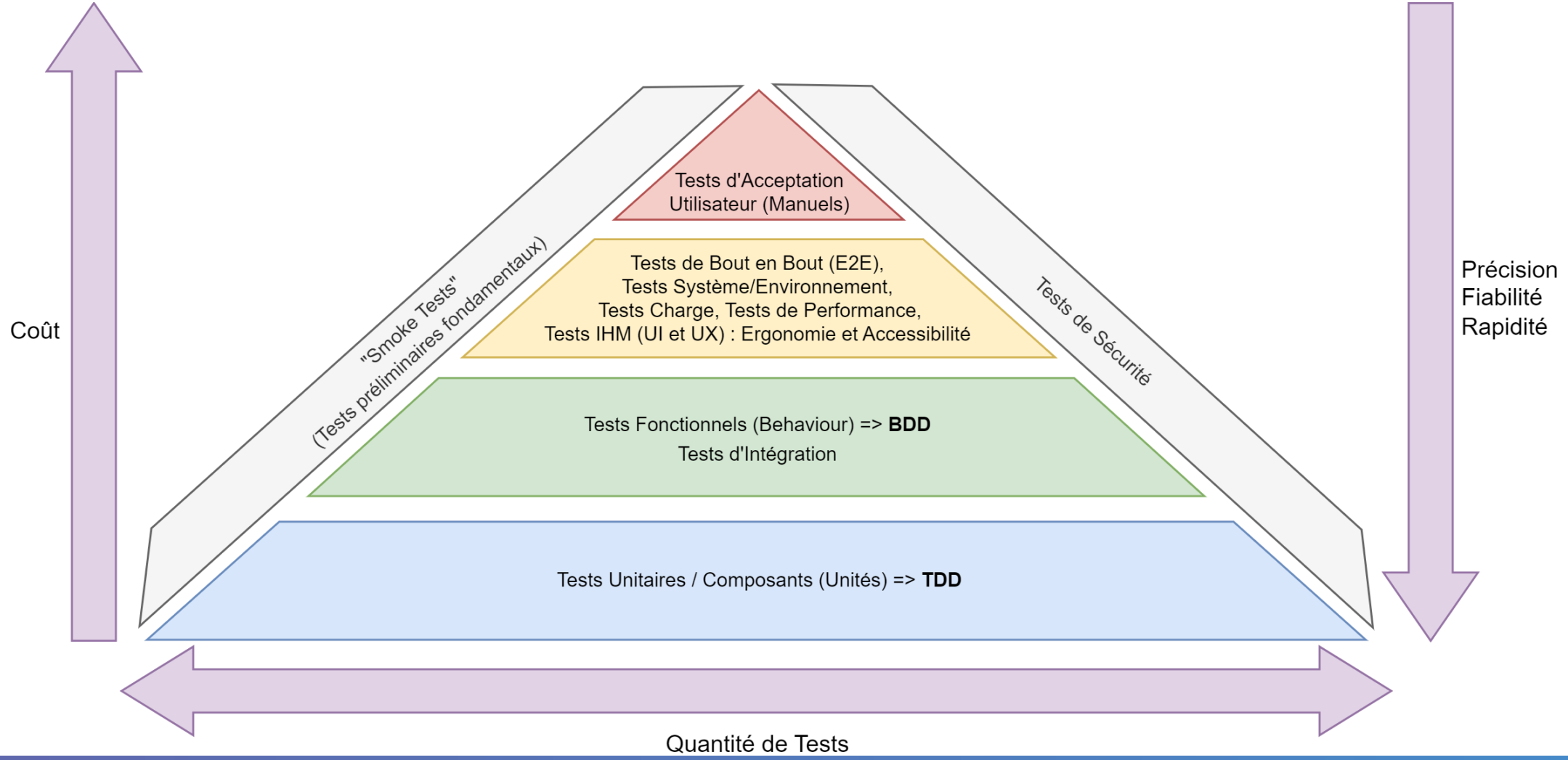
UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Jour 3

Pyramide des tests



Exemple d'outils de tests en Java

Outil	Type de tests
JUnit	Tests Fonctionnels Tests d'Intégration Tests Unitaires
Selenium PostMan	Tests End-to-End
Selenium Jest	Tests IHM (UI et UX) : Ergonomie et Accessibilité
JMeter	Tests Charge Tests de Performance

Clean Test

Le génie logiciel est une question de savoir-faire où nous devons écrire toutes les parties du logiciel avec le même soin, qu'il s'agisse de code de production ou de test.

- Rédiger des tests fait partie de notre savoir-faire.
  Nous ne pouvons avoir un code propre que si nous avons des tests propres.
- Un test propre se lit comme une histoire.
- Un test propre doit contenir toutes les informations nécessaires pour comprendre ce qui est testé

Clean Test

Le génie logiciel est une question de savoir-faire où nous devons écrire toutes les parties du logiciel avec le même soin, qu'il s'agisse de code de production ou de test.

- Rédiger des tests fait partie de notre savoir-faire.
  Nous ne pouvons avoir un code propre que si nous avons des tests propres.
- Un test propre se lit comme une histoire.
- Un test propre doit contenir toutes les informations nécessaires pour comprendre ce qui est testé

Écriture d'un test clean

- Le nom d'un test doit révéler le cas de test exact, y compris le système testé.
- Il doit spécifier l'exigence du cas de test aussi précisément que possible.
- L'objectif principal d'un bon nom de test est que si un test échoue, nous devrions être en mesure de récupérer la fonctionnalité cassée à partir du
  nom du test.
- Les conventions de dénomination populaires :
  - ShouldWhen : Should_ExpectedBehavior_When_StateUnderTest
    (ex : ShouldHaveUserLoggedIn_whenUserLogsIn)
  - MethodName_StateUnderTest_ExpectedBehavior
    (ex : isAdult_AgeLessThan18_False)
  - testFeatureBeingTested
    (ex : testIsNotAnAdultIfAgeLessThan18)
  - GivenWhenThen (BDD): Given_Preconditions_When_StateUnderTest_Then_ExpectedBehavior
    (ex : GivenUserIsNotLoggedIn_whenUserLogsIn_thenUserIsLoggedInSuccessfully)

Écriture d'un test clean - AAA

Le modèle **Arrange-Act-Assert** est une manière descriptive et révélatrice d'intention de structurer des cas de test. Il prescrit un ordre des opérations:

- La section **Arrange** doit contenir la logique de configuration des tests. Ici, les objets sont initialisés et préparés pour l'exécution des tests.
- La section **Act** invoque le système que nous sommes sur le point de tester. Il peut s'agir par exemple d'appeler une fonction, d'appeler une API REST ou d'interagir avec certains composants.
- La section **Assert** vérifie que l'action du test se comporte comme prévu. Par exemple, nous vérifions ici la valeur de retour d'une méthode, l'état final du test , les méthodes que le test a appelées, ou les éventuelles exceptions attendues et les résultats d'erreur

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Développement en TDD

Rappel des frameworks java pour les Test

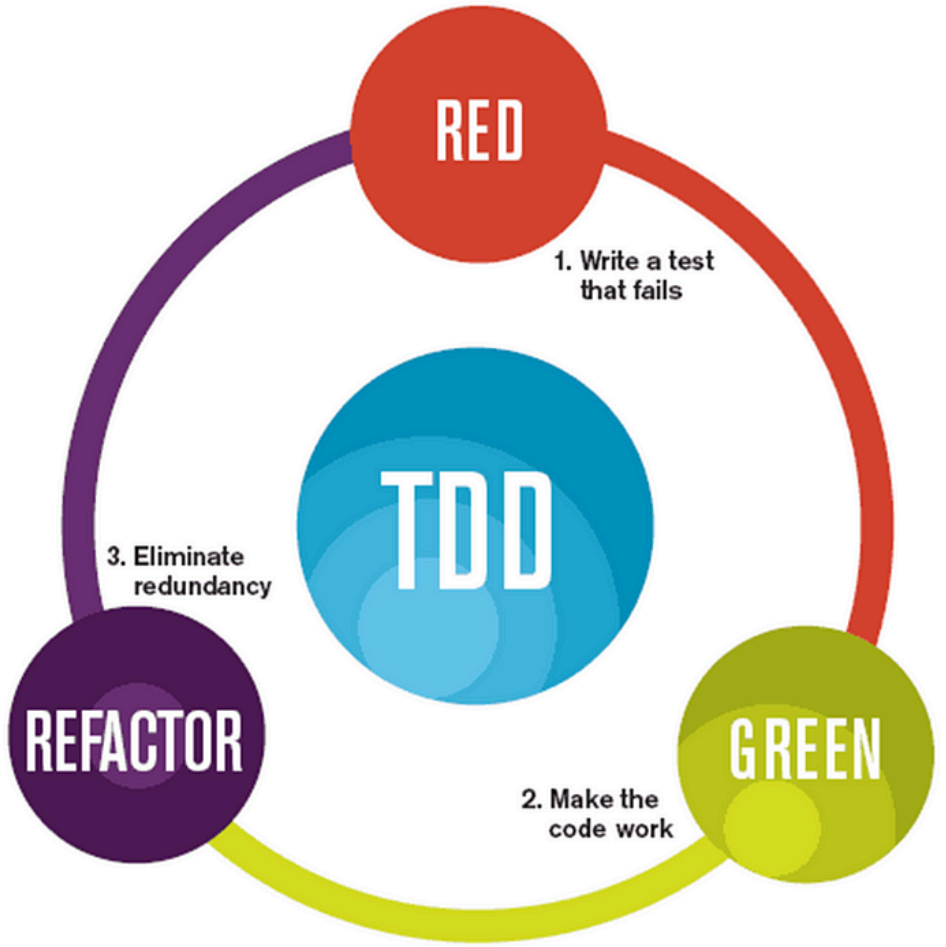
En Java, plusieurs frameworks sont disponibles pour vous aider à créer et à gérer des tests unitaires, d'intégration et fonctionnels.

- **JUnit** : C'est le framework de test le plus populaire pour Java. Il est utile pour les tests unitaires et peut être intégré à des outils d'automatisation de construction comme Maven et Gradle.
- **Mockito** : Il s'agit d'un framework de simulation (mocking) populaire en Java. Mockito est souvent utilisé en combinaison avec JUnit. Il vous permet de créer et de gérer des objets fictifs (mocks) pour simuler le comportement de classes et d'interfaces complexes.

Les paradigmes du TDD

Le **Test Driven Development (TDD)**, ou Développement Dirigé par les Tests, est une méthode de développement de logiciel qui encourage l'écriture de tests avant l'écriture du code de production. En Java, spécifiquement avec le framework Spring, le TDD suit généralement ce processus :

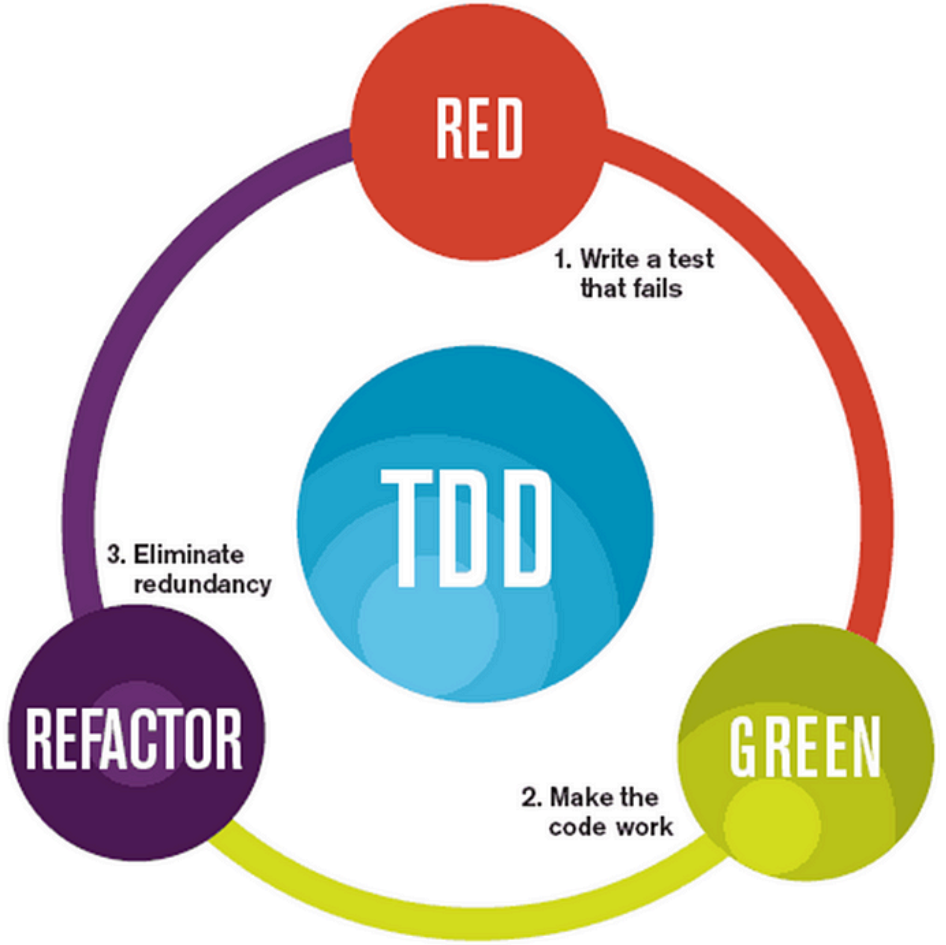
- **Écrire un test** : Vous commencez par écrire un test pour la nouvelle fonctionnalité que vous voulez implémenter. À ce stade, le test échouera, car vous n'avez pas encore écrit le code de production.
- **Exécuter tous les tests et voir si le nouveau test échoue** : Ceci est fait pour s'assurer que le nouveau test ne passe pas par accident. C'est une étape importante pour valider que le test est bien écrit et teste la bonne chose.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Les paradigmes du TDD

- **Écrire le code de production** : Vous écrivez maintenant le code qui fera passer le test. À ce stade, vous ne vous concentrez que sur le fait de faire passer le test et non sur la propreté ou l'efficacité du code.
- **Exécuter les tests** : Vous exécutez maintenant tous les tests pour vous assurer que le nouveau code de production passe le nouveau test et que tous les autres tests passent toujours.
- **Refactoriser le code** : Si les tests passent, vous pouvez alors refactoriser le code de production pour l'améliorer tout en gardant les mêmes fonctionnalités. L'objectif est d'améliorer la structure du code sans changer son comportement. Après la refactorisation, vous exécutez à nouveau les tests pour vous assurer qu'ils passent toujours.



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

Les bonnes pratiques du TDD

- **Comprendre les exigences** : Avant d'écrire vos tests, vous devez avoir une compréhension claire de ce que le code doit réaliser. Cela comprend la compréhension des exigences fonctionnelles et non fonctionnelles.
- **Garder les tests simples** : Chaque test doit être indépendant et tester une seule fonctionnalité ou un seul comportement. Cela signifie que vous devriez vous efforcer de garder vos tests aussi simples et spécifiques que possible.
- **Ne pas anticiper les fonctionnalités futures** : Lorsque vous écrivez vos tests, concentrez-vous sur la fonctionnalité actuellement requise, pas sur ce que vous pensez qu'elle pourrait être requise à l'avenir. C'est ce qu'on appelle le principe YAGNI (You Aren't Gonna Need It).
- **Écrire le code juste nécessaire pour passer le test**: Après avoir écrit un test, écrivez le code de production minimum nécessaire pour passer le test. Ne vous inquiétez pas de la perfection à ce stade, il suffit que le code passe le test.

Les bonnes pratiques du TDD

- **Refactorisation régulière** : Une fois que votre code passe les tests, prenez le temps de le refactoriser. Cela signifie le rendre plus lisible, le simplifier, éliminer les duplications, etc. N'oubliez pas de réexécuter vos tests après chaque refactorisation pour vous assurer que rien n'a été brisé.
- **Faire des tests automatisés** : Les tests doivent être automatisés afin qu'ils puissent être exécutés à chaque modification du code. Cela vous aidera à attraper les bugs tôt et facilitera l'intégration continue.
- **Mise en place des doubles de test (test doubles)** : Utilisez des bouchons (stubs), des mocks et des objets factices (dummy objects) pour isoler le code que vous testez. Cela vous permet de concentrer vos tests sur le code que vous écrivez, plutôt que sur ses dépendances.
- **Exécuter les tests régulièrement** : Les tests doivent être exécutés régulièrement, idéalement à chaque changement du code, pour s'assurer que tout fonctionne toujours comme prévu.

Les bonnes pratiques du TDD

- **Maintenir une bonne couverture de test** : Toutes les parties du code doivent être testées. Vous devriez viser une haute couverture de code par les tests, bien que 100% ne soit pas toujours réaliste ou nécessaire.
- **Tester aux différents niveaux** : Il ne suffit pas de faire des tests unitaires, pensez également aux tests d'intégration, aux tests système, aux tests d'acceptation, etc

Principe FIRST

F - **Fast (Rapide)** : Les tests doivent être rapides, car cela encourage les développeurs à les exécuter fréquemment, ce qui facilite la détection précoce des problèmes.

I - **Independent (Indépendant)** : Les tests doivent être indépendants les uns des autres, ce qui signifie qu'ils ne doivent pas avoir de dépendances mutuelles. Cela permet d'isoler les problèmes plus facilement et d'améliorer la maintenance.

R - **Repeatable (Reproductible)** : Les tests doivent être reproductibles dans n'importe quel environnement. Ils ne doivent pas dépendre de conditions externes instables ou d'états changeants, afin de garantir des résultats cohérents.

S - **Self-validating (Auto-vérifiable)** : Les tests doivent s'auto-vérifier et retourner un résultat clair (passé/échoué) sans nécessiter de vérification manuelle. Cela facilite l'intégration des tests dans les processus d'intégration continue et d'automatisation.

T - **Timely (Opportun)** : Les tests doivent être écrits en même temps que le code, voire avant. Les retards dans l'écriture des tests peuvent entraîner des problèmes de qualité et de maintenabilité.

Kata TDD

Vous avez été embauché en tant que développeur chez FridgeCraft, un fabricant de réfrigérateurs qui se targue de fournir des réfrigérateurs de qualité artisanale.
FridgeCraft a décidé d'adopter la tendance des réfrigérateurs « intelligents », et c'est votre mission de créer le logiciel à intégrer dans le nouveau modèle de test.

Instructions

L'équipe de recherche et développement vous a fourni les exigences suivantes pour la première itération du modèle de test :

1. Suivre les articles placés dans le réfrigérateur et ceux qui en sont sortis.
2. Lorsqu'un article est ajouté, le réfrigérateur doit capturer les informations concernant cet article :
    - Nom de l'article
    - Date d'expiration
    - Horodatage du moment où il a été ajouté

Kata TDD

3. Lorsque le réfrigérateur est ouvert, les articles déjà à l'intérieur voient leur date de péremption se dégrader :
    - De 1 heure (si l'item est scellé)
    - De 5 heures (si l'item est ouvert)
4. Fournir un affichage formaté pour visualiser le contenu et le temps restant avant expiration :
    - Les articles expirés sont affichés en premier avec : `"EXPIRÉ : $item.name"`
    - Les articles non expirés sont affichés ensuite avec : `"$item.name : n jour(s) restant(s)"`

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Développement en BDD

Les paradigmes du BDD

Le **Behavior Driven Development (BDD)** est une pratique de développement de logiciel qui met l'accent sur la **collaboration entre les différentes parties prenantes d'un projet** (comme les développeurs, les testeurs, les responsables produit, etc.) et l'**explication du comportement du système en termes compréhensibles par tous**. Il est souvent utilisé dans le développement Agile

- **Communication et collaboration** : Le BDD insiste sur la nécessité d'une collaboration étroite entre toutes les parties prenantes du projet. Il met l'accent sur le "partage des connaissances", en veillant à ce que tout le monde comprenne le comportement désiré du système.
- **Développement basé sur le comportement** : Comme son nom l'indique, le BDD met l'accent sur **le comportement du système** plutôt que sur les détails techniques de son implémentation. Il s'agit de décrire **ce que le système doit faire de manière compréhensible par tous**, et non de se concentrer sur la manière dont ces fonctionnalités seront mises en œuvre.
- **Spécification exécutable** : Le BDD utilise un **langage naturel** pour **décrire le comportement du système**. Ces descriptions sont ensuite utilisées comme spécifications exécutables, qui peuvent être exécutées comme tests. Cela signifie que les spécifications servent à la fois de documentation et de vérification du système.

Les paradigmes du BDD

- **Tests orientés comportement** : Le BDD utilise un format spécifique pour les tests, connu sous le nom de "**Given-When-Then**" (Étant donné - Quand - Alors). Cela décrit le contexte (Given), l'action qui est effectuée (When), et le résultat attendu (Then). Cela aide à structurer les tests de manière à ce qu'ils reflètent le comportement désiré du système.
- **Développement itératif** : Comme d'autres pratiques Agile, le BDD suit **une approche itérative du développement**. Il s'agit de construire progressivement le système, en ajoutant un comportement à la fois, et en vérifiant constamment que le système se comporte comme prévu.

Les bonnes pratiques du BDD

- **Définissez clairement les comportements** : Vos spécifications **devraient décrire clairement le comportement attendu du système**. Elles ne devraient pas se concentrer sur les détails techniques de l'implémentation, mais plutôt sur **ce que l'utilisateur peut faire et ce qu'il peut s'attendre à voir**.
- **Utilisez le format Given-When-Then** : Ce format est un excellent **moyen de structurer vos scénarios de manière claire et compréhensible**. Il vous aide à vous concentrer sur le contexte (Given), l'action (When) et le résultat (Then). Il est compatible avec le français (**Étant donné-Quand-Alors**)
- **Gardez vos scénarios courts et concentrés** : Chaque scénario doit **tester une seule fonctionnalité ou comportement**. S'il y a trop de choses dans un seul scénario, il devient difficile à comprendre et à maintenir.
- **Automatisez vos scénarios** : Les scénarios BDD doivent **être automatisés pour pouvoir les exécuter régulièrement**. Cela vous permet de vérifier rapidement que votre système se comporte toujours comme prévu, même après des modifications.
- **Revoyez et affinez vos scénarios régulièrement** : Comme tout autre aspect de votre système, vos scénarios BDD devraient être revus et affinés régulièrement. Cela vous aide à maintenir leur pertinence et leur utilité.

Cucumber

Cucumber : Cucumber est l'un des frameworks les plus populaires pour le BDD. Il vous permet d'**écrire des scénarios de tests en langage naturel** qui peuvent être **exécutés comme des tests automatisés**. Cucumber dispose d'une **intégration étroite avec Spring**, ce qui vous permet d'utiliser des fonctionnalités comme l'injection de dépendances dans vos tests.

Gherkin

Gherkin est un langage de spécification utilisé dans le BDD pour écrire des scénarios de tests de manière lisible par les humains.
Il utilise une syntaxe simple et naturelle pour décrire les comportements attendus sous forme de "Features" (fonctionnalités) et de "Scenarios" (scénarios).

Mot Anglais	Mot Français	Description
Feature	**Fonctionnalité**	Description de la fonctionnalité à tester.
Scenario	**Scénario**	Exemple concret illustrant une fonctionnalité.
Given	**Étant donné que**	Contexte initial du scénario.
When	**Quand**	Action ou événement déclencheur.
Then	**Alors**	Résultat attendu après l'action.
And/But	**Et/Mais**	Étapes supplémentaires dans le scénario.

Gherkin

```
Feature: Jeu du Pendu

  Scenario: Proposition correcte d'une lettre
    Given le mot à deviner est "chat"
    And l'état actuel du mot est "_ _ _ _"
    When le joueur propose la lettre "a"
    Then l'état actuel du mot doit être "_ _ a _"
    And le nombre de tentatives restantes est inchangé

  Scenario: Proposition incorrecte d'une lettre
    Given le mot à deviner est "chat"
    And l'état actuel du mot est "_ _ _ _"
    When le joueur propose la lettre "z"
    Then l'état actuel du mot doit rester "_ _ _ _"
    And le nombre de tentatives restantes doit être diminué de 1

  Scenario: Le joueur gagne en devinant toutes les lettres
    Given le mot à deviner est "chat"
    And l'état actuel du mot est "c h a _"
    When le joueur propose la lettre "t"
    Then l'état actuel du mot doit être "c h a t"
    And le joueur doit voir un message de victoire

  Scenario: Le joueur perd après avoir épuisé toutes les tentatives
    Given le mot à deviner est "chat"
    And le nombre de tentatives restantes est 1
    When le joueur propose la lettre "z"
    Then le joueur doit voir un message de défaite
    And le mot complet "chat" doit être révélé
```

```
Fonctionnalité: Jeu du Pendu

  Scénario: Proposition correcte d'une lettre
    Étant donné que le mot à deviner est "chat"
    Et que l'état actuel du mot est "_ _ _ _"
    Quand le joueur propose la lettre "a"
    Alors l'état actuel du mot doit être "_ _ a _"
    Et le nombre de tentatives restantes est inchangé

  Scénario: Proposition incorrecte d'une lettre
    Étant donné que le mot à deviner est "chat"
    Et que l'état actuel du mot est "_ _ _ _"
    Quand le joueur propose la lettre "z"
    Alors l'état actuel du mot doit rester "_ _ _ _"
    Et le nombre de tentatives restantes doit être diminué de 1

  Scénario: Le joueur gagne en devinant toutes les lettres
    Étant donné que le mot à deviner est "chat"
    Et que l'état actuel du mot est "c h a _"
    Quand le joueur propose la lettre "t"
    Alors l'état actuel du mot doit être "c h a t"
    Et le joueur doit voir un message de victoire

  Scénario: Le joueur perd après avoir épuisé toutes les tentatives
    Étant donné que le mot à deviner est "chat"
    Et que le nombre de tentatives restantes est 1
    Quand le joueur propose la lettre "z"
    Alors le joueur doit voir un message de défaite
    Et le mot complet "chat" doit être révélé
```

🧙‍♂️ Kata Potter – Énoncé (version BDD)

Il était une fois une série de **5 livres** racontant l'histoire d'un héros très britannique nommé **Harry Potter**.
(À l'époque où ce kata a été inventé, il n'y avait que 5 tomes. Depuis, ils se sont multipliés, mais nous restons sur ces 5-là).
Les enfants du monde entier trouvaient ces livres fantastiques… et bien sûr, l'éditeur aussi.
Dans un élan de « générosité » (et surtout pour augmenter les ventes), il mit en place la **grille de réduction suivante** :

- **Un seul exemplaire d'un tome coûte 8 €**.
- Si vous achetez **2 tomes différents**, vous obtenez **5 % de réduction** sur ces deux livres.
- Si vous achetez **3 tomes différents**, vous obtenez **10 % de réduction**.
- Si vous achetez **4 tomes différents**, vous obtenez **20 % de réduction**.
- Si vous achetez les **5 tomes différents**, vous obtenez une **réduction de 25 %**.

La « Potter mania » déferle et les parents se ruent sur ces ouvrages.
Votre mission est d'**écrire du code qui calcule le prix total d'un panier d'achat**, en appliquant **la meilleure remise possible**.

- Vous devez **absolument résoudre ce kata en suivant la démarche BDD (Behavior Driven Development)**.
- Les comportements attendus doivent être décrits à l'aide de **scénarios Gherkin (Given / When / Then)**.
- Chaque fonctionnalité doit être validée par un **scénario lisible en langage métier**, afin que les règles de calcul puissent être comprises aussi bien par les développeurs que par les non-techniciens.