

Test Driven Development en Java

Le développement piloté par les tests

Mai 2024

Plan

- 1 Les techniques d'écriture de tests
- 2 Tests unitaires et couverture du code
- 3 Test du code hérité
- 4 Les tests fonctionnels avec Cucumber
- 5 Simplification de l'écriture des tests
- 6 Tests fonctionnels avec FitNesse
- 7 Selenium et JUnit

Organisation des tests

Organisation des tests

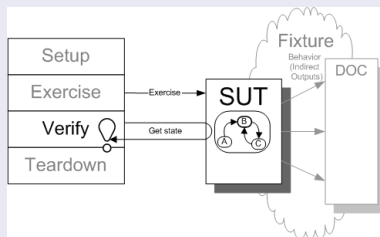
➡ Les tests peuvent être organisés de différentes façons

- ❶ Classe de test par classe : on met toutes les méthodes de test pour une classe de système sous test (SUT) sur une seule Classe de test
- ❷ Classe de test par fonctionnalité : on regroupe les méthodes de test sur les classes de test en fonction de la fonction testable de le SUT qu'ils exercent.
- ❸ Classe de test par fixature : on organise les méthodes de test dans des classes de test basées sur la similitude du test fixation.

Principales stratégies pour vérifier les résultats d'un test

Vérification de l'état

- On détermine si la méthode exercée a fonctionné correctement en examinant l'état du SUT et de ses collaborateurs après la a été exercée
 - 📖 L'état d'un objet est défini par les valeurs de ses attributs
- Approche de test classique



Principales stratégies pour vérifier les résultats d'un test

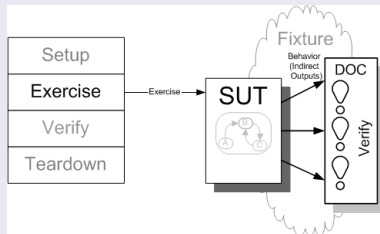
Vérification de l'état : principe de fonctionnement

- ➊ Initialisation : Instancier le SUT et ses collaborateurs
- ➋ L'exécution : l'étape où la méthode à vérifier est exécutée
- ➌ La vérification : l'exécution **des assertions** qui permettent de vérifier l'**état** du SUT et de ses collaborateurs.
- ➍ Le nettoyage (teardown en anglais) : l'étape où les données de test sont réinitialisées, supprimées ou autre.

Principales stratégies pour vérifier les résultats d'un test

Vérification comportementale

- Elle consiste à vérifier le comportement du SUT et de sa collaboration avec les autres objets.
 - 📖 Elle garantit que le SUT se comporte vraiment comme spécifié plutôt que juste se retrouver dans le bon état post-exercice
- Elle est basée sur l'utilisation des Mocks



Principales stratégies pour vérifier les résultats d'un test

Vérification comportementale : principe de fonctionnement

① Initialisation

- Instancier le SUT et les **mocks** à partir des classes interfaces qui représentent les collaborateurs
- Initialiser les attentes (les "Expectations") : on écrit le comportement attendu par l'objet mocké

② L'exécution et vérification : la méthode à vérifier est exécutée ce qui induit que les appels des méthodes des Mocks sont surveillés par Mockito

④ Le nettoyage (teardown en anglais) : l'étape où les données de test sont réinitialisées, supprimées ou autre.

Exemple : vérification de l'état vs comportementale

Exemple : énoncé

Il y a deux objets : un entrepôt (warehouse) et une commande (order). L'entrepôt contient différents types de produits dans des quantités limitées. Une commande concerne un produit pour une quantité donnée. S'il y a suffisamment de stock dans l'entrepôt, la commande est **passée**; sinon, **rien ne se passe**.

Principe

- ➡ L'objet à tester (SUT) : une commande
 - 🚩 Vérifier que la commande change d'état lorsqu'il y a suffisamment de stock dans l'entrepôt
- ➡ L'objet collaborateur : un entrepôt

Principales stratégies pour vérifier les résultats d'un test

Exemple : Méthodes de la classe de tests

```
1 private static String GUINNESS = "Guinness";
2 private static String CHIMAY = "Chimay";
3 private Warehouse warehouse;
4
5 // Initialization, processed before each test thanks to the Before annotation
6 @BeforeEach
7 protected void setUp() throws Exception {
8     // The fixtures or, collaborators
9     warehouse = new Warehouse();
10    warehouse.add(GUINNESS, 50);
11    warehouse.add(CHIMAY, 25);
12 }
13
14 // Teardown, processed after each test thanks to the After annotation
15 @AfterEach
16 protected void tearDown() throws Exception {
17     warehouse = null;
18 }
```

Vérification de l'état

Vérifier l'état d'une commande

- ➡ Vérifier qu'une commande a été effectuée lorsque le stock contient suffisamment de produits

```
1 public void testOrderIsFilledIfEnoughInWarehouse() {  
2     // The System Under Testing  
3     Order order = new Order(GUINNESS, 50);  
4     // Exercise  
5     order.fill(warehouse);  
6  
7     // Check  
8     assertTrue(order.isFilled());  
9     assertEquals(0, warehouse.getInventory(GUINNESS));  
10 }
```

Vérifier l'état d'une commande

- ➡ Vérifier qu'une commande n'était pas effectuée lorsque le stock ne contient pas suffisamment de produits

```
1 public void testOrderDoesNotRemoveIfNotEnough() {  
2     Order order = new Order(GUINNESS, 51);  
3     order.fill(warehouse);  
4     assertFalse(order.isFilled());  
5     assertEquals(50, warehouse.getInventory(GUINNESS));  
6 }
```

Vérification comportementale

Principe

- L'objet à tester (SUT) : une commande
- L'objet collaborateur : un entrepôt
 - 👉 Cet objet sera mocké et surveillé

Vérification comportementale

Vérifier une commande

- ➡ Vérifier qu'une commande a été effectuée lorsque le stock contient suffisamment de produits

```
1 public void testFillingRemovesInventoryIfInStock() {
2     // Setup
3     Order order = new Order(GUINNESS, 50);
4     Warehouse warehouseMock = mock(Warehouse.class);
5
6     // stubs
7     when(warehouseMock.hasInventory(GUINNESS, 50)).thenReturn(true);
8
9     // exercise
10    order.fill(warehouseMock);
11
12    // verify
13    InOrder inOrder = inOrder(warehouseMock);
14    inOrder.verify(warehouseMock).hasInventory(GUINNESS, 50);
15    inOrder.verify(warehouseMock).remove(GUINNESS, 50);
16 }
```

Vérification comportementale

Vérifier une commande

- ➡ Vérifier qu'une commande n'était pas effectuée lorsque le stock ne contient pas suffisamment de produits

```
1 public void testFillingDoesNotRemoveIfNotEnoughInStock() {
2     Order order = new Order(GUINNESS, 51);
3     Warehouse warehouseMock = mock(Warehouse.class);
4
5     // stubs
6     when(warehouseMock.hasInventory(GUINNESS, 51)).thenReturn(false);
7
8     // exercise
9     order.fill(warehouseMock);
10
11    // verify
12    verify(warehouseMock).hasInventory(GUINNESS, 51);
13    verify(warehouseMock, never()).remove(anyString(), anyInt());
14 }
```

Plan

- 1 Les techniques d'écriture de tests
- 2 Tests unitaires et couverture du code**
- 3 Test du code hérité
- 4 Les tests fonctionnels avec Cucumber
- 5 Simplification de l'écriture des tests
- 6 Tests fonctionnels avec FitNesse
- 7 Selenium et JUnit

Couverture du code

Tester quoi ?

- ➡ Plusieurs opinions sur le sujet
 - 📖 Absolument tout
 - 📖 L'interface publique
 - 📖 Les morceaux complexes, critiques
 - 📖 Comportement vs. état
- ➡ Règles générales
 - 📖 S'assurer que tout le code écrit fonctionne

Couverture du code

Définition

- ➡ Taux de code source exécuté par les tests
 - 📖 Fournit une valeur quantitative permettant de mesurer de manière indirecte la qualité des tests
 - 📖 Met en évidence les parties d'un programme qui ne sont pas testées
 - 📖 Permet d'indiquer s'il est nécessaire d'ajouter de nouveaux tests

Idée générale

- ➡ On veut **tester tous les chemins d'exécution possibles**
 - 📖 Plus on a de tests, moins on risque de régression
 - 📖 Pas besoin de tester le code trivial (getter / setter)

Critères de couverture

Principales métriques de couverture de code

- Couverture des instructions
 - 📊 Pourcentage d'instructions du code source qui ont été exécutées
- Couverture des branches
 - 📊 Pourcentage de branches conditionnelles (`if/else`) qui ont été explorées
- Couverture des méthodes
 - 📊 Pourcentage de méthodes qui ont été appelées
- Couverture des classes
 - 📊 Pourcentage de classes dont au moins une méthode a été appelée

Critères de couverture

Statement Coverage

- ➡ Est-ce que toutes les instructions du code ont été exécutées ?
- ➡ Est-ce qu'un statement coverage de 100% garantit que les tests sont complets ?

```
1 int addAndCount(List list) {  
2     if (list != null) {  
3         list.add("Sample text");  
4     }  
5     return list.size();  
6 }
```

🚩 Un Test avec `list != null` donne une couverture de 100%

🚩 Et si `list == null`??

Avertissement

Ne pas écrire des cas de tests juste pour satisfaire votre outil de couverture de code

Critères de couverture

Decision Coverage

- ➡ Est-ce que les conditions dans les structures de contrôle ont été évaluées à true et false ?
- ➡ Similaire : Branch coverage

Path Coverage

- ➡ Est-ce que tous les chemins d'exécution possibles au sein de chaque méthode ont été empruntés ?

```
1  if (A) {  
2      if (B) {}  
3  }  
4  if (C) {  
5  }
```

- ➡ Le nombre de chemin augmente de manière exponentiel avec le nombre de branches. (10 `if` impliquent 1024 chemins possibles)
- ➡ Parfois tous les chemins ne sont pas atteignables.

L'outil JaCoCo

Présentation générale




- JaCoCo (Java Code Coverage) est une bibliothèque de couverture de code Java gratuite distribuée sous la licence publique Eclipse (open source)
- Il peut être installé depuis Eclipse Market Place ou bien intégré à l'aide de Maven
- Mesures de couverture supportées
 - ☞ Instruction (bytecode instruction)
 - ☞ Branches
 - ☞ Lines
 - ☞ Methods
 - ☞ Classes (si au moins une méthode est exécutée)
 - ☞ Cyclomatic Complexity (Mesure de la complexité de la structure d'une fonction, différent du nombre de branches et du nombre de chemins)
- Site officiel : depuis <http://www.eclemma.org/jacoco>

L'outil JaCoCo

Installation

- ➡ Aller au site www.mvnrepository.com et récupérer les méta-données relatives au plugin JaCoCo

**JaCoCo :: Maven Plugin » 0.8.6**

The JaCoCo Maven Plugin provides the JaCoCo runtime agent to your tests and allows basic report creation.

License	EPL 2.0
Categories	Maven Plugins
Date	(Sep 15, 2020)
Files	maven-plugin (52 KB) View All
Repositories	Central
Used By	59 artifacts

[Maven](#) [Gradle](#) [SBT](#) [Ivy](#) [Grape](#) [Leiningen](#) [Buildr](#)


```
<dependency>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.6</version>
</dependency>
```

- ➡ Insérer les méta-données dans le fichier `POM.xml` et recompiler le projet

L'outil JaCoCo

Installation

- ➡ Aller au site www.mvnrepository.com et récupérer les méta-données relatives au plugin JaCoCo

**JaCoCo :: Maven Plugin » 0.8.6**

The JaCoCo Maven Plugin provides the JaCoCo runtime agent to your tests and allows basic report creation.

License	EPL 2.0
Categories	Maven Plugins
Date	(Sep 15, 2020)
Files	maven-plugin (52 KB) View All
Repositories	Central
Used By	59 artifacts

[Maven](#) [Gradle](#) [SBT](#) [Ivy](#) [Grape](#) [Leiningen](#) [Buildr](#)

```
<dependency>
  <groupId>org.jacoco</groupId>
  <artifactId>jacoco-maven-plugin</artifactId>
  <version>0.8.6</version>
</dependency>
```

- ➡ Insérer les méta-données dans le fichier `pom.xml` et recompiler le projet

L'outil JaCoCo

Installation

➡ Exécuter JaCoCo à partir de menu contextuel Coverage As -> JUnit Test

```
3 public class Palindrome {
4     public boolean isPalindrome(String inputString) {
5         if (inputString.length() == 0) {
6             return true;
7         } else {
8             char firstChar = inputString.charAt(0);
9             char lastChar = inputString.charAt(inputString.length() - 1);
10            String mid = inputString.substring(1, inputString.length() - 1);
11            return (firstChar == lastChar) && isPalindrome(mid);
12        }
13    }
14 }
```

Element	Coverage	Covered Instruction	Missed Instruction
└ j3exemple3	40,0 %	20	30
└ src/main/java	21,1 %	8	30
└ j3exemple3	21,1 %	8	30
└ Palindrome.java	21,1 %	8	30
└ Palindrome	21,1 %	8	30
└ isPalindrome(String)	14,3 %	5	30
└ src/test/java	100,0 %	12	0
└ j3exemple3	100,0 %	12	0
└ TestPalindrome.java	100,0 %	12	0

L'outil JaCoCo

Analyse d'un rapport

- ➡ Les rapports JaCoCo aident à analyser visuellement la couverture de code en utilisant des diamants avec des couleurs pour les branches et des couleurs d'arrière-plan pour les lignes
 - 🔴 Le losange rouge signifie qu'aucune branche n'a été exercée pendant la phase de test
 - 🟡 Le diamant jaune indique que le code est partiellement couvert ; certaines branches n'ont pas été exercées
 - 🟢 Le diamant vert signifie que toutes les branches ont été exercées au cours de la tester

L'outil JaCoCo

Exemple illustratif

- ➡ Afin d'atteindre une couverture de code à 100%, nous devons introduire des tests, qui couvrent les parties manquantes.

```
1  @Test
2  public void whenPalindrom__thenAccept() {
3      Palindrome palindromeTester = new Palindrome();
4      assertTrue(palindromeTester.isPalindrome("noon"));
5  }
6
7  @Test
8  public void whenNearPalindrom__thanReject() {
9      Palindrome palindromeTester = new Palindrome();
10     assertFalse(palindromeTester.isPalindrome("neon"));
11 }
```

L'outil JaCoCo

Installation

- ➡ Maintenant, toutes les lignes/branches/chemins de notre code sont entièrement couverts

The screenshot shows an IDE with two tabs: `Palindrome.java` and `TestPalindrome.java`. The `Palindrome.java` file contains the following code:

```
3 public class Palindrome {
4     public boolean isPalindrome(String inputString) {
5         if (inputString.length() == 0) {
6             return true;
7         } else {
8             char firstChar = inputString.charAt(0);
9             char lastChar = inputString.charAt(inputString.length() - 1);
10            String mid = inputString.substring(1, inputString.length() - 1);
11            return (firstChar == lastChar) && isPalindrome(mid);
12        }
13    }
14 }
```

Below the code editor, the `Coverage` tab is active, displaying the `TestPalindrome` report for the date and time 19 sept. 2020 11:11:35. The report table is as follows:

Element	Coverage	Covered Instruction	Missed Instruction
└ j3example3	100,0 %	68	0
└ src/main/java	100,0 %	38	0
└ j3example3	100,0 %	38	0
└ Palindrome.java	100,0 %	38	0
└ Palindrome	100,0 %	38	0
└ isPalindrome(String)	100,0 %	35	0
└ src/test/java	100,0 %	30	0

Plan

- 1 Les techniques d'écriture de tests
- 2 Tests unitaires et couverture du code
- 3 Test du code hérité**
- 4 Les tests fonctionnels avec Cucumber
- 5 Simplification de l'écriture des tests
- 6 Tests fonctionnels avec FitNesse
- 7 Selenium et JUnit

Test du code hérité

C'est quoi le code hérité (legacy code) ?

- ➡ Du code legacy, c'est du code :
 - 👉 Sans tests unitaires (Déf. Michael Feathers)
 - 👉 Plus vieux que ceux y travaillent
 - 👉 Modification en mode patchage
 - 👉 Nécessitant un guide de marais pour s'y retrouver

Test du code hérité

Problèmes avec le code hérité

- C'est un code qui ne possède pas un test suite alors il est difficile à maintenir.
- Il est parfois désordonné, plein d'erreurs
- Utilise des anciens API et Framework.
- Il utilise parfois des anciens systèmes qui ne sont plus utilisés

Test du code hérité

Exemple de test du code hérité

- Soit le code modélisant une machine à café qui fabrique différentes boissons à partir d'ingrédients définis.
- L'initialisation des recettes pour chaque boisson doit être codée, bien qu'il devrait être relativement facile d'ajouter de nouvelles boissons.
- La machine doit afficher le stock d'ingrédients (+ coût) et le menu au démarrage, et après chaque entrée utilisateur valide.
- Le coût de la boisson est déterminé par la combinaison des ingrédients. Par exemple, le café est 3 unités de café (75 cents par), 1 unité de sucre (25 cents par), 1 unité de crème (25 cents par).
- Les ingrédients et les éléments du menu doivent être imprimés par ordre alphabétique. Si la boisson est en rupture de stock, elle doit s'imprimer en conséquence.
- Si la boisson est en stock, elle doit afficher "Distribution:". Pour sélectionner une boisson, l'utilisateur doit saisir un numéro approprié. S'ils soumettent `nrz` ou `nRz`, les ingrédients doivent se réapprovisionner, et `nqz` ou `nQz` doivent cesser. Les lignes vides doivent être ignorées et une entrée non valide doit imprimer un message d'entrée non valide.

Test du code hérité

Code Java de la machine à café

- On a trois classes : Ingredient, Drink et DrinkMachine
- Extrait du code Java de la classe DrinkMachine

```
1 public class DriveMachine {
2     public static void startIO() {
3         BufferedReader reader=new BufferedReader(new InputStreamReader(System.in));
4         String input=""; int compteur=0;
5         while(true) {
6             try {
7                 input=reader.readLine().toLowerCase();
8                 if (input.equals(""))
9                     continue;
10                else if (input.equals("q"))
11                    System.exit(0);
12                else if (input.equals("r")) {
13                    compteur=0;
14                    System.out.println("Compteur :"+compteur);
15                }
16                else if (Integer.parseInt(input)>=1 && Integer.parseInt(input)<=10) {
17                    compteur++;
18                    System.out.println("Compteur :"+compteur);
19                }
20                else throw new IOException();
21            }
22            catch (Exception e) {
23                System.out.println("Invalid selection "+input+"\n");
24            }
25        }
26    }
27    public static void main(String[] args) {
28        System.out.println("Effectuer un choix : q,r , 1-10 ");
29        startIO();
30    }
31 }
```

Test du code hérité

Exemple de test du code hérité

- On doit d'abord examiner et observer le fonctionnement de programme en testant les différentes entrées
- Exécuter le programme et essayer différentes entrées

```
DrinkMachine [Java Application] /usr/lib/jvm/java-11-openjdk-amd64/bin/java (11 sept. 2020 à 00:30:09)
```

```
Inventory:
```

```
Cocoa,10
```

```
Coffee,10
```

```
Cream,10
```

```
Decaf Coffee,10
```

```
Espresso,10
```

```
Foamed Milk,10
```

```
Steamed Milk,10
```

```
Sugar,10
```

```
Whipped Cream,10
```

```
Menu:
```

```
1,Caffe Americano,€3,30,true
```

```
2,Caffe Latte,€2,55,true
```

```
3,Caffe Mocha,€3,35,true
```

```
4,Cappuccino,€2,90,true
```

```
5,Coffee,€2,75,true
```

```
6,Decaf Coffee,€2,75,true
```


Test du code hérité

Exemple de test du code hérité

- ➡ On doit ajouter un test case pour commencer les tests : File->New->JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

☐ New JUnit 3 test ☐ New JUnit 4 test ☒ **New JUnit Jupiter test**

Source folder:

Package:

Name:

Superclass:

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()
☐ setUp() ☐ tearDown()
☐ constructor

Do you want to add comments? (Configure templates and default value [here](#))

☐ Generate comments

Class under test:

⚠ JUnit 5 requires a Java 8 project. [Configure](#) project compliance and the project build path.

Tester du code hérité

Code Java de la machine à café

➡ On teste la méthode `main` de la classe `DrinkMachine`

```
1 package drinkmachine;
2 import static org.junit.jupiter.api.Assertions.*;
3 import org.junit.jupiter.api.Test;
4
5 class DrinkMachinTest {
6     @Test
7     void testMain() {
8         DrinkMachine.main(new String[] {});
9     }
10 }
```

☛ Le teste ne cesse de tourner sans donner la main pour vérifier les résultats !!!

Tester du code hérité

Code Java de la machine à café

- ➡ L'envoi des données au programme s'effectue à travers `System.setIn()`, et la récupération des résultats (affichage) s'effectue à travers la fonction `System.setOut()`

```
1 void testMain() {  
2     byte[] bytes="q\n".getBytes();  
3     System.setIn(new ByteArrayInputStream(bytes));  
4     ByteArrayOutputStream output=new ByteArrayOutputStream();  
5     System.setOut(output);  
6     DrinkMachine.main(new String[] {});  
7     assertEquals("", output);  
8 }
```

- ➡ Afin d'éviter que le programme se termine sans pouvoir récupérer les résultat, il faut remplacer dans la classe `DrinkMachine` l'appel à `System.exit(0)` par `break`

Tester du code hérité

Code Java de la machine à café

➡ La comparaison des résultats d'affichage est effectuée en utilisant `assertThat`

```
1  assertThat(output.toString()).isEqualToNormalizingNewLines("Inventory:\r\n"+
2  "\r\n"+
3  "Cocoa,10\r\n"+
4  "\r\n"+
5  "Coffee,10\r\n"+
6  "\r\n"+
7  "Cream,10\r\n"+
8  "\r\n"+
9  "Decaf Coffee,10\r\n"+
10 "\r\n"+
11 "Espresso,10\r\n"+
12 "\r\n"+
13 "Foamed Milk,10\r\n"+
14 "\r\n"+
15 "Steamed Milk,10\r\n"+
16 "\r\n"+
17 "Sugar,10\r\n"+
18 "\r\n"+
19 "Whipped Cream,10\r\n"+
20 ...
```

Plan

- 1 Les techniques d'écriture de tests
- 2 Tests unitaires et couverture du code
- 3 Test du code hérité
- 4 Les tests fonctionnels avec Cucumber
 - Présentation de Cucumber
 - Paramétrage des étapes
- 5 Simplification de l'écriture des tests
- 6 Tests fonctionnels avec FitNesse
- 7 Selenium et JUnit

Cucumber en bref

Présentation générale

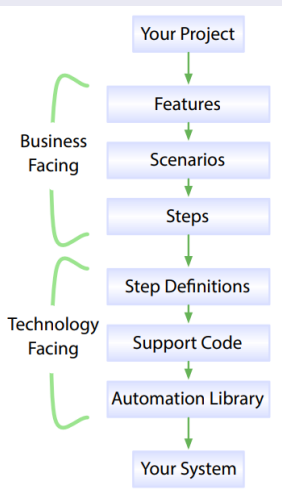


- ➡ Cucumber est initialement écrit dans le langage de programmation Ruby
 - 📖 Plusieurs implémentations ont permis de le porter pour d'autres langages
- ➡ Livrés en deux versions
 - 📖 Cucumber open : version open source
 - 📖 Cucumber Pro : version commerciale. Elle fournit, de plus, des outils collaboratifs

- └ Les tests fonctionnels avec Cucumber
- └ Présentation de Cucumber

Tests avec Cucumber

Fonctionnement de Cucumber



Prise en main de Cucumber

Intégration de Cucumber dans un projet



- Installer Cucumber Plugin à partir de Marketplace dEclipse
- Créer un projet Maven
- À partir du site `www.mvnrepository.com`, récupérer les méta-données de dépendances
 - 🔗 La bibliothèque du framework Cucumberă: `cucumber-java`
 - 🔗 La bibliothèque permettant d'utiliser JUnit 4ă: `cucumber`

Tests avec Cucumber

Exemple de code métier à tester

➡ On considère le code métier donné par la classe suivante :

```
1  public class Order {
2      private String from;
3      private String to;
4      private String message;
5      private List<String> contents = new ArrayList<String>();
6
7      public void declareOwner(String start) {
8          this.from = start;
9      }
10
11     public void declareTarget(String dest) {
12         this.to = dest;
13     }
14
15     public List<String> getCocktails() {
16         return contents;
17     }
18
19     public void withMessage(String something) {
20         this.message = something;
21     }
22
23     public String getTicketMessage() {
24         return "From " + from + " to " + to + ": " + message;
25     }
26 }
```

Tests avec Cucumber

Cucumber et Gherkin

- ➡ Cucumber lit les spécifications à partir d'un fichier texte
- ➡ Le fichier de spécifications est écrit selon la syntaxe de l'outil Gherkin (cornichon)
 - 📄 Un tel fichier porte l'extension `feature`
- ➡ Gherkin supporte plusieurs langues tel que le Français
- ➡ Cucumber exécute des scripts par les fichiers `feature`

Tests avec Cucumber

Écriture d'un fichier *feature*

- ➡ Exemple de spécification d'une fonctionnalité (une story)

Titre : Cocktail Ordering

Description : As Romeo, I want to offer a drink to Juliette so that we can discuss together (and maybe more).

- ➡ On utilise le mot clé **Feature** pour décrire une fonctionnalité

Feature: Cocktail Ordering

As Romeo, I want to offer a drink to Juliette so that we can discuss together (and maybe more).

Tests avec Cucumber

Écriture d'un fichier `feature`

- Une story est composée de scénarios et chaque scénario est composé d'étapes.
- Chaque scénario est introduit par le mot clé `Scenario`. Ce mot clé peut être suivi ou non d'un titre qui décrit explicitement le critère d'acceptation de la story associée à ce scénario,
- Un scénario étant un exemple concret qui illustre une règle métier, il est composé de plusieurs étapes.
- Les différentes étapes d'un scénario sont décrites à partir des trois principaux mots clés :
Given, When et Then
 - 📖 Given décrit les conditions initiales du scénario, c.-à-d. le contexte dans lequel va se dérouler le scénario,
 - 📖 When décrit une action effectuée par un utilisateur, c.-à-d. un événement qui va réellement déclencher le scénario,
 - 📖 Then décrit le comportement attendu, ce qui devrait se produire lorsque les conditions initiales sont remplies et l'action est effectuée

Tests avec Cucumber

Écriture d'un fichier feature

➡ Exemple de scénario

Scenario: Creating an empty order

Given Romeo who wants to buy a drink

When an order is declared for Juliette

Then there is no cocktail in the order

Tests avec Cucumber

Écriture d'un fichier `feature`

- Créer un répertoire `src/test/resources` dans lequel on place le fichier `cocktail.feature`
- Le plugin Cucumber permet reconnait le fichier créé et permet son édition directement à partir d'Eclipse
- Le contenu complet du fichier `cocktail.feature`
`#Author: chiheb.abid@gmail.com`

Feature: Cocktail Ordering

As Romeo, I want to offer a drink to Juliette so that we can discuss together (and maybe more).

Scenario: Creating an empty order

Given Romeo who wants to buy a drink

When an order is declared for Juliette

Then there is no cocktail in the order

Tests avec Cucumber

Configurer le lanceur de Test

- Il s'agit de créer une classe permettant de lancer le test

```
1 import io.cucumber.junit.Cucumber;  
2 import io.cucumber.junit.CucumberOptions;  
3  
4 import org.junit.runner.RunWith;  
5  
6 @CucumberOptions(features="src/test/resources",plugin="html:out.html")  
7 @RunWith(Cucumber.class)  
8 public class RunCucumberTest { }
```

- On a utilisé l'annotation `CucumberOptions` pour spécifier l'emplacement des fichiers feature, et demander la génération d'un rapport dans le fichier `out.html`
- L'annotation `RunWith` permet d'indiquer à JUnit que cette classe doit être exécutée par Cucumber
- Pour lancer les scénarios du fichier `.feature`, il suffit d'exécuter le lanceur de test `RunCucumberTest` comme un simple fichier JUnit via la commande habituelle `Run As -> JUnit Test`

Tests avec Cucumber

Implémentation du code de test des steps

- Les steps seront implémentées dans une classe enregistrée dans le répertoire `src/test/java`
- Implémenter chaque step de manière qu'elle produise le comportement attendu du scénario et puisse ainsi être exécutée en Java
- À chaque step correspond une méthode annotée par `Given`, `When` ou `Then`
- Pour que le mapping existe entre la step (.java) et l'étape d'un scénario (.feature), la valeur de l'annotation Java doit contenir la phrase décrivant l'étape du scénario à quelques caractères spéciaux près.

- └ Les tests fonctionnels avec Cucumber
- └ Présentation de Cucumber

Tests avec Cucumber

Implémentation du code de test des steps

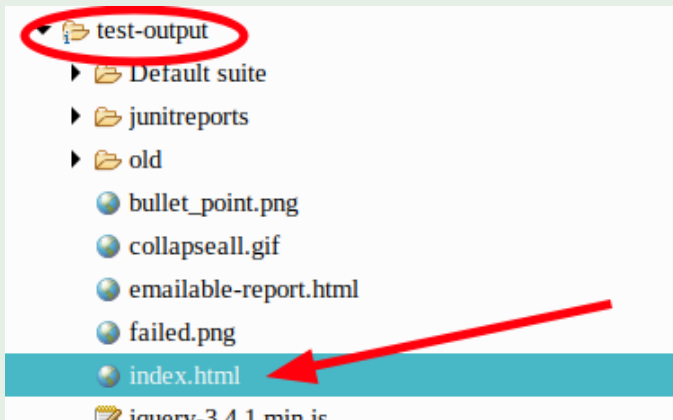
➡ Le code Java des steps pour le scénario de l'exemple

```
1  import static org.junit.Assert.assertEquals;
2  import java.util.List;
3  import io.cucumber.java.PendingException;
4  import io.cucumber.java.en.*;
5
6  public class CocktailSteps {
7      private Order order;
8      @Given("Romeo who wants to buy a drink")
9      public void romeo_who_wants_to_buy_a_drink() {
10         order = new Order();
11         order.declareOwner("Romeo");
12     }
13     @When("an order is declared for Juliette")
14     public void an_order_is_declared_for_Juliette() {
15         order.declareTarget("Juliette");
16     }
17     @Then("there is no cocktail in the order")
18     public void there_is_no_cocktail_in_the_order() {
19         List<String> cocktails = order.getCocktails();
20         assertEquals(0, cocktails.size());
21     }
22 }
```

Tests avec Cucumber

Exécution d'une story

- Exécuter la story avec JUnit à travers le lanceur de test `RunCucumberTest` via la commande habituelle `Run As -> JUnit Test`
- Vérifier que le test s'est déroulé correctement : on est au vert
- Pour consulter le rapport Cucumber, consulter le fichier `out.html` généré dans le projet



Paramétrage des étapes

Paramétrage des étapes

- L'utilisation des paramètres dans les steps permet d'éviter la duplication du code
- Le paramétrage des steps s'effectue en utilisant des expressions Cucumber ou régulières pour matcher (détecter) un motif dans la description textuelle d'une étape
- Par défaut, l'utilisation des expressions Cucumber est activée. Pour activer les expressions régulières, il faut délimiter la step avec ^ et \$
- Exemples d'expressions régulières utilisées par Cucumber
 - 📄 (. *) : matcher un string
 - 📄 (\\d+) : matcher un entier
 - 📄 ^pattern : matcher une chaîne qui commence par pattern
 - 📄 pattern\$: matcher une chaîne qui se termine par pattern
- Exemples d'expressions Cucumber
 - 📄 {string} : matcher un string
 - 📄 {int} : matcher un entier
 - 📄 {float} : matcher un réel
 - 📄 { } : matcher n'importe quoi

Paramétrage des étapes

Paramétrage des étapes

- ➡ On souhaite impliquer d'autres personnes que Romeo et Juliette et paramétrer le nombre de cocktails passés en commande

```
1 Feature Cocktail Ordering
2
3 As Romeo, I want to offer a drink to Juliette so that we can discuss together (and maybe more).
4
5 Scenario Creating an empty order
6   Given "Romeo" who wants to buy a drink
7   When an order is declared for "Juliette"
8   Then there is 0 cocktails in the order
9
10 Scenario Creating another new empty order
11   Given "Tom" who wants to buy a drink
12   When an order is declared for "Jerry"
13   Then there is 0 cocktails in the order
```

- └ Les tests fonctionnels avec Cucumber
 - └ Paramétrage des étapes

Paramétrage des étapes

Paramétrage des étapes

- ➡ Au code Java, dans les steps on spécifie dans les annotations l'utilisation des expressions régulières ou Cucumber, et il faut ajouter les arguments dans les méthodes

```
1  @Given("^"(.*)\" who wants to buy a drink$")
2  public void romeo_who_wants_to_buy_a_drink(String romeo) {
3      order = new Order();
4      order.declareOwner(romeo);
5  }
6  @When("an order is declared for {string}")
7  public void an_order_is_declared_for_Juliette(String juliette) {
8      order.declareTarget(juliette);
9  }
10 @Then("there is {int} cocktails in the order")
11 public void there_is_nb_cocktails_in_the_order(int nb) {
12     List<String> cocktails = order.getCocktails();
13     assertEquals(nb, cocktails.size());
14 }
```

Paramétrage d'un scénario

Paramétrage d'un scénario

- ➡ Le paramétrage d'un scénario permet d'éliminer les duplications en factorisant les descriptifs des étapes

```
1 Scenario Outline Creating an empty order
2 Given "<from>" who wants to buy a drink
3 When an order is declared for "<to>"
4 Then there is <nbCocktails> cocktails in the order
5
6 Examples
7 | from | to | nbCocktails |
8 | Romeo | Juliette | 0 |
9 | Tom | Jerry | 0 |
```

Tests avec Cucumber

Exercice

- ➡ En continuant le même exemple, vérifier les scénarios suivants :
- ➊ Romeo wants to buy a drink. An order is then declared for Juliette with a message saying "Wanna chat ?". Then, the ticket must say "From Romeo to Juliette : Wanna chat ?"
 - ➋ Romeo wants to buy a drink. An order is then declared for Jerry with a message saying "Hei !". Then, the ticket must say "From Romeo to Jerry : Hei !"

Plan

- 1 Les techniques d'écriture de tests
- 2 Tests unitaires et couverture du code
- 3 Test du code hérité
- 4 Les tests fonctionnels avec Cucumber
- 5 Simplification de l'écriture des tests**
- 6 Tests fonctionnels avec FitNesse
- 7 Selenium et JUnit

Spécification des étapes avec And et But

And et But

- ➡ And et But sont utilisées pour alléger l'écriture des étapes

Exemple sans la section Background

```
1 Scenario: Sending a message with an order
2 Given "Romeo" who wants to buy a drink
3 When an order is declared for "Juliette"
4 And a message saying "Wanna chat?" is added
5 Then the ticket must say "From Romeo to Juliette: Wanna chat?"
```

Background

Background

- ➡ La section Background dans fichier feature définit l'ensemble des étapes communes à tous les scénarios

Exemple sans la section Background

```
1 Feature: Change PIN
2 Customers being issued new cards are supplied with a Personal
3 Identification Number (PIN) that is randomly generated by the
4 system.
5 In order to be able to change it to something they can easily
6 remember, customers with new bank cards need to be able to
7 change their PIN using the ATM.
8 Scenario: Change PIN successfully
9 Given I have been issued a new card
10 And I insert the card, entering the correct PIN
11 When I choose "Change PIN" from the menu
12 And I change the PIN to 9876
13 Then the system should remember my PIN is now 9876
14 Scenario: Try to change PIN to the same as before
15 Given I have been issued a new card
16 And I insert the card, entering the correct PIN
17 When I choose "Change PIN" from the menu
18 And I try to change the PIN to the original PIN number
19 Then I should see a warning message
20 And the system should not have changed my PIN
```

Background

Exemple avec la section Background

```
1 Feature: Change PIN
2 As soon as the bank issues new cards to customers, they are
3 supplied with a Personal Identification Number (PIN) that
4 is randomly generated by the system.
5 In order to be able to change it to something they can easily
6 remember, customers with new bank cards need to be able to
7 change their PIN using the ATM.
8 Background:
9 Given I have been issued a new card
10 And I insert the card, entering the correct PIN
11 And I choose "Change PIN" from the menu
12 Scenario: Change PIN successfully
13 When I change the PIN to 9876
14 Then the system should remember my PIN is now 9876
15 Scenario: Try to change PIN to the same as before
16 When I try to change the PIN to the original PIN number
17 Then I should see a warning message
18 And the system should not have changed my PIN
```

Table de données pour Gherkin

Table de données

- ➡ Les tables de données permettent de rassembler les données de Given, When ou Then dans des tableaux

Exemple sans table de données

- ➡ Sans les tables de données

```
1 Given a User "Michael Jackson" born on August 29, 1958
2 And a User "Elvis" born on January 8, 1935
3 And a User "John Lennon" born on October 9, 1940
```

- ➡ Avec une table de données

```
1 Given these Users:
2 | name | date of birth |
3 | Michael Jackson | August 29, 1958 |
4 | Elvis | January 8, 1935 |
5 | John Lennon | October 9, 1940 |
```

Table de données pour Gherkin

Exemple illustratif avec les tables de données

- On considère le développement du jeu tic-tac-toe
- L'objectif est d'illustrer la lecture des données à partir des tables de données
- Le contenu du fichier `board.feature`

```
1 Feature: First move
2 Scenario: First move in an empty board
3 Given a board like this:
4 | | 1 | 2 | 3 |
5 | 1 | | | |
6 | 2 | | | |
7 | 3 | | | |
8
9 When player x plays in row 2, column 1
10 Then the board should look like this:
11 | | 1 | 2 | 3 |
12 | 1 | | | |
13 | 2 | x | | |
14 | 3 | | | |
```

Table de données pour Gherkin

Exemple illustratif avec les tables de données

➡ N'oublier pas de créer la classe de lanceur de test Cucumber

➡ Le code Java des steps

```
1  public class BoardSteps {
2      private List<List<String>> board;
3
4      @Given("^a board like this:$")
5      public void aBoardLikeThis(DataTable table) throws Throwable {
6          this.board = table.asLists();
7      }
8
9      @When("^player x plays in row (\\d+), column (\\d+)$")
10     public void playerXPlaysInRowColumn(int arg1, int arg2) throws Throwable {
11         System.out.println(board.toString());
12     }
13
14     @Then("^the board should look like this:$")
15     public void theBoardShouldLookLikeThis(DataTable arg1) throws Throwable {
16         this.board = arg1.asLists();
17         System.out.println(board.toString());
18     }
19 }
```

Table de données pour Gherkin

Exercice

- ➡ Améliorer les résultats affichés sur la console pour avoir un affichage similaire au suivant :

```
<terminated> RunCucumberTest (1) [JUnit] /usr/lib/jvm/java-11-op  
Table When  
|   | 1 | 2 | 3 |  
| 1 |   |   |   |  
| 2 |   |   |   |  
| 3 |   |   |   |  
Table Then  
|   | 1 | 2 | 3 |  
| 1 |   |   |   |  
| 2 | x |   |   |  
| 3 |   |   |   |
```

Les tags

Les tags

- ➡ Gherkin permet d'ajouter des tags aux scénarios
 - 🔖 Rédiger une classe de tests qui n'exécutera que les scénarios identifiés par un ou plusieurs tags particuliers
 - 🔖 Simplifier la recherche des scénarios avec le plugin Cucumber
- ➡ Pour ajouter un tag, il suffit de mettre @tag juste avant le mot clé Feature ou Scenario
- ➡ Pour filtrer l'exécution des tags, on utilise le paramètre tags dans @CucumberOptions

```
@CucumberOptions(tags= "@nouveau and not @old")
```


Travailler avec d'autres langues

La description des steps en français

- Gherkin supporte l'utilisation de diverses langues
- L'objectif est de simplifier l'écriture des tests selon la langue utilisée par le client
- Pour utiliser la langue française, il suffit d'ajouter le commentaire suivant dans le fichier .feature

```
1 # language: fr
```

- Les correspondances des mots clé de Gherkin

feature => Fonctionnalité

background => Contexte

scenario => Scénario scenario outline => Plan du scénario,
Plan du Scénario

examples => Exemples

given => " * ", "Soit ", "Etant donné que ", "Etant donné
qu' ", "Etant donné ", "Etant donnée ", "Etant donnés ", etc.

when => " * ", "Quand ", "Lorsque ", "Lorsqu' "

then => " * ", "Alors "

and => " * ", "Et que ", "Et qu' ", "Et "

but => " * ", "Mais que ", "Mais qu' ", "Mais "

Travailler avec d'autres langues

Exercice

- ➡ Refaire l'exemple précédent, dans un nouveau projet, en utilisant une recette écrite en français
- ➡ On utilise le même code métier

Plan

- 1 Les techniques d'écriture de tests
- 2 Tests unitaires et couverture du code
- 3 Test du code hérité
- 4 Les tests fonctionnels avec Cucumber
- 5 Simplification de l'écriture des tests
- 6 Tests fonctionnels avec FitNesse**
- 7 Selenium et JUnit

Tests d'acceptation / tests unitaires

Tests d'acceptation / tests unitaires

➡ Tests unitaires

- 📖 Orientés **petite unité de code**
- 📖 Compréhension réservée aux développeurs

Tests unitaires = Est-ce que j'ai écrit le code correctement ?

➡ Tests d'acceptation (Tests de recette)

- 📖 Orientés **fonctionnalité**
- 📖 Compréhension accessible au client

Tests d'acceptation = Est-ce que j'ai écrit le bon code comme demandé par le client ?

FitNesse en bref

Présentation générale



- Un outil type wiki permettant de spécifier des tests d'acceptation au milieu de texte informel
- Serveur autonome, donc facile à installer
- Il est basé sur le Framework de test intégré de Ward Cunningham et est conçu pour prendre en charge les tests d'acceptation plutôt que les tests unitaires
 - 📖 Il facilite la création d'une description lisible et détaillée de la fonction du système
- Développé en langage Java
 - 📖 Fourni et expédié sous la forme d'un fichier jar exécutable unique.
 - 📖 À télécharger depuis <http://www.fitnesse.org/>

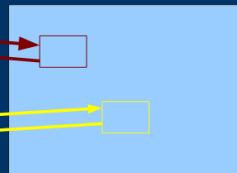
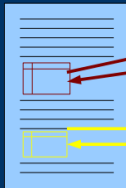
FitNesse en bref

Fonctionnement général

Document FitNesse

Fixtures (code java)

Application (java)



FitNesse en bref

Installation de FitNesse

- ❶ Récupérer l'archive (`fitnessse.jar`) et mettre le jar là où l'on veut installer le logiciel
 - 📄 `http://fitnessse.org/FitNesseDownload`
- ❷ Exécuter une première fois l'archive
 - 📄 `java -jar fitnessse.jar`
- ❸ Exécuter une deuxième fois l'archive
 - 📄 Soit : `java -jar fitnessse.jar` : dans ce cas le port standard 80 est choisi
 - 📄 Soit : `java -jar fitnessse.jar -pXXXX` : le port XXXX est choisi
- ❹ Lancer son navigateur et se connecter sur la bonne machine et le bon port
 - 📄 `http://localhost` ou `http://localhost:XXXX`

Utilisation de FitNesse

Exemple illustratif

On se propose de vérifier le bon fonctionnement d'une calculatrice permettant de réaliser les opérations de soustraction et d'addition.

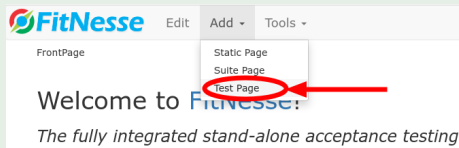
Nous supposons ainsi qu'on a défini la classe `Calculator` qui contient volontairement une erreur dans la méthode `minus`.

```
1 public class Calculator {  
2     public int add(int a, int b) {  
3         return a + b;  
4     }  
5     public int minus(int intToBeSubtracted, int minus) {  
6         return intToBeSubtracted + minus;  
7     }  
8 }
```


Utilisation de FitNesse

Exemple illustratif

- Accéder à FitNesse à travers le navigateur et créer une nouvelle page de test.



- Nommer cette page CalculatorTests et y ajouter le contenu suivant :
!contents -R2 -g -p -f -h
Définit que le test doit être exécuté avec SLIM
!define TEST_SYSTEM {slim}

Utilisation de FitNesse

Exemple illustratif

- ➡ Accéder à la page de test créée en spécifiant comme adresse `http://localhost/FrontPage.CalculatorTests`, et y ajouter une page fille nommée `CalculatorTest`. Cette dernière contient le code wiki pour les spécifications sous forme d'un tableau.

```
|Add two number |  
|first int|second int|result of addition?|  
|0 |1 |1 |  
|2 |0 |2 |  
|2 |3 |5 |  
|Subtract an integer to another |  
|int to be subtracted|minus|result of subtraction?|  
|0 |1 |-1 |  
|4 |2 |2 |  
|3 |1 |2 |
```

Utilisation de FitNesse

Exemple illustratif

- ➡ On crée les classes (custom fixtures) permettant d'effectuer la liaison entre le code wiki et la classe à tester
- ❶ Cette première classe est utilisée comme fixture pour la fonctionnalité d'addition

```
1 public class AddTwoNumber {
2     private int firstInt;
3     private int secondInt;
4     private Calculator calculator;
5     public AddTwoNumber() {
6         calculator = new Calculator();
7     }
8     public void setFirstInt(int firstInt) {
9         this.firstInt = firstInt;
10    }
11    public void setSecondInt(int secondInt) {
12        this.secondInt = secondInt;
13    }
14    public int resultOfAddition() {
15        return calculator.add(firstInt, secondInt);
16    }
17 }
```

Utilisation de FitNesse

Exemple illustratif

② Cette deuxième classe est utilisée comme fixture pour la fonctionnalité de soustraction

```
1 public class SubtractAnIntegerToAnother {
2     private int intToBeSubtracted;
3     private int minus;
4     private Calculator calculator;
5     public SubtractAnIntegerToAnother() {
6         calculator = new Calculator();
7     }
8     public void setIntToBeSubtracted(int intToBeSubtracted) {
9         this.intToBeSubtracted = intToBeSubtracted;
10    }
11    public void setMinus(int minus) {
12        this.minus = minus;
13    }
14    public int resultOfSubtraction() {
15        return calculator.minus(intToBeSubtracted, minus);
16    }
17 }
```

Utilisation de FitNesse

Exemple illustratif

- ➡ Avant d'effectuer les tests, il est nécessaire de spécifier dans la page de configuration `CalculatorTests` le chemin vers les fichiers binaires de notre système à tester en ajoutant le code wiki suivant :
`!path <chemin des fichiers binaires>`
- ➡ Aller à la page `CalculatorTest` et exécuter les tests

Utilisation de FitNesse

Avantages

- ✓ Fitnessse est un outil collaboratif. D'un coté, les analystes et les clients peuvent écrire la spécification du logiciel dans Fitnessse à l'aide du code Wiki, de l'autre coté les développeurs et les testeurs peuvent écrire les Custom Fixtures pour faire la liaison entre les spécifications et le système à tester.
- ✓ Les tests sont lisibles par tous. Analystes, clients, développeurs,
- ✓ Il n'y a pas de redondance d'information, les spécifications sont les tests.
- ✓ Tout ceci étant basé sur un serveur, la spécification ainsi que les tests sont donc toujours à jour, et toujours disponibles.
- ✓ Fitnessse gère les versions. Il est possible de revenir en arrière sur un test.

Utilisation de FitNesse

Inconvénients

- ✗ L'interface est assez laide, et vieillotte.
- ✗ Il pourrait être pratique de spécifier un certain nombre de paramètre via l'interface au lieu que ce soit fait dans du code Wiki (Le type de test (FIT ou SLIM), le path pour les fichiers binaires, le path pour les JARs externes,).
- ✗ Il n'y a pas d'éditeur WYSIWYG intégré à Fitnessse pour l'écriture des tests. Il existe cependant un plugin-in plus ou moins buggé pour remédier à cela.
- ✗ On fait souvent des fautes de frappe entre le nom des entêtes dans le Wiki et le nom des fonctions ou des classes dans le code JAVA.
- ✗ L'intégration avec `maven` est très délicate. Il n'y a pas de solution officielle, chacun y va de sa solution plus ou moins bancale

MERCI POUR VOTRE ATTENTION



Questions ?