descubrimiento.

© JMA 2020. All rights reserved

Formulación: Qué debe hacer

- Tan pronto como hayamos identificado al menos un ejemplo valioso de nuestras sesiones de descubrimiento, ahora podemos formular cada ejemplo como documentación estructurada. Esto nos brinda una forma rápida de confirmar que realmente tenemos una comprensión compartida de qué construir.
- A diferencia de la documentación tradicional, utilizamos un medio que puede ser leído tanto por humanos como por máquinas, de modo que:
 - Podemos obtener comentarios de todo el equipo sobre nuestra visión compartida de lo que estamos construyendo.
 - Podremos automatizar estos ejemplos para guiar nuestro desarrollo de la implementación.
- Al escribir esta especificación ejecutable en colaboración, establecemos un lenguaje compartido para hablar sobre el sistema. Esto nos ayuda a usar la terminología del dominio del problema en todo el código.

© JMA 2020. All rights reserved

Automatización: lo que realmente hace

- Ahora que tenemos nuestra especificación ejecutable, podemos usarla para guiar nuestro desarrollo de la implementación.
- Tomando un ejemplo cada vez, lo automatizamos conectándolo al sistema como prueba. La prueba falla porque aún no hemos implementado el comportamiento que describe. Ahora desarrollamos el código de implementación, utilizando ejemplos de nivel inferior del comportamiento de los componentes internos del sistema para guiarnos según sea necesario.
- Los ejemplos automatizados funcionan como guías, ayudándonos a mantener nuestro trabajo de desarrollo encaminado.
- Cuando necesitemos volver y mantener el sistema más tarde, los ejemplos automatizados nos ayudarán a comprender qué está haciendo el sistema actualmente y a realizar cambios de manera segura sin romper nada involuntariamente.
- Esta retroalimentación rápida y repetible reduce la carga de las pruebas de regresión manual, liberando a las personas para que realicen un trabajo más interesante, como las pruebas exploratorias.

© JMA 2020. All rights reserved

Taller de descubrimiento

- Un taller de descubrimiento es una conversación en la que personas técnicas y de negocios colaboran para explorar, descubrir y ponerse de acuerdo tanto como puedan sobre el comportamiento deseado para una historia de usuario.
- Se debería realizar lo más tarde posible antes de que comience el desarrollo de una nueva historia de usuario, para evitar que se pierdan detalles y le da al equipo suficiente espacio para cambiar sus planes en caso de que surjan nuevos detalles.
- Una buena regla general es que asistan de 3 a 6 personas, pero como mínimo deben estar presentes los tres amigos: un propietario del producto, un desarrollador y un probador. El propietario de su producto identificará el problema que el equipo debería intentar resolver, el desarrollador abordará cómo crear una solución para dicho problema y el probador abordará cualquier caso extremo que pueda surgir.
- Idealmente, un taller de descubrimiento debería durar solo entre 25 y 30 minutos por historia. Si se necesita más tiempo, es probable que la historia sea demasiado extensa y deba desglosarse, o que falten algunos detalles. En este último caso, se debe dejar de lado la historia ya que el equipo necesita investigar más.
- El propósito de un taller de descubrimiento es brindar a todas las partes interesadas, tanto técnicas como no técnicas, una comprensión compartida del trabajo en cuestión. Hacerlo fomenta la colaboración interfuncional, un aumento en la retroalimentación y cubre cualquier detalle perdido o suposiciones incorrectas realizadas.
- El taller de descubrimiento es una pieza muy importante del ciclo de vida de BDD. Sin él, seguramente se encontrarán con problemas de comunicación y el equipo no descubrirá su desconocimiento, lo que realmente podría obstaculizar el éxito de su proyecto.

© JMA 2020. All rights reserved

Formulación

- Para empezar con la formulación sólo nos hace falta conocer 5 palabras, con las que construiremos sentencias con las que vamos a describir las funcionalidades:
 - Feature (característica): Indica el nombre de la funcionalidad que vamos a probar. Debe ser un título claro y explícito. Incluimos aquí una descripción en forma de historia de usuario: “Como [rol] quiero [característica] para [los beneficios]”. Sobre esta descripción empezaremos a construir nuestros escenarios de prueba.
 - Scenario (escenario): Describe cada ejemplo que vamos a probar.
 - Given (dado): Provee el contexto para el escenario en que se va a ejecutar el test, tales como el punto donde se ejecuta el test, o prerequisites en los datos. Incluye los pasos necesarios para poner al sistema en el estado que se desea probar.
 - When (cuando): Especifica el conjunto de acciones que lanzan el test. La interacción del usuario que acciona la funcionalidad que deseamos testear.
 - Then (entonces): Especifica el resultado esperado en el test. Observamos los cambios en el sistema y vemos si son los deseados.

© JMA 2020. All rights reserved

Estilos

- Los escenarios deben describir el comportamiento previsto del sistema, no la implementación. De esa manera, cuando cambie la implementación de una función, solo se tendrá que cambiar los pasos del proceso entre bastidores. El comportamiento no tiene que cambiar solo porque la implementación lo haga.
- Una forma de hacer que los escenarios sean más fáciles de mantener y menos frágiles es usar un estilo declarativo. El estilo declarativo describe el comportamiento de la aplicación (que), en lugar de los detalles de implementación (como). Los escenarios declarativos se leen mejor como "documentación viva". Un estilo declarativo ayuda a concentrarse en el valor que obtiene el cliente, en lugar de las pulsaciones de teclas que utilizará.
- Las pruebas imperativas comunican detalles y, en algunos contextos, este estilo de prueba es apropiado, pero al estar tan estrechamente vinculadas a la mecánica de la interfaz de usuario actual, a menudo requieren más trabajo para mantenerlos. Cada vez que cambia la implementación, las pruebas también.

© JMA 2020. All rights reserved

Estilo imperativo

FEATURE: los suscriptores ven diferentes artículos según su nivel de suscripción

SCENARIO: los suscriptores gratuitos solo ven los artículos gratuitos

GIVEN: los usuarios con una suscripción gratuita pueden acceder a "FreeArticle1" pero no a "PaidArticle1"

WHEN: escribo "freeFrieda@example.com" en el campo de correo electrónico

AND: escribo "validPassword123" en el campo de contraseña

AND: presiono el botón "Enviar"

THEN: Veo "FreeArticle1" en la página de inicio

AND: no veo "PaidArticle1" en la página de inicio

SCENARIO: el suscriptor con una suscripción de pago puede acceder a "FreeArticle1" y "PaidArticle1"

GIVEN: estoy en la página de inicio de sesión

WHEN: escribo "paidPatty@example.com" en el campo de correo electrónico

AND: escribo "validPassword123" en el campo de contraseña

AND: presiono el botón "Enviar"

THEN: veo "FreeArticle1" y "PaidArticle1" en la página de inicio

© JMA 2020. All rights reserved

Estilo declarativo

FEATURE: los suscriptores ven diferentes artículos según su nivel de suscripción

SCENARIO: los suscriptores gratuitos solo ven los artículos gratuitos

GIVEN: Free Frieda tiene una suscripción gratuita

WHEN: Free Frieda inicia sesión con sus credenciales válidas

THEN: ve un artículo gratuito.

SCENARIO: el suscriptor con una suscripción de pago puede acceder a artículos gratuitos y de pago

GIVEN: Paid Patty tiene una suscripción de pago de nivel básico

WHEN: Paid Patty inicia sesión con sus credenciales válidas

THEN: ve un artículo gratuito y un artículo de pago.

© JMA 2020. All rights reserved

Estilo declarativo

- Con un estilo declarativo, cada paso comunica una idea, pero no se especifican los valores exactos. Los detalles de cómo el usuario interactúa con el sistema, como qué artículos específicos son gratuitos o de pago, y el nivel de suscripción de los diferentes usuarios de prueba, se especifican en las definiciones de los pasos (el código de automatización que interactúa con el sistema).
- Los paquetes de suscripción podrían cambiar en el futuro. La empresa podría cambiar el contenido que está disponible para los suscriptores en planes gratuitos y pagos, sin tener que cambiar este escenario y otros escenarios que usan las mismas definiciones de pasos.
- Si se agrega otro nivel de suscripción más tarde, es fácil agregar un escenario para eso. Al evitar términos como "hacer clic en un botón" que sugieren implementación, el escenario es más resistente a los detalles de implementación de la interfaz de usuario.
- La intención del escenario sigue siendo la misma, incluso si la implementación cambia más adelante. Además, tener demasiados detalles de implementación en un escenario dificulta la comprensión del comportamiento previsto que ilustra.

© JMA 2020. All rights reserved

Ciclo del desarrollo orientado a comportamiento

1. Describe el comportamiento del software a través de pruebas
2. Crea el test que verifique el comportamiento descrito
3. Ejecuta los test y comprueba que el nuevo test falla
4. Implementa el nuevo comportamiento
5. Ejecuta los test
6. Mejora el código
7. Ejecuta los test
8. Repite

© JMA 2020. All rights reserved

Cucumber

- Cucumber es una herramienta que admite el desarrollo impulsado por el comportamiento (BDD).
- Cucumber lee las especificaciones ejecutables escritas en texto sin formato y valida que el software haga lo que dicen esas especificaciones. Las especificaciones consisten en múltiples ejemplos o escenarios.
 Scenario: Breaker guesses a word
 Given the Maker has chosen a word
 When the Breaker makes a guess
 Then the Maker is asked to score
- Cada escenario es una lista de pasos que Cucumber debe seguir. Cucumber verifica que el software cumpla con la especificación y genera un informe que indica ☒ éxito o **✗** falla para cada escenario.
- Para que Cucumber entienda los escenarios, deben seguir unas reglas básicas de sintaxis, llamadas Gherkin.
- Cucumber está disponible para la mayoría de los principales lenguajes de programación. Se recomienda elegir la misma implementación que para la plataforma o lenguaje de programación que el código de producción.

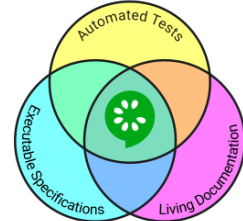
© JMA 2020. All rights reserved

GHERKIN

© JMA 2020. All rights reserved

Gherkin

- Gherkin es un conjunto de reglas gramaticales que hace que el texto sin formato esté lo suficientemente estructurado para que Cucumber lo entienda.
- Gherkin sirve para múltiples propósitos:
 - Especificación ejecutable inequívoca
 - Pruebas automatizadas usando Cucumber
 - Documentar cómo se comporta realmente el sistema
 - Única fuente de verdad
- La gramática Cucumber está traducida a muchos idiomas hablados para que cada equipo pueda usar las palabras clave en su propio idioma.
- Los documentos Gherkin utilizan archivos de texto .feature y, por lo general, se versionan en el control de código fuente junto con el software.



© JMA 2020. All rights reserved

Sintaxis

- Gherkin utiliza un conjunto de palabras clave especiales (o su traducción) para dar estructura y significado a las especificaciones ejecutables.
- Los comentarios comienzan con cero o más espacios, seguidos de un signo de almohadilla (#) y el texto. Se comentan líneas completas y pueden ir en cualquier parte del archivo. Gherkin no admite actualmente los bloques comentados.
- Cada línea que no sea una línea en blanco o comentada debe comenzar con una palabra clave Gherkin, seguida de cualquier texto que se desee, literal descriptivo. Las únicas excepciones son las descripciones de características y escenarios.
- Se pueden utilizar espacios o tabulaciones para la sangría. El nivel de sangría recomendado es de dos espacios.
- Las palabras clave son sensibles a mayúsculas y minúsculas (notación Pascal). Algunas palabras clave van seguidas de dos puntos (:) y otras no. Si se agrega dos puntos después de una palabra clave que no debe tenerlos, se ignorarán sus pruebas.
- El literal descriptivo de cada paso se compara con un bloque de código, llamado definición de paso, que implementa el paso en el caso de prueba.

© JMA 2020. All rights reserved

Palabras clave (ingles)

Palabras clave principales

- Feature
- Rule (*a partir del Gherkin 6*)
- Scenario (o Example)
- Given, When, Then, And, But (o *) *para pasos*
- Background
- Scenario Outline (o Scenario Template)
- Examples (o Scenarios)

Palabras clave secundarias

- "" (Cadenas de documentos)
- | (Tablas de datos)
- @ (Etiquetas)
- # (Comentarios)

© JMA 2020. All rights reserved

Localización

English Keyword	Equivalencia # language: es
Feature	Característica, Necesidad del negocio, Requisito
Background	Antecedentes
Scenario	Ejemplo, Escenario
Scenario Outline	Esquema del escenario
Examples	Ejemplos
Given	*, Dado, Dada, Dados, Dadas
When	*, Cuando
Then	*, Entonces
And	*, Y, E
But	*, Pero
Rule	Regla, Regla de negocio

El encabezado # language: en la primera línea de un archivo de características le dice a Cucumber qué idioma debe usar (inglés de forma predeterminada), por ejemplo, # language: es.

© JMA 2020. All rights reserved

Feature:

- El propósito de la palabra clave Feature (Característica, Necesidad del negocio, Requisito) es proporcionar una descripción de alto nivel de una función de software y agrupar escenarios relacionados. Por fichero .feature solo puede haber una sola Feature.
- La primera palabra clave principal en un documento Gherkin siempre debe ser Feature, seguida de una : y un nombre como texto breve que describa la función.
- Se puede agregar texto en formato libre de varias líneas debajo de Feature (siempre que ninguna línea comience con una palabra clave) para agregar una descripción ampliada. El texto se puede formatear con Markdown dado que los formateadores, incluido el formateador HTML oficial, lo admiten. La descripción finaliza cuando una línea comienza con la palabra clave Background, Rule, Scenario/Example o Scenario Outline (o sus alias).
- El nombre y la descripción opcional no tienen un significado especial para Cucumber, son ignoradas en tiempo de ejecución, pero están disponibles para generar informes (se incluyen en herramientas de informes como el formateador HTML oficial). Su propósito es proporcionar un lugar para documentar aspectos importantes de la función, como una breve explicación y una lista de reglas de negocios (criterios generales de aceptación).
- Las descripciones en formato libre también se pueden colocar debajo de Example/Scenario, Background, Scenario Outline y Rule.

© JMA 2020. All rights reserved

Rule:

- El propósito de la palabra clave Rule (Regla, Regla de negocio) es representar una regla de negocio que debe implementarse. Proporcionan información adicional para una función.
- Las reglas se utilizan para agrupar varios escenarios que pertenecen a dicha regla de negocio. Una Rule debe contener uno o más escenarios que ilustren la regla en particular.
- La palabra clave Rule se incorporó a Gherkin en la versión 6 y es todavía bastante nueva. Ya se ha portado en muchas implementaciones de Cucumber pero hay implementaciones que no son compatibles.

© JMA 2020. All rights reserved

Scenario: o Example:

- Un escenario es un ejemplo concreto que ilustra una regla de negocio y consiste en una lista de pasos Given, When, Then.
- Se definen con la palabra clave Scenario o Example (Escenario, Ejemplo), seguida de : y el literal descriptivo. Puede contar con una descripción en formato libre en las líneas siguientes.
- Un ejemplo puede tener tantos pasos como desee, pero se recomienda de 3 a 5 pasos por ejemplo. Tener demasiados pasos hará que el ejemplo pierda su poder expresivo como especificación y documentación. Además de ser una especificación y documentación, un ejemplo también es una prueba. En su conjunto, los ejemplos son casos de uso, una especificación ejecutable del sistema.
- Los ejemplos siguen el patrón:
 - Describir un contexto inicial (pasos Given)
 - Describir un evento o acción (pasos When)
 - Describir un resultado esperado (pasos Then)
- Cucumber ejecuta de uno en uno cada paso del escenario, en la misma secuencia en la se han escrito. Para ejecutar un paso, busca una definición de paso coincidente. Las palabras clave no se tienen en cuenta al buscar la definición del paso, por lo que no se puede tener un paso con el mismo texto que otro paso aunque sea de otro tipo.

© JMA 2020. All rights reserved

Given, When, Then, And y But

- Se pueden tener varios pasos Given, When o Then sucesivos, pero para hacer que el ejemplo tenga una estructura más fluida, se pueden reemplazar los sucesivos Given, When o Then con And o But.
- Gherkin también admite el uso de un asterisco (*) en lugar de cualquiera de las palabras And o But. Esto puede ser útil cuando algunos pasos son una lista de cosas, por lo que se puede expresar más elegantemente como una lista con viñetas donde el lenguaje natural con And o But podría no leerse tan bien.
- Los pasos **Given** (Dado, Dada, Dados, Dadas) se utilizan para describir el contexto inicial del sistema: la escena del escenario. Por lo general, en pasado, es la situación de partida.
- Cuando Cucumber ejecuta un paso Given, configura el sistema para que esté en un estado bien definido, como crear y configurar objetos o agregar datos a una base de datos de prueba (las precondiciones de los casos de uso).
- El propósito de los pasos Given es llevar al sistema a un estado conocido antes de que el usuario (o el sistema externo) comience a interactuar con el sistema (en los pasos When). Se debe evitar hablar sobre las interacciones del usuario en los pasos Given.

© JMA 2020. All rights reserved

Given, When, Then, And y But

- Los pasos **When** (Cuando) se utilizan para describir un evento o una acción. Puede ser una persona que interactúa con el sistema (hago algo) o puede ser un evento desencadenado por otro sistema (pasa algo).
- Las descripciones de los pasos deben abstraerse de las tecnologías o interfaces de usuario concretas, los detalles de implementación deben ocultarse en definiciones de los pasos.
- Se recomienda un solo paso When por escenario. Si se necesitan más, generalmente es una señal de que se debe dividir el escenario en varios escenarios.
- Los pasos **Then** (Entonces) se utilizan para describir un resultado o resultado esperado. La definición de paso de un paso Then debe usar una aserción o expectativa para comparar el resultado real (lo que el sistema realmente hace) con el resultado esperado (lo que dice el paso que se supone que debe hacer el sistema).
- Un resultado debe ser en una salida observable. Es decir, algo que sale del sistema (informe, interfaz de usuario, mensaje) accesible para el usuario que define el ejemplo, y no un comportamiento profundamente enterrado dentro del sistema (como un registro en una base de datos).

© JMA 2020. All rights reserved

Argumentos en el paso

- El literal descriptivo de los pasos pueden contener el valor de los argumentos que se pasaran a los parámetros de la definición del paso. Los valores se extraen del literal (los numéricos) o delimitados con "dobles comillas" (las cadenas) y deben coincidir con el orden de los parámetros.
- En algunos casos, es posible que se desee pasar más datos a un paso de los que caben en una sola línea. Para ello Gherkin tiene Doc Strings y Data Tables.
- Las Doc Strings son útiles para pasar un texto más grande a una definición de paso. Se pasará automáticamente como el último argumento en la definición del paso. El texto debe estar delimitado por tres comillas dobles (""") o triple acento grave (```) en propias sus líneas. La sangría dentro de las comillas triples es significativa, en los delimitadores no.

```
Given a blog post named "Random" with Markdown body
'''
  Lorem ipsum
  =====
  Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  Nulla in orci non elit consequat accumsan vitae facilisis mi.
'''
```

© JMA 2020. All rights reserved

Argumentos en el paso

- La Data Tables son útiles para pasar una lista de valores a una definición de paso. Se pasará automáticamente como el último argumento en la definición del paso. El texto de las filas y columnas debe estar delimitado por pipes (|), cada fila en su línea.

Given the following users exist:

name	email	twitter
Aslak	aslak@cucumber.io	@aslak_hellesoy
Julien	julien@cucumber.io	@jbpros
Matt	matt@cucumber.io	@mattwynne

- Si desea utilizar un carácter de nueva línea en una celda de la tabla, se puede escribir como `\n`, el `|` se puede escapar como `\|`, y finalmente, la `\` se puede escapar con `\\`.

© JMA 2020. All rights reserved

Background

- Ocasionalmente, un conjunto de los mismos pasos Given se repiten en todos los escenarios de un archivo Feature.
- Dicha repetición es una indicación de que esos pasos no son esenciales para describir los escenarios; son detalles troncales de la característica. Literalmente, se puede mover dichos Given pasos al fondo, agrupándolos en una sección Background, expresándolos como antecedentes de los escenarios.
- Un Background (Antecedentes) permite agregar algo de contexto a los escenarios que le siguen. Pueden contener uno o más pasos Given, que se ejecutan antes de cada escenario, pero después de cualquier gancho Antes .
- El Background se coloca antes del primer Scenario/Example, al mismo nivel de sangría.
- Solo se puede tener un Background por Feature o Rule. Si se necesitan diferentes Background para diferentes escenarios, se debe repartir los escenarios en más de una Rule o Feature.

© JMA 2020. All rights reserved

Scenario Outline o Scenario Template

- La palabra clave Scenario Outline (Esquema del escenario), o su sinónimo Scenario Template, se puede utilizar para ejecutar el mismo Scenario varias veces, con diferentes combinaciones de valores en vez de copiar y pegar escenarios para usar diferentes valores que se vuelve tedioso, repetitivo e inmantenible.
- Los esquemas de escenario nos permiten expresar estos escenarios de manera más concisa mediante el uso de una plantilla con parámetros delimitados con < > y una tabla de Examples (Ejemplos) con los valores de los argumentos.
- Un Scenario Outline debe contener una o más secciones Examples (o Scenarios) debajo de ella. Los pasos se interpretan como una plantilla que nunca se ejecuta directamente. En su lugar, el Scenario Outline se ejecuta una vez para cada fila en la sección Examples (sin contar la primera fila de encabezado).
- Los pasos pueden usar parámetros delimitados <> que hacen referencia a los encabezados en la tabla de ejemplos (la primera fila). Cucumber reemplazará estos parámetros con valores de la tabla antes de intentar hacer coincidir el paso con una definición de paso.

Scenario Outline: eating
Given there are <start> cucumbers
When I eat <eat> cucumbers
Then I should have <left> cucumbers

Examples:
start	eat	left
12	5	7
20	5	15

© JMA 2020. All rights reserved

Etiquetas

- Las etiquetas son una manera de organizar las características o escenarios, complementando la documentación. Se pueden utilizar para dos propósitos:
 - Para ejecutar o ignorar un subconjunto de características o escenarios
 - Para la restricción de hooks a un subconjunto de escenarios
- Los tags son nombres de libre disposición que empiezan por @ y preceden inmediatamente (línea anterior) al elemento que etiquetan. Un elemento puede tener tantas etiquetas como se desee, separándolas con espacios. Las etiquetas se pueden colocar encima de los siguientes elementos Gherkin:
 - Feature
 - Scenario, Example
 - Scenario Outline
- Un Scenario Outline puede usar tener varias tablas de Examples si se etiquetan los diferentes ejemplos.
- Las etiquetas afectan a sus elementos secundarios, las etiquetas de una característica engloba a todos sus escenarios.

© JMA 2020. All rights reserved

Ejemplo

```
# language: es
Característica: Suma de dos números
Como matemático novato
Yo quiero obtener la suma de dos cifras
Para aprender a sumar

Escenario: Sumar dos números positivos
Dado que estoy en la aplicación
Cuando pongo los números 1 y 3
Y solicito el resultado del cálculo
Entonces el resultado debe ser 4

Escenario: Sumar dos números negativos
Dado que estoy en la aplicación
Cuando pongo los números -1 y -3
Y solicito el resultado del cálculo
Entonces el resultado debe ser -4

Escenario: Sumar un número positivo y uno negativo
Dado que estoy en la aplicación
Cuando pongo los números -2 y 3
Y solicito el resultado del cálculo
Entonces el resultado debe ser 1
```

© JMA 2020. All rights reserved

Ejemplo

```
#language: es
Característica: Pagar un menú
Reglas:
- 1 punto por cada euro.
- 10 puntos equivalen a un descuento de 1 euros.
- El IVA es del 10%

Antecedentes:
  Dados los siguientes menús:
  | número | precio |
  | 1      | 10     |
  | 2      | 12     |
  | 3      | 8       |

Escenario: Ganar puntos al pagar en efectivo
Dado que he comprado 5 menús del número 1
Cuando pido la cuenta recibo una factura de 55 euros
Y pago en efectivo con 55 euros
Entonces la factura está pagada
Y he obtenido 50 puntos

Escenario: Pagar con dinero y puntos
Dado que he comprado 5 menús del número 1
Cuando pido la cuenta recibo una factura de 55 euros
Y pago con 10 puntos y 54 euros
Entonces la factura está pagada
Y he obtenido 0 puntos

Escenario: Pagar con puntos
Dado que he comprado 5 menús del número 1
Cuando pido la cuenta recibo una factura de 55 euros
Y pago con 500 puntos y 5 euros
Entonces la factura está pagada
Y he obtenido 0 puntos

Escenario: Intentar pagar el IVA con puntos
Dado que he comprado 5 menús del número 1
Cuando pido la cuenta recibo una factura de 55 euros
Y pago con 550 puntos y 0 euros
Entonces quedan 5 euros por pagar

Escenario: Comprar menús de varios tipos
Dado que he comprado 1 menú del número 1
Y que he comprado 2 menús del número 2
Y que he comprado 2 menús del número 3
Cuando pido la cuenta recibo una factura de 55 euros
Y pago en efectivo con 55 euros
Entonces la factura está pagada
Y he obtenido 50 puntos
```

© JMA 2020. All rights reserved

Java

CUCUMBER

© JMA 2020. All rights reserved

Instalación

- Cucumber-JVM se publica en el repositorio central de Maven. Puede instalarlo agregando dependencias a su proyecto.

```
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-java8</artifactId>
  <version>7.0.0</version>
  <scope>test</scope>
</dependency>
```

- Integración JUnit 5

```
<dependency>
  <groupId>io.cucumber</groupId>
  <artifactId>cucumber-junit-platform-engine</artifactId>
  <version>${cucumber.version}</version>
  <scope>test</scope>
</dependency>
```

© JMA 2020. All rights reserved

Creación del proyecto con el arquetipo

```
mvn archetype:generate \
  "-DarchetypeGroupId=io.cucumber" \
  "-DarchetypeArtifactId=cucumber-archetype" \
  "-DarchetypeVersion=7.0.0" \
  "-DgroupId=curso-cucumber" \
  "-DartifactId=curso-cucumber" \
  "-Dpackage=com.example" \
  "-Dversion=1.0.0-SNAPSHOT" \
  "-DinteractiveMode=false"
```

```
mvn archetype:generate -DarchetypeGroupId=io.cucumber -DarchetypeArtifactId=cucumber-archetype -DarchetypeVersion=7.0.0 -DgroupId=curso-cucumber -DartifactId=curso-cucumber -Dpackage=com.example -Dversion=1.0.0-SNAPSHOT -DinteractiveMode=false
```

© JMA 2020. All rights reserved

Definiciones de pasos

- Una definición de paso es un método Java con una expresión que lo vincula a uno o más pasos de Gherkin. Cuando Cucumber ejecuta un paso Gherkin en un escenario, buscará una definición de paso coincidente para ejecutar.

```
import io.cucumber.java.en.Given;
public class StepDefinitions {
    @Given("I have {int} cukes in my belly")
    public void i_have_n_cukes_in_my_belly(int cukes) {
        System.out.format("Cukes: %n\n", cukes);
    }
}

import io.cucumber.java8.En;
public class StepDefinitions implements En {
    public StepDefinitions() {
        Given("I have {int} cukes in my belly", (Integer cukes) -> {
            System.out.format("Cukes: %n\n", cukes);
        });
    }
}
```

© JMA 2020. All rights reserved

Definiciones de pasos

- La expresión de una definición de paso puede ser una expresión regular o una expresión Gherkin. Se pueden usar expresiones Cucumber o, si se prefiere usar expresiones regulares, cada grupo de captura de la coincidencia se pasará como argumento al método de definición de paso.

```
@Given("I have {int} cukes in my belly")  
public void i_have_n_cukes_in_my_belly(int cukes) {  
}
```
- Si la expresión del grupo de captura es idéntica a uno de los tipos de parámetros regexp registrados, la cadena capturada se transformará antes de pasar al método de definición de paso.
- Una definición de paso puede transferir el estado a una definición de paso posterior almacenando el estado en variables de instancia.
- Las definiciones de pasos no están vinculadas a un archivo de características o escenario en particular. El nombre de archivo, clase o paquete de una definición de paso no afecta a los pasos de Gherkin con los que coincidirá. Lo único que importa es la expresión de la definición del paso.
- Cuando Cucumber encuentra un paso Gherkin sin una definición de paso coincidente, imprimirá en consola un fragmento de definición de paso con una Expresión Cucumber coincidente. Se puede utilizar esto como punto de partida para las nuevas definiciones de pasos.

© JMA 2020. All rights reserved

Definiciones de pasos

- Un paso es análogo a una llamada de método o invocación de función.
- Para buscar pasos coincidentes:
 1. Cucumber compara un paso con la definición Regexp de un paso
 2. Cucumber reúne cualquier grupo de captura o variables
 3. Cucumber los pasa al método de definición de pasos y lo ejecuta
- Las definiciones de pasos comienzan con una preposición o un adverbio (Given, When, Then, And, But).
- Todas las definiciones de pasos se cargan (y definen) antes de que Cucumber comience a ejecutar el texto sin formato en el archivo de funciones.
- Una vez que comienza la ejecución, para cada paso, Cucumber buscará una definición de paso registrada con un archivo Regexp. Si encuentra uno, lo ejecutará, pasando todos los grupos de captura y variables de Regexp como argumentos al método o función.
- La preposición/adverbio específico utilizado no tiene importancia cuando Cucumber está registrando o buscando definiciones de pasos.

© JMA 2020. All rights reserved

Expresiones

- Cucumber admite tanto Cucumber Expressions como Regular Expressions para definir definiciones de pasos, pero no puede mezclar la sintaxis de Cucumber Expression con la sintaxis de Expresiones regulares en la misma expresión.
- Para usar expresiones regulares, se agregan anclas (que comiencen ^ y terminen con \$) o barras inclinadas (/).
- Cucumber Expressions es una alternativa con una sintaxis más intuitiva.
- La expresión Cucumber evaluará el texto literal tal cual y puede contar con parámetros de salida delimitados por {tipo}:
@Then("el resultado es {int}")
public void el_resultado_es(int resultado) {
- El texto entre llaves hace referencia a un tipo de parámetro . Cucumber viene con los siguientes tipos de parámetros integrados: {byte}, {short}, {int}, {long}, {biginteger}, {float}, {double}, {bigdecimal}, {word} (coincide con palabras), {string} (cadenas entre comillas simples o dobles, las comillas se descartan, admite cadenas vacías), {} (anónimo: coincide con cualquier cosa /.*/).

<https://cucumber.github.io/cucumber-expressions>

© JMA 2020. All rights reserved

Expresiones

- Para que las construcciones sean gramaticalmente correctas, las expresiones aceptan texto opcional y texto alternativo.
- El texto opcional se encierra entre paréntesis. La definición:
Then("devuelve {int} elemento(s)")
- coincide con los pasos:
Entonces devuelve 1 elemento
Entonces devuelve 3 elementos
- El texto alternativo se separa con / sin espacios en blanco entre las partes alternativas.
When("el/los elemento(s) coincide(n)")
- Si alguna vez se necesita hacer coincidir () o {} literalmente, puede escapar con una barra invertida en la apertura \ (o \{ dado que el cierre está implícito, \\ para escapar la barra invertida, actualmente / no se puede escapar.

© JMA 2020. All rights reserved

Definiciones de pasos

- Los pasos se definen en una o varias clases, para facilitar su estructuración, pero que no afecta a los pasos de Gherkin con los que coincidirán.

```
public class StepDefinitions {  
}
```
- Los pasos están decorados con las correspondientes anotaciones:

```
import io.cucumber.java.en.Given;  
import io.cucumber.java.en.Then;  
import io.cucumber.java.en.When;
```
- O sus correspondencias idiomáticas:

```
import io.cucumber.java.es.Cuando;  
import io.cucumber.java.es.Dado;  
import io.cucumber.java.es.Entonces;
```
- La anotación utilizada aporta semántica pero no afecta a la localización del paso, solo influye expresión regular o expresión Gherkin asociada, siendo equivalente:

```
@Dado("que estoy en la aplicación")  
@Cuando("que estoy en la aplicación")
```

© JMA 2020. All rights reserved

Resultados de los pasos

- Los pasos Then deben hacer afirmaciones que comparen los resultados esperados con los resultados reales de su aplicación. Cucumber no viene con una biblioteca de aserciones, en su lugar, utilice los métodos de aserción de la herramienta de prueba unitaria.
- Los resultados de los pasos son:
 - Éxito: Cuando Cucumber encuentra una definición de paso coincidente, la ejecutará. Si el bloque en la definición del paso no genera un error, el paso se marca como exitoso (verde). Se ignoran los valores de retorno.
 - Pasos fallidos (failed): Cuando el método o la función de una definición de paso se ejecuta y genera un error, el paso se marca como fallido (rojo).
 - Indefinido (undefined): Cuando Cucumber no puede encontrar una definición de paso coincidente, el paso se marca como indefinido (amarillo) y se omiten todos los pasos posteriores del escenario.
 - Pendiente (pending): Cuando el método o la función de una definición de paso invoca el método pending, el paso se marca como pendiente (amarillo, como con los indefinidos), lo que indica que hay trabajo por hacer.
 - Omitido (skipped): Los pasos que siguen a undefined, pending o failed pasos nunca se ejecutan, incluso si hay una definición de paso coincidente. Estos pasos están marcados como omitidos (cian).
 - Ambiguo: Las definiciones de pasos deben ser únicas para que Cucumber sepa qué ejecutar. Si usa definiciones de pasos ambiguas, Cucumber generará un `AmbiguousStepDefinitionsException`.

© JMA 2020. All rights reserved

Hooks

- Los hooks son bloques de código que pueden ejecutarse en varios puntos del ciclo de ejecución de Cucumber. Por lo general, se usan para configurar y dismantelar el entorno antes y después de cada escenario.
- El lugar donde se define un hook no tiene impacto en los escenarios o pasos para los que se ejecuta. Si se desea un control más detallado, puede usar hooks condicionales. Se puede declarar hooks en cualquier clase.
 - `@BeforeAll`: se ejecuta antes de que se ejecute cualquier escenario.
 - `@Before`: se ejecutan antes del primer paso de cada escenario.
 - `@BeforeStep`: se ejecutan antes de cada paso.
 - `@AfterStep`: se ejecutan después de cada paso, independientemente de su resultado.
 - `@After`: se ejecutan después del último paso de cada escenario, incluso cuando el resultado del paso es failed, undefined, pending o skipped.
 - `@AfterAll`: se ejecuta después de que se hayan ejecutado todos los escenarios.
- Los hooks se pueden seleccionar condicionalmente para su ejecución en función de las etiquetas del escenario.
 - `@After("@browser and not @headless")`

© JMA 2020. All rights reserved

Step Definitions

```
public class CalculadoraStepDefinitions {
    Calculadora calc;
    int op1, op2, rslt;

    @Dado("que estoy en la aplicación")
    public void que_estoy_en_la_aplicación() {
        calc = new Calculadora();
    }

    @Cuando("pongo los números {int} y {int}")
    public void pongo_los_números_y(Integer a, Integer b) {
        op1 = a;
        op2 = b;
    }

    @Cuando("solicito el resultado del cálculo")
    public void solicito_el_resultado_del_cálculo() {
        rslt = calc.suma(op1, op2);
    }

    @Entonces("el resultado debe ser {int}")
    public void el_resultado_debe_ser(Integer r) {
        assertEquals(r, rslt);
    }
}
```

© JMA 2020. All rights reserved