Chapitre 2
Savoir-faire

1. Introduction

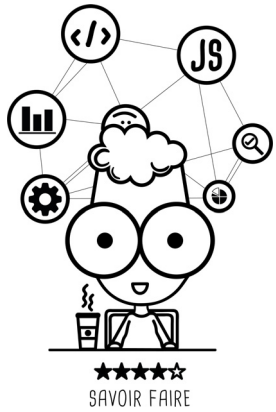
« *The problem is not the amount of unexpected changes in a software project; the problem is our inability to cope with them.* » - *The software craftsman*

Bienvenue dans la deuxième partie. Vous entrez dans le second tunnel de l'aventure, pour vous initier à différentes techniques d'assurance qualité, qui vous aideront à maîtriser les besoins d'évolution logicielle. Et cela en complémentarité avec les boucles de feedback identifiées en première partie. Derrière ces techniques existent les éléments nécessaires pour produire du code et du produit de qualité.

Nous mettrons ainsi l'accent sur l'importance des tests dans la réalisation des solutions. La différence entre un produit A et un produit B, à fonctionnalités égales, est la qualité, tant perceptible que non perceptible par le client final.

Ce principe se retrouve chez les artisans horlogers. Deux montres, d'apparences identiques, n'auraient pas forcément le même prix. En effet, elles ne seraient pas architecturées, montées et testées de la même façon. En développement informatique, le Test avec un grand T est le garant d'un produit de qualité.

Découvrons-les !



Tant qu'à faire quelque chose, autant le faire bien !

2. TDD, au-delà du DD

« Quality is not expensive. The lack of skills is what makes well-crafted software expensive. TDD doesn't slow developers down. Typing is not a bottleneck. Learning and mastering a new skill, practice, or technology is. » - Sandro Mancuso (The Software Craftsman: Professionalism, Pragmatism, Pride - série Uncle Bob)

Le TDD (*Test Driven Development*, développement et documentation par les tests) est une technique de développement où les tests sont moteurs du design applicatif. Contrairement à ce que l'on peut parfois entendre, il ne se résume pas à la réalisation de tests unitaires ou à l'utilisation de JUnit dans le cadre de projets Java. Mais en quoi cette distinction nous intéresse-t-elle en tant qu'aspirants craft ?

Rappelons-nous quelques valeurs du manifeste, à savoir la capacité à réaliser des logiciels bien conçus et l'apport constant de valeur. Ces deux aspects se placent sous couvert d'un apport constant de qualité en matière de design et en matière de valeur produit, afin de répondre pertinemment aux attentes des utilisateurs. Le TDD y contribue !

Avant de plonger dans différents étonnements structurant le sujet, il est opportun de parcourir les invariants et principes clés du TDD.

– **Test First** : on rencontre souvent des équipes ou des entreprises qui se vantent de faire du TDD. Les tests y sont souvent rédigés, dans le cadre des projets, pour valider du code déjà produit et non pour produire du code. Soyons justes, quand on fait du TDD, on écrit le test en premier, en phase avec une attente produit, et ce n'est qu'ensuite que l'on écrit l'algorithme permettant d'y répondre. L'un des bénéfices du Test First, c'est qu'il aide à améliorer et approfondir en amont notre compréhension du besoin métier en traduisant celui-ci par des intentions de test.
– **KISS and YAGNI** (YAGNI – *You Are not Gonna Need It*, évitez l'over-architecture et l'utilisation de concepts et de code dont vous n'aurez pas forcément besoin) : d'un côté, la simplicité est l'un des facteurs clés du TDD. Plus notre code est simple, plus il sera maintenable, car découpé en unités simples, lisibles et facilement testables. De l'autre côté, nous avons tendance, pour de bonnes ou de mauvaises raisons, à vouloir être proactif et utiliser des concepts avancés en faisant appel à certains patrons de conception trop tôt dans le cycle de notre logiciel. Nous utilisons alors des pratiques qui n'ont pas de valeur ajoutée à ce moment. L'une des mauvaises conséquences est l'ajout de complexité inutile. Nous introduisons prématurément une structure technique sans valeur ajoutée pour le besoin métier exprimé.
– **Refactor** : sans refactor, on ne peut parler de TDD. Le refactor est souvent pensé à tort dans les esprits comme étant une fonctionnalité d'Eclipse, voir Intellij pour les initiés, mais ce n'est pas cela dont on parle ici. Le refactor vient donner un sens au TDD et à l'amélioration continue de notre code, dans la mesure où cette démarche vise à modifier la forme d'un code sans en changer le fond. Le refactor permet de changer la structure d'un logiciel pour répondre à des exigences techniques (patterns, conventions de nommage, performance, code smells...) sans en altérer le comportement capturé par des tests préalablement décrits.

Dans ce chapitre, nous allons voir en quoi le TDD est indispensable pour tout craft aspirant à créer des logiciels bien conçus et à valeur ajoutée. Nous allons décrire le cycle type de cette méthodologie et voir en quoi il est vertueux. On parle souvent du coût de test. Cela serait ainsi le bon moment d'apprécier la distinction entre coûts préventifs et coûts correctifs.

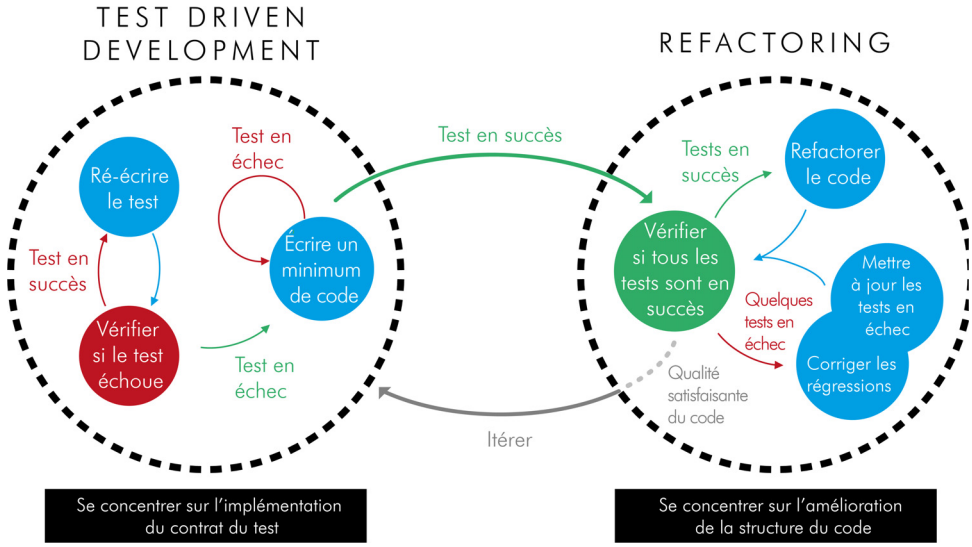
Nous verrons par ailleurs en quoi cela représente la meilleure documentation pour une équipe de développement et en quoi il nous aide, en tant qu'individus et équipes de développement, à être confiants vis-à-vis de modifications. Nous finirons par identifier quelques bonnes pratiques et anti-patterns, histoire de tirer profit de quelques bonnes bases.

2.1 Un cycle vertueux

« *Refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify.* » - *MartinFowler*

Le Test Driven Development est une pratique de développement qui vient avec son lot de règles, articulées autour d'un cycle bien défini : on commence par créer un test, on le fait échouer, on écrit le minimum de code pour qu'il passe. Ensuite, on rentre dans la boucle de refactoring.

[Add test > Run and Fail new tests > Write code > Run tests > Refactor] - Wikipedia*

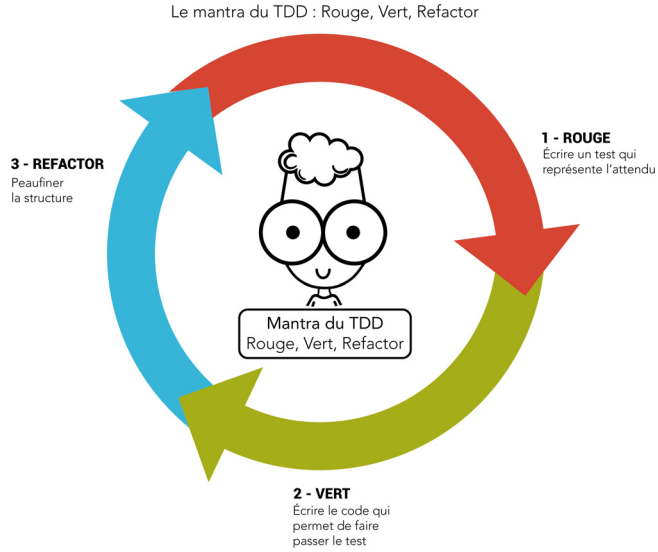


Un cycle bien rodé - Xavier Pigeon

Le but du refactoring n'est pas d'écrire le code pour faire réussir un test ni de rajouter du code issu d'une demande d'évolution ; c'est tout sauf cela. Le refactoring vise à améliorer le design d'un code déjà couvert à l'aide d'au moins un test unitaire, afin de le rendre plus aisé et moins coûteux à modifier. En outre, il permet de supprimer des codes smells (voir section Gestion de la dette technique au chapitre Savoir structurer), de réduire la dette technique, tout en bénéficiant de la protection du harnais de test. Les tests permettent de devenir plus confiant pour faire évoluer la base de code.

« *American software engineer Kent Beck, who is credited with having developed or "rediscovered" the technique, stated in 2003 that TDD encourages simple designs and inspires confidence* » - *Wikipedia (https://en.wikipedia.org/wiki/Test-driven_development)*

Avec le TDD, c'est l'expérience de développement qui est au centre du sujet, bien plus que la qualité intrinsèque du code. On parle souvent d'expérience utilisateur (UX), mais qu'en est-il de l'expérience de réalisation (DX) ? Pour nous accompagner dans ce cycle, cette journey du TDD, il existe de nombreuses librairies de code (frameworks), patterns et outils. Le sujet est en effet tellement crucial que, à l'image du pionnier, Kent Benck (informaticien américain, inventeur de l'eXtreme Programming (XP) et auteur des livres de référence sur la méthode. C'est un des signataires du manifeste Agile), avec JUnit, de nombreux développeurs et entreprises ont créé des frameworks autant pour la couche backend (ex. : TestNG), pour la couche d'accès aux données (ex. : DBUnit), que pour la couche frontend (ex. : JSUnit). Les IDE (*Integrated Development Environment*) sont aussi enrichis avec des plugins, ce qui améliore la productivité et rend l'expérience plus agréable.



Ainsi, histoire de rester dans les rails et dans la vertu du cycle, des patterns ont été popularisés. Kent Beck parle dans son livre, *TDD by Example*, de trois Green Bar Patterns (http://blog.baudson.de/blog/test-driven-development-green-bar-patterns).

– **Obvious Implementation** : commencer par la solution la plus évidente.
– **Fake It (till you make it)** : faire passer le test le plus tôt possible même si ça revient à renvoyer une simple valeur statique par l'unité à tester (UUT : *Unit Under Test*, le système, l'application, le module, le bloc de code ciblé par des tests), puis avancer par petits incréments. Cette approche est généralement utilisée quand on a une bonne idée de l'algorithme qui sera implémenté.
– **Triangulation** : confronter plusieurs cas de tests pour l'UUT. Cette approche apporte du contraste et la sécurité nécessaire quand la solution à mettre en place n'est pas maîtrisée.

Puis, cerise sur le gâteau, arrive le TDD Mantra. On remarque l'utilisation du mot mantra, un terme fort pris au bouddhisme et qui place la pratique au niveau du culte et de la méditation continue. On a pu parler de boucle TDD et de boucle de refactoring, dont la combinaison représente une boucle de maintenance.