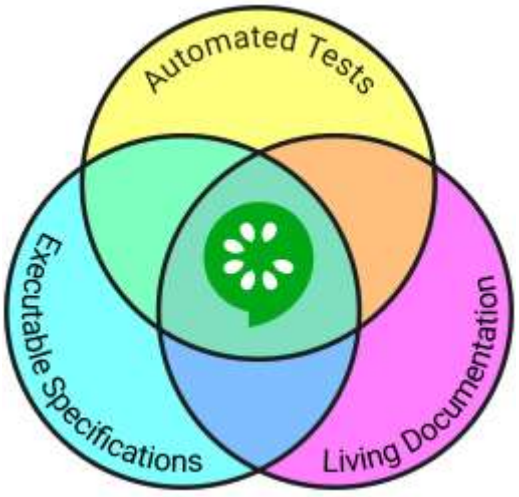
Buenas prácticas de Cucumber

BDD es una estrategia de desarrollo, pero más allá que no sigas esta práctica, nos resulta beneficioso utilizar Cucumber (o alguna herramienta similar) ya que "**te obliga" a documentar tus pruebas automatizadas antes de implementarlas**. Se hace fundamental que estos tests los hagamos legibles y claros para un usuario que a priori no conozca el comportamiento de la funcionalidad que se describe, y más mantenibles para reducir los costos de hacer modificaciones en los pasos de los tests. Al final de esta guía dejamos algunos artículos adicionales sobre Cucumber, Gherkin y BDD.
Esta imagen refleja la idea de combinar pruebas automatizadas, tener una documentación viva, y a su vez, contar con especificaciones que son ejecutables. Todo esto gracias al enfoque de utilizar alguna herramienta del estilo de Cucumber.



Para trabajar con Cucumber básicamente se van a necesitar estos archivos:

- **Feature file**, archivo de texto donde se escriben los criterios de aceptación en formato Gherkin. Estos criterios de aceptación se podrían ver como las pruebas que vamos a preparar.
- **Step Definition**, que son archivos en el lenguaje de programación usado, donde Cucumber va a poder asociar qué acciones ejecutar asociadas a cada paso de cada criterio de aceptación definido en las distintas features.
- Luego, dependiendo de a qué nivel hacemos los tests, se podrán necesitar otros archivos. Por ejemplo, si esto lo estamos haciendo a nivel de capa de presentación, de interfaz gráfica Web, entonces vamos a usar algo como Selenium, para lo cual sería bueno seguir algún patrón de diseño cómo Page Object. Pero esto ya es algo más específico y dependiente de qué es lo que estamos probando, y en este artículo queremos poner foco en la parte de Cucumber en sí.

Feature files (escenarios en Gherkin)

Creación de una feature

Para empezar creamos una carpeta en el proyecto donde guardaremos las features que vamos a escribir en Gherkin. En este ejemplo nos vamos a basar en un ejercicio de BDD (que solemos hacer en el curso de Testing en Contextos Ágiles, de Abstracta y Peregrinus) donde se quiere modelar el comportamiento de un cajero mediante funcionalidades en Gherkin y lo vamos a hacer siguiendo estas prácticas.
Supongamos que nos interesa modelar el comportamiento de un cajero cuando queremos extraer dinero:

- Creamos dentro de la carpeta un archivo con extensión .feature (por ejemplo “extraer-dinero.feature”).
- Definimos un título que diga qué es la funcionalidad, algo como: “Feature: Extracción de dinero”.
- Comenzamos a escribir escenarios para nuestra funcionalidad.

La descripción de un escenario se suele escribir de la siguiente forma:

```
Scenario: Como [usuario concreto]
quiero [realizar acción concreta]
para [resultado o beneficio]
```

Lo importante es que explique brevemente qué se quiere lograr con el escenario.

Una forma de empezar a escribir nuestra Feature puede ser ésta:

```
Feature: Extracción de dinero
  Scenario: Como usuario existente y habilitado del cajero,
quiero realizar una extracción para obtener dinero.
```

Algunos puntos importantes sobre Feature files:

- Lo más recomendable es utilizar una Feature por funcionalidad del sistema, y que lo especificado en la Feature sea específico de una sola funcionalidad en concreto y lo más independiente posible de otras funcionalidades.

- La mejor forma de hacer nuestras Feature files entendibles para un cliente es utilizando el mismo lenguaje que éste utiliza para describir la funcionalidad, por lo tanto siempre es mejor describir las acciones como lo haría el cliente.

Features y escenarios

En Gherkin los escenarios son ejemplos de un comportamiento individual para establecer criterios de aceptación, por lo que nos puede interesar escribir varios por funcionalidad para observar distintos resultados y hacer más completo nuestro test (se recomienda escribir los escenarios positivos primero).

Para describir los escenarios se utilizan sentencias de Gherkin: Given, When, Then, But y And.

Lo más importante es que los pasos describan brevemente qué se quiere hacer en la funcionalidad y no cómo se quiere hacer (esto es responsabilidad de los step definitions, explicados más abajo). Un ejemplo de escenario mal escrito puede ser éste:

```
Scenario: Como usuario existente y habilitado del cajero,
quiero realizar una extracción para obtener dinero.
Given Me autentiqué con una tarjeta habilitada
And El saldo disponible en mi cuenta es positivo
And El cajero tiene suficiente dinero
And El cajero tiene papel suficiente para imprimir recibos
When Introduzco la tarjeta en el cajero
And Escribo en el teclado el pin de la tarjeta
And Presiono el botón de confirmar pin
And Presiono el botón próximo a la opción extraer dinero
And Ingreso una cantidad menor o igual a mi saldo disponible
And Presiono el botón de confirmar extracción
And Presiono el botón de confirmar imprimir recibo
.....
```

Es mejor evitar escribir los escenarios de esta manera pues los hace muy largos, con muchos detalles innecesarios y difícil de leer y entender. Otra desventaja de escribirlos así es que los hace difíciles de mantener, ya que al mínimo cambio que tengamos que hacer al comportamiento es muy probable que tengamos que modificar muchos pasos en el escenario. Una forma más resumida, descriptiva y clara de escribir el escenario puede ser ésta:

```
Scenario: Como usuario existente y habilitado del cajero,
quiero realizar una extracción para obtener dinero.
Given Me autentiqué con una tarjeta habilitada
And El saldo disponible en mi cuenta es positivo
When Selecciono la opción de extraer dinero
And Ingreso la cantidad de dinero menor al saldo disponible y disponible
del cajero
Then Obtengo dinero
And El dinero que obtuve se resta del saldo disponible de mi cuenta
And El sistema devuelve la tarjeta automáticamente
And El sistema muestra el mensaje de transacción  finalizada
```

Aquí hay algunos puntos importantes sobre escenarios y pasos en Gherkin:

- Las sentencias se deben escribir en orden Given-When-Then. Esto es porque Given representa una precondición, When una acción y Then un resultado o consecuencia de la acción (criterio de aceptación del usuario). Por lo que escribir un When después del Then, por ejemplo, no estaría bien conceptualmente y no sería muy claro.
- Tampoco deberían repetirse los Given-When-Then por escenario, para extender cualquiera de las sentencias se utiliza And. El motivo de ésto es que un escenario representa un comportamiento individual, y si definimos algo del estilo: Given-When-Then-When…, seguramente podemos dividirlo en más de un escenario.
- Las sentencias tienen que ser consistentes entre sí y con la descripción del escenario, o sea, si la descripción del escenario está escrita en primera persona las sentencias también deberían escribirse en primera persona (desarrollaremos la cuestión de las personas gramaticales que se utilizan en la redacción de los escenarios en la siguiente sección del artículo).

- En la medida de lo posible no usar muchos pasos para un solo escenario, la idea es que un usuario que no conozca la funcionalidad sea capaz de entenderla leyendo el escenario. *Mientras menos tenga que leer para entenderlo, mejor*.
- Para esto también se recomienda que las sentencias sean explicativas y breves.
- *Deberíamos escribir los escenarios como nos gustaría que nos lo presentaran.*
- La sentencia But funciona igual que Then, pero se utiliza cuando queremos verificar que no se observa un resultado concreto, por ejemplo:

```
Given Cumplo una precondición
When Ejecuto una acción
Then Observo este resultado
But No debería poder observar este otro resultado
```

- Es muy importante que los escenarios sean lo más independientes entre sí mientras sea posible, es decir: no puede pasar que se acoplen escenarios. Por ejemplo, no es conveniente que si en un escenario insertamos registros en una base de datos, el resultado de escenarios siguientes dependan de la existencia de esos registros. Tener escenarios acoplados puede generar errores, por ejemplo, si tenemos que ejecutarlos en paralelo, o si uno falla.

¿Cómo separar los archivos?

Al separar los features la cantidad de archivos puede llegar a ser enorme, entonces hay que pensar cómo hacer la división de features en diferentes archivos.

Una opción muy usada es tener un archivo con las features que agrupen todo lo relacionado con un aspecto de la aplicación, e incluso organizar en directorios.

En un caso concreto, para un sistema de espectáculos, podríamso tener esto:

- En el primer nivel podríamos tener una carpeta “espectáculos”.
- Dentro, distintos features de crear, editar, eliminar y todo lo que tenga que ver con ellos.

Así se organiza mejor y es más fácil ubicar cada test y dónde encontrar cada cosa.

Escenarios con datos concretos

BDD es en cierta forma similar a SBT (Sample based testing), en cuanto a que se busca reducir ambigüedades al mostrar ejemplos. Considerando esto, tal vez el ejemplo anterior sería mejor si lo bajamos a datos concretos, como en el siguiente caso:

```
Scenario: Como usuario existente y habilitado del cajero,
quiero realizar una extracción para obtener dinero.
Given Me autentiqué con una tarjeta habilitada
And El saldo disponible en mi cuenta es de $10.000
And El cajero cuenta con $100.000 en efectivo
When Selecciono la opción de extraer dinero
And Indico que quiero extraer $1.000
Then Obtengo $1.000 en dos billetes de $500
And El saldo de mi cuenta pasa a ser $9.000
And El cajero se queda con $99.000 en efectivo
And El sistema devuelve la tarjeta automáticamente
And El sistema muestra el mensaje de transacción finalizada
```

¿En primera persona o en tercera persona?

Una duda muy común que surge a la hora de redactar un escenario (y sus respectivos pasos) es el punto de vista que debe utilizarse. La pregunta habitual es: **¿debo escribir los escenarios en primera o tercera persona?** La cuestión es más compleja de lo que parece.
Los ejemplos utilizados en la documentación oficial de Cucumber utilizan ambos puntos de vista, así que no es una referencia exacta para resolver el problema.

Analicemos los argumentos a favor de cada una de las posturas.

Uso de la primera persona

Dan North (considerado como el creador de BDD), según encontramos en una referencia en Stack Overflow, recomienda el uso de la primera persona, y de hecho es la que utiliza para escribir sus escenarios en su artículo Introducing BDD.
El uso de la primera persona permite escribir el escenario siendo coherentes con su descripción, que, como se mencionó anteriormente, suele seguir la forma "*Como [usuario concreto] quiero [realizar acción concreta] para [resultado o beneficio]*".
Dado que el rol o usuario concreto para el cual se construye el escenario está especificado en la descripción, y la idea es ponerse en los pies de ese usuario, el uso de la primera persona puede resultar una forma coherente de redacción.

Uso de la tercera persona

Los defensores de esta postura argumentan que el uso de la primera persona hace que el lector del escenario pierda la referencia sobre el rol o el usuario del que se está hablando. Si escribo en un step "Elimino un artículo del sistema", ¿quién es el que elimina? ¿Un administrador, un usuario en particular? ¿Un conjunto de roles?

De alguna forma, el uso de la tercera persona disminuye el riesgo o la dificultad de que el lector haga presunciones erróneas sobre quién es el o los stakeholders involucrados.

También se sostiene que el uso de la tercera persona, especialmente en inglés, presenta la información de una forma más formal y objetiva.

Conclusión

No hay una regla general sobre el punto de vista a utilizar para escribir los escenarios. Lo importante en este punto, como ya se mencionó, mantener la consistencia entre la descripción del escenario y sus pasos (no alternar personas gramaticales), respetar los criterios utilizados en el caso de que estemos agregando escenarios a un proyecto ya existente y favorecer la claridad de lo que se escribe.

Otras palabaras clave para describir escenarios

Background

Si en todos los escenarios de una misma feature se cumplen algunas precondiciones, es mucho más práctico usar un **Background** que escribir lo mismo varias veces. Esto sirve como una serie de pasos que se van a ejecutar antes de todos los escenarios de la feature. Veamos un ejemplo:

```
Feature: Extracción de dinero

Background:

Given La tarjeta de crédito está habilitada

And El saldo disponible en mi cuenta es positivo

And El cajero tiene suficiente dinero

Scenario: ...
```

Se recomienda que los Backgrounds sean lo más cortos posibles en cuanto a cantidad de pasos, ya que si se hacen muy largos puede ser difícil entender los escenarios que siguen al Background. Siempre es mejor tener escenarios lo más autocontenidos posible, y en caso de tener un Background que sea lo más breve posible.

Scenario Outline

Scenario Outline son un tipo de escenario donde se especifican datos de entrada. Son muy prácticos ya que gracias a esto no es necesario escribir un escenario por dato de entrada, por ejemplo:

```
Scenario outline: Extraer dinero con distintas claves de tarjeta.

Given La tarjeta de crédito está habilitada

And El saldo disponible en mi cuenta es positivo

And El cajero tiene suficiente dinero

When Introduzco la tarjeta en el cajero

And Ingreso el <pin> de la tarjeta

...
```

Examples:

```
| pin |
| 1234 |
| 9876 |
```

Doc Strings

Los **Docs Strings** son útiles para agregar cadenas de caracteres largas a un step de una forma más prolija.
Para utilizarlos, se debe agregar el texto deseado en el step entre tres comillas dobles (“”“). Por ejemplo:

```
Scenario outline: ...
Given ...
When ...
Then Obtengo dinero
And Se muestra mensaje de confirmación con el texto:
  """
  Estimado cliente:
  Se ha retirado de su cuenta n° <cuenta> el siguiente monto: <monto>.
  Gracias por utilizar nuestros servicios.
  """
```

Examples:

```
  | pin  | cuenta    | monto |
  | 1234 | 112235489 | 5000  |
  | 9876 | 668972214 | 6000  |
```

Como se puede ver en el ejemplo anterior, una Doc String (que es en sí mismo un dato de entrada) puede ser utilizada en combinación con otros datos de entrada para mostrar datos propios del escenario que se está ejecutando.

Data Tables

Las Data Tables, en su estructura y utilidad, son muy similares a Scenario Outlines. Sus dos diferencias principales son:

- Las Data Tables son definidas a nivel de un step y no a nivel del escenario, por lo cual sirven para pasar datos de entrada a un solo paso dentro del escenario.
- No es necesario definirles un cabezal, aunque es útil y recomendable para poder referenciar los datos con mayor facilidad.

Ejemplo de su uso:

```
Scenario: Extraer dinero con distintas claves de tarjeta.

Given La tarjeta de crédito está habilitada

And El saldo disponible en mi cuenta es positivo

And El cajero tiene suficiente dinero

When Introduzco la tarjeta en el cajero

And Ingreso los siguientes <pin> y obtengo el resultado <resultado> :

    | pin  | resultado |
    | 1234 | Error     |
    | 9876 | OK        |
```

En el ejemplo anterior, agregamos una segunda columna “resultado”, para indicar el resultado esperado de acuerdo al PIN ingresado (“1234” es incorrecto y “9876” es correcto). No es necesario utilizar las Data Table de esa forma, pero se incluye como un ejemplo de cómo pueden ser utilizados los datos de entrada en un escenario.

Languages

Cucumber brinda la posibilidad de escribir los escenarios en distintos idiomas, siguiendo las mismas convenciones que normalmente utilizamos en inglés.

Para hacer uso de esta característica, se debe encabezar la funcionalidad con “# language:”, seguido del código del dialecto a utilizar (por ejemplo, “# language: es” para español).

En la documentación oficial de Cucumber se puede encontrar toda la información necesaria para utilizar esta característica, incluyendo el código de cada dialecto y las palabras que deberían utilizarse para cada idioma para sustituir las habituales (por ejemplo y para idioma español, se utiliza “Característica” en lugar de “feature”, “Escenario” en lugar de “scenario”, etc.).

Tags

En ciertas ocasiones puede ocurrir que queramos no ejecutar todos los escenarios de nuestro test y en lugar de eso queremos agrupar ciertos escenarios y ejecutarlos por separado. Cucumber proporciona una forma de configurar esto mediante etiquetas (tags). Los tags son anotaciones que sirven para agrupar y organizar escenarios e incluso features, estos se escriben con el símbolo @ seguido de un texto significativo, ejemplos:

```
@gui
Feature: ...
@SmokeTest @wip
Scenario: ...
@RegressionTest
Scenario: . . .
```

Es importante notar que los tags que especifiquemos a los títulos de las Feature files serán heredados por los escenarios de la misma, incluidos Scenario Outlines.

Si tenemos un Scenario outline bajo un tag, todos los ejemplos de datos que tenga el escenario van a ejecutarse bajo ese tag.

Habiendo asignado nuestros tags, existen muchas formas de configurarlos en la ejecución en el apartado de tags de @CucumberOptions. Algunos ejemplos:

tags = {“@SmokeTest”} Ejecuta todos los escenarios bajo el tag @SmokeTest

tags = {“@gui”} Ejecuta todos los escenarios bajo el tag @gui, como en el ejemplo tenemos una feature bajo este tag, se ejecutarán todos los escenarios de esa feature.

tags = {“@SmokeTest, @wip”} Ejecuta todos los escenarios que estén bajo el tag @SmokeTest o bajo el tag @wip (condición OR).

tags = {“@SmokeTest”, “@RegressionTest”} Ejecuta todos los escenarios que estén bajo los tags @SmokeTest y @RegressionTest (condición AND).

tags = {“~@SmokeTest”} ignora todos los escenarios bajo el tag @SmokeTest

tags = {“@RegressionTest, ~@SmokeTest”} ejecuta todos los escenarios bajo el tag @RegressionTest, pero ignora todos los escenarios bajo el tag @SmokeTest

tags = {“@gui“, “~@SmokeTest”, “~@RegressionTest”} ignora todos los escenarios bajo el tag @SmokeTest y @RegressionTest pero ejecuta todos los que estén bajo el tag “@gui”, si seguimos el ejemplo es como ejecutar todos los escenarios de la feature que no estén bajo ningún otro tag.

En definitiva, los tags no sólo son útiles para organizar y agrupar nuestros escenarios/features (lo cual aporta mucho a la claridad del test), sino que también nos permiten ejecutarlos selectivamente, como por ejemplo, ejecutar más frecuentemente los escenarios más rápidos.

Step Definitions (implementación de los pasos)

Lo que realizamos anteriormente fue la especificación de los pasos de nuestros escenarios, describimos qué procesos va a seguir nuestro test, pero no definimos cómo queremos que se haga. Esta responsabilidad pasa a ser de la implementación de las sentencias Gherkin que escribimos en los escenarios (step definitions).

Las step definitions sirven a Cucumber como traducción de los pasos que escribimos en acciones a ejecutar para interactuar con el sistema.

Si bien los ejemplos que se darán a continuación para la implementación de los pasos están desarrollados en Java, hay que mencionar que Cucumber también puede utilizarse con JavaScript, Ruby, C++ y otros lenguajes.

Para comenzar a escribir step definitions, si estamos trabajando en un IDE con dependencias de Gherkin y Cucumber ya instaladas, éste nos sugerirá implementarlas (van a aparecer subrayadas), y nos va a permitir crear un archivo .java o elegir uno donde ya tengamos pasos implementados.

Escogiendo cualquiera de estas dos opciones se nos generará un método en la clase, por ejemplo si decidimos crear un step definition para el paso:

```
  Given La tarjeta de crédito está habilitada
```

se nos generará automáticamente un método con una anotación, donde el texto del encabezado va a coincidir con la descripción del paso:

```
@Given("^La tarjeta de crédito está habilitada$")
    public void verificarTarjetaHabilitada() throws Throwable {
       // Write code here that turns the phrase above into concrete
actions
        throw new PendingException();
}
```

En el caso de que el paso incluya datos de entrada definidos a través de Scenario Outline o Data Tables, esos datos son incluidos en la anotación como expresiones regulares, y en el método como parámetros recibidos:

```
@When("^Ingreso el \"([0-9]+)\" de la tarjeta $")
    public void ingresarPIN(int pin) throws Throwable {
      // Write code here that turns the phrase above into concrete
actions
      throw new PendingException();
}
```

Automáticamente cuando hacemos esto, el paso en la feature (la sentencia en Gherkin) ya reconoce dónde está la implementación. Aquí hay algunos puntos importantes a la hora de implementar step definitions:

- Lo más recomendable es crear step definitions que sólo se tengan que implementar una vez y reusar en muchos escenarios (incluso de distintas features).
- Esto es práctico no sólo para ahorrar la cantidad de código que hay que escribir, también aporta mucho a la mantenibilidad de los tests ya que eventualmente va a

ser menor la cantidad de step definitions que vamos a tener que modificar en cualquier caso.

- Una manera de reutilizar step definitions es definirlos en Scenario outlines y parametrizarlos.