

SERENITY BDD con CUCUMBER

2

Creando mi primer Historia de Usuario

(Tiempo ejecución 4 horas)



Introducción a BDD, Gherkin y Cucumber

BDD (Behavior Driven Development): Desarrollo guiado por el comportamiento, consiste básicamente en una estrategia de desarrollo y lo que plantea es la definición de los requisitos desde el punto de vista del comportamiento de la aplicación, desde el negocio, creado en un lenguaje común para el negocio y para los técnicos.

Gherkin, es un lenguaje común, que lo puede escribir alguien sin conocimientos en programación, pero que lo puede comprender un programa, de forma tal que se pueda utilizar como especificación de pruebas.

Estas pruebas se almacenan en archivos “.feature” los cuales deberían ser versionados junto al código fuente que se está probando.

Gherkin es considerado un lenguaje Business Readable DSL (Lenguaje específico de dominio legible por el negocio)

Para empezar a hacer BDD, sólo se necesita conocer 5 palabras con lo que vamos a describir las funcionalidades:

Feature: Nombre de la funcionalidad que vamos a probar.

Scenarió: habrá uno por cada prueba que quiera especificar para esta funcionalidad

Given: precondiciones

When: acciones que se van a ejecutar

Then: Se especifica el resultado esperado, las verificaciones a realizar.

Una feature puede contener varios escenarios de prueba.

Ejemplo:

Feature : Transferencias entre cuentas de depósito.

Transferencias desde cuentas de ahorro o corriente

A cuentas de depósito de ahorro o corriente propias

A cuentas de depósito de ahorro o corriente de terceros

Scenarió: Transferir de una cuenta de Ahorro propia a una cuenta de Ahorro propia

Given: Autenticación exitosa en la aplicación

And: Consulta el saldo de la cuenta a debitar

And: Ingresar a la funcionalidad transferencias

When: Prepara la transacción transferencias

And: Verifica información ingresada

And: Verifica confirmación de la transacción

And: Consulta el saldo de la cuenta a debitar

Then: Verifica débito del monto transferido

Cucumber, es una de las herramientas framework que podemos utilizar para automatizar nuestras pruebas BDD, permite ejecutar descripciones funcionales en texto plano como pruebas de software automatizadas.



¿Qué es un archivo POM?

pom, responde a las siglas de Project Object Model, es un fichero **XML**, que es la “unidad” principal de un proyecto Maven. Contiene información a cerca del proyecto, fuentes, test, dependencias, plugins, version...

¿Qué es y para qué sirve Maven?

Maven es una herramienta de software para la gestión y construcción de proyectos Java.

¿Qué es y para qué sirve Junit?

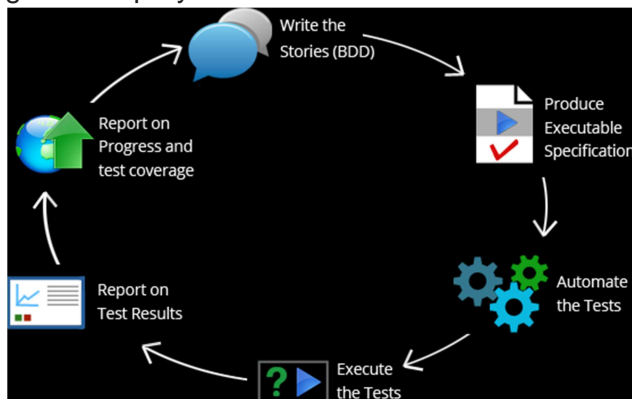
JUnit es un conjunto de clases (framework) que permite realizar la ejecución de clases **Java** de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera.

¿Qué es Serenity BDD?

Serenity es una biblioteca de código abierto que le ayuda a redactar pruebas de aceptación automatizadas de mayor calidad de forma más rápida.

Serenity te ayuda a:

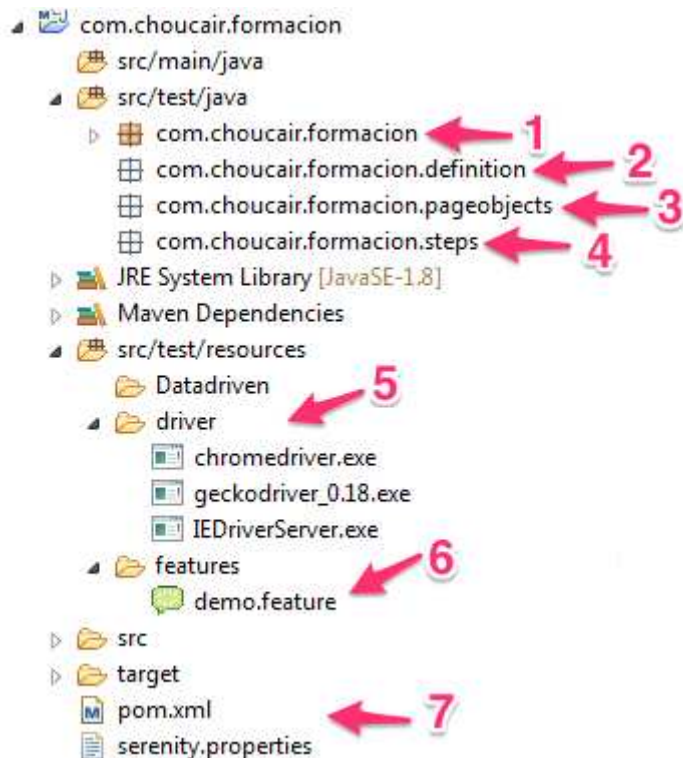
- ✓ Escribir pruebas que sean más flexibles y fáciles de mantener
- ✓ Producir informes ilustrados y narrativos sobre sus pruebas
- ✓ Asigne sus pruebas automatizadas a sus requisitos
- ✓ Vea cuánto de su aplicación se está probando realmente
- ✓ Y controle el progreso del proyecto



Proyecto Base

Conozcamos la estructura

El proyecto compartido en la guía anterior y que ha sido importado en el IDE Eclipse, contiene una estructura base que servirá de modelo para la construcción de nuestras pruebas automatizadas BDD.



1. Scenario Runner: En este paquete encontrará las clases "RunnerFeatures" y "RunnerTags" desde donde se lanzan las ejecuciones de los features.
2. Definitions, en cucumber cada línea de el scenario gherkin de la feature mapea a un método de una clase java existente en este paquete, con ayuda de las anotaciones @Given, @When and @Then.
3. Pageobjects, en este paquete se tendrán las clases en donde se definirán los mapeos de los objetos requeridos para la prueba.
4. Steps, es el repositorio de las clases con los pasos o acciones a realizar para una "definition" respectiva.
5. En el folder src/test/resources, se organizan todos los recursos necesarios para la prueba, uno de ellos son los driver de los diferentes navegadores.
6. Src/test/resources/features, folder en el cual se organizan las diferentes **features** del proyecto.
7. Archivos Pom.xml y serenity.properties, que contienen información requerida para el proyecto de configuración.

Un modelo BDD está compuesto por la siguiente estructura de conexión

Feature: cada línea de un scenario mapea con un método en la clase .definition

Definition: cada método mapea con unos métodos en la clase .steps

Steps: para la construcción de los pasos es necesaria la definición de objetos en .pageobjects

Pageobjects: contiene las clases con la definición de los objetos



Mi primera prueba BDD

Historia de Usuario: Verificar el diligenciamiento de la pantalla “Popup Validation”.

Criterios de Aceptación:

- Verificar diligenciamiento exitoso.
- Verificar mensaje de validación para cada campo

url de prueba: <https://colorlib.com/polygon/metis/login.html>

Pasos para la ejecución de la prueba.

1. Autenticacion en colorlib

- a. Abrir navegador con la url de prueba
- b. Ingresar usuario demo
- c. Ingresar password demo
- d. Click en botón Sign in
- e. Verificar la Autenticación (label en home)

2. Ingresar a Funcionalidad Popup Validation

- a. Clic en Menu “Forms”
- b. Clic en submenú “Form Validation”
- c. Verificación : se presenta pantalla de la funcionalidad con título Popup Validation

3. Diligenciar Formulario Popup Validation

- a. Diligenciar todos los campos del formulario
- b. Clic en botón Validate

4. Verificar Respuesta Exitosa/Fallida

Análisis:

Existen diferentes formas de analizar el flujo de una transacción, para efectos de este taller vamos a agrupar los pasos en acciones, y cada una de estas acciones corresponderán a una línea **gherkin** en nuestra *feature*.

Como siempre en cualquier prueba es importante que determinemos la data requerida para la ejecución de la prueba.



Crear la feature.

Lo primero es crear la **feature** para la funcionalidad **Popup Validation**.

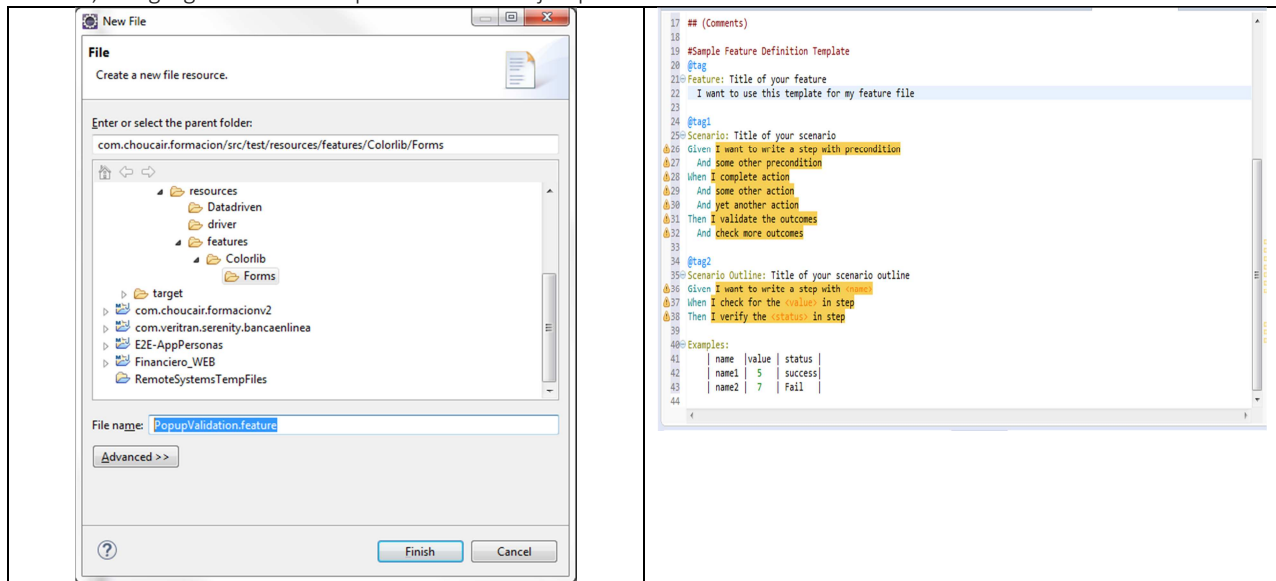
Paso 1, en la ruta src/test/resources/features, vamos a agregar un folder llamado "Colorlib"

Paso 2, al interior del folder "Colorlib", agreguemos un folder "Forms"

Paso 3, Crear un archivo tipo **feature**, dar clic derecho al folder ../Forms > New > File

Asignar nombre a la historia ejemplo: "PopupValidation.**feature**" (importante que finalice con .feature)

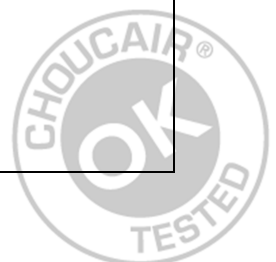
Paso 4, Se agregará un archivo plantilla con un ejemplo base de un archivo .feature



Nota:

- Las filas que inicien por #, con comentarios.
- Las filas que inicien por @ son etiquetas con las que se pueden organizar los escenarios de prueba, por ejemplo, por su propósito, de ésta forma al momento de ejecutar se puede hacer por grupo de tags.

@regresion	Feature		En esta estructura tenemos la posibilidad de ejecutar 1 escenario que pertenece a @RutaCritica
@RutaCritica	Scenari		O podríamos ejecutar los dos escenarios del @CasoAlterno.
	Given		Si quisiéramos ejecutar la totalidad de escenarios, enviaríamos la etiqueta @Regresion.
	When		
	Then		
@CasoAlterno	Scenari		
	Given		
	When		
	Then		
	Scenari		
	Given		
	When		
	Then		



Paso 5, modifiquemos la etiqueta de la **feature** por **@Regresion**.

Paso 6, documente el campo **feature**, con las características de la funcionalidad y lo que se espera de ella.

Paso 7, modifiquemos la etiqueta del primer escenario por **@CasoExitoso**.

Paso 8, documente el campo **Scenari**o, con la descripción del mismo.

Paso 9, crear los pasos usando la estructura **Given**, **When** y **Then**, si quiere extender alguno de ellos puede usar **"And"**.

Recuerde que para este ejemplo las acciones que resaltamos previamente harán referencia a una línea gherkin en la **feature**.

Nos quedaría lo siguiente:

```
15 #<> (placeholder)
16 #""
17 ## (Comments)
18
19 #Sample Feature Definition Template
20 @Regresion
21 Feature: Formulario Popup Validation
22     El usuario debe poder ingresar al formulario los datos requeridos.
23     Cada campo del formulario realiza validaciones de obligatoriedad,
24     longitud y formato, el sistema debe presentar las validaciones respectivas
25     para cada campo a través un globo informativo.
26
27 @CasoExitoso
28 Scenario: Diligenciamiento exitoso del formulario Popup Validation,
29     no se presenta ningún mensaje de validación.
30 Given Autentico en colorlib con usuario "demo" y clave "demo"
31 And Ingreso a la funcionalidad Forms Validation
32 When Diligencio Formulario Popup Validation
33 Then Verifico ingreso exitoso
34
35 @tag2
36 Scenario Outline: Title of your scenario outline
37 Given I want to write a step with <name>
38 When I check for the <value> in step
39 Then I verify the <status> in step
40
41 Examples:
42 | name | value | status |
43 | name1 | 5 | success |
44 | name2 | 7 | Fail |
45
```

Nota:

- En la línea **Given**, observemos una forma de enviar datos a una definición, en este caso los datos hacen parte de la línea gherkin **"demo"**, el sistema es capaz de identificar la recepción de dos parámetros con valores.
- Como se observa cada línea gherkin se encuentra resaltada, indicando que aún no tiene implementado un código en el paquete **".definition"**

Ya tenemos nuestro archivo de características listo, pero aún **cucumber** no sabe realmente qué parte del código se va a ejecutar, esto requiere la necesidad de un archivo de **definición** de paso intermedio. El archivo de definición de pasos almacena la asignación entre cada paso del escenario definido en el archivo de características con un código de función para ser ejecutado.



Paso 10, ahora es necesario crear los definitions, para esto la herramienta es capaz de proponer los métodos requeridos para el mapeo.

Para esto, es necesario editar la clase **RunnerTags.java** que se encuentra en “src/test/java/com.choucair.formacion”, esta clase permita ejecutar un tag de una feature en particular, para esto vamos a agregar la siguiente línea al runner o modificar una existente.

Quedaría así:

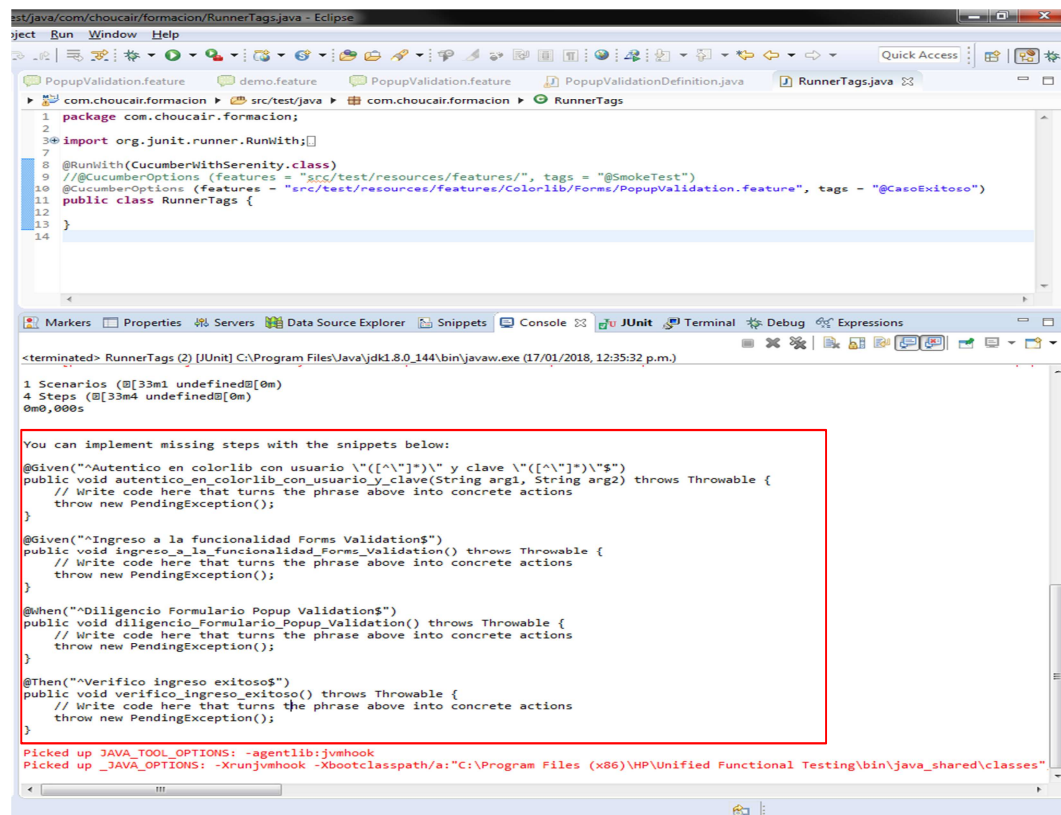
```
1 package com.choucair.formacion;
2
3 import org.junit.runner.RunWith;
4
5
6 @RunWith(CucumberWithSerenity.class)
7 // @CucumberOptions (features = "src/test/resources/features/", tags = "@SmokeTest")
8 @CucumberOptions (features = "src/test/resources/features/Colorlib/Forms/PopupValidation.feature", tags = "@CasoExitoso")
9 public class RunnerTags {
10
11 }
12
13 }
```

Donde **features**, es la ruta complete de la feature a ejecutar y **Tags**, es el nombre de la etiqueta respectiva.

Guardar cambios.

Paso 11, Ejecutar, clic derecho sobre el runner > Run As > JUnit Test

Paso 12, en la pestaña consola el sistema nos propone los métodos a implementar en la clase **.definitions**.



```
<terminated> RunnerTags (2) [JUnit] C:\Program Files\Java\jdk1.8.0_144\bin\javaw.exe (17/01/2018, 12:35:32 p.m.)
1 Scenarios ([33m1 undefined@0m)
4 Steps ([33m4 undefined@0m)
0m0,000s

You can implement missing steps with the snippets below:

@Given("^Autentico en colorlib con usuario \"([^\"]*)\" y clave \"([^\"]*)\"")
public void autentico_en_colorlib_con_usuario_y_clave(String arg1, String arg2) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Given("^Ingreso a la funcionalidad Forms Validation$")
public void ingreso_a_la_funcionalidad_Forms_Validation() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

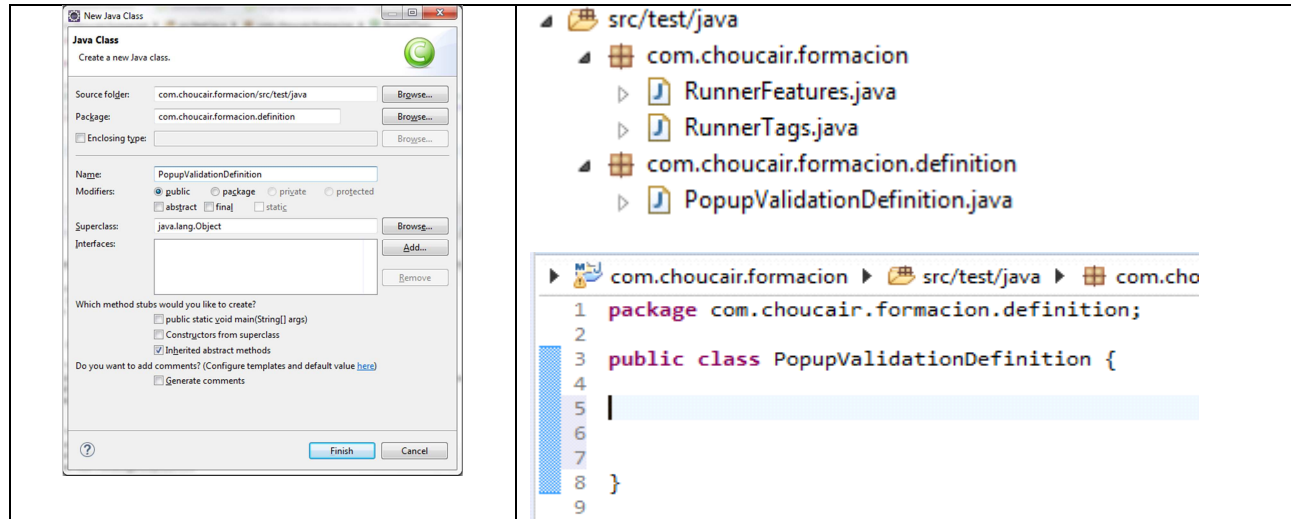
@When("^Diligencio Formulario Popup Validation$")
public void diligencio_Formulario_Popup_Validation() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Then("^Verifico ingreso exitoso$")
public void verifico_ingreso_exitoso() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

Picked up JAVA_TOOL_OPTIONS: -agentlib:jdwp=transport=dt_socket,address=127.0.0.1,server=y,suspend=n
Picked up _JAVA_OPTIONS: -Xrunjvmonhook -Xbootclasspath/a:"C:\Program Files (x86)\HP\Unified Functional Testing\bin\java_shared\classes"
```



Paso 13, crear una clase definition en el paquete **.definition**, por ejemplo con el nombre **"PopupValidationDefinition"**.



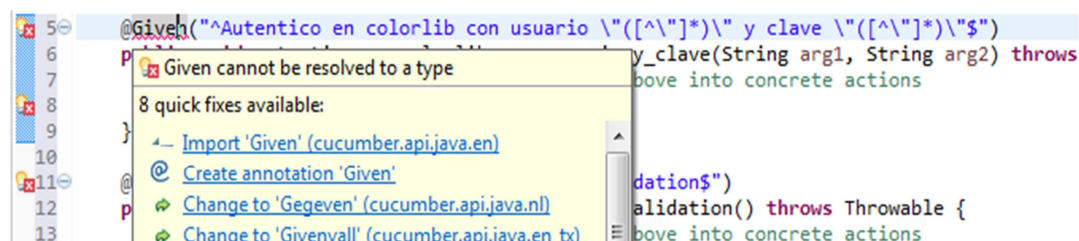
Paso 14, agregar los métodos propuestos a la clase de definiciones creada.

```

1 package com.choucair.formacion.definition;
2
3 public class PopupValidationDefinition {
4
5     @Given("^Autentico en colorlib con usuario \"([^\"]*)\" y clave \"([^\"]*)\"$")
6     public void autentico_en_colorlib_con_usuario_y_clave(String arg1, String arg2) throws Throwable {
7         // Write code here that turns the phrase above into concrete actions
8         throw new PendingException();
9     }
10
11     @Given("^Ingreso a la funcionalidad Forms Validation$")
12     public void ingreso_a_la_funcionalidad_Forms_Validation() throws Throwable {
13         // Write code here that turns the phrase above into concrete actions
14         throw new PendingException();
15     }
16
17     @When("^Diligencio Formulario Popup Validation$")
18     public void diligencio_Formulario_Popup_Validation() throws Throwable {
19         // Write code here that turns the phrase above into concrete actions
20         throw new PendingException();
21     }
22
23     @Then("^Verifico ingreso exitoso$")
24     public void verifico_ingreso_exitoso() throws Throwable {
25         // Write code here that turns the phrase above into concrete actions
26         throw new PendingException();
27     }
28 }

```

Como se observa, las etiquetas se encuentran subrayadas, por lo que ubique el cursor sobre la misma y seleccione la opción import 'Given' (cucumber.api.java.en) y así mismo para el When y Then.



Por último, limpiar los métodos para que queden de la siguiente forma:

```
3 import cucumber.api.java.en.Given;
4 import cucumber.api.java.en.Then;
5 import cucumber.api.java.en.When;
6
7 public class PopupValidationDefinition {
8
9     @Given("^Autentico en colorlib con usuario \"([^\"]*)\" y clave \"([^\"]*)\"")
10     public void autentico_en_colorlib_con_usuario_y_clave(String arg1, String arg2) {
11
12     }
13
14     @Given("^Ingreso a la funcionalidad Forms Validation$")
15     public void ingreso_a_la_funcionalidad_Forms_Validation() {
16
17     }
18
19     @When("^Diligencio Formulario Popup Validation$")
20     public void diligencio_Formulario_Popup_Validation() {
21
22     }
23
24     @Then("^Verifico ingreso exitoso$")
25     public void verifico_ingreso_exitoso() {
26
27     }
28 }
```

Como se observa, cada línea **gherkin** en la **feature** mapea con una anotación en la **Definition**.

Observe como la línea "Given Autentico en colorlib con usuario "demo" y clave "demo" los parámetros son interpretados a través de expresiones.

