

# Fiche 11 : Test Driven Development

## Table des matières

1	Objectifs.....	2
2	Concepts.....	2
2.1	Introduction au Test Driven Development.....	2
2.2	Spécifier les tests.....	3
2.2.1	Comment faire ?.....	3
2.2.2	Un exemple.....	3
2.3	Implémenter les tests et le code de l'application .....	4
2.3.1	Écrire un test qui échoue.....	4
2.3.2	Écrire le code pour faire passer le test.....	6
2.3.3	Refactor du code .....	7
2.3.4	Continuer les itérations .....	7
2.4	TDD en résumé .....	9
3	Exercices.....	9
3.1	Exercice 1 : très simple TDD .....	9
3.2	Exercice 2 : nouveaux scénarios de tests .....	9
3.3	Exercice 3 : TDD lors de la mise à jour de fonctionnalités existantes .....	10
3.4	Exercice 4 : TDD pour les nouvelles fonctionnalités de la classe Task .....	10
3.5	Exercice 5 : TDD pour les nouvelles fonctionnalités de la classe TodoList.....	11

# 1 Objectifs

ID	Objectifs
AJ01	Test Driven Development (TDD)
AJ02	Spécifier des tests

# 2 Concepts

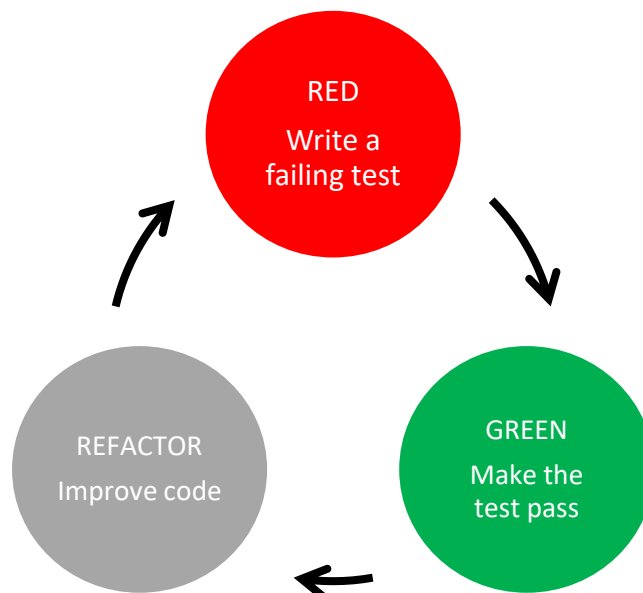
## 2.1 Introduction au Test Driven Development

Le TDD est un processus de développement de logiciel où les tests sont utilisés comme une spécification pour concevoir et écrire le code.

Les 3 étapes du TDD :

1. **Ecrire un test qui échoue** : on écrit un test qui décrit ce que notre code devra faire.
2. **Ecrire le code pour faire passer ce test** : on écrit le code le plus simple permettant le succès du test.
3. **Refactor du code** : on va considérer une mise à jour du code selon les bonnes pratiques de l'OO. S'il y a des aspects qui peuvent être améliorées, on va faire un refactor du code pour qu'il soit plus facilement maintenable, en éliminant les duplications, en diminuant les dépendances, en renommant des méthodes, variables, en augmentant l'efficacité... tout cela sans changer le comportement du code : le test doit toujours réussir.

Voici donc comment on schématise le TDD :



Voici certains des avantages du TDD :

- Le programmeur reçoit un rapide feedback sur ce qu'il produit :
  - o Tout changement de code peut se faire en toute confiance car si tous les tests passent, c'est que les modifications n'ont rien changé !
  - o Cela diminue donc la peur de changer son code et encourage à l'améliorer.
- Le programmeur doit se mettre dans la peau de l'utilisateur lors de l'écriture du test permettant de produire du code répondant mieux aux besoins.
- Le programmeur doit se focaliser sur l'écriture de classes concrètes, sans en faire trop, en évitant ainsi des généralisations et optimisations du code prématurées.

## 2.2 Spécifier les tests

### 2.2.1 Comment faire ?

Mais comment pouvons-nous spécifier des tests ?

Dans un premier temps, il faut d'abord savoir ce que l'on souhaite développer : )

Pour cela, nous vous recommandons une approche toute simple. Il suffit d'identifier les fonctionnalités à développer sous forme de liste. Nous appellerons ces fonctionnalités les **cas d'utilisation** (use cases ou UC).

Puis, par cas d'utilisation, nous allons identifier :

- D'abord, le comportement attendu habituel ;
- Puis, le comportement inhabituel.

Nous vous recommandons de donner un nom à chaque comportement que vous spécifiez et que vous allez tester. À chacun de ces comportements correspondra un scénario de test qui sera identifié par le nom que vous lui avez donné.

### 2.2.2 Un exemple

Nous souhaitons développer une mini application qui permette de gérer une liste de tâches.

Nous allons donc identifier tous les cas d'utilisation et le comportement attendu :

- (UC1) Ajouter une tâche à la liste :
  - o **addTask** : la tâche est contenue dans la liste, on informe du succès de l'opération
  - o **addEmptyTask** : on tente d'ajouter une tâche vide (constituée uniquement de caractères « blancs » ou nulle), la tâche n'est pas contenue dans la liste, on informe de l'échec de l'opération
  - o **addExistingTask** : on tente d'ajouter une tâche déjà présente, on informe de l'échec de l'opération
- (UC2) Vérifier qu'une tâche est contenue dans la liste :
  - o La tâche est présente et on l'indique (*pas besoin d'identifier ce scénario de tests car c'est couvert par les scénarios associés à l'UC1*)
  - o La tâche n'est pas présente et on l'indique (*pas besoin d'identifier ce scénario de tests car c'est couvert par les scénarios associés à l'UC1*)
- (UC3) Supprimer une tâche de la liste :
  - o **removeTask** : la tâche n'est plus contenue dans la liste, on informe du succès de l'opération

- **removeUnexistingTask** : on tente de supprimer une tâche inexistante, on informe de l'échec de l'opération

Cette liste permet d'identifier les scénarios de tests et peut vite devenir longue. Si vous souhaitez travailler de manière incrémentale et aller plus rapidement dans votre code, n'hésitez pas, à créer votre spécification de tests de manière incrémentale :

- Vous identifiez tous les scénarios de tests associés à un UC.
- Vous faite du TDD pour chacun des scénarios de tests.
- Vous identifiez tous les scénarios de tests associés à une autre UC.
- Vous faite du TDD pour chacun de ces nouveaux scénarios de tests.
- ...

L'idée est de commencer les tests avec l'UC et le scénario de test qui semble le plus important, un scénario qui nous encourage à mettre en place le squelette de notre application.

## 2.3 Implémenter les tests et le code de l'application

### 2.3.1 Écrire un test qui échoue

Pour l'implémentation des tests, on va travailler à l'envers par rapport à ce qu'on fait dans des tests traditionnels.

#### Point-clé : travailler à l'envers à partir des assertions

Nous allons d'abord identifier ce que l'on veut vérifier, en écrivant les **asserts** attendus. On écrira donc nos assertions pour vérifier le résultat de l'appel d'une méthode sur un objet pour une méthode et un objet qui souvent n'existent pas encore.

Puis nous allons utiliser notre Editeur de Code pour nous aider dans la génération du contexte de notre test, pour créer les classes nécessaires et les objets.

Si vous voulez plus de détails sur les fonctionnalités offertes par IntelliJ pour faire du TDD, vous pouvez consulter la documentation : [Test-driven development](#). Nous allons reprendre ces fonctionnalités dans le tutoriel qui suit.

Sous IntelliJ, veuillez créer un projet **Maven** nommé **AJ\_Atelier\_11**. Pensez à mettre dans **Advanced Setting**, comme **GroupId** : **be.vinci**

Commençons par le scénario de tests que nous trouvons le plus important : « **(addTask)** Ajouter une tâche à la liste : la tâche est contenue dans la liste, on informe du succès de l'opération »

Nous allons d'abord créer la classe de tests **TodoListTest** :

- Dans le dossier racine de tests (**/src/test/java**) : clic droit, **New | Java Class**
- Mettre son curseur dans le corps de la classe, **Alt + Insert, Test...**
- Cliquer une fois sur **Fix** si JUnit5 n'est pas trouvé. Cela ajoutera la dépendance nécessaire dans le fichier **pom.xml**. **Attention** : le message **Fix** reste même si vous avez ajouté la dépendance.
- **Class name** doit garder le nom **TodoListTest** (cela fait en sorte de ne pas créer une nouvelle classe mais de mettre à jour la classe existante)
- Cliquer sur **OK**.

Pour créer le scénario **addTask** :

- Mettre son curseur dans le corps de la classe : **Alt + Insert, Test Method**. Il ne reste plus qu'à donner le nom **addTask** à la méthode de test et d'écrire les tests qu'on veut effectuer.

```
@Test
void addTask() {

    assertAll(
        () -> assertTrue(todoList.addTask("task 1")),
        () -> assertTrue(todoList.containsTask("task 1"))
    );
}
```

### Point-clé : voir le test échouer pour les bonnes raisons

Est-il intéressant à ce stade-ci de voir le test échouer ?

On ne veut pas que le test échoue juste parce que le setup du test est mauvais. Ici, il est clairement mauvais, puisque **todoList** n'existe pas, tout comme les méthodes **addTask** & **containsTask** !

Cela signifie qu'avant d'écrire du code qui va faire réussir notre test, on doit s'assurer que le test échoue pour de bonnes raisons.

Nous allons donc continuer en écrivant le **contexte** du test, le **setup**.

En passant sur **todoList**, on peut accéder aux actions en cliquant dessus. Sinon, en laissant le curseur sur la variable non déclarée, on peut accéder aux actions en faisant **Alt + Enter : Create local variable 'todoList'**. Donnons-lui le type **TodoList**.

```
TodoList todoList;
```

Voici comment créer la classe **TodoList** :

- Passer le curseur sur **TodoList** dans le scénario de test et choisir l'action **Create class 'TodoList'**.
- Indiquer le main (... \src\main\java) pour **Target destination directory**

La variable locale **todoList** doit maintenant devenir un objet de **TodoList**. On fait donc appel au constructeur par défaut.

```
TodoList todoList = new TodoList();
```

Via IntelliJ, nous pouvons automatiquement générer les méthodes **addTask** et **containsTask** (via **Alt + Enter** ou en passant la souris sur ces méthodes) :

- **Create method 'addTask' in TodoList** : on fait en sorte que la méthode fasse échouer le test :

```
public boolean addTask(String task) {
    return false;
}
```

- **Create method 'containsTask' in TodoList** : on fait en sorte que la méthode fasse échouer le test :

```
public boolean containsTask(String task) {
    return false;
}
```

Ça y est, nous avons un bon setup de test qui nous permet de voir échouer le test pour des bonnes raisons !

Voici à quoi ressemble le code du scénario de test.

```
@Test
void addTask() {

    TodoList todoList = new TodoList();
    assertAll(
        () -> assertTrue(todoList.addTask("task 1")),
        () -> assertTrue(todoList.containsTask("task 1"))
    );
}
```

### 2.3.2 Écrire le code pour faire passer le test

**Point-clé : écrire l'implémentation qui est évidente en évitant les étapes triviales de tests**

Il est inutile d'écrire du code trop trivial comme renvoyer une valeur hardcodée dans une fonction quand on sait pertinemment que cette valeur va changer au cours du temps. Dans ce cas, autant directement créer un attribut. Idem si on doit ajouter un élément à une liste, autant créer cette liste. Il est notamment inutile de tester les getters & setters. Et quand une implémentation n'est pas si évidente, alors n'écrivez que le code permettant de passer le test...

Nous allons commencer par la méthode **addTask**. Voici l'implémentation évidente pour ajouter une tâche :

```
public class TodoList {

    private List<String> tasks = new ArrayList<>();

    public boolean addTask(String task) {
        return tasks.add(task);
    }

    public boolean containsTask(String task) {
        return false;
    }
}
```

A ce stade-ci, l'exécution du test **addTask** passe le premier assert, mais pas le deuxième associé à la méthode **containsTask**.

Nous allons donc mettre à jour la méthode **containsTask** :

```
public boolean containsTask(String task) {
    return tasks.contains(task);
}
```

Ça y est, nous avons passé le premier test unitaire !

### 2.3.3 Refactor du code

**Point-clé : appliquer la règle de trois quand il y a de la duplication dans notre code avant de créer du code à réutiliser**

**La réutilisation de code gagne généralement sur la duplication.** Car s'il y a plus de code, il y a plus de risque de bugs, et le code est moins lisible.

Néanmoins, afin de trouver un bon compromis entre créer de l'abstraction trop rapidement, et attendre trop longtemps avant de le faire, en moyenne, on conseille d'attendre qu'il y ait **3 duplications** dans notre code avant de faire un **refactoring**.

Ceci doit être pris comme une moyenne, si vous sentez qu'il y a un pattern qui va souvent revenir dans votre code, vous pouvez bien sûr le faire plus rapidement.

Notons que la duplication peut se trouver tant dans le code des scénarios de test, que dans les objets à tester.

En cas de duplication dans vos scénarios de tests, vérifiez que vos tests restent facilement lisibles. Parfois cela amène à préférer de la duplication dans nos scénarios de tests.

Au niveau du tutoriel sur la gestion d'une `ToDoList`, il n'y a pas encore de duplication de code.

### 2.3.4 Continuer les itérations

Le but est de continuer à développer sur base des scénarios de tests jusqu'à ce que tous les UC soient implémentés.

Nous allons continuer le tutoriel avec ce scénario : « **(addEmptyTask)** Ajouter une tâche à la liste : on tente d'ajouter une tâche vide (constituée uniquement de caractères « blancs » ou nulle), la tâche n'est pas contenue dans la liste, on informe de l'échec de l'opération »

#### 1. Écrire un test qui échoue

```
2. @Test
void addEmptyTask() {
    assertAll(
        () -> assertFalse(todoList.addTask("")),
        () -> assertFalse(todoList.containsTask("")),
        () -> assertFalse(todoList.addTask(null)),
        () -> assertFalse(todoList.containsTask(null))
    );
}
```

En l'exécutant, ce test échoue bien.

#### 3. Écrire le code pour faire passer ce test

```
4. public boolean addTask(String task) {
    if (task == null) {
        return false;
    }
    if (task.isBlank()) {
        return false;
    }
    return tasks.add(task);
}
```

Ce test passe ! Vérifiez que tous les tests continuent à passer !

## 5. Refactor

Ici, on sent au niveau des scénarios de tests qu'il y a un pattern qui revient régulièrement : on aura quasi toujours besoin d'une `ToDoList` vide pour commencer un scénario de test. Nous allons donc créer un attribut, et le réinitialiser avant chaque test au sein de la méthode **setUp** de la classe de test :

- Dans **ToDoListTest**, **Alt + Insert** , **SetUp Method**
- Création d'un attribut et initialisation de celui-ci dans **setUp**
- Utilisation de l'attribut **todoList** dans les scénarios de test

Voici le code à cette étape-ci :

```
public class ToDoListTest {

    private ToDoList todoList;

    @BeforeEach
    void setUp() {
        todoList = new ToDoList();
    }

    @Test
    void addTask() {
        assertAll(
            () -> assertTrue(todoList.addTask("task 1")),
            () -> assertTrue(todoList.containsTask("task 1"))
        );
    }

    @Test
    void addEmptyTask() {
        assertAll(
            () -> assertFalse(todoList.addTask("")),
            () -> assertFalse(todoList.containsTask("")),
            () -> assertFalse(todoList.addTask(null)),
            () -> assertFalse(todoList.containsTask(null))
        );
    }

}
```

Nous allons terminer le tutoriel avec ce scénario : « (**addExistingTask**) Ajouter une tâche à la liste : on tente d'ajouter une tâche déjà présente, on informe de l'échec de l'opération »

### 1. Écrire un test qui échoue

```
@Test
void addExistingTask() {
    todoList.addTask("task 1");
    assertFalse(todoList.addTask("task 1"));
}
```

En l'exécutant, ce test échoue bien.

NB : Notons que si l'on voulait avoir la certitude absolue que la tâche n'a pas été ajoutée, il faudrait créer une nouvelle méthode pour compter le nombre de tâches associée à la `ToDoList`. Néanmoins, pour ce tutoriel, on se satisfait du fait que si la méthode **addTask** renvoie **false**, c'est que d'office la **task.add(task)** n'a pas été appelée !



## 2. Écrire le code pour faire passer ce test

```
public boolean addTask(String task) {  
    if (task.isBlank()) {  
        return false;  
    }  
    if (containsTask(task)) {  
        return false;  
    }  
    return tasks.add(task);  
}
```

Ce test passe ! Vérifiez que tous les tests continuent à passer !

## 3. Refactor

Pas de duplication de code à cette étape-ci.

### 2.4 TDD en résumé

- Commencez par identifier les cas d'utilisation de votre application et identifier les scénarios de tests associés.
- Pour chaque scénario de test :
  - 1. Écrire un test qui échoue**  
Démarrez à partir des assertions et visualisez que le test échoue pour de bonnes raisons !
  - 2. Écrire le code minimum pour faire passer le test**  
Ecrivez une implémentation évidente tout en évitant le code trivial.
  - 3. Refactor du code**  
Appliquez la règle de trois : dès qu'il y a 3 duplications dans votre code, remplacez celui-ci par du code réutilisable ; sinon, laissez votre code en l'état.  
Assurez-vous que tous les tests continuent à passer après le refactor.

## 3 Exercices

### 3.1 Exercice 1 : très simple TDD

Veuillez faire du TDD pour ces deux scénarios de test :

- **(removeTask)** Supprimer une tâche de la liste : la tâche n'est plus contenue dans la liste, on informe du succès de l'opération
- **(removeUnexistingTask)** Supprimer une tâche de la liste : on tente de supprimer une tâche inexistante, on informe de l'échec de l'opération

### 3.2 Exercice 2 : nouveaux scénarios de tests

Nous souhaitons faire évoluer l'application de gestion de tâches. Il doit être possible :

- De créer des tâches en donnant ces informations : un titre (ne peut pas être vide ou null), une description (ne peut pas être nulle).
- D'ajouter une tâche qui a un même titre au sein d'une TodoList. Un doublon devient donc une tâche qui aurait un même titre et une même description !
- Terminer une tâche.
- De modifier le titre d'une tâche seulement si cette tâche n'est pas déjà terminée ; notons que le titre ne peut pas être vide ou nul...


- De modifier la description d'une tâche seulement si cette tâche n'est pas déjà terminée ; notons que la description peut être vide.
- De renvoyer une tâche qui se trouve au sein de la `ToDoList` en donnant une tâche qui contiendrait son titre et sa description.
- De modifier une `ToDoList` en indiquant une tâche à modifier et une nouvelle tâche incluant les nouvelles données.


Veuillez mettre à jour les scénarios de test existant en identifiant les nouveautés **en vert**, et en listant les nouvelles UCs ci-dessous **en vert** :

- (UC1) Ajouter une tâche à la liste :
  - o **addTask** : la tâche est contenue dans la liste, on informe du succès de l'opération
  - o **addEmptyTask** : on tente d'ajouter une tâche vide (constituée uniquement de caractères « blancs » ou nulle), la tâche n'est pas contenue dans la liste, on informe de l'échec de l'opération
  - o **addExistingTask** : on tente d'ajouter une tâche déjà présente, on informe de l'échec de l'opération
- (UC2) Vérifier qu'une tâche est contenue dans la liste :
  - o La tâche est présente et on l'indique (*pas besoin d'identifier ce scénario de tests car c'est couvert par les scénarios associés à l'UC1*)
  - o La tâche n'est pas présente et on l'indique (*pas besoin d'identifier ce scénario de tests car c'est couvert par les scénarios associés à l'UC1*)
- (UC3) Supprimer une tâche de la liste :
  - o **removeTask** : la tâche n'est plus contenue dans la liste, on informe du succès de l'opération
  - o **removeUnexistingTask** : on tente de supprimer une tâche inexistante, on informe de l'échec de l'opération

### 3.3 Exercice 3 : TDD lors de la mise à jour de fonctionnalités existantes

Veuillez faire du TDD pour les scénarios que vous avez dû modifier pour prendre en compte le fait qu'une tâche doit maintenant ne plus être juste une `String` : une tâche contient maintenant un titre et une description au sein d'un objet.

 **Tips** : Commencez par mettre à jour les scénarios de tests de **ToDoListTest** associé à l'ajout de tâches au sein d'une **ToDoList**. Comme une tâche est un objet très simple, il n'est pas utile de créer des Mocks objects de ceux-ci. Vous pouvez créer et directement utiliser un constructeur de tâches...

 Dans cet exercice, vous devriez vous rendre compte à quel point il est confortable de mettre à jour des fonctionnalités existantes lorsqu'on a déjà des tests existants. Le jeu consiste à mettre à jour le code pour faire passer tous les anciens tests. Bien sûr, parfois, il est aussi utile de créer des nouveaux tests pour bien couvrir les mises à jour des fonctionnalités.

Par exemple, avez-vous pensé à vérifier si une tâche qui a été clonée (même titre et même description qu'une tâche existante) amène bien à la suppression de la tâche existante ? Ou avez-vous juste testé que l'on peut supprimer une tâche que sur base de la tâche existante ?

### 3.4 Exercice 4 : TDD pour les nouvelles fonctionnalités de la classe `Task`

Veuillez faire du TDD pour les scénarios de la classe `Task` au sein d'une nouvelle classe **TaskTest**.

Si vous souhaitez exécuter tous les tests se trouvant dans les différentes classes de tests situées dans le dossier **/src/test/java** en une seule fois, vous pouvez le faire ainsi : clic droit sur **/src/test/java**, **Run 'all Tests'**.

### 3.5 Exercice 5 : TDD pour les nouvelles fonctionnalités de la classe `TodoList`

Il est temps de s'occuper de l'opération permettant de retrouver une tâche au sein de la `TodoList`.

De plus, il devrait être possible de modifier une tâche par le biais de la `TodoList` en indiquant tant la tâche que l'on souhaite mettre à jour que les nouvelles données de cette tâche.

Veillez faire du TDD pour les nouveaux scénarios au sein de la classe **`TodoListTest`**.