Table des matières

Cucumber

Bon à savoir

Cucumber

Est un framework de test dédié à l'écriture de tests fonctionnels dans un style *behaviour-driven development* (BDD). La description des tests s'effectue au moyen de Gherkin, un langage non-technique et orienté langage naturel afin de s'adresser à toute l'équipe de développement, y compris les analystes métiers. Gherkin supporte une cinquantaine de langues (les mots-clés ont été traduits) ce qui facilite son utilisation par des utilisateurs non-techniques.

BDD

1. Faire ce qu´il faut de préparation, analyse, ..., mais pas plus.
2. Livrer quelque chose qui a de la valeur : si vous n'êtes pas en train de faire quelque chose qui a de la valeur ou qui permettra d´en rajouter, alors arrêtez tout de suite.
3. Tout est un comportement : que ce soit au niveau du code, de l´application ou au-delà, nous pouvons utiliser la même façon de penser et les mêmes constructions linguistiques pour décrire le comportement à tout niveau de granularité.

De façon plus large, le BDD fait partie des méthodes dites « Agile », de la seconde génération. Les méthodes Agile visent à améliorer le processus de création de logiciels en refondant les grandes lignes de celui-ci. Cela avait été résumé dans le *Agile manifesto* et notamment les « 4 valeurs » :

Davantage l´interaction avec les personnes que les processus et les outils.
Davantage un produit opérationnel qu´une documentation pléthorique.
Davantage la collaboration avec le client que la négociation de contrat.
Davantage la réactivité face au changement que le suivi d´un plan.

Un exemple concret de l´utilisation de Cucumber est le suivant : élaborer dès le départ du projet la liste des fonctionnalités et leurs descriptions avec le client. Imaginez-vous 2 minutes dans une salle de réunion comme nous les connaissons tous. Avec votre équipe, et le ou les clients en face. L´idée est de pousser le client à définir le plus possible ses besoins (avec votre aide) et de se servir de ce processus pour éclairer des zones d´ombres et comprendre ses besoins.
En assénant la question « pourquoi ? » un nombre suffisant de fois, le client détaillera le processus lié à une fonctionnalité demandée, et ce jusqu'à décrire son besoin initial (réduire les coûts, etc.).
Un exemple cité est le suivant (C : client, D : développeur) :

C : « Il nous faut une possibilité d´imprimer » ;
D : « Pourquoi ? » ;
C : « Parce qu´il nous faut pouvoir avoir les données sur une feuille » ;
D : « Pourquoi ? » ;
C : « Parce qu´on s´en sert pour rentrer les données dans ce poste et celui-ci. »

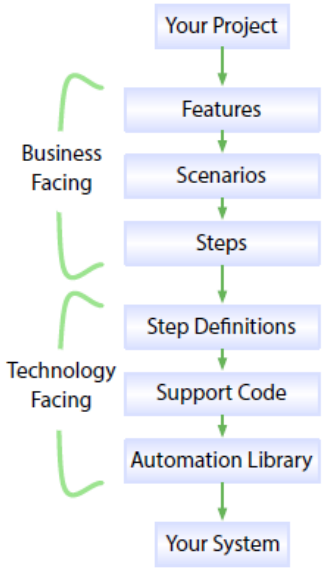
Le lecteur malin comprendra vite que l´impression est donc remplaçable par un transfert de données entre les postes. (Sauvant ainsi un arbre et ses enfants du tronçonnage).
Pour savoir plus
Le développement axé sur le comportement (BDD) s'appuie sur le développement piloté par le test (TDD) en formalisant les bonnes habitudes des meilleurs praticiens du TDD. Les meilleurs praticiens TDD travaillent de l'extérieur, en commençant par un test d'acceptation du client qui échoue, qui décrit le comportement du système du point de vue du client. En tant que praticiens BDD, nous prenons soin de rédiger les tests d'acceptation à titre d'exemples pouvant être lus par tous les membres de l'équipe. Nous utilisons le processus de rédaction de ces exemples pour obtenir les commentaires des parties prenantes de l'entreprise sur la question de savoir si nous sommes en train de créer la bonne chose avant de commencer. Ce faisant, nous faisons un effort délibéré pour élaborer un langage commun et omniprésent pour parler du système.

Tests d'acceptation automatisés

L'idée des tests d'acceptation automatisés trouve son origine dans la programmation extrême (XP), en particulier dans la pratique du développement piloté par les tests (TDD).
Au lieu d'un acteur de l'entreprise (Stackeholder) répondant aux exigences de l'equipe de développement sans avoir de possibilité d'avoir des retours, le développeur et les parties prenantes collaborer écrivent les tests automatisés qui expriment le résultat voulue par les parties prenantes(Stackholder)
Nous les appelons des tests d'acceptation parce qu'ils expriment ce que le logiciel doit faire pour que la partie prenante le trouve acceptable. Le test échoue au moment de l'écriture, car aucun code n'a encore été écrit, mais il capture ce qui intéresse l'intéressé et donne à chacun un signal clair quant à ce qu'il faudra faire.
Ces tests sont différents des tests unitaires, destinés aux développeurs. Il les aident à conduire et à vérifier leurs conceptions logicielles. On dit parfois que les tests unitaires vous permettent de construire la chose correctement, alors que les tests d'acceptation vous assurez de construire la bonne chose.
Les tests d'acceptation automatisés sont une pratique établie parmi les bonnes des équipes XP depuis des années, mais beaucoup d'équipes Agiles moins expérimentées semblent voir TDD comme étant une activité de programmeur uniquement. Comme Lisa Crispin et Janet Gregory pointent des tests agiles: Guide pratique pour les testeurs et les équipes agiles [CG08], sans les tests de réception automatisés pour les entreprises, il est difficile pour le les programmeurs de savoir quels tests unitaires ils doivent écrire. Les Tests d'acceptation automatisés aident votre équipe à se concentrer, assurant le travail que vous effectuez à chaque itération c'est la chose la plus précieuse que vous puissiez faire. Vous ferez encore des erreurs -mais vous en ferez beaucoup moins, ce qui signifie que vous pouvez rentrer chez vous à temps et profitez du reste de votre vie.

Comment Cucumber fonctionne

Avant de plonger dans les fondements du livre, voyons un peu le contexte avec un survol de haut niveau d'une suite de tests typique de Cucumber. Cette structure, allant des fonctionnalités à la bibliothèque d'automatisation, est illustrée dans la figure.
Cucumbre a été créé à l'origine comme un outil de ligne de commande par les membres de la communauté Ruby. Il a depuis été traduit dans plusieurs environnements de développement, y compris Java, pour permettre à un plus grand nombre d'entre nous d'en profiter. Lorsque vous exécutez Cucumber, vos spécifications sont lues à partir de fichiers texte en langage clair appelés **features**, elles sont examinées à la recherche de scénarios à tester et sont exécutées sur votre système. Chaque scénario est une liste d'étapes(**Steps**) que le Cucumber doit suivre.
Pour que Cucumber puisse comprendre ces fichiers de fonctionnalités, ils doivent suivre certaines règles de syntaxe de base. Le nom de cet ensemble de règles est Gherkin. En plus des **features**, vous fournissez à Cucumber un ensemble de définitions step, qui mappent le code lisible par l'entreprise de chaque étape en code (écrit en Java tout au long de ce livre) pour exécuter l'action décrite par l'étape. Dans une suite de tests aboutie, la définition d'étape elle-même ne sera probablement composée que d'une ou deux lignes de code déléguant à une bibliothèque de code de support, spécifique au domaine de votre application, sachant effectuer des tâches courantes sur le système. Cela peut parfois impliquer l'utilisation d'une bibliothèque d'automatisation, comme la bibliothèque d'automatisation du navigateur Selenium, pour interagir avec le système lui-même.



Exemple d'un test d'une fonction

Feature: Sign-up
  Sign-up should be quick and friendly.
  Scenario: Successful sign-up
    New users should get a confirmation email and be greeted
    personally by the site once signed in.
    Given I have chosen to sign up
    When I sign up with valid details
    Then I should receive a confirmation email
    And I should see a personalized greeting message
  Scenario: Duplicate email
    Where someone tries to create an account for an email address
    that already exists.
    Given I have chosen to sign up
    When I sign up with an email address that has already registered
    Then I should be told that the email is already registered
    And I should be offered the option to recover my password

Première tache

Comprendre l'objectif :

La caisse gardera une trace du coût total. Ainsi, par exemple, si le prix des articles disponibles ressemble à ceci :
banana 40c
apple 25c

Et seulement l' élément scanné à la caisse et le suivant :
1 banana

Alors la sortie sera 40c

Similaire. Si vous scannez plusieurs éléments :
3 apple

Alors la sortie sera 75c

Le même test en anglais

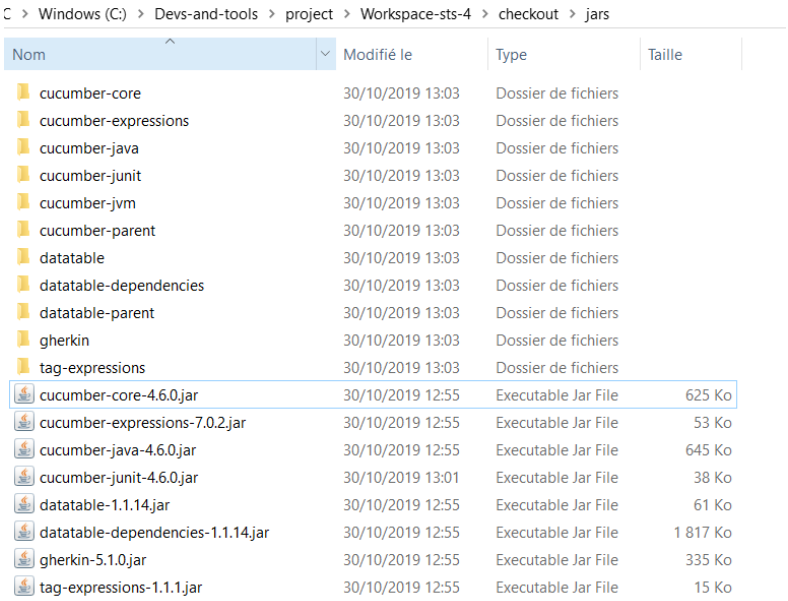
The checkout will keep track of the total cost. So, for example, if the prices
of available items looks like this:
banana 40c
apple 25c
and the only item you scan at the checkout is this:
1 banana
then the output will be 40c.
Similarly, if you scan multiple items:
3 apple
then the output will be 75c.

Création de la Feature

Dans un premier temps on ne va pas utiliser un IDE, on va utiliser seulement le compilateur java en ligne de commande plus les jar de Cucumber

Créer un dossier checkout pour les tests
mkdir checkout
cd checkout

Créer un dossier checkout pour les tests
mkdir jars
Télécharger cucumber et copié les jars dans le dossier jars crée. J'ai utilisé maven pour récupérer la dépendance. La version utiliser c'est 4.6.0


C > Windows (C:) > Devs-and-tools > project > Workspace-sts-4 > checkout > jars

Nom	Modifié le	Type	Taille
cucumber-core	30/10/2019 13:03	Dossier de fichiers	
cucumber-expressions	30/10/2019 13:03	Dossier de fichiers	
cucumber-java	30/10/2019 13:03	Dossier de fichiers	
cucumber-junit	30/10/2019 13:03	Dossier de fichiers	
cucumber-jvm	30/10/2019 13:03	Dossier de fichiers	
cucumber-parent	30/10/2019 13:03	Dossier de fichiers	
datatable	30/10/2019 13:03	Dossier de fichiers	
datatable-dependencies	30/10/2019 13:03	Dossier de fichiers	
datatable-parent	30/10/2019 13:03	Dossier de fichiers	
gherkin	30/10/2019 13:03	Dossier de fichiers	
tag-expressions	30/10/2019 13:03	Dossier de fichiers	
cucumber-core-4.6.0.jar	30/10/2019 12:55	Executable Jar File	625 Ko
cucumber-expressions-7.0.2.jar	30/10/2019 12:55	Executable Jar File	53 Ko
cucumber-java-4.6.0.jar	30/10/2019 12:55	Executable Jar File	645 Ko
cucumber-junit-4.6.0.jar	30/10/2019 13:01	Executable Jar File	38 Ko
datatable-1.1.14.jar	30/10/2019 12:55	Executable Jar File	61 Ko
datatable-dependencies-1.1.14.jar	30/10/2019 12:55	Executable Jar File	1 817 Ko
gherkin-5.1.0.jar	30/10/2019 12:55	Executable Jar File	335 Ko
tag-expressions-1.1.1.jar	30/10/2019 12:55	Executable Jar File	15 Ko

Exécuté la commande

$ java -cp "jars/*" io.cucumber.core.cli.Main -p pretty .

Résultat :
No features found at [.]
0 Scenarios
0 Steps
0m0.000s

-p pretty tells cucumber to use the pretty formatter plugin (you'll see why later)
. a path that points to where our feature files are located

Puisque on va écrire les commande dans l'invite de commande , il sera préférable de créer un fichier .bat pour enregistrer les commandes à exécuter , exemple **cucumber.bat**
Contenu de fichier c'est la ligne précédente **$ java -cp "jars/*" io.cucumber.core.cli.Main -p pretty .**

On va créer un dossier ou se trouve nos features
mkdir features
cd features
Type null > checkout.feature ou touch checkout.feature

On va spécifier ou cucumber peut trouver les features
$ java -cp "jars/*" io.cucumber.core.cli.Main -p pretty features
Exécuter encore le fichier cucumber.bat apres l'avoir modification

Résultat :
No features found at [features]
0 Scenarios
0 Steps
0m0.000s

Chaque test de Cucumbre est appelé scénario et chaque scénario contient des étapes (Steps) indiquant au Cucumber quoi faire. Cette sortie signifie que Cucumber est en train de scanner le répertoire des fonctionnalités, mais il n'a trouvé aucun scénario à exécuter. Créons-en un.

Notre recherche auprès des utilisateurs nous a montré que 67% des acheteurs achètent des bananes. C'est donc ce que nous allons commencer avec.
Dans le fichier checkout.feature , ajouter le scenario de test

```
Feature: Checkout
  Scenario: Checkout a banana
    Given the price of a "banana" is 40c
    When I checkout 1 "banana"
    Then the total price should be 40c
```

Les mots clés Freature , Scenario , Given , When , Then sont la structure pour ecrire un test en langage Natural .L'écriture des test se basent sur **Gherkin**

Exécuter le fichier checkout.feature
Resultat :

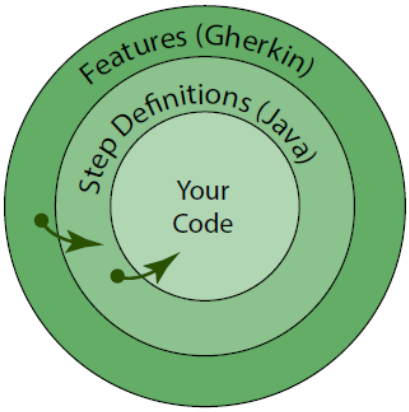
```
Feature: Checkout
  Scenario: Checkout a banana # checkout.feature:2
    Given the price of a "banana" is 40c
    When I checkout 1 "banana"
    Then the total price should be 40c
1 Scenarios (1 undefined)
3 Steps (3 undefined)
0m0.000s

You can implement missing steps with the snippets below:

@Given("^the price of a \"(.*?)\" is (\\d+)c$")
public void the_price_of_a_is_c(String arg1, int arg2) throws Throwable {
  // Write code here that turns the phrase above into concrete actions
  throw new PendingException();
}

@When("^I checkout (\\d+) \"(.*?)\"$")
public void i_checkout(int arg1, String arg2) throws Throwable {
  // Write code here that turns the phrase above into concrete actions
  throw new PendingException();
}
@Then("^the total price should be (\\d+)c$")
public void the_total_price_should_be_c(int arg1) throws Throwable {
  // Write code here that turns the phrase above into concrete actions
  throw new PendingException();
}
```

Cucumber a trouvé la feature et il propose les tests qu'il faut écrire pour les scenarios
Les méthode afficher par Cucumber sont en snake case pour les rendre en camalcase on va ajouter un plugin proposé par cucumber

$ java -cp "jars/*" io.cucumber.core.cli.Main -p pretty --snippets camelcase features

Résultat après l'exécution de fichier cucumber.bat

```
@Given("^the price of a \"(.*?)\" is (\\d+)c$")
public void thePriceOfAIsC(String arg1, int arg2) throws Throwable {
   // Write code here that turns the phrase above into concrete actions
   throw new PendingException();
}
@When("^I checkout (\\d+) \"(.*?)\"$")
public void iCheckout(int arg1, String arg2) throws Throwable {
   // Write code here that turns the phrase above into concrete actions
   throw new PendingException();
}
@Then("^the total price should be (\\d+)c$")
public void theTotalPriceShouldBeC(int arg1) throws Throwable {
   // Write code here that turns the phrase above into concrete actions
   throw new PendingException();
}
```



Avant d'explorer sous la couche d'entités Gherkin faisant face aux entreprises, il convient de jeter un coup d'œil rapide sur la carte au cas où quelqu'un se sentirait perdu. La figure nous rappelle comment les choses s'emboîtent. Nous commençons avec des features, qui contiennent nos scénarios et steps. Les steps de nos scénarios appellent des steps definitions qui fournissent le lien entre les fonctionnalités de Gherkin et l'application en cours de construction. Nous allons maintenant implémenter des step definition afin que notre scénario ne soit plus indéfini.

Création d'un dossier pour step definition

$ mkdir step_definitions

$ cd step_definitions

Créer la classe java que cucumber va utiliser comme step definition et exécuter chaque step comme test unitaire

Classe CheckoutSteps

```
package step_definitions;
import cucumber.api.PendingException;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;

public class CheckoutSteps {
    @Given("the price of a {string} is {int}c")
    public void thePriceOfAIsC(String string, Integer int1) {
        // Write code here that turns the phrase above into concrete actions
        throw new cucumber.api.PendingException();
    }
```

```
    @When("I checkout {int} {string}")
    public void iCheckout(Integer int1, String string) {
        // Write code here that turns the phrase above into concrete actions
        throw new cucumber.api.PendingException();
    }

    @Then("the total price should be {int}c")
    public void theTotalPriceShouldBeC(Integer int1) {
        // Write code here that turns the phrase above into concrete actions
        throw new cucumber.api.PendingException();
    }

}
```

Il faut compiler la classe avant l'exécution du test

```
echo "------------ build class"
javac -cp "jars/*" step_definitions/CheckoutSteps.java
echo "------------ test cucumber"
java -cp "jars/* :." io.cucumber.core.cli.Main -p pretty --snippets camelcase -g step_definitions features
```

Nous ajoutant le chemin actuel « . » au classpath
-g = glue
-g step_definition : nous indiquant à cucumber ou chercher les step definition qui vont être coller à la feature

Exécuter la cucumber.bat

```
Feature: Checkout
  Scenario: Checkout a banana # checkout.feature:2
    Given the price of a "banana" is 40c # CheckoutSteps.thePriceOfAIsC(String,int)
      cucumber.api.PendingException: TODO: implement me
        at step_definitions.CheckoutSteps.thePriceOfAIsC(CheckoutSteps.java:12)
        at *.Given the price of a "banana" is 40c(checkout.feature:3)
    When I checkout 1 "banana" # CheckoutSteps.iCheckout(int,String)
    Then the total price should be 40c # CheckoutSteps.theTotalPriceShouldBeC(int)

1 Scenarios (1 pending)
3 Steps (2 skipped, 1 pending)
0m0.138s

cucumber.api.PendingException: TODO: implement me
        at step_definitions.CheckoutSteps.thePriceOfAIsC(CheckoutSteps.java:12)
        at *.Given the price of a "banana" is 40c(checkout.feature:3)
```

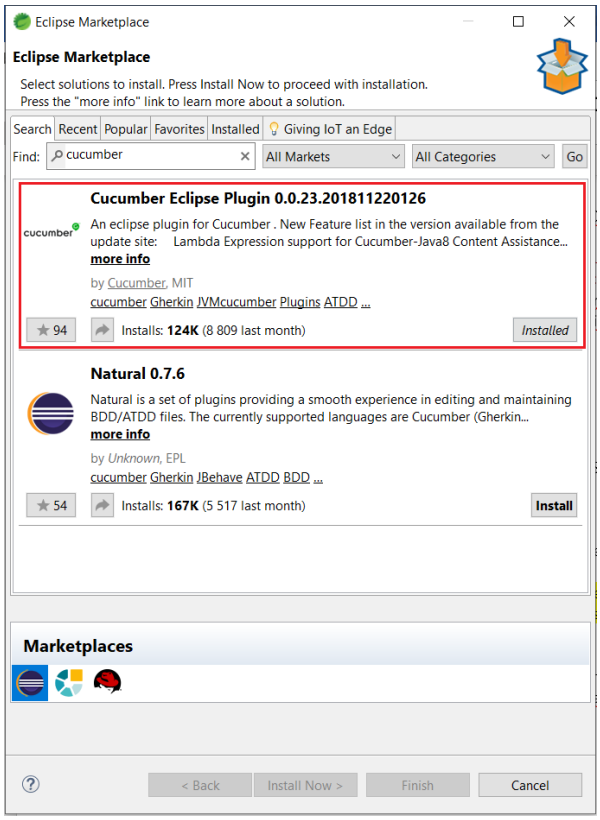
Dans la step definition toutes les methode retournent seulement des exception , il faut adapter le code pour tester correctement
Il faut savoir que cucumber s'il trouve un test en erreur il continu de faire les autres tests sans s'arrêté mais dans le résumé du test il montre bien que c'est un test en erreur

Création d'un projet avec Maven et Cucumber

Pour aller plus vite c'est plus bénéfique d'utiliser un idée avec un environnement
Ide : Eclipse , il faut installer le plugin Cucuber à partir de eclipse market
Créer un projet de type maven

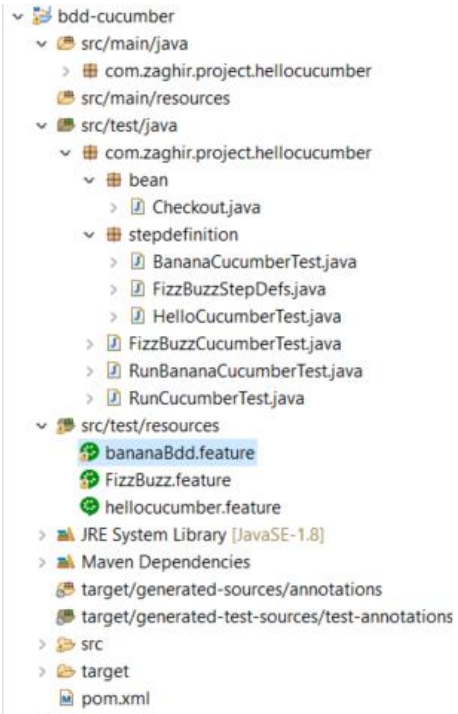
```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
	<modelVersion>4.0.0</modelVersion>
	<groupId>com.zaghir.project</groupId>
	<artifactId>bdd-cucumber</artifactId>
	<version>0.0.1-SNAPSHOT</version>
	<name>bdd-cucumber</name>

	<properties>
		<java.version>1.8</java.version>
		<maven.compiler.plugin.version>3.8.1</maven.compiler.plugin.version>
		<maven.surefire.plugin.version>2.22.0</maven.surefire.plugin.version>
		<lombok.version>1.18.10</lombok.version>
		<maven.compiler.source>1.8</maven.compiler.source>
		<maven.compiler.target>1.8</maven.compiler.target>
		<mockito.version>2.24.0</mockito.version>
		<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
		<cucumber.version>4.6.0</cucumber.version>
		<cucumber-junit.version>4.6.0</cucumber-junit.version>

	</properties>
	<dependencies>
		<!-- https://mvnrepository.com/artifact/io.cucumber/cucumber-java -->
		<dependency>
			<groupId>io.cucumber</groupId>
			<artifactId>cucumber-java</artifactId>
			<version>${cucumber.version}</version>
		</dependency>

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

		<dependency>
			<groupId>org.projectlombok</groupId>
			<artifactId>lombok</artifactId>
			<version>${lombok.version}</version>
			<optional>true</optional>
		</dependency>

		<dependency>
			<groupId>org.mockito</groupId>
			<artifactId>mockito-core</artifactId>
			<version>${mockito.version}</version>
			<scope>test</scope>
		</dependency>
		<dependency>
			<groupId>junit</groupId>
			<artifactId>junit</artifactId>
			<version>4.12</version>
		</dependency>
```

```
		<dependency>
			<groupId>org.slf4j</groupId>
			<artifactId>slf4j-api</artifactId>
			<version>1.7.28</version>
		</dependency>
		<dependency>
			<groupId>org.slf4j</groupId>
			<artifactId>slf4j-simple</artifactId>
			<version>1.7.28</version>
		</dependency>


	</dependencies>

	<build>
		<plugins>
			<plugin>
				<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-surefire-plugin</artifactId>
				<version>${maven.surefire.plugin.version}</version>
				<configuration>
					<argLine>-Duser.language=en</argLine>
					<argLine>-Xmx1024m</argLine>
					<argLine>-XX:MaxPermSize=256m</argLine>
					<argLine>-Dfile.encoding=UTF-8</argLine>
					<useFile>false</useFile>
				</configuration>
			</plugin>

			<plugin>
				<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-compiler-plugin</artifactId>
				<version>3.8.1</version>
				<configuration>
					<source>${maven.compiler.source}</source>
					<target>${maven.compiler.target}</target>
				</configuration>
			</plugin>
		</plugins>
	</build>
</project>
```



Développer le premier Test

Structure de projet :



Classe BananaCucumberTest.java

```
package com.zaghir.project.hellocucumber.stepdefinition;


import org.junit.Assert;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.zaghir.project.hellocucumber.bean.Checkout;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;

public class BananaCucumberTest {

	public static Logger logger =
LoggerFactory.getLogger(BananaCucumberTest.class);

	private int bananaPrice = 0 ;
	private Checkout checkout ;
	@Given("the price of a {string} is {int}c")
	public void thePriceOfAIsC(String name, Integer price) {
	    logger.info(" ---------- thePriceOfAIsC  --> "+name +"  --  "+price);
	    Assert.assertEquals(40 ,price.intValue() );
```

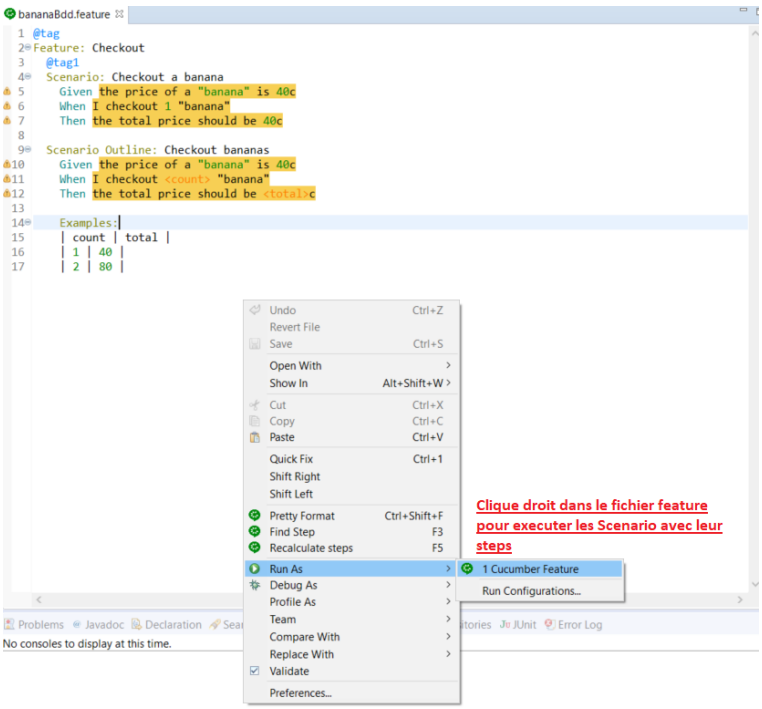
```
            bananaPrice = price ;
        }

        @When("I checkout {int} {string}")
        public void iCheckout(Integer itemCount, String itemName) {
            //logger.info(" ----------- Checkout");
            checkout = new Checkout();
            checkout.add(itemCount, bananaPrice);

        }

        @Then("the total price should be {int}c")
        public void theTotalPriceShouldBeC(Integer total) {
            //logger.info(" ----------- theTotalPriceShouldBeC");
            Assert.assertEquals(checkout.getTotal(), total.intValue());

        }

}
```

Fichier bananaBdd.feature

```
@tag
Feature: Checkout
        @tag1
  Scenario: Checkout a banana
    Given the price of a "banana" is 40c
    When I checkout 1 "banana"
    Then the total price should be 40c

  Scenario Outline: Checkout bananas
             Given the price of a "banana" is 40c
             When I checkout <count> "banana"
             Then the total price should be <total>c

             Examples:
             | count | total |
             | 1 | 40 |
```

```
| 2 | 80 |
```

Le plugin de ecplise scanne le dossier src/test/java, puis cherche avec les expressions régulières les steps définition



Pour compliquer et lancer les tests dans les environnements d'intégration il faut configure un runner pour les tests
On va spécifier une classe pour configurer et exécuter notre test

Classe : RunBananaCucumberTest

```
package com.zaghir.project.hellocucumber;

import org.junit.runner.RunWith;

import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;
import io.cucumber.junit.CucumberOptions.SnippetType;

@RunWith(Cucumber.class)
@CucumberOptions(
		plugin = {
				"pretty",
				"html:target/cucumber-html-report",
				"json:target/cucumber-json-report.json"
			} ,
		snippets = SnippetType.CAMELCASE,
		glue = "src/test/resources/stepdefinition",
		features = {"src/test/resources/bananaBdd.feature"}
		)

public class RunBananaCucumberTest {

}
```

`@RunWith(Cucumber.class)` : c'est le runner de Cucumber qui sera utiliser pour executer le test

```
plugin = {
  "pretty",
  "html:target/cucumber-html-report",
  "json:target/cucumber-json-report.json"
} ,
```

Cucumber propose plusieurs plugins ,
`pretty`: coloration du code en sortie
`html:target/cucumber-html-report` :gérer le report de test en forma html dans le dossier target
`json:target/cucumber-json-report.json :` gérer le report de test en forma html dans le dossier target

`snippets = SnippetType.CAMELCASE, :` pour le camalcase des méthodes générer

`glue = "src/test/resources/stepdefinition"` : spécifier l'endroit ou cucumber chercher les classes de step définition

`features = {"src/test/resources/bananaBdd.feature"} :` spécifier l'endroit ou cucumber chercher les fichiers .feature pour les tests

Les bases de Gherkins

Gherkin c'est le langage avec lequel on écrie les tests fonctionnels

Le chalenge ici c'est d'écrire des bons tests d'acceptances automatisés qui seront lisibles no pas par les ordinateurs mais aussi par les stakeholders. C'est cette lecture humaine qui nous permet d'avoir un feedback sur ce qu'on a créé et ce qu'on va créer, et c 'est ici ou Gherkin entre en jeu

Gherkin nous offre une structure légère pour documenter des exemples du comportement souhaité par nos parties prenantes, de manière à être facilement compris par les parties prenantes et par Cuncumber. Bien que nous puissions appeler le langage de programmation Gherkin, son objectif principal est la lisibilité, ce qui signifie que vous pouvez écrire des tests automatisés qui se lisent comme de la documentation. Voici un exemple :

Feature: Feedback when entering invalid credit card details
  In user testing we've seen a lot of people who made mistakes entering their credit card. We need to be as helpful as possible here to avoid losing users at this crucial stage of the transaction.
  Background:
    Given I have chosen some items to buy
    And I am about to enter my credit card details
  Scenario: Credit card number too short
    When I enter a card number that's only 15 digits long
    And all the other details are correct
    And I submit the form
    Then the form should be redisplayed
    And I should see a message advising me of the correct number of digits
  Scenario: Expiry date must not be in the past
    When I enter a card expiry date that's in the past
    And all the other details are correct
    And I submit the form
    Then the form should be redisplayed
    And I should see a message telling me the expiry date must be wrong

Une caractéristique intéressante de la syntaxe de Gherkin est qu'elle n'est pas liée à une langue parlée particulière. Chacun des mots-clés de Gherkin a été traduit dans plus de quarante langues parlées. Il est donc parfaitement correct de les utiliser pour rédiger vos fonctionnalités Gherkin. Peu importe si vos utilisateurs parlent le norvégien ou l'espagnol, cliquez sur Gherkin kan du livre funksjonalitet i et språk de vil forstå. (Gherkin vous permet d'écrire vos fonctionnalités dans une langue qu'ils comprendront.) Tocino grueso! (Bacon Chunky!) Plus sur cela plus tard.

Format et syntax

Les fichiers Gherkin utilisent l'extension de fichier .feature. Ils sont enregistrés sous forme de texte brut, ce qui signifie qu'ils peuvent être lus et modifiés avec des outils simples. À cet égard, Gherkin est très similaire aux formats de fichiers tels que Markdown, Textile et YAML.

Keywords

Un fichier Gherkin prend sa structure et sa signification en utilisant un ensemble de mots-clés spéciaux.
Il existe un ensemble équivalent de ces mots-clés dans chacun langue parlées , mais pour le moment jetons un coup d'œil à langue anglaise

- Feature
- Background
- Scenario
- Given
- When
- Then
- And
- But
- *
- Scenario Outline
- Examples

Dry Run

Le paramètre --dry-run indique à Cucumber d'analyser le fichier sans l'exécuter. Il vous dira si votre Gherkin n'est pas valide.

```
java -cp "jars/*" io.cucumber.core.cli.Main -p pretty -g step_definition –dry-run features
```

Feature

Chaque fichier Gherkin commence par le mot-clé Feature. Ce mot clé n'affecte pas du tout le comportement de vos tests Cucumber; cela vous donne simplement un endroit pratique pour mettre une documentation sommaire sur le groupe de tests qui suit.
Exemple :

```
Feature: This is the feature title
  This is the description of the feature, which can span multiple lines.
  You can even include empty lines, like this one:
  In fact, everything until the next Gherkin keyword is included in the description.
```

Le texte qui suit immédiatement sur la même ligne que le mot-clé Feature est le nom de la fonctionnalité et les lignes restantes correspondent à sa description. Vous pouvez inclure tout texte de votre choix dans la description, à l'exception d'une ligne commençant par l'un des mots Scénario, Background ou Scenario Outline. La description peut s'étendre sur plusieurs lignes. C'est un bon endroit pour crier lyrique avec des détails sur qui utilisera la fonctionnalité et pourquoi, ou pour

mettre des liens vers des documents de support tels que des wireframes ou des enquêtes de recherche sur les utilisateurs.
Il est classique de nommer le fichier de fonctions en convertissant le nom de la fonction en minuscules et en remplaçant les espaces par des traits de soulignement. Ainsi, par exemple, la fonctionnalité nommée User logs in serait stockée dans user_logs_in.feature.
Dans Gherkin valide, une entité doit être suivie de l'un des éléments suivants:
• Scenario
• Background
• Scenario Outline

Scenario

Pour exprimer le comportement souhaité, chaque feature contient plusieurs scénarios. Chaque scénario est un exemple concret unique de la manière dont le système devrait se comporter dans une situation donnée. Si vous additionnez le comportement défini par tous les scénarios, il s'agit du comportement attendu de la fonctionnalité elle-même.
Lorsque Cucumber exécute un scénario, si le système se comporte comme décrit dans le scénario, le scénario passera sinon, ça va échouer. Chaque fois que vous ajoutez un nouveau scénario à votre suite de tests Cucumber et que vous le faites passer, vous ajoutez de nouvelles fonctionnalités au système, et le moment est venu pour un high-five.
Chaque fonctionnalité comporte généralement entre cinq et vingt scénarios, chacun décrivant différents exemples de la manière dont cette fonctionnalité doit se comporter dans différentes circonstances. Nous utilisons des scénarios pour explorer des cas critiques et différents chemins à travers une fonctionnalité.
Les scénarios suivent tous le même schéma:
1. Obtenez le système dans un état particulier.
2. Poke-le (ou chatouille-le, ou…).
3. Examiner le nouvel état.

Nous commençons donc par un contexte, décrivons ensuite une action, puis vérifions que le résultat obtenu correspond à ce que nous attendions. Chaque scénario raconte une petite histoire décrivant quelque chose que le système devrait pouvoir faire.

Given, When , Then

Dans Gherkin, nous utilisons les mots-clés Given, When et Then pour identifier ces trois parties différentes du scénario:

Scenario: Successful withdrawal from an account in credit
  Given I have $100 in my account *# the context*
  When I request $20 *# the event(s)*
  Then $20 should be dispensed *# the outcome(s)*

Nous utilisons donc **Given** pour définir le contexte dans lequel le scénario se produit,
When interagir avec le système d'une manière ou d'une autre
Then pour vérifier que le résultat de cette interaction correspond bien à nos attentes.

And , But

Chacune des lignes d'un scénario est appelée step. Nous pouvons ajouter plus d'étapes à chaque Given , When, ou Then du scénario à l'aide des mots-clés And and But:

Scenario: Attempt withdrawal using stolen card
  Given I have $100 in my account

But my card is invalid
When I request $50
Then my card should not be returned
And I should be told to contact the bank

Le Cucumber ne s'inquiète pas réellement de l'utilisation de ces keywords, le choix est simplement là pour vous aider à créer le scénario le plus lisible. Si vous ne voulez pas utiliser And ou But, vous pouvez écrire le scénario précédent de la sorte, et cela fonctionnera toujours de la même manière:

Scenario: Attempt withdrawal using stolen card
Given I have $100 in my account
Given my card is invalid
When I request $50
Then my card should not be returned
Then I should be told to contact the bank
Mais ce n'est pas jolie ?

Remplacer Given/When/Then with Bullets

Certaines personnes trouvent Given, When, Then, And et But But un peu verbeux. Il existe un mot clé supplémentaire que vous pouvez utiliser pour démarrer une étape: * (un astérisque). Nous aurions pu écrire le scénario précédent comme ceci :

Scenario: Attempt withdrawal using stolen card
* I have $100 in my account
* my card is invalid
* I request $50
* my card should not be returned
* I should be told to contact the bank

Pour Cucumber, c'est exactement le même scénario. Trouvez-vous cette version plus facile à lire? Peut-être. Une partie du sens s'est-elle perdue ? Peut-être. C'est à vous et à votre équipe de décider comment vous voulez exprimer les choses. La seule chose qui compte, c'est que tout le monde comprenne ce qui est communiqué.

Stateless

Lors de la rédaction de scénarios, voici un concept très important que vous devez comprendre: Chaque scénario doit avoir un sens et pouvoir être **exécuté indépendamment** de tout autre scénario. Cela signifie que vous ne pouvez pas placer d'argent sur le compte dans un scénario et vous attendez à ce que cet argent soit disponible dans le scénario suivant. Le Cucumber ne vous empêchera pas de le faire, mais c'est une très mauvaise pratique: vous allez vous retrouver avec des scénarios qui échouent de manière inattendue et qui sont plus difficiles à comprendre.
Cela peut sembler un peu dogmatique, mais surement, cela aide vraiment à garder vos scénarios simples à utiliser. Cela évite de créer des dépendances fragiles entre les scénarios et vous offre également la possibilité d'exécuter les scénarios nécessaires lorsque vous travaillez sur une partie particulière du système, sans avoir à vous soucier de la configuration des données de test appropriées.
Lors de l'écriture d'un scénario, supposez toujours qu'il s'exécutera sur le système dans un état vide par défaut. Racontez l'histoire dès le début, en utilisant Given Steps pour configurer tout l'état dont vous avez besoin pour ce scénario particulier.

Nom et description

Tout comme une Feature, un mot-clé de scénario peut être suivi d'un nom et d'une description. Normalement, vous utiliserez probablement simplement le nom, mais Gherkin est autorisé à suivre le nom avec une description multiligne - tout jusqu'au premier, Given ou Then sera intégré à la description du scénario.
Les noms de scénarios périmés peuvent être source de confusion. Lorsque vous modifiez des scénarios existants (ou que vous les copiez et les collez), veillez à ce que le nom ait toujours un sens. Comme le nom du scénario n'est qu'une simple documentation, Cucumber n'échouera pas le scénario, même si son nom n'a plus rien à voir avec ce qui se passe réellement dans les étapes. Cela peut être très déroutant pour quiconque qui lira le scénario plus tard.

Comments

En plus des champs de description qui suivent les mots-clés Feature et Scénario, Cucumber vous permet de faire précéder ces mots-clés de commentaires.
Les commentaires commencent par un caractère # et doivent être la première et unique chose sur une ligne (enfin, à part les espaces).

```
# This feature covers the account transaction and hardware-driver modules
Feature: Withdraw Cash
  In order to buy beer
  As an account holder
  I want to withdraw cash from the ATM
  # Can't figure out how to integrate with magic wand interface
  Scenario: Withdraw too much from an account in credit
    Given I have $50 in my account
    # When I wave my magic wand
    And I withdraw $100
    Then I should receive $100
```

Vous pouvez également mettre des commentaires dans un scénario. L'utilisation la plus courante consiste à commenter une étape, comme nous l'avons montré dans l'exemple précédent.
Comme dans tout langage de programmation, les commentaires peuvent rapidement devenir obsolètes et perdre tout leur sens ou être source de confusion. Lorsque cela se produit, le commentaire fait plus de mal que de bien. Il faut les utiliser avec parcimonie et de placer les informations importantes dans des scénarios permettant de les tester.
Voici comment nous pensons à ce sujet : la description, que vous pouvez mettre après chaque mot clé, fait partie du document Gherkin structuré et constitue le lieu approprié pour mettre la documentation pour vos parties prenantes.
Les commentaires, par contre, peuvent être davantage utilisés pour laisser des notes aux testeurs et aux programmeurs qui travaillent avec les Features. Pensez à un commentaire comme quelque chose de plus temporaire, un peu comme un post-it.
N'oubliez pas que les programmeurs et les testeurs ont également besoin de documentation. Si certains détails techniques doivent être documentés à l'aide de la fonctionnalité, vous pouvez également les inclure dans la description, à condition que les membres de votre équipe qui font face à l'entreprise soient à l'aise avec leur présence

Les langages parlés

Ecrire vos fonctionnalités Gherkin dans la langue parlée par vos stackholder c'est possible.

Dans la première ligne de votre fichier .feature specifié la langue à utiliser par un # suivis dans la langue voulu Voici comment.
Si vous omettez cet en-tête, Cucumber passera par défaut à l'anglais, comme vous l'avez déjà vu.
Voici un exemple de fonction écrite en norvégien:

```
# language: no
Egenskap: Summering
  For å unngå at firmaet går konkurs
  Må regnskapsførerere bruke en regnemaskin for å legge sammen tall
  Scenario: to tall
    Gitt at jeg har tastet inn 5
    Og at jeg har tastet inn 7
    Når jeg summerer
    Så skal resultatet være 12
```

Si vous vous demandez si Cucumber sait parler votre langue, vous pouvez lui demander la liste de toutes les langues valides avec cette commande:

```
$ ./cucumber --i18n help
```

Lorsque vous travaillez avec une langue particulière, vous pouvez découvrir les mots-clés en transmettant le code de langue (indiqué dans la commande précédente) à
le commutateur --i18n. Voici le japonais, par exemple:

```
$ ./cucumber --i18n ja
```

L'option --i18n n'est disponible que dans Cucumber-JVM 1.2.0. Auparavant, le moyen le plus simple de connaître les langues prises en charge était de consulter le projet Gherkin à l'adresse lib / i18n.json.5.
L'un des principaux avantages de l'utilisation d'un outil tel que Cucumber réside dans les conversations que vous avez avec vos parties prenantes lors de la rédaction des scénarios. Ces conversations peuvent vous aider à identifier les lacunes et les malentendus qui auraient autrement pu émerger après que vous ayez passé des jours voire des semaines à travailler sur le code. Ainsi, même si vous n'exécutez jamais les tests, les écrire peut vous aider à accélérer les tests.

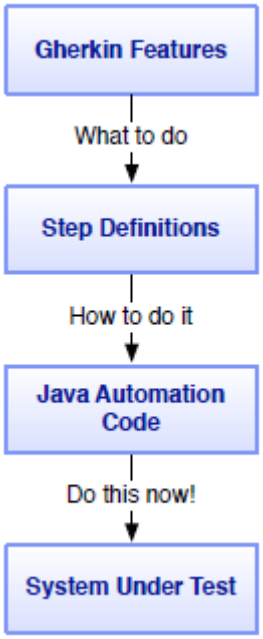
Step definition

Maintenant que vous savez utiliser Gherkin pour décrire ce que vous voulez que vos tests fassent, la tâche suivante consiste à leur expliquer comment le faire. Que vous choisissiez de piloter vos tests d'acceptance à partir de scénarios Cucumber ou de tests JUnit simples, rien ne vous échappera: vous devrez éventuellement écrire du code.
Les définitions d'étape(Step definitions) sont situées à la limite du domaine de métier et du domaine du programmeur. Vous pouvez les écrire dans de nombreux langages JVM (pour l'instant, nous montrerons des exemples en Java). Leur responsabilité est de traduire chaque étape en langage clair de vos scénarios Gherkin en actions concrètes dans votre code. Par exemple, prenons cette étape du scénario ATM de précédent chapitre
Given I have $100 in my Account
This step definition needs to make the following things happen:

- Create an account for the protagonist in the scenario (if there isn't one already).
- Set the balance of that account to be $100.

La manière dont ces deux objectifs sont atteints dépend en grande partie de votre application. Les tests d'acceptance automatisés tentent généralement de simuler les interactions utilisateur avec le

système, et les définitions d'étape permettent de dire à Cucumber comment vous voulez que votre système le fouille. Cela peut impliquer de cliquer sur des boutons sur une interface utilisateur ou d'atteindre sous les couvertures et d'insérer directement des données dans une base de données, d'écrire dans des fichiers ou même d'appeler un service Web. Nous pensons que les définitions d'étape elles-mêmes sont distinctes du code d'automatisation qui effectue la sélection pour que les couches se séparent, comme le montre la figure suivante :



Il y a deux côtés d'une Step definition. À l'extérieur, il traduit le langage courant en code ,et à l'intérieur il indique à votre système quoi faire avec le code d'automatisation. La JVM dispose d'un ensemble incroyablement riche de bibliothèques pour automatiser une grande variété de systèmes, des applications Web JavaScriptheavy aux services Web REST. On va pas montrer comment utiliser toutes ces bibliothèques ; cela viendra plus tard . Ici, nous allons nous concentrer sur la responsabilité principale de cette couche de vos tests Cucumber, qui consiste à interpréter une étape de Gherkin en langage simple et à décider quoi faire.
Nous allons commencer par expliquer certains des mécanismes de la mise en correspondance des définitions d'étapes avec les étapes en langage simple, puis nous montrerons comment écrire une définition en une seule étape capable de gérer de nombreuses étapes différentes. Nous terminerons en expliquant comment Cucumber exécute les Steps definition et traite leurs résultats. Lorsque nous aurons terminé, vous devriez comprendre suffisamment pour commencer à écrire et à exécuter vos propres définitions d'étape.

Step et Step definition

Commençons par clarifier la distinction entre une étape(Step) et une définition d'étape(Step definition).
Chaque scénario de Gherkin est composé d'une série d'étapes écrites en langage clair. En soi, une étape n'est que de la documentation, elle a besoin d'une définition par étapes(step defintion) pour lui donner vie. Un Step definition est un morceau de code qui dit à Cucumber: "Si vous voyez une étape qui ressemble à ceci ..., alors voici ce que je veux que vous fassiez .."
Lorsque Cucumber tente d'exécuter chaque étape, il recherche une définition d'étape correspondante à exécuter. Alors, comment Cucumber correspond-il à une définition d'étape?

Correspondance avec une étape (Matching a step)

Les Steps Gherkin sont exprimées en texte brut. Cucumber analyse le texte de chaque étape à la recherche des motifs qu'il reconnaît, que vous définissez à l'aide d'une expression régulière. Si vous n'avez jamais utilisé d'expressions régulières auparavant, imaginez-les comme une version légèrement plus sophistiquée des caractères génériques que vous utiliseriez pour rechercher un fichier. Bien qu'ils puissent sembler intimidants au début, vous n'avez besoin que d'un petit nombre de modèles pour en faire beaucoup.

```
Feature: Cash withdrawal
  Scenario: Successful withdrawal from an account in credit
    Given I have $100 in my account
    When I request $20
    Then $20 should be dispensed
```

Au fur et à mesure que Cucumber exécute la Feature, il passe au premier Step du scénario. Étant donné que j'ai 100 $ dans mon compte, ilse dit: Maintenant, ai-je des Steps definition qui correspondent à l'expression I have $100 in my Account?

Une expression régulière simple qui correspondra à cette étape ressemblerait à ceci:
"I have \\$100 in my Account"
Notez que nous avons dû échapper au signe dollar avec une double barre oblique inverse. C'est parce que le signe dollar peut avoir une signification particulière dans une expression régulière, mais dans ce cas, nous voulons l'interpréter littéralement. Pour rendre la vie encore plus compliquée, la barre oblique inversée a une signification particulière dans une chaîne de caractère Java. Nous devons donc utiliser une double barre oblique inversée. Si Cucumber voit une définition d'étape avec cette expression régulière, il l'exécutera lorsqu'il s'agira de la première étape de notre scénario. Alors, comment créer un Step definition?

Création d'un Step definition

Les Steps definitions vivent dans des fichiers ordinaires. Pour créer un step definition en Java, vous utilisez une annotation Cucumber spéciale, telle que @Given, comme ceci:

```
@Given("I have \\$100 in my Account")
public void iHave$100InMyAccount() throws Throwable {
   // TODO: code that puts $100 into User's Account goes here
}
```

Souvent, vous allez regrouper plusieurs steps definitions comme celle-ci dans le même fichier source. Etant donné que vous devez indiquer à Cucumber où trouver vos définitions d'étape(Steps definitions), c'est à vous de décider comment vous souhaitez les organiser. Nous vous suggérons de conserver un fichier distinct par entité de domaine afin que les définitions d'étape qui fonctionnent avec des parties similaires du système soient conservées.
Examinons la définition de Step en détail. Ceci est un fichier Java, et nous utilisons l'annotation spéciale Cucumber @Given, qui indique à Cucumber que nous voulons enregistrer un Step definition. Nous passons à l'annotation @Given une expression régulière pour correspondre à une ou plusieurs Steps (le bit entre les guillemets), et nous définissons une méthode Java qui s'exécutera lorsqu'elle correspond. Cucumber stocke un mappage entre l'expression régulière et la méthode. Il peut donc appeler la méthode ultérieurement si une étape correspondante est détectée. Vous pouvez également utiliser les annotations @When ou @Then pour créer une définition d'étape de la même manière.

Giver, When , Then , sont les même

Quelle que soit de ces trois méthode que vous utilisiez pour enregistrer un step definion , Cucumber ignore le keyword lorsqu'il fait correspondre un Step. Sous le capot, toutes les annotations sont des alias pour StepDefAnnotation.
Les annotations ne sont là que pour la documentation supplémentaire qui vous aidera à exprimer l'intention de chaque step ou step definition.
Cela signifie que, qu'elle ait été créée avec les méthodes @Given, @When ou @Then, une définition d'étape correspondra à n'importe quelle étape de Gherkin tant que l'expression régulière correspond au texte principal du Step. Cette figure met en évidence ce que Cucumber voit lorsqu'il analyse un scénario pour rechercher des définitions des Steps definition .
Cette flexibilité peut être très pratique, mais il faut se méfier de tous. Regardons un exemple.

```
Feature: Cash Withdrawal
  Scenario: Attempt withdrawal using stolen card
    Given I have $100 in my account
    But my card is invalid
    When I request $50
    Then my card should not be returned
```

And I should be told to contact the bank

Imaginez que vous ayez déjà implémenté votre scénario de retrait au guichet automatique, notamment en écrivant un step definition pour Given I have $100 in my account . Donc, vous avez une définition d'étape qui correspond au texte Given I have $100 in my account et crée un compte contenant 100 $. Quelques semaines plus tard, ce scénario est devenu vague , et vous obtenez une nouvelle obligation de faire un don de 1 $ à tous les nouveaux comptes. Vous asseyez avec votre expert de domaine et vous écrivez le scénario suivant:

Scenario: New accounts get a $1 gift
Given I have a brand new Account
And I deposit $99
Then I have $100 in my Account

Cela semble raisonnable, n'est-ce pas ? Nous ouvrons le nouveau compte, déposons de l'argent, puis vérifions que le nouveau solde correspond à ce que nous attendons. Mais pouvez-vous voir ce qui se passera si nous exécutons ce nouveau scénario avec notre scénario de retrait DAB original ? Regardant l'ancien test de DAB pour le retrait

```
@Given("I have \\$100 in my Account")
void iHave$100InMyAccount() {
  // TODO: code that puts $100 into User's Account goes here
}
```

Maintenant que nous avons appris que Cucumber ignore l'annotation @Given/ @When / @Then lors de la mise en correspondance d'une étape, nous pouvons voir que cette définition d'origine de l'étape va également correspondre à la dernière étape de notre nouveau scénario. Ensuite, j'ai 100 $ dans mon compte. Surprise! Nous nous attendions à ce que cette étape vérifie le solde du compte, mais au lieu de cela il va mettre 100 $ sur le compte! Nous devons évidemment faire attention dans cette situation, car nous aurions facilement pu avoir un scénario qui nous donnait un faux positif: passer quand ça devrait ont échoué. Cela ne semble peut-être pas être le cas, mais la souplesse de Cucumber nous a réellement aidés ici en exposant une ambiguïté assez subtile dans le langage utilisé dans chacune des étapes. Pour éviter ce genre de problème, le meilleur moyen que nous ayons trouvé est de porter une attention particulière au libellé précis dans vos étapes. Vous pouvez changer les deux étapes pour être moins ambigu:

Given I have deposited $100 in my Account
Then the balance of my Account should be $100

En reformulant(rewording) les étapes de cette manière, vous leur avez permis de mieux communiquer exactement ce qu'elles feront lorsqu'elles seront exécutées. Apprendre à détecter et à supprimer ce type d'ambiguïté est quelque chose qui nécessite de la pratique. Faire attention à la distinction de formulation entre deux étapes comme celle-ci peut également vous donner des indices sur des concepts qui ne sont peut-être pas exprimés dans votre code, mais qui doivent l'être. Cela peut sembler pédant, mais les équipes qui accordent une telle attention aux détails écrivent des logiciels bien meilleurs, plus rapidement.

Traiter les arguments

Vous remarquerez que dans l'étape que nous avons utilisée à titre d'exemple, nous avons parlé de la somme de 100 $ tout le temps. Et si nous avions un autre scénario dans lequel nous devions déposer un montant différent sur le compte? Aurions-nous besoin d'une autre définition d'étape, comme celle-ci?

```
@Given("I have deposited \\$100 in my Account")
public void iHaveDeposited$100InMyAccount() {
  // TODO: code goes here
}
@Given("I have deposited \\$250 in my Account")
public void iHaveDeposited$250InMyAccount() {
  // TODO: code goes here
}
```

Heureusement que nous ne le faisons pas. C'est ici qu'intervient la flexibilité des expressions régulières. Nous pouvons utiliser ici deux des fonctionnalités les plus utiles des expressions régulières pour capturer n'importe quel montant en tant qu'argument de la step definition. Ces fonctionnalités sont des groupes de capture et des caractères génériques(windcards).

Groupes de capture

Lorsque vous entourez une partie d'une expression régulière avec des parenthèses, elle devient un groupe de capture. Les groupes de capture sont utilisés pour mettre en évidence des parties particulières d'un motif que vous souhaitez extraire du texte correspondant et que vous souhaitez utiliser. Dans une définition d'étape Cucumber, le texte correspondant à chaque groupe de capture est transmis au bloc de code en tant qu'argument

```
@Given("I have deposited \\$(100) in my Account")
public void iHaveDeposited$100InMyAccount(int amount) {
  // TODO: code goes here
}
```

Ici, le montant de l'argument de méthode recevra la valeur de chaîne 100 lorsque cette définition d'étape correspond. L'exemple précédent est un peu ridicule, car cette expression régulière ne correspondra toujours qu'aux étapes mentionnant le montant de 100 $. Nous devons utiliser un caractère générique à l'intérieur du groupe de capture pour l'ouvrir à d'autres valeurs.

Alternation

Nous pouvons spécifier un caractère générique dans une expression régulière en utilisant différentes approches. L'un des plus simples est l'alternance, où nous exprimons différentes options séparées par un caractère de pipe |, comme ceci:

```
@Given("I have deposited \\$(100|250) in my Account")
public void iHaveDeposited$InMyAccount(int amount) {
  // TODO: code goes here
}
```

Cette Step definition va maintenant faire correspondre une étape avec l'une des deux valeurs 100 ou 250, et le nombre sera capturé et transmis à la méthode en tant qu'argument. L'alternance peut être utile si vous souhaitez accepter un ensemble fixe de valeurs dans votre définition d'étape, mais vous voudrez normalement quelque chose d'un peu plus lâche.

Le Point

Le point est un métacaractère, ce qui signifie qu'il possède des pouvoirs magiques dans une expression régulière. Littéralement, un point signifie correspondre à n'importe quel caractère. Nous pouvons donc essayer ceci à la place :

```
@Given("I have deposited \\$(...) in my Account")
public void iHaveDeposited$InMyAccount(int amount) {
  // TODO: code goes here
}
```

Cela va maintenant faire correspondre une étape à une somme en dollars à trois chiffres et envoyer le montant correspondant dans la méthode. C'est un pas dans la bonne direction, mais ce que nous avons fait pose quelques problèmes. D'une part, rappelez-vous que le point correspond à n'importe quel caractère, nous pourrions donc capturer des lettres ici au lieu de chiffres. Plus important encore, si nous voulions une étape qui ne dépose que 10 $ dans le compte, ou 1 000 $? Cette définition d'étape ne correspond pas à ces étapes car elle recherche toujours trois caractères. Nous pouvons résoudre ce problème en utilisant un modificateur.

Le modificateur étoile :

Dans les expressions régulières, un modificateur de répétition prend un caractère (ou métacaractère) et nous dit combien de fois il peut apparaître. Le modificateur le plus flexible est l'étoile:

```
@Given("I have deposited \\$(.*) in my Account")
public void iHaveDeposited$InMyAccount(int amount) {
   // TODO: code goes here
}
```

Le modificateur étoile signifie n'importe quel nombre de fois. Donc, avec. *, Nous capturons n'importe quel caractère, autant de fois que nécessaire. Maintenant, nous allons quelque part - cela nous permettra de capturer toutes ces quantités différentes. Mais il reste un problème.
Le modificateur étoile est un peu un instrument contondant. Comme nous l'utilisons avec le point correspondant à n'importe quel caractère, il engloutira tout texte jusqu'à la phrase de mon compte. C'est pourquoi, dans la terminologie des expressions , le modificateur étoile est appelé opérateur glouton. Par exemple, cela correspondrait avec bonheur à cette étape

Given I have deposited $1 and a cucumber in my Account

La quantité capturée par notre expression régulière dans ce cas serait 1 et un Cucumber. Nous devons être plus précis sur les caractères que nous voulons associer et capturer simplement des chiffres. Au lieu d'un point, nous pouvons utiliser autre chose.

Classes de caractère

Les classes de caractères vous permettent d'indiquer au moteur des expressions régulières une correspondance avec l'un des caractères. Vous placez simplement tous les caractères que vous accepteriez entre crochets :

```
@Given("I have deposited \\$([01234567890]*) in my Account")
public void iHaveDeposited$InMyAccount(int amount) {
  // TODO: code goes here
}
```

Pour une gamme continue de caractères comme ceux que nous avons, vous pouvez utiliser un trait d'union :

```
@Given("I have deposited \\$([0-9]*) in my Account")
public void iHaveDeposited$InMyAccount(int amount) {
   // TODO: code goes here
}
```

Nous avons maintenant limité le caractère que nous accepterons d'être numérique. Nous modifions toujours le caractère avec l'étoile pour en accepter un nombre quelconque, mais nous précisons maintenant que nous n'accepterons qu'une chaîne continue de nombres.

Classes abrégées

Pour les modèles de caractères courants tels que [0-9], vous pouvez utiliser quelques classes de caractères abrégées. Vous constaterez peut-être que cela rend simplement vos expressions régulières plus cryptiques, mais il n'ya que quelques-unes à apprendre. Pour un chiffre, vous pouvez utiliser \ d comme raccourci pour [0-9]:

```
@Given("I have deposited \\$(\d*) in my Account")
public void iHaveDeposited$InMyAccount(int amount) {
   // TODO: code goes here
}
```

Voila la liste des classes abrégées les plus utilisées :

\d : (digital) représente le chiffre ou [0-9]
\D : signifie tout caractère sauf un chiffre
\w : (word character) représente le caractère du mot, plus précisément [A-Za-z0-9_]. Notez que les traits de soulignement et les chiffres sont inclus, mais pas les traits d'union.
\s : (whitespace) représente un espace, en particulier [\ t \ r \ n]. Cela signifie un espace, une tabulation ou un saut de ligne.
\b : (word boundary) représente la limite de mot, ce qui ressemble beaucoup à \s mais signifie en réalité le contraire de \ w. Tout ce qui n'est pas un caractère de mot est une limite de mot comme un espace blanc un retour à la ligne (exp \b123\b == 123 123 123123)

Les quantificateurs les plus répandus sont

? : [0,1] qui définit un groupe qui existe zéro ou une fois : toto? correspondant alors à « tot » ou « toto » mais pas « totoo » ;
* : [0 , *] qui définit un groupe qui existe zéro ou plusieurs fois (l'étoile de Kleene) : toto* correspondant à « tot », « toto », « totoo », « totooo », etc. ;
+ : [1, *] qui définit un groupe qui existe une ou plusieurs fois : toto+ correspondant à « toto », « totoo », « totooo », etc. mais pas « tot ».

Vous pouvez également annuler les classes de caractères abrégées en les mettant en majuscule. Ainsi, par exemple, \D signifie tout caractère sauf un chiffre.
Retour à la correspondance de notre montant. On dirait que nous avons terminé, mais il reste un dernier problème à résoudre. Pouvez-vous voir ce que c'est?

Le modificateur plus +

L'étoile est un exemple de modificateur de répétition, mais il y en a d'autres. Un problème subtil avec l'étoile est que n'importe quel nombre de fois peut signifier zéro.
Donc, cette étape correspondrait à:

Given I have deposited $ in my Account

Ce n'est pas bon. Pour résoudre ce problème, nous pouvons utiliser le modificateur +, ce qui signifie au moins un , dans l'exemple la chaine de caractere est filtrer par le \d pour récupérer que les nobre et puis par + qui précise qu'il faut récupérer au moins un nombre

```
@Given("I have deposited \\$(\\d+) in my Account")
public void iHaveDeposited$InMyAccount(int amount) {
  // TODO: code goes here
}
```

Petit exercice
Imaginez que vous construisiez un système pour les écrans de la salle d'embarquement de l'aéroport. Vous devez être capable de capturer des exemples de codes de vol à partir des scénarios de Cucumber.
Pouvez-vous écrire une définition en une seule étape pouvant capturer les codes de vol de toutes ces étapes ?
Given the flight EZY4567 is leaving today
...
Given the flight C038 is leaving today
...
Given a flight BA01618 is leaving today

Commencez par écrire une définition d'étape qui fonctionne pour la première étape, puis rendez-la de plus en plus générique afin qu'elle fonctionne également avec les autres étapes.

Captures Multiple

Vous ne devez pas vous contenter de capturer un seul argument. Concombre transmettra un argument à votre méthode pour chaque groupe de capture de votre expression régulière, afin que vous puissiez saisir autant de détails que vous le souhaitez à partir d'une étape.
Voici un exemple. Imaginons que notre banque veuille offrir à ses clients des comptes d'épargne ainsi que leur compte courant. Les clients peuvent utiliser le guichet automatique pour transférer de l'argent entre leurs comptes. Voici l'un des scénarios pour cette nouvelle fonctionnalité :

```
Scenario: Transfer funds from savings into checking account
  Given I have deposited $10 in my Checking Account
  And I have deposited $500 in my Savings Account
  When I transfer $500 from my Savings Account into my Checking Account
  Then the balance of the Checking Account should be $510
  And the balance of the Savings Account should be $0
```

Essayons d'écrire une définition d'étape capable de gérer les deux premières étapes. En plus du montant déposé, nous devons saisir le type de compte afin de savoir où le placer.
Nous pouvons utiliser une version modifiée de l'expression régulière utilisée précédemment pour capturer le type de compte :

```
@Given("I have deposited \\$(\\d+) in my (\\w+) Account")
```

```
public void iHaveDeposited$InMyAccount(int amount, String accountType) {
   // TODO: code goes here
}
```

Nous utilisons la classe de caractères réduite(shorthand) \ w, modifiée avec le signe plus pour désigner tout caractère du mot, au moins une fois, capturant efficacement un seul mot. Ce mot est ensuite transmis à la méthode nommée accountType dans le deuxième argument.

Flexibilité

La lisibilité des fonctionnalités de concombre aide les équipes à apprendre à utiliser un langage omniprésent lorsqu'ils parlent du système qu'ils construisent. Il s'agit d'un avantage vraiment important, car l'utilisation cohérente de la terminologie contribue à réduire les malentendus et à permettre une communication plus fluide entre tous les membres de l'équipe.
Nous souhaitons donc encourager nos auteurs à être cohérents sur les noms et les verbes qu'ils utilisent dans les fonctions Cucumber, car ils permettent de créer des fonctions facilement compréhensibles par tous les membres de l'équipe. De même, nous souhaitons également que les auteurs de fonctionnalités puissent s'exprimer aussi naturellement que possible, ce qui signifie qu'ils peuvent souvent utiliser une formulation légèrement différente pour signifier exactement la même chose. Les fonctions de concombre concernent la communication avec les gens du metier dans leur langue et il est important de ne pas les forcer à ressembler à des robots. Pour que les fonctionnalités restent lisibles et naturelles, il est utile de développer les compétences nécessaires pour rendre vos définitions d'étape suffisamment souples pour correspondre aux différentes manières dont une personne de metier peut exprimer quelque chose.

Le modificateur de point d'interrogation ?

Lorsque vous faites correspondre un texte Gherkin faisant face à la logique metier, vous voudrez souvent indiquer que vous ne vous souciez pas du caractère impair de votre correspondance, par exemple lorsqu'un mot peut être singulier ou pluriel :

```
Given I have 1 cucumber in my basket
Given I have 256 cucumbers in my basket
```

Comme l'étoile et le plus, le point d'interrogation modifie le caractère qui le précède, en précisant le nombre de fois où il peut être répété. Le modificateur de point d'interrogation signifie zéro ou une fois [0,1], autrement dit, cela rend le caractère précédent facultatif. Dans les définitions d'étape, il est particulièrement utile(useful) pour les pluriels :

```
@Given("I have (\\d+) cucumbers? in my basket")
public void iHaveCucumbersInMyBasket(int number) {
   // TODO: code goes here
}
```

En mettant un point d'interrogation après le symbole s dans les concombres, nous disons que nous ne nous soucions pas de savoir si le mot est au singulier ou au pluriel. Donc, cette définition d'étape correspondra aux deux étapes précédentes.
Une autre technique utile consiste à utiliser un Noncapturing Groups.

Noncapturing Groups

Rappelez-vous dans Alternance, nous avons montré comment répertorier un ensemble de valeurs possibles pour une partie d'une expression régulière, séparées par un symbole de pipe (|).
Nous pouvons utiliser cette même technique pour ajouter de la flexibilité à nos steps definition, en laissant les auteurs metier dire la même chose de manière légèrement différente. Nous devons apporter un petit changement, mais nous y arriverons dans une minute.
Prenez cette étape extrêmement courante pour une application Web :

When I visit the homepage

Supposons que quelqu'un arrive et écrit une autre étape qui ressemble à ceci

When I go to the homepage

Pour le lecteur, ces deux steps ont la même signification, mais malheureusement, un step definition pour la première ne correspondra pas à la seconde sans quelques modifications. Il serait utile de définir les étapes à suivre pour reconnaître les deux expressions, car peu importe que vous disiez visiter ou aller à, elles signifient la même chose. Nous pouvons utiliser un autre moyen d'assouplir la définition de l'étape pour accepter cette formulation légèrement différente :

```
@When("I (?:visit|go to) the homepage")
public void iVisitTheHomepage() {
    // TODO: code goes here
}
```

Notez que nous avons dû préfixer la liste des alternatives avec un autre peu de magie d'expression régulière. Le ?: Au début du groupe le marque comme non capturant, ce qui signifie que Cucumber ne le transmettra pas comme argument à notre bloc.
Pour plus d'explication dans mon exemple j'ai pas besoin des caractères entre i et the homepage , je dis au moteur de regex d'annuler le groupe de caractères avec (?:) que se soit [visit] ou [go to]

Les ancres ^ ,$ (Anchors)

Vous avez peut-être remarqué que les extraits des steps definition imprimés par Cucumber pour des steps non définies commencent par ^ et se terminent par $. Vous êtes peut-être devenu tellement habitué à les voir que vous avez cessé de les remarquer. Ces deux métacaractères sont appelés ancres, car ils servent à rattacher chaque extrémité de l'expression régulière au début et à la fin d'une chaine de caractère chaîne sur lequel ils correspondent.
Vous n'êtes pas obligé de les utiliser et nous les avons délibérément laissés en dehors de l'exemple, car nous voulions attendre d'avoir expliqué ce qu'ils faisaient. Si vous en omettez un ou les deux, vous vous retrouverez avec une définition d'étape beaucoup plus flexible, peut-être trop flexible.
Comme exemple idiot, supposons que nous ajoutions l'ancre ^ au début, mais omettons-le $ à la fin de la définition de l'étape de notre compte bancaire :

```
@Given("^I have deposited \\$(\\d+) in my Account")
public void iHaveDeposited$InMyAccount(int amount) {
   // TODO: code goes here
}
```

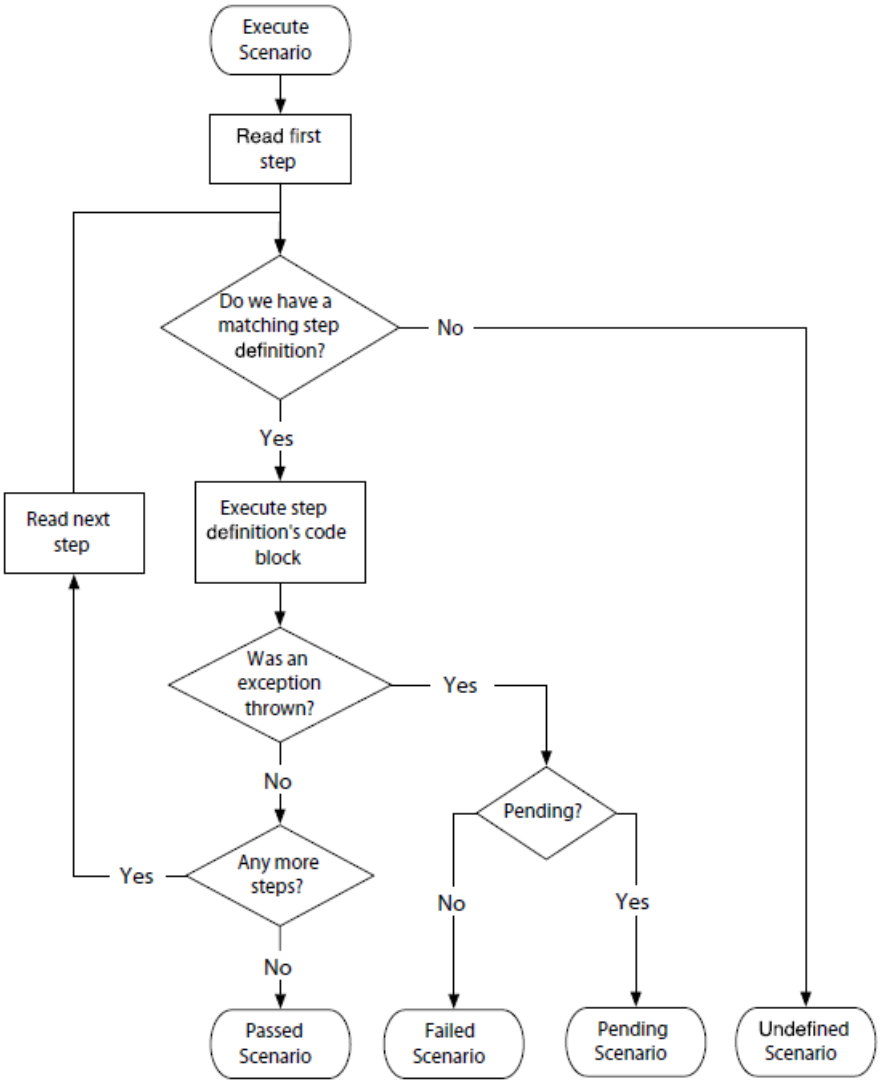
Cela permet à un auteur de fonctionnalité particulièrement créatif d'écrire quelque chose comme :

Given I have deposited $100 in my Account from a check my Grandma gave to me

Généralement , il est préférable de garder les expressions régulières aussi réduites que possible afin de réduire les risques de conflit entre les définitions en deux étapes. C'est pourquoi les extraits (snippets) générés par Cucumber pour des steps non définies incluent toujours les ancres. Néanmoins, laisser tomber les ancres est une astuce qu'elle est utile de savoir, qui peut parfois être utile.

Retourner les résultats

Cucumber est un outil de test. Il s'agit dans le code Java d'un step definition ou nos tests déterminent si un step a réussi dans tous les cas.
Alors, comment un step definition indique-t-il à Cucumber si il a réussi ou non ?



Comme la plupart des autres outils de test, Cucumber utilise des exceptions pour signaler l'échec d'un test. Lorsqu'il exécute un scénario étape par étape, Cucumber suppose qu'une étape est réussie à moins que sa définition ne lève une exception.
Si l'exception levée est une exception PendingException, l'étape est marquée comme étant en attente. Toutes les autres exceptions entraînent l'échec de l'étape. Si une étape est réussie, Cucumber passe à l'étape suivante.
Dans Concombre, les résultats sont un peu plus sophistiqués qu'un simple succès ou échec. Un scénario qui a été exécuté peut aboutir à l'un des états suivants :

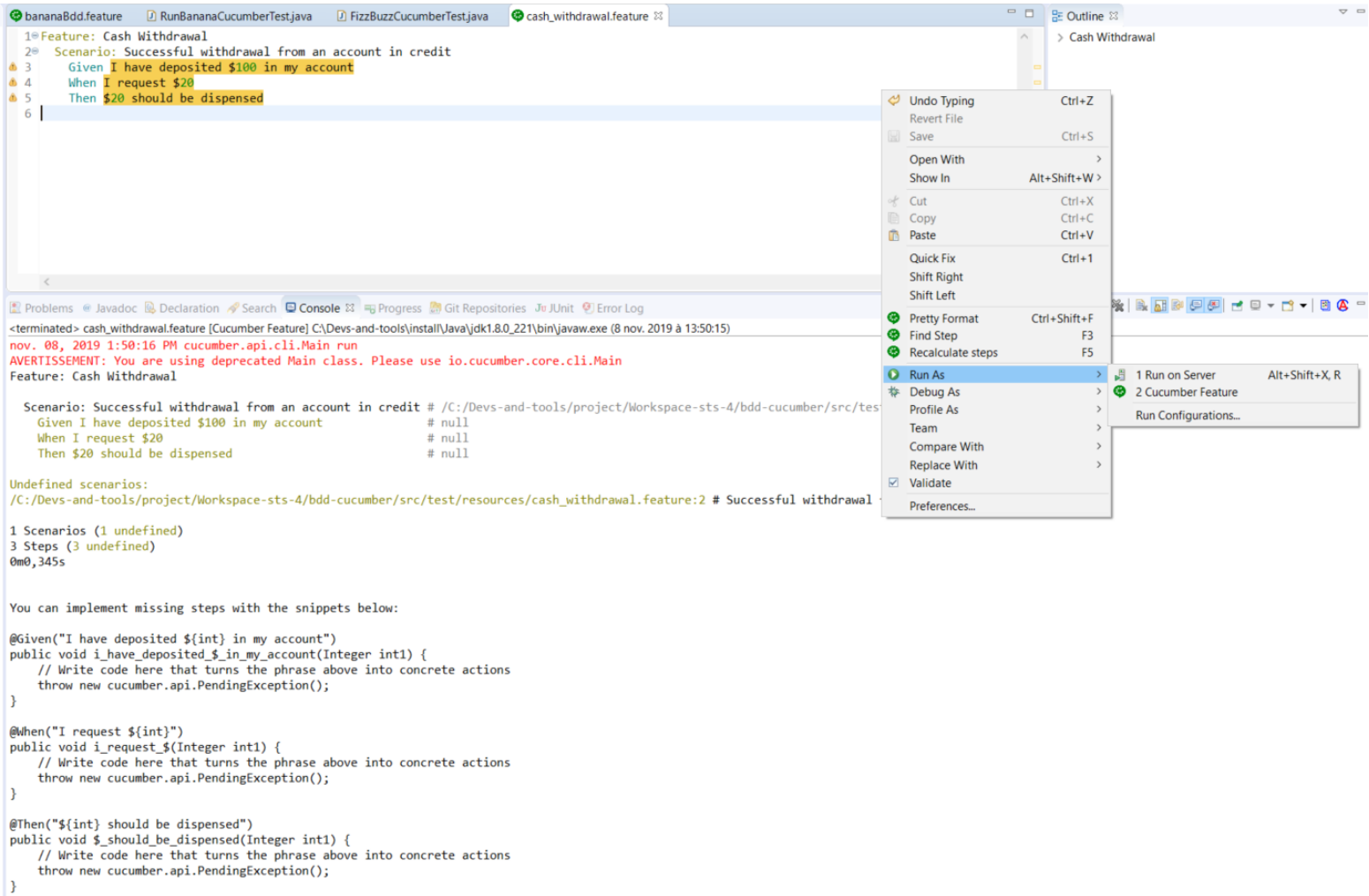
- Failed
- Pending
- Undefined
- Skipped
- Passed

Ces états sont conçus pour vous aider à indiquer les progrès que vous effectuez au cours du développement de vos tests. Examinons un exemple d'automatisation du scénario de retrait ATM pour illustrer notre propos.

Steps indéfinis

Lorsque Cucumber ne peut pas trouver une définition d'étape correspondant à une étape, il marque l'étape comme indéfinie (jaune) et arrête le scénario. Le reste des étapes du scénario sera ignoré ou marqué comme indéfini s'il n'a pas de définition d'étape correspondante lui-même.
Pour vous montrer comment cela fonctionne, nous pouvons exécuter notre scénario de retrait au guichet automatique. Créez un fichier appelé resources / cash_withdrawal.feature et insérez-y les éléments suivants:

Feature: Cash Withdrawal
  Scenario: Successful withdrawal from an account in credit
    Given I have deposited $100 in my account
    When I request $20
    Then $20 should be dispensed



Il faut voir Chaque étape est en jaune, ce qui indique qu'elle n'est ni défaillante (rouge) ni dépassée (vert) mais quelque part entre les deux. Notez également que Cucumber a imprimé un extrait de code pour chaque définition d'étape manquante. Nous pouvons les utiliser comme point de départ pour mettre en œuvre nos propres définitions d'étape.

Steps en attentes

Lorsque Cucumber découvre qu'une définition d'étape est en cours d'implémentation, elle est marquée comme étant en attente (jaune). Encore une fois, le scénario sera arrêté et le reste des étapes sera ignoré ou marqué comme indéfini.
Comment Cucumber sait-il si une définition d'étape a été implémentée ?
Bien sûr, vous devez le dire en lançant une exception PendingException.
Lorsque vous générez une exception PendingException à partir d'une définition d'étape, cela indique au moteur d'exécution de Cucumber que l'étape a échoué, mais d'une manière particulière : la définition de l'étape est toujours en cours de travail. Vous aurez probablement remarqué que les extraits générés par Cucumber pour des étapes non définies génèrent une exception PendingException, alors vous comprenez pourquoi.
Revenons à notre scénario de retrait au guichet automatique pour vous montrer ce que nous voulons dire.
Créez un fichier appelé step_definitions / Steps.java et collez-le dans cette définition d'étape :

```
package com.zaghir.project.hellocucumber.stepdefinition;

import cucumber.api.PendingException;
import io.cucumber.java.en.Given;

public class Steps {

	@Given("^I have deposited \\$(\\d+) in my account$")
	public void iHaveDeposited$InMyAccount(int amount) throws Throwable {
	// Write code here that turns the phrase above into concrete actions
	throw new PendingException();
	}

}
```

Maintenant, lorsque nous exécutons mvn clean test, nous verrons qu'il a seulement essayé d'exécuter la première étape et l'a marquée comme étant en attente. Tout le reste n'est pas encore défini :

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running RunCukesTest
Feature: Cash Withdrawal

  Scenario: Successful withdrawal from an account in credit
    Given I have deposited $100 in my account
      cucumber.api.PendingException: TODO: implement me
        at nicebank.Steps.iHaveDeposited$InMyAccount(Steps.java:11)
        at *.Given I have deposited $100 in my account(cash_withdrawal.feature:3)
    When I request $20
    Then $20 should be dispensed
1 Scenarios (1 undefined)
3 Steps (1 pending, 2 undefined)
```

0m0.121s
cucumber.api.PendingException: TODO: implement me
    at nicebank.Steps.iHaveDeposited$InMyAccount(Steps.java:11)
    at *.Given I have deposited $100 in my account(cash_withdrawal.feature:3)

Le statut en attente est un peu comme les panneaux en construction que vous voyiez sur Internet dans les années 90. Vous pouvez l'utiliser comme indicateur temporaire pour vos coéquipiers que vous êtes en train de travailler sur quelque chose.
Lorsque nous développons un nouveau scénario de l'extérieur comme celui-ci, nous avons tendance à travailler sur le même niveau avant de passer au suivant. Pour le moment, nous nous concentrons sur l'ajout de définitions d'étape. Nous allons donc le faire pour chacune des autres étapes et ajouter un appel en attente à chacune d'elles :

```
package com.zaghir.project.hellocucumber.stepdefinition;

import cucumber.api.PendingException;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;

public class Steps {

	@Given("^I have deposited \\$(\\d+) in my account$")
	public void iHaveDeposited$InMyAccount(int amount) throws Throwable {
		// Write code here that turns the phrase above into concrete actions
		throw new PendingException();
	}

	@When("^I request \\$(\\d+)$")
	public void iRequest$(int arg1) throws Throwable {
		// Write code here that turns the phrase above into concrete actions
		throw new PendingException();
	}

	@Then("^\\$(\\d+) should be dispensed$")
	public void $ShouldBeDispensed(int arg1) throws Throwable {
		// Write code here that turns the phrase above into concrete actions
		throw new PendingException();
	}

}
```

En faisant cela, nous pouvons vérifier si nous pouvons utiliser des définitions d'étape existantes et, dans le cas contraire, en créer une nouvelle qui lève une exception PendingException.
Les scénarios en attente deviennent notre liste de tâches à effectuer pour le travail que nous effectuerons lorsque nous passerons à la couche suivante et que nous commencerons à implémenter les définitions d'étape.

Steps en échecs

Si le bloc de code exécuté par une définition d'étape déclenche une exception, Le concombre marquera cette étape comme ayant échoué (rouge) et arrêtera le scénario. Le reste des étapes du scénario sera ignoré.
En pratique, une définition d'étape **échouera** pour l'une des raisons suivantes :
• Le scénario ne peut pas se terminer car vous avez un bug dans votre code de définition d'étape, ou dans le système testé, qui a provoqué une erreur.

Vous serez habitué à voir ces échecs tout le temps pendant le développement si vous utilisez Cucumber pour piloter votre développement de l'extérieur. Chaque message d'échec vous indique ce que vous devez faire ensuite.
• La définition de l'étape a utilisé une assertion pour vérifier quelque chose sur l'état du système et la vérification n'a pas abouti. Vous obtiendrez généralement ces erreurs à la fin de votre cycle d'entrée-en-dehors ou longtemps après la mise en œuvre de la fonctionnalité si quelqu'un introduit accidentellement un bogue.

Une assertion est une vérification dans vos tests qui décrit une condition à laquelle vous vous attendez à être satisfaite. Les échecs dus aux assertions ont tendance à se produire lors des étapes Then, dont le travail consiste à vérifier l'état de l'état du système. Ce sont les sonnettes d'alarme que vous adaptez au système et qui se déclenchent s'il commence à se comporter de manière inattendue. De toute façon, Cucumber vous montrera le message d'exception et la trace de retour dans sa sortie, il vous incombe donc d'enquêter.
Dans la classe Steps.java

```
@Given("^I have deposited \\$(\\d+) in my account$")
public void iHaveDeposited$InMyAccount(int amount) throws Throwable {
  new Account(amount);
}
```

Que va-t-il se passer quand nous exécutons ca ? Nous n'avons pas encore de classe Account, elle échouera lors de la compilation. Créons donc une classe Account dans notre fichier Steps.java. Notez que nous définissons la classe ici même dans notre fichier steps. Ne vous inquiétez pas, il ne restera pas ici éternellement, mais c'est plus pratique pour nous de le créer ici, là où nous travaillons. Une fois que nous avons une idée claire de la façon dont nous allons travailler avec la classe, nous pouvons ensuite la refactoriser et la transférer dans un endroit plus permanent.

Dans la classe Steps.java

```
class Account {
  public Account(int openingBalance) {
  }
}
```

Maintenant, lorsque nous exécutons mvn clean test, nous obtenons l'échec suivant

```
-------------------------------------------------------
  T E S T S
-------------------------------------------------------
Running RunCukesTest
Feature: Cash Withdrawal

  Scenario: Successful withdrawal from an account in credit
    Given I have deposited $100 in my account
    When I request $20
      cucumber.api.PendingException: TODO: implement me
        at nicebank.Steps.iRequest$(Steps.java:24)
        at *.When I request $20(cash_withdrawal.feature:4)
    Then $20 should be dispensed
1 Scenarios (1 pending)
3 Steps (1 skipped, 1 pending, 1 passed)
0m0.100s
```

```
cucumber.api.PendingException: TODO: implement me
   at nicebank.Steps.iRequest$(Steps.java:24)
   at *.When I request $20(cash_withdrawal.feature:4)
Tests run: 5, Failures: 0, Errors: 0, Skipped: 3, Time elapsed: 0.739 sec
```

Comme vous l'avez vu, notre définition de première étape est en train de réussir. Le concombre a capturé l'exception PendingException émise par la définition de la deuxième étape et nous l'a affichée juste sous l'étape. Dans le résumé en bas, vous pouvez voir qu'une étape a été franchie, une étape a échoué et une autre a été ignorée.

Un petit résumé :

Il est utile de penser à une step definition comme étant une méthode particulière.
Contrairement à une méthode régulière, dont le nom doit correspondre exactement, une définition d'étape peut être invoquée par toute étape correspondant à son expression régulière. Étant donné que les expressions régulières peuvent contenir des caractères génériques, vous avez ainsi la possibilité de rendre les étapes Gherkin agréables et lisibles, tout en maintenant votre code de définition d'étape Java propre et sans duplication.
• Les définitions d'étape fournissent un mappage à partir des descriptions en langage clair des actions de l'utilisateur par les scénarios Gherkin en code Java, qui simule ces actions.
• Les définitions d'étape sont enregistrées avec Cucumber en utilisant @Given, @When, @Then ou l'un des alias de votre langue parlée.
• Les définitions d'étape utilisent des expressions régulières pour déclarer les étapes qu'elles peuvent gérer. Les expressions régulières pouvant contenir des caractères génériques, une définition à une étape peut gérer plusieurs étapes différentes.
• Une définition d'étape communique son résultat à Cucumber en soulevant ou en notant une exception.

Scénarios expressifs

Lorsque vous écrivez les fonctionnalités Cucumber, faites de la lisibilité votre objectif principal. Sinon, un lecteur peut facilement avoir l'impression de lire un programme informatique plutôt qu'un document de spécification, ce que nous souhaitons que vous essayiez d'éviter à tout prix. Après tout, si les non-programmeurs ont du mal à lire vos fonctions, vous pouvez aussi bien écrire vos tests dans un code ancien et clair.
La véritable clé des scénarios expressifs consiste à avoir un vocabulaire sain du langage de domaine à utiliser pour exprimer vos besoins. Cela dit, l'utilisation de l'ensemble des mots clés Gherkin de base peut souvent rendre vos fonctionnalités répétitives, ce qui les rend encombrées et difficiles à lire.Nous allons également vous montrer comment utiliser les balises et les dossiers pour rester organisé au fur et à mesure que vous écrivez plus de fonctionnalités pour votre projet.
Premièrement, nous souhaitons vous aider à éliminer ce fouillis répétitif.
Nous allons vous montrer comment utiliser les contours de scénarios et les tableaux de données pour rendre vos scénarios Gherkin plus lisibles, mais nous commencerons par un nouveau mot-clé appelé Background.

Background

Une section d'arrière-plan dans un fichier de fonctions vous permet de spécifier un ensemble d'étapes communes à chaque scénario du fichier. Au lieu d'avoir à répéter ces étapes pour chaque scénario, vous les déplacez vers le haut dans un élément d'arrière-plan. Voici quelques avantages à cela :
• Si vous avez besoin de changer ces étapes, vous devez les changés à un seul endroit.

• L'importance de ces étapes s'estompe dans l'arrière-plan. Ainsi, lorsque vous lisez chaque scénario, vous pouvez vous concentrer sur ce qui est unique et important dans ce scénario.

Pour vous montrer ce que nous voulons dire, prenons un scénario existant qui utilise uniquement l'élément de base Scénario Gherkin et améliorons sa lisibilité en le remaniant pour utiliser un arrière-plan.

Voici notre article avant le début de la refactorisation :

```
Feature: Change PIN
  Customers being issued new cards are supplied with a Personal
  Identification Number (PIN) that is randomly generated by the system.
  In order to be able to change it to something they can easily remember, customers with new bank
  cards need to be able to change their PIN using the ATM.
  Scenario: Change PIN successfully
    Given I have been issued a new card
    And I insert the card, entering the correct PIN
    When I choose "Change PIN" from the menu
    And I change the PIN to 9876
    Then the system should remember my PIN is now 9876

  Scenario: Try to change PIN to the same as before
    Given I have been issued a new card
    And I insert the card, entering the correct PIN
    When I choose "Change PIN" from the menu
    And I try to change the PIN to the original PIN number
    Then I should see a warning message
    And the system should not have changed my PIN
```

Vous pouvez voir qu'il existe deux scénarios, mais sans les lire attentivement, il est assez difficile de voir exactement ce qui se passe dans chacun d'eux. Les trois premières étapes de chaque scénario, bien que nécessaires pour clarifier le contexte du scénario, sont complètement répétées dans les deux scénarios. Cette répétition est distrayante, ce qui rend plus difficile de comprendre l'essence de ce que chaque scénario teste.

Factorisons les trois étapes répétées dans un background, comme ceci :

```
Feature: Change PIN
  As soon as the bank issues new cards to customers, they are supplied with a Personal Identification
  Number (PIN) that is randomly generated by the system.
  In order to be able to change it to something they can easily remember, customers with new bank
  cards need to be able to change their PIN using the ATM.

  Background:
    Given I have been issued a new card
    And I insert the card, entering the correct PIN
    And I choose "Change PIN" from the menu

  Scenario: Change PIN successfully
    When I change the PIN to 9876
    Then the system should remember my PIN is now 9876

  Scenario: Try to change PIN to the same as before
    When I try to change the PIN to the original PIN number
    Then I should see a warning message
    And the system should not have changed my PIN
```

Notre refactoring n'a pas du tout changé le comportement des tests : au moment de l'exécution, les étapes en arrière-plan sont exécutées au début de chaque scénario, comme auparavant. Ce que nous avons fait est de rendre chaque scénario beaucoup plus facile à lire.
Vous pouvez avoir un seul élément Background par fichier de fonction et il doit apparaître avant tout élément Scenario ou Scenario Outline. Comme tous les autres éléments de Gherkin, vous pouvez lui attribuer un nom et disposer d'un espace pour placer une description multiligne avant la première étape. Par exemple :

Feature: Change PIN
In order to be able to change it to something they can easily remember, customers with new bank cards need to be able to change their PIN using the ATM.

Background: Insert a newly issued card and sign in
Whenever the bank issues new cards to customers, they are supplied with a Personal Identification Number (PIN) that is randomly generated by the system.
Given I have been issued a new card
And I insert the card, entering the correct PIN

L'utilisation d'un élément background n'est pas toujours nécessaire, mais il est souvent utile d'améliorer la lisibilité de vos fonctionnalités en supprimant les étapes répétitives de scénarios individuels. Voici quelques conseils pour bien l'utilisé :
• N'utilisez pas l'arrière-plan pour configurer un état compliqué, à moins que le lecteur ait besoin de savoir cet état. Par exemple, nous n'avons pas mentionné les chiffres réels du code confidentiel généré par le système dans l'exemple précédent, car ces détails n'étaient pertinents pour aucun des scénarios.
• Gardez votre section Background courte. Après tout, vous attendez de l'utilisateur qu'il se souvienne de ces informations lors de la lecture de vos scénarios. Si l'arrière-plan compte plus de quatre lignes, pouvez-vous trouver un moyen d'exprimer cette action en une ou deux étapes seulement ?
• Rendez votre section Background vive. Utilisez des noms colorés et essayez de raconter une histoire, car vos lecteurs peuvent garder une trace de leur histoire beaucoup mieux qu'ils ne le font pour des noms ennuyeux tels qu'Utilisateur A, Utilisateur B, Site 1, etc. Si cela vaut la peine de le mentionner, faites-le vraiment ressortir.
• Gardez vos scénarios courts et n'en avez pas trop. Si l'arrière-plan comporte plus de trois ou quatre étapes, envisagez d'utiliser des étapes de niveau supérieur ou de scinder le fichier de caractéristiques en deux. Vous pouvez utiliser un arrière-plan comme un bon indicateur du moment où une fonctionnalité prend trop de temps: si les nouveaux scénarios que vous souhaitez ajouter ne correspondent pas à l'arrière-plan existant, envisagez de scinder la fonctionnalité.
• Évitez de placer des détails techniques tels que le nettoyage des files d'attente, le démarrage des services principaux ou l'ouverture de navigateurs en arrière-plan. Le lecteur assumera la plupart de ces choses, et il existe des moyens d'insérer ces actions dans votre code de support
Les arrière-plans sont utiles pour effectuer des étapes données (et parfois quand) qui sont répétées dans chaque scénario et les déplacer à un seul endroit. Cela aide à garder vos scénarios clairs et concis.

Data Tables

Parfois, les steps d'un scénario doivent décrire des données qui ne s'intègrent pas facilement sur une seule ligne de données, GIVEN , WHEN , ou THEN . Gherkin nous permet de placer ces détails dans un tableau juste en dessous d'une marche. Les tableaux de données vous permettent d'étendre un Step de Gherkin au-delà d'une ligne afin d'inclure une donnée plus volumineuse.
Given a User *"Michael Jackson"* born on August 29, 1958
And a User *"Elvis"* born on January 8, 1935

And a User *"John Lennon"* born on October 9, 1940

Factoring and Background

Le refactoring est le processus de modification du code pour améliorer sa lisibilité ou sa conception sans modifier son comportement. Cette technique s'applique aussi bien aux fonctionnalités de Gherkin qu'au reste de votre code. Au fur et à mesure que votre compréhension de votre domaine se développe au cours du projet, vous souhaiterez refléter cet apprentissage en mettant à jour vos fonctionnalités.
Souvent, vous ne voyez pas un fond immédiatement. Vous pouvez commencer par écrire un ou deux scénarios, et c'est uniquement lorsque vous écrivez le troisième que vous remarquerez certaines étapes courantes. Lorsque vous repérez une fonctionnalité où les mêmes étapes ou des étapes similaires sont répétées dans plusieurs scénarios, voyez si vous pouvez effectuer un refactoring pour extraire ces étapes dans un arrière-plan.
Cela peut prendre un peu de courage, car il existe un risque de commettre une erreur et de casser quelque chose, mais c'est un refactoring assez sûr. Une fois que vous avez terminé, vous devriez vous retrouver avec la fonctionnalité faisant exactement la même chose qu'avant, mais plus facile à lire.

Ennuyeux ! Nous ne tolérerions pas ce genre de choses répétitives dans un document de spécification, et nous n'avons pas non plus à le tolérer dans une spécification Cucumber. Nous pouvons rassembler ces étapes en une seule étape qui utilise une table pour exprimer les données:

Given these Users:

```
  | name              | date of birth      |
  | Michael Jackson | August 29, 1958 |
  | Elvis               | January 8, 1935  |
  | John Lennon       | October 9, 1940  |
```

C'est beaucoup plus clair. Le tableau commence sur la ligne qui suit immédiatement l'étape et ses cellules sont séparées à l'aide du caractère pipe: |. Vous pouvez aligner les tuyaux en utilisant des espaces pour que le tableau soit propre, bien que Cucumber ne le dérange pas. il supprimera les valeurs dans chaque cellule, en ignorant les espaces blancs environnants.
Dans le tableau précédent, nous avons utilisé un en-tête pour chaque colonne du tableau, mais c'est uniquement parce que cela a du sens pour cette étape particulière. Vous avez la liberté de spécifier des données de différentes manières, par exemple en plaçant les en-têtes sur le côté :

Then I should see a vehicle that matches the following description:

```
  | Wheels       | 2                        |
  | Max Speed   | 60 mph                  |
  | Accessories  | lights, shopping basket |
```

Ou Simplement specifié une liste

Then my shopping list should contain:

```
  | Onions |
  | Potatoes |
  | Sausages |
  | Apples |
  | Relish |
```

Pour expliquer comment utiliser ces différentes tables de formes, nous devons plonger brièvement dans la couche steps definition.

Utiliser les data tables dans les steps definition

Nous allons expliquer comment utiliser une data table dans vos steps definition avec un jeu rapide de tic-tac-toe. Imaginons que nous construisons un jeu de tic-tac-toe et que nous avons commencé à

travailler sur la fonctionnalité de base pour effectuer des mouvements au tableau. Nous commençons avec un scénario comme celui-ci :
Fichier tic_tac_toc.feature

```
Feature:
 Scenario:
  Given a board like this:
    |   | 1 | 2 | 3 |
    | 1 |   |   |   |
    | 2 |   |   |   |
    | 3 |   |   |   |
   When player x plays in row 2, column 1
   Then the board should look like this:
    |   | 1 | 2 | 3 |
    | 1 |   |   |   |
    | 2 | X |   |   |
    | 3 |   |   |   |
```

Nous allons vous montrer comment saisir la table de la première étape, la manipuler dans la seconde et enfin comparer le tableau attendu du scénario avec le tableau réel.

Exécutez ./cucumber pour générer les extraits de définition d'étape, puis collez-les dans step_definitions / BoardSteps.java

```
package com.zaghir.project.hellocucumber.stepdefinition;

import cucumber.api.PendingException;
import io.cucumber.datatable.DataTable;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;

public class BoardSteps {

        @Given("^a board like this:$")
        public void aBoardLikeThis(DataTable arg1) throws Throwable {
            // Write code here that turns the phrase above into concrete actions
            // For automatic transformation, change DataTable to one of
            // List<YourType>, List<List<E>>, List<Map<K,V>> or Map<K,V>.
            // E,K,V must be a scalar (String, Integer, Date, enum etc)
            throw new PendingException();
        }

        @When("^player x plays in row (\\d+), column (\\d+)$")
        public void playerXPlaysInRowColumn(int arg1, int arg2) throws Throwable {
            // Write code here that turns the phrase above into concrete actions
            throw new PendingException();
        }

        @Then("^the board should look like this:$")
        public void theBoardShouldLookLikeThis(DataTable arg1) throws Throwable {
            // Write code here that turns the phrase above into concrete actions
            // For automatic transformation, change DataTable to one of
            // List<YourType>, List<List<E>>, List<Map<K,V>> or Map<K,V>.
            // E,K,V must be a scalar (String, Integer, Date, enum etc)
            throw new PendingException();
        }
```

}
Notez que les extraits de code pour les définitions à deux étapes où nous allons recevoir un tableau qui est un peu différents. Il y a un Un commentaire qui vous informe sur la conversion automatique de l'argument DataTable qui vous a été transmis, mais nous l'ignorerons pour le moment (ne vous inquiétez pas, nous parlerons beaucoup de la conversion automatique plus tard).
Cucumber.api.DataTable est un objet très riche avec de nombreuses méthodes pour interagir avec ses données. Nous allons vous montrer quelques-uns des plus utiles maintenant.
Commençons à préciser ces définitions.

Tourner la table dans une liste des listes

Sous le capot, la table n'est qu'une liste de listes de String : Liste <Liste <String >>. Souvent, nous voudrons travailler avec cela sous cette forme brute, nous pouvons donc appeler la méthode brute dessus pour le faire. Récupérons les données brutes de la table et stockons-les dans un tableau de variables d'instance, que nous pourrons manipuler lors de notre deuxième étape lorsque nous voulons effectuer un déplacement.
A titre expérimental, ajoutez une implémentation pour la définition de la deuxième étape qui imprime simplement le tableau brut afin que nous puissions voir à quoi elle ressemble :

```
@Given("^a board like this:$")
	public void aBoardLikeThis(DataTable table) throws Throwable {
		// table.raw(); n'est plus suporter dans la nouvelle version de
cucumber >= 2 on
		// utilise asLists
		board = table.asLists(String.class);
	}

	@When("^player x plays in row (\\d+), column (\\d+)$")
	public void playerXPlaysInRowColumn(int arg1, int arg2) throws Throwable {
		System.out.println("-----------------------------");
		System.out.println(board.toString());

		throw new PendingException();
	}
```

Exécuter la feature tic_tac_toc.feature
Running RunCukesTest
Feature:
[[, 1, 2, 3], [1, , ,], [2, , ,], [3, , ,]]
 Scenario: # tic_tac_toe/tic_tac_toe.feature:2
  Given a board like this:
  When player x plays in row 2, column 1
  cucumber.api.PendingException: TODO: implement me
   at tic_tac_toe.BoardSteps.playerXPlaysInRowColumn(BoardSteps.java:21)
   at *.When player x plays in row 2, column 1(tic_tac_toe.feature:8)
 Then the board should look like this:
1 Scenarios (1 pending)
3 Steps (1 skipped, 1 pending, 1 passed)

La table brute regroupe les entêtes et les lignes aussi

Comparaison des tables avec la différence

Pour que nous puissions commencer avec un test qui a échoué, nous ne ferons aucune manipulation du tableau dans cette étape pour le moment. Supprimez donc le corps de la définition de la deuxième étape et effectuez l'implémentation suivante dans la définition de la dernière étape :

```
@When("^player x plays in row (\\d+), column (\\d+)$")
public void playerXPlaysInRowColumn(int arg1, int arg2) throws Throwable {
}
@Then("^the board should look like this:$")
public void theBoardShouldLookLikeThis(DataTable expectedTable) throws Throwable {
  expectedTable.diff(board);
}
```

Nous avons utilisé la méthode diff de la table qui décrit l'aspect visuel, en transmettant le résultat au tableau tel que nous le voyons dans notre application. Lorsque vous exécutez à nouveau mvn clean test, vous devez voir que l'étape a échoué car les tables n'étaient pas identiques:

```
package com.zaghir.project.hellocucumber.stepdefinition;

import java.util.List;

import io.cucumber.datatable.DataTable;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;

public class BoardSteps {

	private List<List<String>> board;
	private DataTable actualDataTable ;

	@Given("^a board like this:$")
	public void aBoardLikeThis(DataTable table) throws Throwable {
		// table.raw(); n'est plus suporter dans la nouvelle version de
cucumber >= 2 on
		// utilise asLists
		board = table.asLists(String.class);
		actualDataTable = table ;
	}

	@When("^player x plays in row (\\d+), column (\\d+)$")
	public void playerXPlaysInRowColumn(int arg1, int arg2) throws Throwable {
		System.out.println("-----------------------------");
		System.out.println(board.toString()); // en suprimer l'exception

	}

	@Then("^the board should look like this:$")
	public void theBoardShouldLookLikeThis(DataTable expectedTable) throws
Throwable {
		expectedTable.diff(actualDataTable);
	}
}
```

```
Running RunCukesTest
Feature:
```

```
Scenario:                    # tic_tac_toe/tic_tac_toe.feature:2
  Given a board like this:
  When player x plays in row 2, column 1
  Then the board should look like this:
    cucumber.runtime.table.TableDiffException: Tables were not identical:
    |   | 1 | 2 | 3 |
    | 1 |   |   |   |
  - | 2 | x |   |   |
  + | 2 |   |   |   |
    | 3 |   |   |   |
  at cucumber.runtime.table.TableDiffer.calculateDiffs(TableDiffer.java:32)
  at cucumber.api.DataTable.diff(DataTable.java:169)
  at cucumber.api.DataTable.diff(DataTable.java:159)
  at tic_tac_toe.BoardSteps.theBoardShouldLookLikeThis(BoardSteps.java:24)
  at *.Then the board should look like this:(tic_tac_toe.feature:9)
1 Scenarios (1 failed)
3 Steps (1 failed, 2 passed)
```

Les lignes qui diffèrent de celles attendues seront imprimées deux fois: la première (précédée d'un «-») correspond à ce qui était attendu, suivie d'une autre (précédée d'un «+») qui correspond à ce qui a été réellement renvoyé.
Corrigeons l'étape @When pour que le scénario passe. Ajoutez cette implémentation à la définition de la deuxième étape:

```
@When("^player x plays in row (\\d+), column (\\d+)$")
	public void playerXPlaysInRowColumn(int row, int col) throws Throwable {
		// board.get(row).set(col, "x");
		actualDataTable.row(row).set(col, "X");
	}
```

Executer le scenario , mais on va avoir un probleme de type runtime error

```
java.lang.UnsupportedOperationException
  at java.util.Collections$UnmodifiableList.set(Collections.java:1311)
  at tic_tac_toe.BoardSteps.playerXPlaysInRowColumn(BoardSteps.java:20)
  at *.When player x plays in row 2, column 1(tic_tac_toe.feature:8)
```

Cette erreur est du au fait que la Datatable est immodifiable , c'est le comportement de cucumber
On passe par une copie pour réussir le test

```
@Given("^a board like this:$")
	public void aBoardLikeThis(DataTable table) throws Throwable {
		// table.raw(); n'est plus suporter dans la nouvelle version de
cucumber >= 2 on
		// utilise asLists

		//board = table.asLists(String.class);
		List<List<String>> boardTmp = new ArrayList();
		for(List<String> row : table.asLists()) {
			boardTmp.add( new ArrayList<String>(row));
		}

		//actualDataTable = DataTable.create(boardTmp);
	}

@When("^player x plays in row (\\d+), column (\\d+)$")
	public void playerXPlaysInRowColumn(int row, int col) throws Throwable {
		//actualDataTable.row(row).set(col, "X"); //get(row).set(col, "x");
```

```
            System.out.println("--------------> 
"+actualDataTable.row(row).set(col, "X"));
        }
```

J'ai essayé mais j'ai toujours le probleme de reference de java à voir

Les tableaux de données sont une excellente fonctionnalité de Gherkin. Ils sont vraiment polyvalents et vous aident à exprimer les données de manière concise, comme vous le souhaiteriez dans un document de spécification normal. Avec les arrière-plans et les tableaux de données, vous pouvez faire beaucoup pour réduire le bruit et l'encombrement(cluster) de vos scénarios. Même lorsque vous utilisez ces outils, vous verrez toujours parfois un motif dans lequel un scénario ressemble beaucoup à celui qui le précédait et celui qui le suivait. C'est à ce stade qu'un aperçu de scénario peut aider.

Configuration du Senario

Parfois, vous avez plusieurs scénarios qui suivent exactement le même modèle d'étapes, mais avec des valeurs d'entrée ou des résultats attendus différents. Par exemple, supposons que nous testions chacun des boutons de retrait à montant fixe sur le DAB.

Feature: Withdraw Fixed Amount
The *"Withdraw Cash"* menu contains several fixed amounts to
speed up transactions for users.
Scenario: Withdraw fixed amount of $50
Given I have $500 in my account
When I choose to withdraw the fixed amount of $50
Then I should receive $50 cash
And the balance of my account should be $450

Scenario: Withdraw fixed amount of $100
Given I have $500 in my account
When I choose to withdraw the fixed amount of $100
Then I should receive $100 cash
And the balance of my account should be $400
Scenario: Withdraw fixed amount of $200
Given I have $500 in my account
When I choose to withdraw the fixed amount of $200
Then I should receive $200 cash
And the balance of my account should be $300

Encore une fois, toute la répétition de cette fonctionnalité rend la lecture ennuyeuse. Il est difficile de voir l'essenciel de chaque scénario, c'est-à-dire le montant des sommes investies dans chaque transaction. Nous pouvons utiliser un plan de scénario pour spécifier les étapes une fois, puis jouer plusieurs ensembles de valeurs à travers elles. Voici à nouveau ce scénario, modifié pour utiliser un plan de scénario :

Feature: Withdraw Fixed Amount
The *"Withdraw Cash"* menu contains several fixed amounts to
speed up transactions for users.
Scenario Outline: Withdraw fixed amount
Given I have <Balance> in my account
When I choose to withdraw the fixed amount of <Withdrawal>
Then I should receive <Received> cash
And the balance of my account should be <Remaining>

Examples:

Balance	Withdrawal	Received	Remaining
$500	$50	$50	$450
$500	$100	$100	$400
$500	$200	$200	$300

Nous indiquons des espaces réservés dans l'esquisse de scénario à l'aide de chevrons(angle brackets) (<..>) où nous souhaitons que les valeurs réelles soient substituées. Le plan de scénario lui-même est inutile sans un tableau Exemples, qui répertorie les lignes de valeurs à substituer pour chaque espace réservé.
Vous pouvez avoir un nombre quelconque d'éléments de scénario dans une fonction et un nombre quelconque de tableaux Exemples dans chaque plan de scénario. En coulisse, Cucumber convertit chaque ligne du tableau Exemples en un scénario avant de l'exécuter.
L'un des avantages de l'utilisation d'un schéma de scénario est que vous pouvez clairement voir les lacunes dans vos exemples. Dans notre exemple, nous n'avons testé aucun cas particulier, par exemple lorsque vous essayez de retirer plus d'argent que ce dont vous disposiez. Cela devient beaucoup plus évident lorsque vous pouvez voir toutes les valeurs alignées dans un tableau.
N'oubliez pas que, bien que la syntaxe utilisée pour les écrire dans Gherkin soit identique, ces tableaux sont totalement différents des tableaux de données décrits précédemment.Les tableaux de données décrivent simplement un bloc de données à associer à une étape d'un scénario. Dans un plan de scénario, chaque ligne d'un tableau représente un scénario complet à exécuter par Cucumber. En fait, vous pouvez utiliser le mot-clé Scénarios (notez les caractères supplémentaires) à la place de Exemples si vous le trouvez plus lisible.

Grands espaces réservés (Bigger Placeholders)

Il est facile d'imaginer que vous ne pouvez utiliser des espaces réservés d'esquisse de scénario que s'il existe une donnée dans l'étape. En fait, lorsque Cucumber compile un tableau d'exemples en scénarios prêts à être exécutés, il se soucie peu de savoir où se trouvent les espaces réservés. Vous pouvez donc substituer autant que vous le souhaitez à partir du texte de n'importe quelle étape.
Illustrons cela en testant le cas où nous essayons de retirer plus d'argent que nous n'avons. Que devrions-nous faire dans ce cas ? Donner aux utilisateurs autant que possible, compte tenu de leur solde, ou simplement leur montrer un message d'erreur ?
Nous demandons des éclaircissements à nos parties prenantes, qui sont heureuses que nous leur montrions un message d'erreur. Voici comment nous écrivons le scénario en premier :

Scenario: Try to withdraw too much
Given I have $100 in my account
When I choose to withdraw the fixed amount of $200
Then I should see an error message
And the balance of my account should be $100

Il y a encore beaucoup de duplication ici avec le flux des scénarios précédents, mais étant donné que l'étape Then est si différente, nous ne pouvons pas l'inscrire dans le outline de scénario. Ou pouvons-nous ?
Modifions le plan du scénario en remplaçant l'espace réservé <Received> par un résumé <résultat> plus abstrait :

Scenario Outline: Withdraw fixed amount
Given I have <Balance> in my account
When I choose to withdraw the fixed amount of <Withdrawal>
Then I should <Outcome>

And the balance of my account should be <Remaining>
Examples:

Balance	Withdrawal	Remaining	Outcome
$500	$50	$450	receive $50 cash
$500	$100	$400	receive $100 cash
$500	$200	$300	receive $200 cash

Maintenant, nous pouvons simplement ajouter notre cas d'échec au bas de ce tableau :

Scenario Outline: Withdraw fixed amount
Given I have <Balance> in my account
When I choose to withdraw the fixed amount of <Withdrawal>
Then I should <Outcome>
And the balance of my account should be <Remaining>
Examples:

Balance	Withdrawal	Remaining	Outcome
$500	$50	$450	receive $50 cash
$500	$100	$400	receive $100 cash
$500	$200	$300	receive $200 cash
$100	$200	$100	see an error message

Nous pouvons utiliser un espace réservé pour remplacer n'importe quel texte que nous aimons lors d'une étape. Notez que l'ordre dans lequel les espaces réservés apparaissent dans le tableau n'a pas d'importance. ce qui compte, c'est que l'en-tête de colonne corresponde au texte de l'espace réservé dans le plan du scénario.
Même s'il s'agit d'une technique utile, veillez à ce que votre programmeur n'ait pas l'instinct de réduire les doublons à tout prix. Si vous déplacez trop le texte d'une étape dans le tableau des exemples, il peut être très difficile de lire le déroulement du scénario. N'oubliez pas que votre objectif est la lisibilité. Ne le poussez pas trop loin. Testez toujours vos fonctionnalités en demandant à d'autres personnes de les lire régulièrement et de vous donner leurs impressions.

Combien d'exemple dois je utilisé ?
Une fois que vous avez un plan de scénario avec quelques exemples, il est très facile de penser à plus d'exemples, et encore plus facile de les ajouter. Avant de vous en rendre compte, vous disposez d'un vaste tableau d'exemples très complet et d'un problème. Pourquoi?
Sur un système de toute complexité sérieuse, vous pouvez très rapidement commencer à expérimenter ce que les mathématiciens appellent une explosion combinatoire, où le nombre de combinaisons différentes d'entrées et de sorties attendues devient ingérable. En essayant de couvrir toutes les éventualités possibles, vous vous retrouvez avec des lignes et des lignes de données d'exemple à exécuter par Cucumber. N'oubliez pas que chacune de ces petites lignes représente un scénario complet dont l'exécution peut prendre plusieurs secondes et qui peut rapidement commencer à s'additionner. Lorsque l'exécution de vos tests prend plus de temps, vous ralentissez votre boucle de rétroaction, ce qui rend toute l'équipe moins productive.
Une très longue table est également très difficile à lire. Il est préférable d'essayer de rendre vos exemples plus illustratifs ou représentatifs qu'exhaustifs. Essayez de vous en tenir à ce que Gojko Adzic appelle les exemples clés.a Si vous étudiez le code que vous testez, vous constaterez souvent que certaines lignes de votre tableau d'exemples couvrent la même logique qu'une autre ligne du tableau. Vous pouvez également constater que les cas de test de votre table sont déjà couverts par des tests unitaires du code sous-jacent. S'ils ne le sont pas, demandez-vous s'ils devraient l'être. Rappelez-vous que la lisibilité est le plus important. Si vos parties prenantes se sentent rassurées par des tests exhaustifs, peut-être parce que votre logiciel fonctionne dans un environnement critique

pour la sécurité, alors intégrez-les bien. N'oubliez pas que vous ne pourrez jamais pour prouver qu'il n'y a pas de bugs. Comme le disent les logiciens, l'absence de preuve n'est pas une preuve d'absence.

Des exemples multiple tables

Concombre se fera un plaisir de gérer n'importe quel nombre d'éléments Exemples situés sous un schéma de scénario, ce qui signifie que vous pouvez regrouper différents types d'exemples si vous le souhaitez. Par exemple :

```
Scenario Outline: Withdraw fixed amount
  Given I have <Balance> in my account
  When I choose to withdraw the fixed amount of <Withdrawal>
  Then I should <Outcome>
  And the balance of my account should be <Remaining>

  Examples: Successful withdrawal
    | Balance | Withdrawal | Outcome             | Remaining  |
    | $500    | $50        | receive $50 cash    | $450       |
    | $500    | $100       | receive $100 cash   | $400       |

  Examples: Attempt to withdraw too much
    | Balance | Withdrawal | Outcome               | Remaining |
    | $100    | $200       | see an error message | $100      |
    | $0      | $50        | see an error message | $0        |
```

Comme d'habitude, vous avez la possibilité d'attribuer un nom et une description à chaque tableau Exemples. Lorsque vous avez un grand nombre d'exemples, le fractionner en plusieurs tableaux peut être plus facile à comprendre pour le lecteur.

Expliquer vous

Vous utiliserez normalement un schéma de scénario et un tableau d'exemples pour vous aider à spécifier l'implémentation d'une règle de gestion. N'oubliez pas d'inclure une description en langage clair de la règle sous-jacente que les exemples sont supposés illustrer !
C'est incroyable de voir combien de personnes oublient de le faire.
Par exemple, regardez cette fonctionnalité :

```
Feature: Account Creation
  Scenario Outline: Password validation
    Given I try to create an account with password "<Password>"
    Then I should see that the password is <Valid or Invalid>
    Examples:
      | Password | Valid or Invalid |
      | abc      | invalid          |
      | ab1      | invalid          |
      | abc1     | valid            |
      | abcd     | invalid          |
      | abcd1    | valid            |
```

Si vous devez implémenter le code pour faire passer cette fonctionnalité, pouvez-vous indiquer la règle sous-jacente?
Pas très facilement. Modifions donc la fonctionnalité pour la rendre plus explicite, comme ceci :

Feature: Account Creation
Scenario Outline: Password validation
Given I try to create an account with password *"<Password>"*
Then I should see that the password is <Valid or Invalid>

Examples: Too Short
Passwords are invalid if less than 4 characters

Password	Valid or Invalid
abc	invalid
ab1	invalid

Examples: Letters and Numbers
Passwords need both letters and numbers to be valid

Password	Valid or Invalid
abc1	valid
abcd	invalid
abcd1	valid

En séparant les exemples en deux ensembles et en attribuant à chacun un nom et une description, nous avons expliqué la règle et donné des exemples de la règle en même temps.

Trop d'informations

Trouver le bon niveau de détail, ou d'abstraction, à utiliser dans vos scénarios est une compétence qui prend du temps à maîtriser. Ce que beaucoup de gens ne réalisent pas, c'est que différents niveaux de détail sont appropriés pour différents scénarios dans le même système - parfois dans la même fonctionnalité - en fonction de ce qu'ils décrivent.
Par exemple, voici un scénario pour l'utilisateur de notre guichet automatique s'authentifiant avec leur PIN :

Scenario: Successful login with PIN
Given I have pushed my card in the slot
When I enter my PIN
And I press *"OK"*
Then I should see the main menu

Il est tout à fait approprié d'aborder ce processus avec autant de détails sur le processus d'authentification, car c'est là que nous nous concentrons. Examinons maintenant le scénario de retrait d'argent présenté précédemment, qui a un objectif différent mais doit encore être authentifié. Est-il judicieux d'exprimer les étapes d'authentification par code PIN de ce scénario au même niveau de détail? Essayons:

Scenario: Withdraw fixed amount of $50
Given I have $500 in my account
And I have pushed my card into the slot
And I enter my PIN
And I press *"OK"*
When I choose to withdraw the fixed amount of $50
Then I should receive $50 cash
And the balance of my account should be $450

C'est terrible ! Il y a tellement de bruit dans l'authentification que nous remarquons à peine l'essentiel : le retrait de l'argent. Ce détail était utile dans le scénario de code confidentiel où il était pertinent, mais maintenant, il ne fait que distraire.
Pour le moment, nous souhaitons vous encourager à extraire ces détails dans une définition séparée pour que notre scénario reste facile à lire.

Extrater les details

Prenons les trois étapes d'authentification et résumons ce qu'elles font en une seule étape de haut niveau :

Given I have authenticated with the correct PIN

Supprimez ces trois lignes du scénario et remplacez-les par cette étape unique. Maintenant, exécutez ./cucumber pour générer l'extrait de définition d'étape pour votre nouvelle étape de haut niveau. Ça devrait ressembler à ça :

```
@Given("^I have authenticated with the correct PIN$")
public void iHaveAuthenticatedWithTheCorrectPIN() throws Throwable {
 // Express the Regexp above with the code you wish you had
 throw new PendingException();
}
```

Maintenant, créez une méthode dans votre fichier de définition d'étape appelée authenticateWithPIN qui fait le nécessaire pour s'authentifier avec un code confidentiel et modifiez la définition de votre étape de la manière suivante :

```
@Given("^I have authenticated with the correct PIN$")
public void iHaveAuthenticatedWithTheCorrectPIN() throws Throwable {
 authenticateWithPIN();
}
```

Votre authenticateWithPIN peut effectuer exactement les mêmes appels que les définitions à trois étapes qu'il remplace, ou il peut exister un autre moyen d'obtenir le guichet automatique dans un état authentifié. Quoi qu'il en soit, le scénario est maintenant beaucoup plus lisible :

```
Scenario: Withdraw fixed amount of $50
 Given I have $500 in my account
 And I have authenticated with the correct PIN
 When I choose to withdraw the fixed amount of $50
 Then I should receive $50 cash
 And the balance of my account should be $450
```

Doc Strings

Les chaînes de documents vous permettent de spécifier un texte plus volumineux que ce que vous pourriez tenir sur une seule ligne. Par exemple, si vous devez décrire le contenu précis d'un mail, vous pouvez le faire comme suit :

```
Scenario: Ban Unscrupulous Users
 When I behave unscrupulously
 Then I should receive an email containing :
```

```
"""
Dear Sir,
Your account privileges have been revoked due to your unscrupulous behavior.
Sincerely,
The Management
"""
And my account should be locked
```

Tout comme pour une table de données, la chaîne entière entre les "" "guillemets triples est attachée à l'étape située au-dessus de celle-ci. L'indentation de l'ouverture" "" n'est pas importante, bien que la pratique courante consiste à indenter deux espaces de l'étape qui les entoure, comme nous avons montré. L'indentation à l'intérieur des guillemets triples est toutefois importante: imaginez la marge gauche qui descend du début du premier "" ". Si vous souhaitez inclure une indentation dans votre chaîne, vous devez l'indenter dans cette marge.
Les chaînes de documentation ouvrent toutes sortes de possibilités pour spécifier des données dans vos étapes.
Nous avons vu des équipes utiliser ces arguments pour spécifier des extraits de données JSON ou XML lors de l'écriture de fonctionnalités pour une API, par exemple
Vous devez être prudent lorsque vous incluez autant de détails dans un scénario.
Il est facile de créer beaucoup de fouillis avec une grande quantité de données, ce qui rend difficile la lecture du scénario dans son ensemble. Vous pouvez également créer facilement des scénarios fragiles, où la moindre modification apportée au système entraîne l'échec du scénario, car il se comporte légèrement différemment de la façon dont il a été décrit dans la chaîne de documentation.

Rester organisé avec des balises(Tags) et des sous-dossiers(Subfolders)

Il est facile d'être organisé lorsque vous n'avez que quelques fonctionnalités, mais au fur et à mesure que votre suite de tests se développe, vous souhaiterez garder les choses en ordre afin que la documentation soit facile à lire et à parcourir. Une façon simple de procéder consiste à utiliser des sous-dossiers pour classer vos fonctionnalités. Cela ne vous donne cependant qu'un seul axe d'organisation. Vous pouvez donc également utiliser des balises pour associer une étiquette à un scénario, ce qui vous permet de découper autant de fois que vous le souhaitez vos fonctionnalités.

Subfolders

C'est le moyen le plus simple d'organiser vos fonctionnalités. Vous pouvez vous trouver déchiré quant à la manière de choisir une catégorie. Toutefois, organisez-vous par type d'utilisateur, avec un dossier features / admins, un dossier features / logs_in_users et un dossier features / visiteurs, par exemple? Ou les organisez-vous par entité de domaine ou autre chose?
Bien sûr, c'est une décision à prendre pour vous et votre équipe, mais nous pouvons vous donner quelques conseils. Nous avons eu beaucoup de succès en utilisant des sous-dossiers pour représenter différentes tâches de haut niveau qu'un utilisateur peut essayer de faire. Donc, si nous construisions un système de reporting intranet, nous pourrions l'organiser de la manière suivante:

```
features/
  reading_reports/
  report_building/
  user_administration/
```

Ne vous attardez pas trop pour que votre structure de dossiers soit correcte du premier coup. Prenez la décision d'essayer une structure, de réorganiser tous les fichiers de fonctionnalités existants, puis de vous y tenir pendant un certain temps lorsque vous ajoutez de nouvelles fonctionnalités. Mettez une note dans le calendrier pour vous détendre dans quelques semaines et demandez-vous si la nouvelle structure fonctionne.

Si vous considérez vos fonctionnalités comme un livre décrivant ce que fait votre système, les sous-dossiers sont comme les chapitres de ce livre. Alors, alors que vous racontez l'histoire de votre système, que voulez-vous que le lecteur voie quand il numérise la table des matières?

Les Features ne sont pas des histroires d'utilistateurs (User Stories)

Il y a bien longtemps, Cucumber était un outil appelé RSpec Story Runner. À cette époque, les tests en langage clair utilisaient une extension .story. Lorsque j'ai créé Cucumber, j'ai pris la décision délibérée de nommer les caractéristiques des fichiers plutôt que les histoires. Pourquoi j'ai fait ça? Les histoires d'utilisateurs sont un excellent outil de planification. Chaque récit contient un peu de fonctionnalités que vous pouvez hiérarchiser, créer, tester et publier. Une fois qu'un article a été publié, nous ne voulons pas qu'il laisse une trace dans le code. Nous utilisons la refactorisation pour nettoyer la conception de sorte que le code absorbe le nouveau comportement spécifié par la user story, en le laissant
en regardant comme si ce comportement avait toujours été là.
Nous souhaitons que la même chose se produise avec nos fonctions Cucumber. Les fonctionnalités doivent décrire le comportement actuel du système, mais elles n'ont pas besoin de documenter l'historique de sa construction; c'est à quoi sert un système de contrôle de version!
Nous avons vu des équipes dont le répertoire de fonctionnalités ressemble à ceci:

```
traits/
story_38971_generate_new_report.feature
story_38986_run_report.feature
story_39004_log_in.feature
...
```

Nous vous encourageons fortement à ne pas le faire. Vous vous retrouverez avec des fonctionnalités fragmentées qui ne fonctionnent tout simplement pas comme documentation pour votre système. Une user story peut correspondre à une fonctionnalité, mais une autre peut vous amener à ajouter ou modifier des scénarios dans plusieurs fonctionnalités existantes, par exemple si la story change la façon dont les utilisateurs doivent s'authentifier. Il est peu probable qu'il y ait toujours un mappage individuel de chaque user story à chaque fonctionnalité, aussi n'essayez pas de la forcer. Si vous devez conserver un identifiant d'histoire pour un scénario, utilisez plutôt une balise.

Tags

Si les sous-dossiers sont les chapitres de votre livre de fonctionnalités, les balises(Tage) sont les notes autocollantes que vous avez placées sur des pages que vous souhaitez pouvoir trouver facilement. Vous marquez un scénario en mettant un mot précédé du caractère @ sur la ligne précédant le mot clé Scenario, comme ceci :

```
@widgets
Scenario: Generate report
  Given I am logged in
  And there is a report "Best selling widgets"
  ...
```

En fait, vous pouvez associer plusieurs balises au même scénario, séparées par des espaces :

```
@slow @widgets @nightly
Scenario: Generate overnight report
  Given I am logged in
  And there is a report "Total widget sales history"
  ...
```

Si vous souhaitez baliser tous les scénarios d'une fonctionnalité à la fois, il suffit de baliser l'élément Feature en haut de la liste pour que tous les scénarios héritent de la balise. Vous pouvez également baliser des scénarios individuels.

```
@nightly @slow
Feature: Nightly Reports
  @widgets
  Scenario: Generate overnight widgets report
   ...
  @doofers
  Scenario: Generate overnight doofers report
  ...
```

Dans l'exemple précédent, le scénario appelé Générer un rapport sur les widgets de nuit comportera trois balises: @nightly, @slow et @widgets, alors que le scénario intitulé Générer de doublons d'une nuit contient les balises @nightly, @slow et @doofers. Vous pouvez également baliser les éléments de scénario et les tableaux individuels d'exemples qui s'y trouvent.
Les scénarios de marquage ont trois raisons principales :
• Documentation : vous souhaitez utiliser une balise pour associer une étiquette à certains scénarios, par exemple pour les étiqueter avec un ID provenant d'un outil de gestion de projet.
• Filtrage : Cucumber vous permet d'utiliser des balises en tant que filtre pour sélectionner des scénarios spécifiques à exécuter ou à générer des rapports. Vous pouvez même demander à Cucumber d'échouer votre test si une balise donnée apparaît trop souvent.
• Crochets (Hooks): exécutez un bloc de code chaque fois qu'un scénario avec une balise particulière est sur le point de commencer ou vient de se terminer.