Mumbles

Geste de savoir

Un retex sur le Test Driven Development

12 décembre 2022

Table des matières

Introduction

Le Test Driven Development est un outil qui se répand de plus en plus dans les équipes de développement. Mais ce concept peut être un peu compliqué à aborder ou être mal compris (condition malheureusement alimentée par un gatekeeping de plus en plus pesant).

L'objectif n'est pas de faire un cours magistral sur le TDD, son origine, les différents types, etc. L'article de Wikipedia le fait déjà.

Je ne cherche pas non plus à en faire la promotion: le TDD n'est qu'un simple outil parmi beaucoup d'autres. Et, il n'a rien de magique…

Ce billet est plutôt un condensé de ce que j'en retiens après plusieurs années de pratiques. Ainsi que les quelques subtilités et pièges qui m'ont fait (un peu) ramer, au début.

Le TDD en quelques mots

Le TDD est une **stratégie de design** qui repose sur l'écriture de tests de manière itérative. Chaque test que nous allons écrire nous servira d'exemple dont notre code doit se comporter.

C'est surtout un prétexte à la discussion et à la réflexion: les tests ne sont qu'une base de réflexion, comme le serait un schéma sur un tableau blanc.
En ça, le TDD **n'est pas une méthode pour écrire des tests**[1] : ils ne servent que de support à nos discussions et la plupart partiront à la poubelle.

En résumé, on utilise donc du code pour réfléchir au code.

C'est un exercice qui se prête bien au binômage: une personne pour écrire le test, une seconde pour le faire passer au vert. On échange enfin les rôles à l'itération suivante.

Il faut que chaque itération soit la plus courte possible afin d'éviter l'effet tunnel et de partir dans des développements longs: le TDD est là pour nous aider à avancer pas à pas

Une itération se base sur la boucle TDD qui se décompose en trois phases:

1. Du moins, ce n'est pas une bonne méthode pour écrire de bons tests et construire une campagne de tests de non-régression solide

Test rouge

C'est la phase d'exploration. On va réfléchir à la façon dont on veut que notre code se comporte (par exemple, ce que va retourner une fonction) en se basant sur un exemple simple (on avance pas à pas). Puis, on va pouvoir définir ce dont on a besoin: les entrées nécessaires, les sorties attendues, etc. On va également s'assurer qu'on a bien compris les règles métier et qu'on sait ce qu'on doit faire

Il s'agit là d'écrire un test en imaginant le comportement de notre futur code.

On écrit ou modifie un **unique** test de façon à ce qu'il ne passe pas (test rouge). Il peut échouer parce qu'il ne compile pas ou parce que les assertions ne sont pas vérifiées. Si jamais un nouveau test passe directement OK, c'est signe soit que le test n'est pas correct (il manque une assertion par exemple[2]); soit que nous ne sommes pas en train d'ajouter de nouvelles fonctionnalités.

Dans le dernier cas, ce n'est pas dramatique en soi: il peut apporter un nouveau cas d'exemple ou simplement révéler que nous allons trop vite dans la phase d'implémentation

Par contre, il vaut mieux modifier les valeurs des entrées pour s'assurer que le test peut passer au rouge.

Test vert

On fait passer le test au vert. Tous les tests doivent continuer de l'être.

Il convient d'écrire le code le plus simple qui répond au strict besoin pour que le test passe. Cela évite d'ajouter de la complexité inutile et d'avancer au plus proche du besoin actuel (i.e. l'étape actuelle).

Peu importe que le code écrit ne soit pas très beau. L'important est d'obtenir le plus rapidement possible une première version qui fonctionne.

Refactoring

Cette phase n'est pas obligatoire mais est importante.

Une fois que tous les tests sont vert, nous pouvons nous demander si le code et les tests nous conviennent en l'état et corriger ce qui nous gêne.

La seule règle: le comportement doit être exactement identique (i.e. le code ne fait pas plus ou moins de choses, ne traite pas plus de cas, etc.).

C'est pendant cette étape qu'on va pouvoir améliorer notre code, explorer de nouvelles façon de faire, de nouvelles approches, etc. Dans tous les cas, on a déjà une implémentation fonctionnelle

2. D'où l'intérêt de toujours commencer par écrire les assertions

Ce que n'est pas le TDD

Le TDD n'est pas une méthode pour écrire des tests. On va bien se retrouver avec une panoplie de tests, mais ce n'est pas le but. Cela sous-entend aussi que les tests que nous allons écrire n'ont pas forcément pour objectif d'être pérennes. Ils ne seront également pas suffisants (pas de tests d'intégration, end-to-end, etc.). Voire, ne pas être maintenables.

Cela dit, on peut tout à fait l'imaginer comme étant un bon moyen de se mettre à écrire des tests. Et, ça peut toujours être mieux que rien.
L'important, c'est d'y trouver un intérêt

Ce n'est pas une pratique si simple et évidente. Il faut du temps avant d'en voir les bénéfices. Les débuts sont souvent brouillons et difficiles (rien que l'automatisme d'écrire un test avant le code prend du temps)

Contrairement à ce que prône certains gourous trop prolixes[3], le TDD n'est pas une silver bullet: si vous avez du mal à boucler les développements dans les temps, le TDD ne vous aidera pas. Il ne garantit pas non plus que vous n'aurez plus aucun bug.

Le TDD est un outil. Et comme tout outil, s'il ne nous aide pas, c'est que ce n'est pas le bon outil qui ne sera rien d'autre qu'un frein.

Nous ne sommes pas obligés de faire du TDD sur l'ensemble de la *codebase*: il est adapté pour implémenter les règles métier et le code fonctionnel. Moins pour tout ce qui est code technique (glu, infra, CRUD, etc.).
En gros, dès le moment où on a besoin de réfléchir à comment construire le code, le TDD peut nous aider.

Pour bien débuter

Le kata fizzbuzz est un bon point d'entrée au TDD

on peut voir le TDD comme une boussole ou un compas: il est là pour nous aiguiller quand la destination n'est pas évidente et qu'on est un peu perdu sur le chemin à suivre

Les tests ne sont pas la finalité mais uniquement un effet de bord.
Il n'y a donc pas de tests trop simples ou inutiles. On réfléchit en les écrivant. Au pire, on peut les supprimer juste après (pas toujours utile de garder des tests qui vérifient qu'on sait appeler le constructeur)

L'objectif est de rentrer le plus vite possible dans le code: ne pas perdre de temps à réfléchir au nommage ou à la structure du code. Tout ça finira par émerger naturellement (les outils de refactoring des IDEs sont très efficaces).

Un code coûte à la lecture, pas à l'écriture: si on doit écrire le code le plus simple à chaque itération, il faut qu'il soit lisible et maintenable (on ne cherche pas forcément à écrire le code le plus court). Ne pas oublier cette règle pendant la phase de refactoring

Si passer de la phase rouge au vert prend plus de 20 minutes, c'est signe que les itérations sont trop longues et qu'on essaie de faire de trop grandes étapes (généralement, ça ne devrait pas prendre plus de quelques minutes)

3. Pas la peine, non plus, d'avoir lu 10 livres et suivies 15 formations en ligne pour se lancer

Il arrive bien souvent qu'on ait besoin d'implémenter une couche sous-jacente ou de descendre d'un niveau (par exemple, devoir créer une nouvelle classe alors qu'on implémente un service). Dans ce cas, on laissera le test actuel dans l'état rouge[4] (c'est important pour ne pas l'oublier plus tard) et on recommence une boucle TDD sur notre nouvelle classe. Une fois terminée, on reprendra là où s'était arrêté.

Savoir faire preuve de mauvaise foi

Quand vient la phase verte, il faut savoir écrire le code le plus simple qui soit. Et, parfois savoir faire preuve de mauvaise foi: ajouter une condition à l'arrache dans une fonction, retourner une valeur en dur, etc.

Pour deux raisons:

Quand le test passe au vert, cela signifie qu'on a une implémentation qui répond aux besoins actuels. Donc, on sait qu'il y a au moins une solution à notre problème. Ce qui veut dire qu'on peut passer à l'étape de refactoring et rendre notre code plus propre, robuste, etc. Si jamais notre refactoring ne donne rien, on peut toujours revenir à l'implémentation d'origine et recommencer. Dans tous les cas, on a notre backup

Ca oblige à chercher des approches et méthodes de tests suffisamment poussées pour rendre plus difficile d'ajouter juste un `if`. Ca permet généralement de chercher à tester une propriété (tout nombre multiple de 5 doit afficher "buzz") plus q'un simple cas (10 doit afficher "buzz"). Les tests seront plus couvrants et notre code plus proche des règles métier

Prendre soin de ses tests

On va rapidement se retrouver avec énormément de tests unitaires utilisant une base de code en construction, donc mouvante.

Il arrivera certainement un moment où il faudra changer une signature de méthode, un constructeur, un type de variable, etc. Cela peut vite devenir fastidieux. Il faut y penser et s'assurer que nos tests ne se basent pas sur l'implémentation (utilisation de builders ou de helpers, par exemple). Qui plus est, cela rendra les tests plus faciles à lire.

La durée d'exécution des tests qui va s'accroître avec l'ajout de nouveaux tests. On va devoir lancer très souvent l'ensemble des tests unitaires. Il est hors de question de devoir patienter plusieurs minutes (une minute, c'est déjà beaucoup trop long) à chaque fois. Les tests doivent être très rapides et tous passer en quelques secondes grand maximum

Conclusion

Pour terminer, une vidéo d'une mise en application du TDD pédagogique et rondement menée.

4. si le test ne compile pas, il est possible de commenter son contenu, puis de lever une exception ou ajouter une assertion fausse.