SOFTWARE CRAFT

TDD, Clean Code et autres pratiques essentielles

Cyrille Martraire
CTO et co-fondateur d'Arolla

Arnaud Thiéfaine
Développeur, coach craft et formateur chez Arolla

Dorra Bartaguiz
Développeuse, coach technique et formatrice chez Arolla

Fabien Hiegel
Développeur chez Shodo Nantes

Houssam Fakih
CTO de La Combe du Lion Vert

DUNOD

Illustration de couverture : © Dhujmontra – Shutterstock

11 rue Paul Bert, 92240 Malakoff
www.dunod.com
ISBN 978-2-10-082520-2

Table des matières

PARTIE 2
Techniques avancées et élargissements

PARTIE 3
Craft et attitudes

Avant-propos

Le Software Craft est une approche de développement logiciel qui connaît une certaine popularité depuis 2009 dans le monde entier. Cette approche, désormais revendiquée par de nombreux professionnels, s'inscrit dans le prolongement des approches agiles et les complète avec les aspects d'ingénierie.

◆ *L'essor du Software Craft dans le monde*

Chaque année des milliers de développeurs et développeuses se retrouvent lors du Global Day of Code Retreat dans 250 villes du monde pour une journée de pratique intensive autour d'exercices de code en simultané, durant 24 heures au total[1].

En outre, depuis 2011, des centaines de professionnels du développement logiciel se retrouvent durant trois jours dans un lieu tranquille pour échanger leurs expériences en pratiquant ensemble. Les plus connues de ces conférences sans programme préétabli, qu'on appelle ainsi des « non-conférences », sont les événements SoCraTES (International Conference of Software Craft & TESting) qui existent désormais dans de nombreux pays, dont bien entendu la France[2] à l'initiative de Houssam Fakih (un des co-auteurs de ce livre). À Paris, la communauté Software Crafters[3] créée par Cyrille Martraire (également co-auteur de ce livre) fin 2011 comptait en 2020 plus de 4 200 membres, dont environ 60 d'entre eux se retrouvent une soirée chaque mois pour échanger et pratiquer.

Les entreprises suivent et cherchent à adopter ces pratiques, dans une optique d'amélioration de leur processus de développement. Les compétences de Software Craft deviennent désirables voire requises sur les CV des candidats. Les entreprises veulent sponsoriser les communautés Software Crafters. Les entreprises de services se positionnent en réponse à la demande croissante de leurs clients. L'offre commerciale est aujourd'hui bien développée avec des formations, des formules d'accompagnement en coaching ou avec des « coachs craft embarqués », avec des personnes rompues au Software Craft qui intègrent une équipe pour lui transmettre les pratiques et techniques de l'intérieur.

Même si ce livre peut sembler tardif sur ce sujet, il reste pourtant absolument d'actualité, ce qui est étonnant dans une industrie où une technologie est souvent caduque après dix ans. Mais le Software Craft n'est pas une technologie de plus qu'on télécharge, qu'on installe et qu'on apprend avec un tutoriel avant de l'abandonner au profit d'une autre ; c'est un ensemble *d'attitudes* qu'on apprend à adopter pour être plus

---

1. Pour en savoir plus sur cet événement annuel, rendez-vous sur le site https://www.coderetreat.org
2. Toutes les informations sur l'événement SoCraTES sont disponibles ici : https://socrates-fr.github.io/
3. Le site officiel de la communauté Software Crafters Paris se situe à l'adresse suivante : https://www.meetup.com/fr-FR/paris-software-craftsmanship

efficace sur la façon d'utiliser les technologies, quelles qu'elles soient. Il s'agit d'acquérir de nouveaux réflexes, de faire le deuil de certaines illusions, de prêter attention à des enjeux qu'on ignorait ou négligeait auparavant. Il s'agit surtout de redéfinir ce qu'on appelle un « bon » logiciel. Et ce qui est remarquable avec les attitudes est qu'elles sont bien plus pérennes que les technologies sur lesquelles on les applique.

Changer sa vision du monde et sa façon de se comporter demande du temps, des efforts et un peu d'inconfort temporaire ; il faut considérer cela comme un investissement avec des bénéfices persistants sur une longue période. Apprendre le Test-Driven Développement (TDD) en Java, C#, Typescript ou dans tout autre langage consiste avant tout à adopter un état d'esprit porteur d'attitudes fondamentales et essentielles qui seront tout aussi utiles dans n'importe quel autre langage existant ou encore à inventer. Une fois les attitudes acquises, vous les appliquerez avec profit dans n'importe quel univers ; par exemple, vous voudrez probablement utiliser une approche similaire au TDD même sur des configurations d'infrastructures, des bases de données ou du cloud, et même dans des environnements orientés data ou machine learning.

Votre parcours vers le Software Craft consiste donc à pratiquer des techniques sélectionnées, idéalement en compagnie d'autres praticiens, en vue d'adopter de nouvelles attitudes. Pour cela, préparez-vous à être surpris !

◆ *Les enjeux du développement logiciel remis en question*

Votre voyage dans le Software Craft commence par changer ce qui vous paraît le plus important, en prenant conscience du fait que les enjeux suivants sont parmi les plus importants aujourd'hui :

Réussir à faire fonctionner le logiciel n'est que la première étape

Working code is a low bar.

Autrement dit, un logiciel qui fonctionne, ce n'est que le strict minimum. Le vrai enjeu du développement logiciel est de rester évolutif dans la durée, c'est-à-dire permettre d'ajouter facilement des fonctionnalités, et de rester maintenable, c'est-à-dire permettre de changer facilement des fonctionnalités existantes.

Un logiciel n'est jamais fini et change en permanence

Un logiciel, ça change tout le temps. Si le logiciel lui-même ne s'use pas, son utilité dépend en revanche de son adéquation au besoin, et ce besoin change régulièrement. Un logiciel n'est donc jamais fini, sauf lorsqu'il est mis à la poubelle en fin de vie.

En anglais le mot correspondant à « logiciel » est *software* et désigne ce qui est facile à changer, par opposition au *hardware*, le matériel. Pourtant, sans soin particulier, le code devient rapidement difficile à changer, jusqu'à parfois devenir totalement impossible à faire évoluer. C'est fâcheux et il faut alors envisager de le réécrire, ce qui est à la fois très coûteux et très risqué.

Développer c'est lire et comprendre le code au moins autant que l'écrire

Conséquence du point précédent, changer du code nécessite de le lire au préalable afin de le comprendre. Dans nos métiers, le code est bien plus souvent lu et relu qu'écrit (ou réécrit), et il convient donc d'en tirer les conséquences en termes de facilité de lecture.

Définir le besoin est au moins aussi difficile que d'écrire le code correspondant

Les développeurs ont l'habitude d'attendre une expression de besoin parfaitement définie dans chaque détail. Plus précisément, ils ou elles ont l'habitude de se plaindre de ne pas avoir une telle qualité de définition du besoin, reproches adressés aux analystes ou Product Owners.

Le Software Craft accepte l'idée que définir le besoin est éminemment difficile, avec comme conséquences, d'une part, que c'est un processus de découverte progressive, souvent même en cours d'écriture du code, et d'autre part que c'est une démarche collective incluant ceux et celles qui écrivent le code, au côté des futurs utilisateurs et des experts du domaine.

Pas de hacker ni de virtuose

Dans les communautés Software Craft, les termes *hacker* et *recherche de virtuosité* ne sont pas en vogue. Il ne s'agit pas d'impressionner par des astuces originales que seul leur auteur comprend. Il ne s'agit pas de taper le plus vite possible au clavier. Tout au contraire, c'est la réflexion sur l'opportunité même d'écrire ou non du code, puis de choisir parmi les alternatives possibles qui importent et qui méritent de prendre du temps.

Il ne s'agit pas de connaître par cœur des listes entières de commandes, alors que comprendre le métier et l'avoir en tête serait nettement plus utile. Il ne s'agit pas de faire du code qui prévoit tous les cas futurs, quand on a accepté que le futur sera toujours plein d'imprévus et qu'on préfère la simplicité et la capacité de changer facilement.

Il ne s'agit pas de trouver, dans son coin, une solution complète et qui fonctionne du premier coup au prix d'un énorme effort cérébral, alors que le travail se fait en équipe et que les problèmes vraiment intéressants sont trop complexes pour « tenir dans une seule tête ».

◆ *Retour historique sur une cascade de réactions dans l'industrie logicielle*

Pour bien comprendre Software Craft, il faut rappeler le contexte historique dans lequel il est apparu, à commencer par le *Manifeste agile* publié en 2001. Durant les années précédant ce manifeste, nombre de praticiens avaient proposé des approches ou méthodes de développement itératives, à cycle court, en accordant de l'importance à la communication entre les personnes impliquées et aux boucles de rétroaction.

Si la méthode Scrum est devenue depuis la plus populaire, l'approche Extreme Programming (XP) était la plus complète, combinant les pratiques d'organisation du

travail et d'ingénierie de code dans un tout cohérent. Extreme Programming, portant en substance les germes du craft, a connu au départ un succès d'estime et continue d'avoir une grande influence.

À partir de 2001, l'agilité, principalement avec Scrum, a connu un formidable essor jusqu'à devenir l'approche de référence de l'industrie du développement logiciel. La dynamique agile était une réaction au cycle en V de moins en moins adapté face à la complexité croissante des logiciels.

À peine dix ans plus tard, l'agilité avait remplacé progressivement le cycle en V dans la gestion du cycle de développement, mais trop souvent les pratiques d'ingénierie de code avaient été négligées. Les équipes ont gagné en efficacité de production du code, mais, à défaut de techniques d'ingénierie, elles empilent du code sur du code, et cette accumulation entraîne de la souffrance car elle devient de plus en plus difficile à entretenir. Des équipes ressentent alors ce que Sandro Mancuso appelle la « gueule de bois agile » (*the Agile hangover*).

Certaines figures connues du développement logiciel s'en émeuvent et réagissent. Robert C. Martin reprend le terme « Software Craftsmanship » issu du livre *Software craftsmanship: the new imperative* (2001, Addison-Wesley) et en fait un manifeste destiné à compléter le *Manifeste agile* (http://manifesto.softwarecraftsmanship.org) :

« *We have come to value:*

Not only working software, but also well-crafted software

Not only responding to change, but also steadily adding value

Not only individuals and interactions, but also a community of professionals

Not only customer collaboration, but also productive partnerships

That is, in pursuit of the items on the left we have found the items on the right to be indispensable. »

En initiant la dynamique Software Craftsmanship, l'idée est de rappeler l'importance des pratiques de code, souvent négligées depuis des années. Ces pratiques et techniques sont celles qui permettent qu'un logiciel évolue dans la durée sans se dégrader trop vite. Pour une large part, il s'agit de redonner une seconde vie à des éléments issus de l'Extreme Programming pour compléter Scrum. De façon caricaturale, on pourrait être tenté d'affirmer que le Software Craft apporte ce qui manque à Scrum pour reconstituer un semblant de Extreme Programming. Tout inexacte et grossière que soit cette affirmation, si vous connaissez ou apprenez Extreme Programming, vous avez, de fait, déjà un engagement significatif dans le Software Craft.

On a vu que l'agilité était une réaction au cycle en V et à la rigidité des processus de développement logiciel à la fin des années 1990 ; le Software Craft est à son tour une réaction à la dérive de l'agilité qui a oublié ou négligé ses racines techniques. Ironiquement c'est donc une réaction à une réaction. À ce titre, le Software Craft est aussi affaire de marketing, avec un terme qui devient populaire pour passer des idées et donner envie de changer son comportement. Ce n'est pas honteux, c'est même utile.

◆ *Une communauté bienveillante et soucieuse de la transmission des savoir-faire*

Notre expérience après dix ans dans des communautés Software Craft dans toute l'Europe et au-delà est de croiser des personnes ouvertes d'esprit et avec beaucoup de bienveillance. Ça fait très bisounours mais les exceptions sont rares. Lorsqu'il a été constaté que le nom même de « Software Craftsmanship » était genré au masculin, la décision de renommer en a été immédiate dans la plupart des communautés. La communauté de Paris s'est ainsi renommée en Software Crafters Paris.

En insistant sur l'excellence technique, le risque est de dériver vers l'élitisme, mais le Software Craft inclut un antidote qui est l'importance de la transmission des savoir-faire. Il s'agit non seulement de progresser soi-même vers l'excellence technique, mais d'aider d'autres, de tout niveau, à progresser aussi. Et bien entendu les deux objectifs se renforcent plus qu'ils ne s'opposent.

◆ *Une boîte à outils de techniques*

Le Software Craft comprend bien entendu une sélection de techniques formant une boîte à outils à bien connaître et à perfectionner sans cesse. Les techniques majeures sont :

- ✓ TDD (Test-Driven Development) ou « développement dirigé par les tests », qui n'est pas une technique de test mais bien une technique de développement et de conception, avec tous les refactorings.
- ✓ BDD (Behavior-Driven Development) ou « développement dirigé par le comportement », issu de TDD mais qui a développé un état de l'art propre, en particulier autour de l'expression du besoin.
- ✓ Techniques dites de propreté de code (Clean Code), qui rassemblent les pratiques-clés pour écrire du code propre à l'échelle de la ligne de code.
- ✓ Domain-Driven Design (conception dirigée par le métier), pour guider la conception avancée et l'architecture jusqu'à l'échelle du système d'information.
- ✓ Techniques spécifiques de « Legacy Remediation » (remaniement de code hérité), pour reprendre le contrôle, mettre sous tests et réparer du code ancien et mal entretenu.
- ✓ Techniques de conception orientée objet (OO) et style de programmation fonctionnelle (FP), principes de conception SOLID, etc.
- ✓ Techniques de collaboration, en particulier le binômage (pair programming) jusqu'au mob programming (programmation en groupe), car collaborer efficacement cela s'apprend et se travaille.
- ✓ Enfin, le Software Craft comprend aussi l'application et l'adaptation de tout ce qui précède pour les technologies comme le cloud, le machine learning, et les infrastructures data, qui sont plus récentes et par conséquent moins matures en termes de pratiques que la programmation généraliste traditionnelle.

Ces techniques sont décrites en détail dans les différents chapitres de ce livre, avec des exemples et exercices pour apprendre à les appliquer.

◆ *Comment lire ce livre*

Cet ouvrage peut être lu d'un bout à l'autre comme un parcours pédagogique progressif, ordonné par difficulté croissante et en commençant par les prérequis. Une approche conseillée serait de consacrer par exemple une semaine par chapitre, pour lire puis pratiquer le sujet avec les exercices proposés.

Ce parcours pédagogique reprend le contenu et la progression qui ont fait le succès du programme d'apprentissage de plus de 100 recrues de la société La Combe du Lion Vert, créée par les mêmes dirigeants que la société Arolla où se sont connus les auteurs.

Les exemples sont donnés en utilisant un des langages de programmation les plus populaires aujourd'hui, à savoir Java, JavaScript, C# ou Typescript. Les lecteurs et les lectrices pourront extrapoler les recommandations à leur langage habituel, ce qui ne doit pas poser de problème, le Software Craft étant largement agnostique aux détails de chaque langage.

Ce livre peut aussi se consulter comme un guide de référence, organisé par chapitres thématiques, pour répondre aux interrogations et hésitations que vous rencontrez dans votre pratique.

Chaque chapitre référence les ouvrages ou liens, le plus souvent en langue anglaise, qui seront utiles pour approfondir chaque sujet. Ce livre peut donc être vu comme une porte d'entrée vers un vaste univers, avec les conseils des auteurs pour se repérer et trouver plus vite les sources les plus inspirantes et les plus pertinentes.

◆ *À propos des auteurs*

Cyrille Martraire

Cyrille est CTO co-fondateur d'Arolla, une société qui rassemble 90 enthousiastes du Software Craft. Il est aussi le fondateur de la communauté Paris Software Crafters, qui compte aujourd'hui plus de 4 000 membres, et est l'auteur du livre *Living Documentation* chez Addison-Wesley. Cyrille est orateur régulier dans des conférences en Europe et au-delà.

Avec une expérience accumulée depuis 1999, Cyrille intervient toujours en conseil en architecture et en conception, avec l'aide de Domain-Driven Design et avec une vraie curiosité pour tous les métiers, de la finance à la logistique ou au médical.

Arnaud Thiéfaine

Arnaud est développeur depuis de nombreuses années, ainsi que coach craft et formateur. Arnaud apporte son expertise aussi bien auprès des grandes entreprises que des startups, en sensibilisant les équipes aux pratiques du craft au moyen de coding dojos, de *brown-bag lunches* et d'accompagnement des équipes.

Ayant à cœur de transmettre ses compétences et d'élever le niveau technique, Arnaud consacre également du temps à enseigner auprès des consultants du groupe Arolla, et anime régulièrement les meetups *Craft Your Skills* et *Jam de Code*, qui

permettent à des développeurs de tous horizons passionnés par le craft de se retrouver et de parfaire leur pratique.

Dorra Bartaguiz

Dorra est développeuse depuis 2008, coach technique et formatrice. Elle partage son savoir-faire en publiant des articles et en animant des conférences et meetups. Elle a aussi enseigné dans une école d'ingénieurs à Paris pendant des années.

Au fil de son expérience, elle a acquis des valeurs autour du Clean Code, du craft et de l'agilité qu'elle adore partager en accompagnant des clients à travers du coaching ou des formations pour aider les équipes à mieux apporter de la valeur.

Fabien Hiegel

Fabien est un développeur passionné ; quel que soit le langage de programmation, tout est prétexte à créer et à y prendre du plaisir. Par ses diverses expériences, il a découvert et aiguisé ses pratiques du craft, notamment par des ateliers de partage et de veille collective.

En animant des communautés de pratiques, en facilitant l'internalisation de formations ou encore en ludifiant des ateliers de pratique, il espère pouvoir transmettre cette passion aux personnes qui l'entourent et les aider à apprendre les compétences techniques qui les font rêver.

Houssam Fakih

Houssam Fakih est CTO, dés-organisateur de conférences, agent de carrières, formateur, coach technique et développeur. Il a créé avec ses équipes à La Combe du Lion Vert un programme de training d'un mois sur les compétences XP et craft. Suivi par près d'une centaine de développeurs, ce programme fut un vrai accélérateur de carrières.

D'autre part, il a contribué au lancement des premières versions des conférences open space SoCraTES dans plusieurs pays dont la France où il co-organise l'événement depuis 2015. En 2022, il facilitera l'événement pour la première édition de SoCraTES Maroc. Également orateur, il intervient dans plusieurs conférences sur des thèmes en rapport avec l'agilité et le craft.

◆ *À propos des illustrations*

Les illustrations de ce livre sont réalisées par notre co-auteure **Dorra Bartaguiz**, inspirée par les idées et les échanges avec tous les auteurs.

◆ *Remerciements*

Les auteurs se remercient mutuellement pour l'opportunité de collaborer sur cet ouvrage dans la durée, et ce dans une atmosphère amicale. Ils tiennent à remercier profondément Jean-Luc Blanc, des éditions Dunod, pour son soutien et ses

encouragements du projet initial jusqu'au manuscript final. Leurs remerciements vont aussi à Brice Martin, pour sa relecture.

Arnaud Thiéfaine

Arnaud remercie son épouse Delphine et ses enfants Anaïs et Noémie qui ont dû porter une part des sacrifices nécessaires à l'activité d'écriture. Elles ont su être discrètes pendant les phases demandant beaucoup d'attention et apporter leur soutien dans les moments les plus difficiles.

Arnaud tient également à exprimer sa gratitude envers Bruno Boucard et Thomas Pierrain pour lui avoir transmis la passion du craft en lui montrant qu'en matière de développement logiciel, une voie beaucoup plus vertueuse était possible.

Dorra Bartaguiz

Avec un bébé, l'aventure pouvait être compromise mais c'était sans compter sur le soutien et la patience de son chéri. Donc merci à lui pour tout.

Cyrille Martraire

Cyrille remercie particulièrement Yunshan son épouse, ainsi que Norbert et Gustave, qui, en plus d'être compréhensifs pour les moments où papa ne jouait pas avec eux, ont même pris goût à participer aux points de suivi vidéo avec Jean-Luc Blanc.

Fabien Hiegel

À Houssam qui, en plus d'avoir été un binôme enrichissant, m'a inclus et retenu dans cette aventure.

Houssam Fakih

Houssam remercie l'ensemble des alchimistes et coaches à La Combe du Lion Vert, J. B. Rainsberger, ses anciens collègues chez BISAM ainsi qu'Hadrien Mens-Pellens, Thomas Carpaye et Thomas Ployon.

Ce livre a été conçu avec la préoccupation permanente de rester accessible aux novices du craft. Nous n'avons pas l'ambition de proposer une bible exhaustive faisant référence sur le craft.

Ainsi, si vous déjà une bonne maîtrise du sujet, il se peut que vous trouviez des motifs d'insatisfaction dans nos choix de présentation. Nous sommes conscients des multiples façons d'aborder chaque thème, même si nous n'en avons retenu qu'une, qui n'est peut-être pas votre façon privilégiée. Enfin, pour garder le texte accessible, nous avons évité de mentionner tout ce qu'il aurait été possible de mentionner pour chaque sujet, mettant parfois de côté certaines des références les plus avancées.

Même lorsque le masculin est employé, on parle de personnes indépendamment du genre. Il peut arriver que certaines formulations aient échappé à notre vigilance : que cela ne soit en aucun cas considéré comme une volonté de discriminer.

PARTIE 1

Les pratiques incontournables du craft

Le Développement Dirigé par les Tests (TDD)

Le développement dirigé par les tests (TDD), proposé par Kent Beck à la fin des années 1990, est une technique de développement au sens large, dans la mesure où son impact ne se limite pas qu'aux tests. Cette technique incontournable et centrale constitue un des piliers de l'Extreme Programming (XP) et du craft.

1. 1 UNE TECHNIQUE POUR PROGRAMMER EFFICACEMENT DES FONCTIONNALITÉS COMPLEXES

Pour comprendre tout l'intérêt de cette technique, il est utile de préciser son origine. Traditionnellement, lorsqu'on apprend le développement logiciel, le focus est tout naturellement en premier lieu sur l'apprentissage du langage et de sa syntaxe, avec la satisfaction de réussir à faire fonctionner le code comme désiré. Puis, petit à petit, on écrit des applications plus complexes qui nécessitent davantage de réflexion.

L'accroissement du nombre de règles de gestion à prendre en compte oblige à bien organiser un nombre élevé d'étapes de traitements. Dans de nombreux cas, on parvient à le faire mentalement, mais, lorsque le problème à résoudre dépasse les capacités de notre cerveau, il devient nécessaire de s'aider d'un bloc-notes.

Réaliser une fonctionnalité sous forme de code consiste à en concevoir l'algorithme et imbriquer des `while`, des `if`, ajouter une clause `else` nécessaire pour gérer une alternative, stocker les états intermédiaires dans des variables, avant de prendre soin de vérifier les paramètres passés, sans oublier de gérer les exceptions… La multiplication de ces imbrications nous fait ressentir certaines inquiétudes :

- ✓ Comment démarrer, par quel côté démarrer les développements (ou comment éviter le syndrome de la page blanche) ?
- ✓ Comment s'assurer de ne pas oublier des détails essentiels lors du développement d'une fonctionnalité ?
- ✓ Comment savoir précisément que le développement d'une fonctionnalité est vraiment terminé ?
- ✓ Comment rester concentré sur le problème à résoudre, sans se laisser distraire par des détails non nécessaires ?

Dans son livre *Test Driven Development: By Example*[1], Kent Beck explique que le fait d'avoir peur est légitime quand on fait du développement. Attention, il ne s'agit pas de la peur dans le sens négatif, celle qui nous empêche d'avancer. Il s'agit plutôt de la peur qui fait réfléchir, de la peur de ne pas voir le bout de la problématique. Cette peur nous pousse à essayer diverses solutions pour découvrir laquelle est la mieux adaptée. Cette peur peut se montrer négative dans la mesure où elle risque aussi de nous empêcher de communiquer et de rechercher un feedback puisqu'on n'est pas certain du résultat. Pire encore, elle peut nous empêcher de démarrer.

Le TDD, par son aspect très encadré, se propose de réduire les peurs et frustrations.

1. 2 LES PRINCIPES ET LES ÉTAPES DU TDD

Le développement dirigé par les tests se caractérise essentiellement par le fait d'écrire le code de test avant le code de production. L'idée est de fixer l'objectif du besoin et de définir à l'avance le comportement attendu.

Dans le cercle des développeuses et développeurs, on dit souvent qu'un bon développeur est un développeur « fainéant ». Cette citation est applicable aussi à une développeuse. Le TDD vient renforcer cette idée. En effet, en appliquant le TDD, le but est d'écrire le moins de code possible tout en satisfaisant la fonctionnalité attendue.

1. Beck, K. *Test Driven Development: By Example*, Addison-Wesley Professional, 2002.

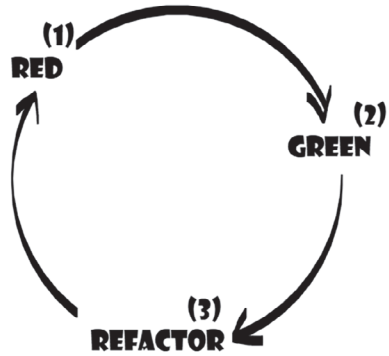
Robert C. Martin (surnommé « Uncle Bob ») évoque ainsi les trois règles du TDD dans son blog[2] :

- ✓ *You must write a failing test before you write any production code.*
- ✓ *You must not write more of a test than is sufficient to fail, or fail to compile.*
- ✓ *You must not write more production code than is sufficient to make the currently failing test pass.*

En français, cela donne :

- ✓ On doit écrire un test qui échoue avant d'écrire n'importe quel code de production.
- ✓ On ne doit pas écrire plus de tests que ce qui est nécessaire pour échouer ou ne pas compiler.
- ✓ On ne doit écrire que le code suffisant pour que le test actuellement en échec réussisse.

Le développement se fait en plusieurs itérations et chacune de ces itérations se décline en trois étapes : l'étape rouge, l'étape verte et l'étape de refactoring (en anglais : *red* / *green* / *refactor*). À chaque étape de l'itération, on va écrire le minimum de code pour satisfaire le besoin.



1.2.1 Étape rouge (red)

Lorsqu'on entreprend une nouvelle fonctionnalité en approche TDD, on écrit tout d'abord un test qui correspond au comportement attendu. Cette étape correspond à l'étape rouge, en référence à la convention employée par les IDE (pour Integrated Development Environment) pour signifier l'échec d'un test.

À ce stade, le code de production permettant de réaliser la fonctionnalité n'est pas encore développé. Par conséquent, le test n'est pas supposé réussir. En écrivant le test avant l'implémentation, on s'assure de cadrer et diriger l'effort de développement et ne pas se disperser. Il est donc très important de vérifier que le test échoue pour les bonnes raisons en s'assurant que les données d'entrée et le résultat attendu de notre test sont pertinents et correspondent bien au besoin attendu. En cas de doute, il est souhaitable de s'en assurer avec le métier (la ou les personnes qui ont exprimé le besoin).

2. https://blog.cleancoder.com/uncle-bob/2014/12/17/TheCyclesOfTDD.html

Et si le résultat initial de mon test est vert ?

Il arrive parfois que le test nouvellement ajouté réussisse immédiatement, sans même nécessiter l'ajout ou la modification de code de production. En approche TDD, ce n'est pas bon signe et cela peut être dû à une des raisons suivantes :

- ✓ le test peut être erroné : les valeurs du résultat attendu ou les données d'entrée sont fausses ;
- ✓ le comportement attendu a déjà été développé par accident lors d'une précédente itération.

1.2.2 Étape verte (green)

Une fois que le test est écrit, nous pouvons entrer dans la deuxième étape de chaque itération du TDD, l'étape verte (ici aussi, c'est la couleur adoptée par les IDE pour indiquer que les tests ont réussi). L'idée principale de cette phase est de faire passer le test le plus rapidement possible, avec un minimum d'effort, et le minimum de code pour répondre au besoin. Bien souvent, l'implémentation est très naïve et se limite à retourner une valeur en dur.

On évite ainsi de trop réfléchir, à contre-courant de l'approche habituelle consistant à passer du temps en amont pour faire une belle conception, avant de produire un code élégant et sophistiqué. Au contraire, nous cherchons à garder un code suffisamment simple et qui apporte immédiatement de la valeur en supportant un nouveau cas métier.

1.2.3 Étape de refactoring (refactor)

Lors de la dernière phase de l'itération, on cherche à améliorer le code pour une meilleure lisibilité. Il peut s'agir d'un renommage de variable. Ou encore de l'extraction d'un bloc de code vers une nouvelle méthode, ou bien de la création d'une nouvelle classe[3]. L'idée est d'utiliser au maximum l'environnement de développement pour faire les actions de refactoring[4], pour se prémunir des erreurs humaines et surtout éviter l'ajout d'un nouveau comportement non couvert par les tests.

1. 3 RIGUEUR NÉCESSAIRE À CHAQUE ÉTAPE

Pour chaque itération du TDD, la rigueur est de mise. Pour l'écriture du test, on s'assure de n'utiliser que les variables nécessaires. On n'initialise que des variables qui seront utilisées comme données d'entrée de notre fonctionnalité à tester. Pour la phase verte, il faut se contraindre à écrire le minimum de code. Quand on écrit une deuxième ligne de code ou rajoute une condition (un if par exemple), c'est le moment de se demander si on ne peut pas écrire moins de code.

---

3. Il existe beaucoup d'autres techniques de refactoring, décrites en détail dans le chapitre 5, dédié à ce sujet.
4. On pourrait le traduire en « refactorisation » mais le terme « refactoring » reste couramment utilisé en français.

Lors de la phase de refactoring, on se met à la place d'une autre personne arrivant pour la première fois devant ce code et on se pose la question : « Est-ce qu'on arrive à le lire et à comprendre à quel besoin il répond ? »

On est toujours tenté de brûler les étapes pour arriver à la solution finale. L'idée est de faire vraiment attention et d'avancer d'un pas sûr à chaque itération et de simplifier au maximum.

> **Remarque :** Pour éviter les valeurs de retour en dur dans le code, on peut utiliser la technique de triangulation.
>
> Cela consiste à créer plusieurs tests (au moins deux) pour la même règle de gestion, avec des données d'entrée différentes. Comme le code retourne une valeur en dur, le nouveau test sera rouge. Ainsi l'implémentation devient plus générique pour répondre à la règle de gestion. Cette technique concerne plusieurs itérations red/green/refactor. On ne peut pas l'implémenter dès la première itération.

1.4 ANATOMIE D'UN TEST

Dans la plupart des langages, un test est délimité par une méthode. En termes de découpage, chaque test sert à vérifier que le code de production est conforme à une règle de gestion sur un cas métier. Un test est caractérisé à la fois par son nom et par son corps. Pour que l'objectif d'un test soit facilement compréhensible, il est crucial d'être soigneux sur chacun de ces deux aspects.

1.4.1 Nommage du test

Dans l'idéal, le nom d'un test retranscrit la règle de gestion qui est testée. Prenons pour exemple le test d'une opération de division. Une des règles de gestion liée aux divisions consiste à lancer une exception si le dénominateur est égal à zéro. Le nom du test peut être par exemple :

```
Divide_should_raise_an_error_when_denominator_is_zero
```

Lorsque la règle de gestion est exprimée avec précision, on peut facilement déterminer la cause de l'échec du test.

Afin de conserver une certaine régularité dans les tests et en rendre la lecture plus fluide, on peut s'inspirer des patterns de nommage inspiré du Behaviour-Driven Development[5], à savoir `should`/`when` et `given`/`when`/`then`.

- ***Should/when***

`Should`/`when` se traduit en « devrait/quand ». L'idée est d'articuler le nom de la fonctionnalité à tester autour de ces mots-clés. Si on reprend l'exemple ci-dessus de la fonctionnalité de division par zéro, on obtient les possibilités suivantes :

5. Les notions propres au BDD sont plus largement décrites dans le chapitre 3, dédié à ce sujet.

```
Divide_should_raise_an_error_when_denominator_is_zero
Should_raise_an_error_when_denominator_is_zero
```

On peut aussi remonter le préfixe `DivideShould` directement au niveau de la classe de test et par conséquent simplifier le nom de la méthode de test :

```
Raise_an_error_when_denominator_is_zero
```

Le nom complet de la règle testée s'obtient ainsi en combinant le nom de la classe et celui de la méthode, ce qui donne :

```
DivideShould.Throw_an_invalid_operation_when_denominator_is_zero
```

◆ ***Given/when/then***

Ce format est quant à lui plus proche de la syntaxe Gherkin[6], et est destiné à faciliter les échanges entre les développeurs et les métiers. En effet, la traduction serait « étant donné que… / lorsque… / alors… ».

Si on applique ce format à notre exemple de la division par zéro, on obtient :

```
Given_five_as_nominator_and_zero_as_denominator_when_I_divide_
nominator_by_denominator_then_an_error_is_raised
```

Ce format peut sembler plus verbeux, mais il a le mérite d'être plus explicite, notamment sur les valeurs précises utilisées en exemple.

Sur les formats `should/when` et `given/when/then`, aucun n'est objectivement meilleur que l'autre, il s'agit surtout d'une sensibilité personnelle et du choix d'un standard de nommage des tests au niveau de l'équipe. Il arrive même que tous les tests d'une base de code ne partagent pas le même format. Le plus important est de faire en sorte que l'objectif d'un test apparaisse avec clarté à la lecture du nom.

1.4.2 Corps du test

Si le nom du test décrit ce que le test doit vérifier, son corps décrit comment il le fait. Le corps du test est divisé en trois sections :

- ✓ La première correspond à l'initialisation des données nécessaires pour le test.
- ✓ La deuxième correspond à l'appel de la fonctionnalité à tester.
- ✓ La troisième correspond à la vérification du résultat fourni par l'appel de la fonctionnalité.

Pour permettre une lecture fluide du test, il est important que ces trois parties soient bien délimitées et ne se chevauchent pas.

◆ ***Triple A***

Le découpage du corps de test se retrouve sous l'appellation anglophone de `Arrange/Act/Assert` qu'on peut traduire en « Préparer/Agir/Vérifier ». Lorsqu'on

6. Le langage Gherkin, qui permet d'exprimer des exemples sous une forme naturelle, est également présenté dans le chapitre 3, consacré au BDD.

débute en TDD, afin d'éviter de s'éparpiller, on peut, avant de commencer, baliser les trois sections au niveau de la méthode et ensuite se demander comment les remplir :

```
{
  // Arrange

  // Act

  // Assert

}
```

1.5 AUTRES CONSEILS SUR LES TESTS

Quel que soit le format utilisé, il ne faut pas avoir peur d'écrire un nom de test trop long. Le plus important est de rendre explicite la règle de gestion ainsi que la cause d'échec du test, le cas échéant.

Lorsqu'on débute dans les tests, de façon naïve, on a tendance à d'abord préparer les données d'entrée, puis appeler la méthode à tester et enfin vérifier le résultat. En procédant ainsi, on risque de créer plus de code que nécessaire.

Pour rester dans l'optique d'écrire le minimum de code possible, il vaut mieux commencer par la vérification. Ensuite comme le résultat à vérifier vient d'un appel à un traitement, on va écrire cet appel. Et comme ce traitement a besoin de données d'entrée, on va les initialiser. Cette nouvelle gymnastique, qui peut paraître difficile à appliquer, s'apprend rapidement en s'exerçant : d'autant plus que, avec les IDE actuels, la génération des variables et du code manquants est beaucoup plus fluide, donc on va facilement apprendre.

1.6 TDD EN PRATIQUE SUR UN KATA TRÈS SIMPLE

Pour bien comprendre le cheminement des étapes du TDD, prenons un exemple concret avec le kata[7] FizzBuzz, très simple (et célèbre pour être fréquemment utilisé dans les tests de recrutement).

1.6.1 Le kata FizzBuzz

Le kata FizzBuzz nous demande d'écrire un programme qui affiche, pour chaque nombre de 1 à 100 :

- ✓ 'Fizz' si le nombre est un multiple de 3 ;
- ✓ 'Buzz' si le nombre est un multiple de 5 ;

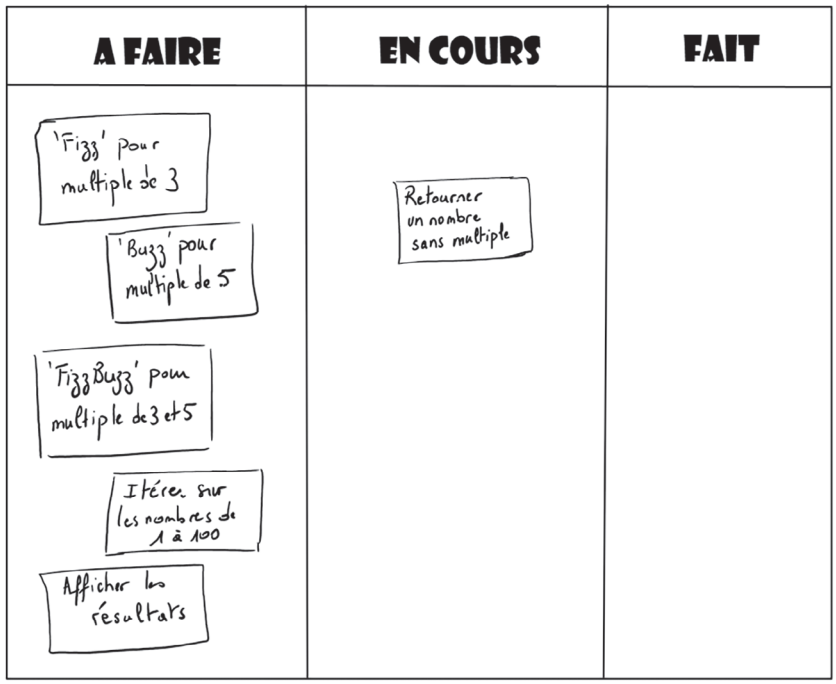
7. Dans les arts martiaux, l'efficacité en situation de combat s'acquiert par de l'entraînement délibéré sur des exercices précis, effectués à vitesse réduite : les katas. Le craft en a emprunté l'approche et le nom : les crafters s'entraînent régulièrement sur des katas pour acquérir de nouveaux savoir-faire et renforcer ceux déjà acquis. Un kata de code est un exercice de programmation qui permet de progresser par la répétition.
Chaque nouvelle répétition d'un kata offre la possibilité de faire mieux (en gagnant en rapidité et en fluidité), d'expérimenter une autre approche, ou de s'imposer de nouvelles contraintes (par exemple, en progressant par purs baby steps, ou en s'interdisant les if). De la métaphore des arts martiaux jaillit l'idée que « *de la répétition de la pratique naît la perfection* ».

- ✓ 'FizzBuzz' si nombre est un multiple de 3 et de 5 ;
- ✓ le nombre lui-même dans tous les autres cas.

Avant de commencer, il vaut mieux décomposer le problème pour faciliter l'évolution du développement.

Pour notre exemple, on peut séparer le calcul de l'affichage dans un premier temps. La méthode d'affichage va appeler la méthode de calcul. Cette dernière prend un nombre et retourne une chaîne de caractères. Dans un second temps, on peut séparer le calcul du résultat pour un nombre, de l'itération sur les nombres de 1 à 100. Ce découpage et sa priorisation doivent être faits bien sûr en concertation avec le métier.

Pour faciliter le suivi de l'avancement du développement, on peut imaginer chaque cas dans un Post-it sur un tableau à trois colonnes pour simplifier. Ces colonnes représentent les statuts d'avancement du développement (« À faire », « En cours » et « Fait »). Pour notre exemple, on imagine donc plusieurs Post-it comme présenté dans le tableau ci-dessous.



1.6.2 Lister les exemples de la fonctionnalité

Pour chaque règle de gestion citée, nous préparons un ou plusieurs exemples. On peut imaginer par exemple quatre cas de tests pour la méthode de calcul :

- ✓ si le nombre est égal à 1 alors on affiche '1' ;
- ✓ si le nombre est égal à 3 alors on affiche 'Fizz' ;
- ✓ si le nombre est égal à 5 alors on affiche 'Buzz' ;
- ✓ si le nombre est égal à 15 alors on affiche 'FizzBuzz'.

On peut imaginer une priorisation par rapport au pourcentage d'occurrences de mise en œuvre de la règle, pour apporter une implémentation qui apporte de la valeur au plus tôt au métier. On va donc avoir une majorité des nombres de 1 à 100 qui correspondent à la règle de gestion du premier cas de test. Cette dernière correspond à retourner le chiffre lui-même. Ensuite on s'occupera de la règle de gestion du retour de la valeur 'Fizz', suivie de celle du 'Buzz' et enfin 'FizzBuzz'. On peut toujours compter sur le métier pour confirmer la priorisation identifiée[8].

1.6.3 Première itération

Reprenons les étapes du TDD : rouge/vert/refactor (red/green/refactor). Pour la première itération, on va prendre l'exemple :

Pour le nombre 1, on affiche « 1 ».

◆ *Étape rouge / red*

On va donc commencer par écrire notre test sachant que rien n'est developpé encore. Comme on l'a précisé auparavant, le but est de faciliter la réflexion, de se fixer un objectif.

```
[TestMethod]
public void Should_return_same_number_when_it_is_not_multiple_of_three_or_five()
{
   FizzBuzz fizzBuzz = new FizzBuzz();
   int number = 1;
   string value = fizzBuzz.Print(number);
   Assert.AreEqual("1", value);
}
```

À ce moment, on n'a ni la classe `FizzBuzz`, ni la méthode `Print`. Écrire un code qui ne compile pas n'est pas intuitif mais avec l'approche TDD on apprend à lâcher prise et utiliser la puissance de l'IDE pour générer le code attendu. Tout dépend des langages et de l'IDE, la génération peut créer une méthode qui renvoie une exception de type `NotImplementedException`, ou qui retourne une valeur par défaut telle que zéro si la méthode retourne un entier.

```
public class FizzBuzz
{
 public string Print(int number)
   {
      throw new NotImplementedException();
   }
}
```

8. La décomposition et la priorisation de tâches sont également présentées dans le chapitre 7, consacré à l'étude détaillée du kata Fraction.

Il faut faire attention à l'étape rouge. Un code qui ne compile pas ne veut pas dire qu'on a fini l'étape. On a besoin que le code compile et que les tests s'exécutent bien sûr. Il faut que le test soit en état rouge pour indiquer que le résultat attendu n'est pas celui retourné par la méthode testée. Si le test est vert dès la première exécution alors il faut revoir le test, peut être que l'assertion ou les données d'entrée sont erronées.

Statut	Tests	Nom du test
✗	FizzBuzz.Tests	
✗		Should_return_same_number_when_it_is_not_multiple_of_three_or_five

◆ *Étape verte / green*

Pour cette étape, l'idée est d'écrire le minimum de code pour que le test passe au vert. C'est un peu difficile de se contraindre à écrire une seule ligne de code alors qu'on voit probablement d'avance l'implémentation finale. Avec notre connaissance du langage et de la plateforme, on se dirige naturellement vers un `return number.ToString()`, ce qui est parfaitement acceptable. Nous verrons toutefois dans la section 9.3 (« L'idée du TDD poussée au maximum : *TDD as if you meant it* ») une approche poussant la naïveté de l'implémentation et l'emploi des baby steps jusqu'au bout, certes contre-intuitive mais se révélant parfois, voire souvent, bénéfique. Sur ce premier exemple, on procéderait simplement à un `return "1"`.

```
public class FizzBuzz
{
    public string Print(int number)
    {
        return number.ToString();
    }
}
```

Statut	Tests	Nom du test
✓	FizzBuzz.Tests	
✓		Should_return_same_number_when_it_is_not_multiple_of_three_or_five

◆ *Étape de refactoring / refactor*

Une fois que le test est vert, on va modifier le code pour le rendre plus lisible et plus compréhensible. On peut aussi le refactorer pour une meilleure maintenabilité et pour simplifier l'ajout des évolutions. Le refactoring concerne autant le code de production que le code de test. Il peut correspondre au renommage de variables ou de classes pour que le code soit plus lisible. Mais il ne doit en aucun cas changer

le comportement de la fonctionnalité. Les tests doivent rester au vert après chaque action de refactoring. D'ailleurs, il est préférable de lancer les tests à chaque refactoring réalisé pour s'assurer de ne pas altérer le comportement développé.

Remarque : Certains IDE proposent la fontionnalité *live testing* (ou des plugins d'exécution de tests automatique). Quand on l'active, les tests sont lancés automatiquement dès l'enregistement d'une modification. Ainsi, on est au courant en temps réél si on altère un ou plusieurs tests.
Pour notre exemple, on peut voir qu'on a oublié de renommer la classe de test par exemple, donc c'est le moment de le faire.

◆ *Fin de l'itération*

La première itération du TDD pour cet exemple est finie. On va réappliquer ce cycle rigoureusement pour chaque exemple identifié lors de la décomposition du problème.

Au niveau de l'écriture du test rouge, on vérifie l'exemple en s'assurant des paramètres d'entrée et de sortie de la méthode à tester. Parfois, après l'implémentation du code de production, le test ne passe toujours pas au vert. Dans ce cas il vaut mieux revérifier les valeurs des données utilisées dans le test.

Pour passer les tests au vert, on se contraint à écrire le minimum de code pour éviter de développer en un seul coup plusieurs règles de gestion (dont certaines ne seraient donc pas couvertes par des tests).

Enfin pour l'étape de refactoring, on se met à la place d'un autre développeur découvrant le code pour la première fois. Le code est-il suffisamment explicite et compréhensible ? Que pouvons-nous apporter pour réduire le temps de lecture et favoriser la compréhension ?

1.6.4 Deuxième itération

Pour cette itération, on s'attaque à la règle de gestion consistant à afficher « Fizz » si le nombre est un multiple de 3. On prend donc comme exemple :

Pour le nombre 3, on affiche « Fizz ».

◆ *Étape rouge / red*

Le test couvrant ce cas est le suivant :

```
[TestMethod]
public void Should_return_fizz_when_number_is_multiple_of_three()
{
    FizzBuzz fizzBuzz = new FizzBuzz();
    int number = 3;
    string value = fizzBuzz.Print(number);
    Assert.AreEqual("Fizz", value);
}
```

Statut	Tests	Nom du test
✗	FizzBuzz.Tests	
✗		Should_return_fizz_when_number_is_multiple_of_three
✓		Should_return_same_number_when_it_is_not_multiple_of_three_or_five

◆ *Étape verte / green*

Pour cette étape, l'idée est d'écrire encore une fois le minimum de code pour que le test passe au vert. Inconsciemment on va ajouter un `if(number % 3 == 0)` pour retourner « Fizz ». Effectivement, c'est une solution simple pour répondre au test et le faire passer au vert, mais encore une fois on peut faire mieux si on suit la méthode « TDD if you meant it ». Pour cela on peut juste ajouter une condition encore plus simple qui est `if(number == 3)`. On verra dans le chapitre correspondant comment faire.

Pour ce chapitre, on va se contenter de la solution `if(number % 3 == 0)`. La méthode sera donc comme suit :

```
public string Print(int number)
{
    if (number % 3 == 0)
    {
        return "Fizz";
    }
    return number.ToString();
}
```

Statut	Tests	Nom du test
✓	FizzBuzz.Tests	
✓		Should_return_fizz_when_number_is_multiple_of_three
✓		Should_return_same_number_when_it_is_not_multiple_of_three_or_five

◆ *Étape de refactoring / refactor*

Pour améliorer la lisibilité du code, on peut modifier ce qu'on appelle le *magic number* par une constante pour mieux exprimer l'intention. Un magic number est une valeur numérique en dur dans le code dont la signification n'apparaît pas immédiatement. Le magic number peut ainsi être transformé en une constante dont le nom décrit le rôle.

Pour notre exemple, le magic number 3 peut être extrait en une constante `FizzMultiplier`, plus explicite.

◆ *Fin de l'itération*

La deuxième itération est finie. En enchaînant les itérations, il faut toujours s'assurer que tous les tests sont verts. En activant la fonctionnalité *live testing* de l'IDE, la boucle de feedback est très courte. Ainsi on peut rectifier le tir si on altère un test en revenant à un état stable et réfléchir à une meilleure solution. Dans la grande majorité des cas, lorsqu'un test échoue, c'est le dernier ajouté. Mais il arrive qu'on casse par inadvertance un des tests précédents, alors que le plus récent est au vert : si un test plus ancien passe au rouge, ça veut dire qu'on a apporté une régression à une fonctionnalité précédente. C'est pourquoi il est important de relancer régulièrement l'ensemble des tests, et pas seulement celui que l'on vient d'écrire.

1.6.5 Troisième itération

Pour cette itération, on se concentre sur la règle de gestion qui correspond à afficher « Buzz » si le nombre est un multiple de 5. Nous prenons l'exemple suivant :

Pour le nombre 5, on affiche « Buzz ».

◆ *Étape rouge / red*

Écrivons le test suivant (qui sera bien sûr rouge à l'exécution) :

```
[TestMethod]
public void Should_return_buzz_when_number_is_multiple_of_five()
{
    FizzBuzz fizzBuzz = new FizzBuzz();

    int number = 5;
    string value = fizzBuzz.Print(number);

    Assert.AreEqual("Buzz", value);
}
```

Statut	Tests	Nom du test
✗	FizzBuzz.Tests	
✗		Should_return_buzz_when_number_is_multiple_of_five
✓		Should_return_fizz_when_number_is_multiple_of_three
✓		Should_return_same_number_when_it_is_not_multiple_of_three_or_five

◆ *Étape verte / green*

Une fois de plus, on se restreint au minimum de code pour passer le test au vert et on ajoute une simple condition, qui vérifie si le nombre passé est multiple de 5, pour sortir au plus tôt. La méthode évolue ainsi :

```
public string Print(int number)
{
    if (number % 5 == 0)
    {
        return "Buzz";
    }

    if (number % FizzMultiplier == 0)
    {
        return "Fizz";
    }

    return number.ToString();
}
```

Statut	Tests	Nom du test
	FizzBuzz.Tests	
		Should_return_buzz_when_number_is_multiple_of_five
		Should_return_fizz_when_number_is_multiple_of_three
		Should_return_same_number_when_it_is_not_multiple_of_three_or_five

◆ *Étape de refactoring / refactor*

Ça commence à devenir une seconde nature : une fois que le test est au vert, on regarde ce qu'on peut refactorer. Comme au cycle précédent, on extrait une constante pour le magic number 5 pour gagner en lisibilité (cette fois encore, en s'aidant de l'IDE). La méthode sera donc comme suit :

```
public string Print(int number)
{
    if (number % BuzzMultiplier == 0)
    {
        return "Buzz";
    }
```

```
            if (number % FizzMultiplier == 0)
            {
                return "Fizz";
            }

            return number.ToString();
        }
```

1.6.6 Quatrième itération

Pour cette itération on va prendre le dernier exemple défini à la présentation du kata :

Pour le nombre 15, on affiche « FizzBuzz ».

- ***Étape rouge / red***

Le test ressemble au code suivant :

```
    [TestMethod]
    public void Should_return_fizzbuzz_when_number_is_multiple_of_three_and_five()
    {
        FizzBuzz fizzBuzz = new FizzBuzz();
        int number = 15;
        string value = fizzBuzz.Print(number);
        Assert.AreEqual("FizzBuzz", value);
    }
```

Statut	Tests	Nom du test
✗	FizzBuzz.Tests	
✓		Should_return_buzz_when_number_is_multiple_of_five
✓		Should_return_fizz_when_number_is_multiple_of_three
✗		Should_return_fizzbuzz_when_number_is_multiple_of_three_and_five
✓		Should_return_same_number_when_it_is_not_multiple_of_three_or_five

- ***Étape verte / green***

Si on regarde le corps de la méthode `Print()`, il est tentant de refactorer de suite pour pouvoir passer le test directement au vert. Mais il faut garder en tête que nous ne sommes pas encore en phase de refactoring. En phase verte, nous ne sommes autorisés qu'à écrire le minimum de code pour faire passer le test. Pour cela, on ajoute

une nouvelle condition, spécifique à notre nouveau cas de test et permettant de le faire fonctionner au plus tôt :

```
public string Print(int number)
{
    if (number % FizzMultiplier == 0 && number % BuzzMultiplier == 0)
    {
        return "FizzBuzz";
    }

    if (number % BuzzMultiplier == 0)
    {
        return "Buzz";
    }

    if (number % FizzMultiplier == 0)
    {
        return "Fizz";
    }

    return number.ToString();
}
```

Statut	Tests	Nom du test
✓	FizzBuzz.Tests	
✓		Should_return_buzz_when_number_is_multiple_of_five
✓		Should_return_fizz_when_number_is_multiple_of_three
✓		Should_return_fizzbuzz_when_number_is_multiple_of_three_and_five
✓		Should_return_same_number_when_it_is_not_multiple_of_three_or_five

- ***Étape de refactoring / refactor***

Maintenant qu'on a tous les tests au vert, une nouvelle phase de refactoring peut commencer. Les chaînes de caractères « Fizz » et « Buzz » apparaissent deux fois, elles peuvent être extraites en constantes. À l'aide de l'IDE, on peut directement sélectionner la valeur 'Buzz' pour l'extraire en constante. L'outil, fort serviable, va même jusqu'à nous proposer de remplacer directement toutes les occurrences. L'effet sur notre code est alors le suivant :

```
        public string Print(int number)
        {
            if (number % FizzMultiplier == 0 && number % BuzzMultiplier == 0)
            {
                return FIZZ + BUZZ;
            }
            if (number % BuzzMultiplier == 0)
            {
                return BUZZ;
            }
            if (number % FizzMultiplier == 0)
            {
                return FIZZ;
            }
            return number.ToString();
        }
```

Si on observe plus attentivement les conditions pour vérifier le multiplicateur, une forme de duplication commence à nous sauter aux yeux. Il est possible d'éliminer cette duplication en l'extrayant, dans une méthode par exemple. Une fois encore, l'IDE peut se charger gracieusement de cette besogne. Le code de la méthode peut ressembler à ce qui suit :

```
        public string Print(int number)
        {
            if (IsMultipleOf(number, FizzMultiplier) &&
IsMultipleOf(number, BuzzMultiplier))
            {
                return FIZZ + BUZZ;
            }
            if (IsMultipleOf(number, BuzzMultiplier))
            {
                return BUZZ;
            }
            if (IsMultipleOf(number, FizzMultiplier))
            {
                return FIZZ;
            }
            return number.ToString();
        }
        private bool IsMultipleOf(int number, int divider)
        {
            return number % divider == 0;
        }
```

Une autre alternative consiste à refactorer la construction des retours « Fizz », « Buzz » et « FizzBuzz ». Voici ce que ça donnerait :

```
public string Print(int number)
{
    string fizzBuzzValue = null;
    if (IsMultipleOf(number, FizzMultiplier))
    {
        fizzBuzzValue += FIZZ;
    }
    if (IsMultipleOf(number, BuzzMultiplier))
    {
        fizzBuzzValue += BUZZ;
    }
    if (fizzBuzzValue != null)
    {
        return fizzBuzzValue;
    }
    return number.ToString();
}
```

À partir du moment où on considère que le code est suffisamment lisible et facilement maintenable et extensible, on peut décider d'arrêter de refactorer[9].

1.7 CONCLUSION

Le but du TDD est d'aider à définir un objectif à atteindre (sous la forme d'un test en échec), d'y parvenir le plus rapidement possible (en faisant passer le test avec un minimum de code et d'efforts), et ensuite de prendre le temps de rendre le code suffisamment lisible (par du refactoring).

Nous avons insisté sur la nécessité d'écrire le minimum de code pour passer le test au vert. En effet, on considère que, en écrivant le moins de code possible, on réduit le délai du feedback et on avance sereinement sur les développements. Par ailleurs, comme on limite la réflexion au strict nécessaire, le design qui en résulte est beaucoup plus simple. Et en gardant la réflexion et l'amélioration du design pour la phase de refactoring, on ouvre les portes vers du design émergent. En d'autres termes, les tests orientent le design vers ce qui est strictement nécessaire pour répondre au besoin, garantissant ainsi un maximum de simplicité.

La pratique du TDD se trouve renforcée par les IDE modernes, qui permettent de lancer les tests en continu à chaque modification et d'être prévenu immédiatement

9. D'une certaine manière, le refactoring est quelque chose d'infini : en cherchant bien, on peut toujours trouver des possibilités d'amélioration. C'est pourquoi il est important de savoir s'arrêter une fois qu'on atteint un code qui paraît satisfaisant.

en cas de régression. Ces outils, par la présence de fonctionnalités de génération du code manquant (lors de la phase rouge) et de refactoring, permettent de gagner en productivité, fluidité et efficacité.

En appliquant la méthode TDD, la couverture de code sera totale puisqu'on respecte les trois règles d'Uncle Bob. Ainsi ces tests deviennent notre harnais de sécurité contre les régressions qui peuvent venir se glisser dans le code par inadvertance.

Vous avez compris le principe ; maintenant, à vous de jouer. Rappelons qu'à chaque itération vous devez toujours garder tous les tests au vert.

Points de contrôle / checkpoints

La pratique de TDD permet par ailleurs une discipline très élevée de gestion de versions et de sauvegarde régulière des changements de code (*commit* en anglais). Un commit marque l'ajout d'un incrément à notre fonctionnalité. C'est pourquoi on conseille souvent de créer un commit dès qu'un test passe au vert.

Au cours d'une itération de TDD, dans cette approche, on va donc sauvegarder le code deux fois :

- ✓ quand le test réussit (à la fin de la phase green) ;
- ✓ et après un refactoring qui garde les tests au vert.

En revanche, si un test échoue après un refactoring, on ne doit pas créer une sauvegarde (ou un commit) puisqu'on a fait un changement qui ne correspond pas à ce qui est attendu.

On peut même annuler les changements effectués (*rollback* en anglais) pour trouver une autre solution de refactoring. C'est tout l'intérêt des petits commits. Ça nous évite de revenir trop loin dans les modifications et ça sécurise les incréments.

Techniques et principes de propreté de code (Clean Code)

En apprenant à programmer, nous avons été habitués à écrire du code dans le but qu'il soit syntaxiquement correct et qu'il fonctionne, c'est-à-dire que la machine l'interprète et l'exécute correctement. La machine n'émet aucun jugement de valeur sur le code : soit il est exécutable, soit il ne l'est pas.

Seulement, un certain nombre d'enseignements sur la programmation ont tendance à oublier que la machine n'est pas la seule destinataire du code. N'importe quel programme va vraisemblablement être repris par d'autres développeurs, qui auront besoin de comprendre de quoi il retourne. Or cet aspect de lisibilité du code est trop fréquemment mis de côté.

2. 1 PROGRAMMER COMME UN EXERCICE DE COMMUNICATION

La réalité du métier de développeur est qu'on passe nettement plus de temps à relire et essayer de comprendre le code des autres (lorsqu'il est mal ficelé, cryptique et piégeux) qu'à écrire son propre code. Dès lors, il devient important de se préoccuper des autres développeurs lorsqu'on écrit du code.

Ceci est résumé à merveille dans cette citation de Martin Fowler[1] :

« *Any fool can write code that a computer can understand. Good programmers write code that humans can understand.* »

On regroupe sous le nom de Clean Code les principes et techniques qui permettent de tendre vers du code non seulement lisible et compréhensible par les autres, mais également facile à maintenir et à faire évoluer et exempt de bugs.

1. Fowler, M. *Refactoring : comment améliorer le code existant,* Dunod, 2019.

Michael Feathers, pour sa part, estime que la qualité majeure du code propre est de donner l'impression d'être écrit par quelqu'un qui fait attention à la qualité de son travail :

« *I could list all of the qualities that I notice in clean code, but there is one overarching quality that leads to all of them. Clean code always looks like it was written by someone who cares.* »

Dans ce chapitre, nous expliquons de façon accessible un certain nombre de principes et techniques essentiels qui permettent d'accroître sensiblement la qualité du code. Nous commençons par les lignes de conduite orientant vers un code de bonne qualité, avant de rentrer dans les détails et de décrire les principes historiquement associés au Clean Code. Enfin, nous donnons quelques idées permettant de favoriser la qualité du code au sein d'une équipe.

2. 2 L'IMPORTANCE D'ÊTRE EXEMPLAIRE

En sociologie, la **théorie de la vitre brisée** part du principe que la moindre dégradation dans un environnement urbain, si elle n'est pas rapidement réparée, va entraîner d'autres dégradations successives et rapides. Ainsi, lorsqu'une vitre cassée d'un bâtiment n'est pas immédiatement remplacée, les autres vitres vont rapidement suivre, parce qu'en l'absence de réaction on peut penser que la bâtisse est abandonnée. C'est le point de départ d'un cercle vicieux.

Cette théorie suggère que l'être humain a une propension au laisser-aller face à un environnement dégradé, et qu'à l'inverse il éprouve des scrupules à ne pas respecter les règles dans un environnement sain. Cette analogie, déclinable dans de nombreuses situations, est également vraie dans le cadre d'une base de code.

Face à ce genre de dérive, l'exemplarité doit primer : lorsqu'un ou plusieurs développeurs ont à cœur la propreté du code, les autres sont rapidement incités à faire de même. Il faut aussi appliquer un principe de tolérance zéro : dès qu'on se met à s'autoriser des négligences qu'on estime acceptables, on ouvre la porte à des négligences futures plus graves.

Ces conseils sont bien beaux, et paraissent pouvoir s'appliquer à des bases de codes déjà saines, mais que faire lorsque le code d'une application est déjà dans un état déplorable ? Pouvons-nous améliorer les choses ? Nos efforts ne risquent-ils pas d'être vains ?

Il serait regrettable de se résigner. Sans se lancer dans des réécritures de grande ampleur du code legacy (d'autres chapitres de ce livre sont dédiés à ce sujet), on peut néanmoins chercher à améliorer le code par petites touches successives, même lorsqu'elles paraissent insignifiantes (parfois un simple renommage de variable ou un reformatage de code apporte beaucoup). Cette attitude rejoint ce qu'Uncle Bob appelle la **règle du boy-scout**. Selon lui, quand on touche à une portion de code, on devrait la laisser dans un état meilleur que celui où on l'a trouvée. Cette règle

constitue l'opposé vertueux de la théorie des vitres brisées, qu'on pourrait même être tenté de qualifier de *théorie des vitres réparées*. Lorsque tous les développeurs respectent la règle du boy-scout, le code va progressivement s'améliorer et chacun prendra de plus en plus de plaisir à travailler.

Même si c'est exigeant et contraignant, au fil de la pratique et de la montée en expérience, ça va devenir une seconde nature.

2.3 PRINCIPES ESSENTIELS POUR UN CODE LISIBLE ET ÉVOLUTIF

Plutôt que de commencer par fournir un catalogue de techniques, principes et autres patterns dédiés à l'écriture de code propre, nous préférons nous concentrer en premier lieu sur les questions essentielles à considérer pour s'orienter vers du code de qualité :

- ✓ Le code n'est-il pas trop complexe pour ce qu'il fait ? N'est-il pas possible de le simplifier ? Les abstractions et autres généralisations introduites le sont-elles à bon escient ? Est-ce qu'on n'anticipe pas trop sur un possible futur ?
- ✓ L'intention du code est-elle claire ? Posé autrement, est-ce qu'on comprend à la première lecture ce que le code cherche à faire ?
- ✓ Le code ne tend-il pas à se répéter ? Est-il pertinent d'en réduire la duplication ?
- ✓ Le design mis en place autorise-t-il suffisamment de liberté pour la suite ?

D'une manière générale, on doit chercher à guider les développeurs qui vont reprendre notre code après nous, à réduire leur charge mentale de façon qu'ils puissent se concentrer sur l'essentiel.

2.3.1 Rester simple et aller à l'essentiel

La pratique du développement, par la quantité et la complexité des concepts mis en œuvre, nécessite un minimum de connaissances, de compétences et d'esprit logique. Or, en tant que développeurs, nous avons souvent une tendance naturelle (ou narcissique, en voulant prouver notre supériorité) à écrire et concevoir les choses de façon compliquée. Cette tendance est, bien entendu, totalement contre-productive, pour plusieurs raisons. Tout d'abord pour les autres : il y a un risque non négligeable que notre projet soit maintenu à l'avenir par des personnes moins expérimentées, qui donc auront du mal à percevoir les subtilités de notre code. Ensuite, pour nous-même : la complexité d'un projet ayant tendance à croître exponentiellement, nous risquons d'en être la première victime, alors autant commencer par faire des choses simples.

Cela peut paraître paradoxal, mais implémenter de façon simple et compréhensible par les autres demande en réalité beaucoup de soin, d'astuce, d'expertise et d'efforts. Cela demande de prendre du temps de réflexion, de s'appuyer sur la pratique et l'expérience accumulée, pour arriver à sentir le juste niveau de complexité qui convient à chaque situation. D'une manière générale, le code ne devrait pas paraître

plus complexe que le problème métier qu'il cherche à résoudre ; sinon, comme Arnaud Lemaire[2] aime à nous le rappeler, on ajoute à la complexité essentielle du métier la complexité accidentelle de notre code.

Voici des acronymes utiles à garder à l'esprit :

- ✓ KISS, pour *Keep It Simple and Stupid*, nous invite à la simplicité, et à ne pas rendre le code plus compliqué que nécessaire.
- ✓ YAGNI, pour *You Ain't Gonna Need It*, nous empêche d'écrire du code superflu[3] et nous invite à rester concentrés sur l'essentiel.

La pratique stricte du Test-Driven Development offre également un bon point de départ : le design qui en émerge est naturellement simple et nous empêche de produire plus de code que nécessaire.

2.3.2 Quatre règles simples pour un design simple

Ce n'est pas parce qu'on nous enjoint à écrire du code simple que c'est facile à faire. En réalité, écrire du code simple n'est pas simple. Afin de nous guider, Kent Beck a proposé les *Four Rules of Simple Design*[4].

En premier lieu, le code doit fonctionner comme prévu, c'est-à-dire **passer les tests** avec succès. En absence ou carence de tests, le fonctionnement attendu est nébuleux. Quand on ne sait pas exactement ce qu'on doit faire, il est difficile de tendre vers un design épuré. Lorsque la robustesse du code est garantie par les tests, alors nous avons suffisamment confiance pour le faire évoluer. Les changements seront d'autant plus aisés et fréquents avec des tests qui tournent rapidement.

Ensuite, il est crucial de **révéler l'intention**. Le code doit faire preuve de clarté, être facile à comprendre. Rappelons qu'on écrit surtout pour des humains. La machine a juste besoin de savoir comment faire, alors que pour les relecteurs du code il est essentiel de pouvoir comprendre ce que son auteur cherchait à accomplir (le quoi et le pourquoi). Le nommage, que nous détaillerons plus loin, fournit un atout majeur pour véhiculer efficacement l'intention.

La troisième règle nous encourage à **éviter la duplication**. Cette règle se situe dans le prolongement d'un autre acronyme, DRY (pour *Don't Repeat Yourself*). D'une manière générale, toute connaissance (règle métier, élément de configuration…) ne devrait être exprimée qu'une fois dans le code. On cible vraiment la duplication de connaissance dans le code plutôt que la duplication de lignes de code. Ainsi, si une connaissance vient à changer, les impacts doivent se limiter à un seul endroit dans le code. Limiter la duplication constitue un moyen efficace de renforcer la maintenabilité du code.

---

2. Lemaire, A. *Certaines complexités sont plus utiles que d'autres*. https://www.lilobase.me/certaines-complexites-sont-plus-utiles-que-dautres/

3. On ne sait jamais, ce code pourrait servir dans six mois si on nous demande telle ou telle fonctionnalité… ou pas.

4. Kent, B. *XP Simplicity Rules*. http://wiki.c2.com/?XpSimplicityRules

Enfin, il est important de rester **petit** et de ne conserver que ce qui est important. Tout ce qui ne sert pas les trois premières règles devrait être éliminé : on retrouve ici la nécessité de supprimer ce qui est inutile, comme le code mort, ou encore de résister à la tentation de concevoir du code complexe en prévision d'un avenir hypothétique. L'objectif est d'aider à réduire la charge mentale associée à une base de code en n'introduisant pas de code superflu.

Ces quatre règles ont le mérite d'être assez faciles à retenir, avec une rentabilité réelle sur la qualité et la simplicité du code produit.

Ne conserver que deux règles ?

Selon J. B. Rainsberger[5], ces quatre règles peuvent même être raffinées. D'abord, passer les tests fait partie de la routine de TDD et c'est donc quelque chose qui vient implicitement avec l'activité de programmer. Cela rend la quatrième règle également caduque : le respect des trois lois du TDD et le design émergent amènent à rester petit et à ne pas écrire de code superflu. Il reste un point qui pourrait être plus précis : exprimer l'intention. Pour cela, il est proposé de renommer cette règle en *amélioration du nommage*. Ainsi, les règles essentielles du design simple ne seraient plus qu'au nombre de deux (ce qui est encore plus facile à retenir) : **améliorer le nommage** et **réduire la duplication**.

2.3.3 Exprimer l'intention

Exprimer l'intention dans le code, c'est expliquer ce qu'on cherche à faire et pourquoi on veut le faire plutôt que de détailler comment on le fait. Les principes exposés précédemment, renforçant l'abstraction et l'encapsulation, contribuent à leur façon à l'expression de l'intention. L'objectif de cette section est de décrire des pratiques qui aident à expliciter l'intention :

- ✓ en mettant l'accent sur le nommage approprié des éléments du code (classes, méthodes, attributs, variables) ;
- ✓ en organisant et en découpant le code de façon à en faciliter la découverte et la navigation ;
- ✓ en apprenant à utiliser les commentaires avec parcimonie (c'est-à-dire se restreindre à un emploi vraiment utile).

◆ *L'art du nommage*

L'effort sur le nommage vise à fournir des noms révélant explicitement l'intention. La connaissance du métier se révèle précieuse dans le travail de nommage, qui devrait être aussi conforme que possible à la terminologie employée par les experts du domaine. Ainsi, on s'évite d'être trop imaginatif au départ et surtout on limite le mapping mental entre le code et les concepts métiers par la suite.

5. Rainsberger, J. B. https://blog.jbrains.ca/permalink/the-four-elements-of-simple-design

C'est peut-être l'heuristique la plus importante dans le nommage : s'appuyer sur le métier. Néanmoins, les astuces qui suivent aident à rendre le code plus lisible :

- ✓ Mettre de côté les noms imprononçables, notamment quand ils comportent des acronymes ou abréviations ; on peut faire exception à cette règle quand le raccourci lexical est couramment employé par le métier.
- ✓ Éviter les termes fourre-tout qui ne véhiculent pas de connaissance tels que « Data », « Info » ou « Manager » : ils ont tendance à perturber la lecture en ajoutant du bruit.
- ✓ Privilégier des noms communs pour les classes et des verbes pour les méthodes.

Parfois, l'absence de nom correct ou un nom très long peuvent être révélateurs d'un code qui n'est pas bien organisé, mal découpé, qui fait trop de choses ou dont les éléments ne vont pas ensemble. Passer à d'autres actions sur le code permet souvent de débloquer la situation avant de revenir sur le nommage.

Un processus itératif

Trouver des noms appropriés se révèle souvent très ardu. Parfois le nom vient facilement, et c'est tant mieux. Mais souvent aucun nom réellement satisfaisant ne vient à l'esprit. Ce n'est pas nécessairement un drame : tout nom qui apportera un peu plus de connaissance que le précédent sera bon à prendre. Il faut faire preuve d'un lâcher-prise dans la recherche de noms parfaits, en gardant à l'esprit qu'un nom n'a rien de définitif : si nécessaire, on fera mieux dans une prochaine étape de renommage, au fur et à mesure que notre compréhension progressera.

Ne pas trouver le nom parfait immédiatement peut même se révéler sain, à condition de faire évoluer itérativement le nommage. Surtout, on ne s'interdit pas de renommer fréquemment un élément du code étant donné qu'avec les IDE actuels cette action est quasi gratuite.

Les étapes de renommage

Les quatre étapes de renommage préconisées par Kent Beck, et clarifiées par J. B. Rainsberger sur son blog, sont les suivantes :

- ✓ *Nonsense* (« sans signification ») : on ne sait pas ce que le code extrait fait, on donne juste un nom, par exemple `foo()`.
- ✓ *Accurate-but-vague* (« ciblé mais vague ») : on exprime l'idée générale de ce qui est réalisé par le code, par exemple `computeCost()`.
- ✓ *Precise* (« précis ») : on se rend compte que le code fait un peu plus, on renomme `findItemThenComputeCost()` ; les adverbes tels que `then` et les conjonctions (`and`, `or`) nous indiquent qu'il y a plus d'une responsabilité. On peut alors aller plus loin en découpant la méthode.
- ✓ *Intention-revealing* (« révélant l'intention ») : on se rend compte que la méthode `computeCost()` fait d'autres choses, comme le calcul des taxes et leur ajout pour calculer le prix net. On pourrait renommer en `computeNetCost()` ou procéder à davantage de découpage.

Ce processus est vertueux parce qu'il permet de mettre en évidence des opportunités de refactoring supplémentaires au renommage.

Il reste important de garder à l'esprit qu'il est contre-productif de chercher le nom parfait tout de suite, ça peut faire perdre beaucoup de temps en discussions interminables. Il faut considérer le renommage comme un processus itératif, dans lequel on améliore le niveau de compréhension du code par petits incréments.

Le nommage est un super-pouvoir pour révéler l'intention. Comme tout pouvoir, il s'accompagne de grandes responsabilités. Un nommage erroné ou approximatif peut induire en erreur et engendrer la création de bugs, d'autant plus lorsque le nommage ment par omission (par exemple : `getPrice()` au lieu de `getAndUpdatePrice()`). Si le code fait quelque chose de risqué, de mal maîtrisé, ou dont on est moyennement fier, il est préférable de le mentionner (par exemple : `performDarkMagicCalculation()`, `doSomethingEvilToTheDatabase()`, `computePriceSomehow()`).

Pièges à éviter et autres anti-patterns

- ✓ Nommage en fonction du type

  Lorsque le nom d'une variable ou d'un paramètre correspond au nom de son type, on se trouve face à une redondance d'information, déjà portée par le type. Dans ces situations, il existe quasi toujours une meilleure possibilité de nommage. Il est préférable d'expliciter par le nommage à quoi sert la variable ou le paramètre, et quel est son rôle dans ce contexte d'utilisation. On évitera aussi les noms (plus ou moins courts) habituels qui dénotent le type tels que `s` ou `str` pour les chaînes de caractères ou `n`, `x`, `num` ou même `number` pour les nombres, qui n'apportent pas davantage d'information. Et dans la mesure du possible, on préférera `position` ou `index` à `i` et `j` pour les compteurs de boucles dès qu'ils sont passés en paramètres, ou encore mieux on se passera de compteurs de boucles en adoptant un style de programmation plus fonctionnel.

- ✓ Noms de paramètres et d'arguments identiques

  Le plus souvent par paresse, on est amené à utiliser pour une variable passée en argument le même nom que le paramètre déclaré. Différencier le nom de la variable passée en argument est un bon moyen d'exprimer l'intention localement et de fournir un peu de contexte.

  Par exemple, la méthode `toUpperCase(String text)` nomme son paramètre `text` pour être utilisée de façon très générale. Mais pour l'utilisation précise de mettre en capitales un nom de famille, on choisira `lastname` pour la variable passée en argument, et pas `text`.

- ✓ Paraphrasage maladroit

  Parfois, pour nommer une fonction on fournit un nom à rallonge, décrivant par le menu tout ce que fait la fonction. Cela donne des noms difficiles à lire mais aussi à maintenir (obligeant à aligner le nom lors du moindre changement de comportement de la fonction… lorsqu'on pense à le faire). On peut aussi prendre de la hauteur et changer de niveau d'abstraction, en se demandant quel service global la fonction nous rend, ou dans quelle situation on souhaiterait l'utiliser. Cela revient, une fois encore, à séparer le pourquoi (dans le nom et la signature de la fonction) du comment (le corps de la fonction).

Il faut garder à l'esprit que le nommage est une occasion en or de transmettre de la connaissance qui ne pourrait pas être déduite par une lecture, même attentive, du code.

Voici un exemple avec la condition suivante :

```
if (person.numberOfInjections >= 2
   || (person.contractedCovid && person.numberOfInjections >= 1)
   || person.hasPCRNegativeForLastSevenDays(today))
```

Pour laquelle on pourrait extraire une fonction :

```
boolean isSanitaryPassOk(Person person, Date atDate)
```

Ce sera d'autant plus commode pour les clients de cette fonction que les conditions d'attribution d'un passe sanitaire sont amenées à évoluer régulièrement.

2.3.4 Structurer le code

◆ *L'art du storytelling*

Lorsqu'on raconte une histoire, il est important d'avoir un cheminement pertinent avec un début, un milieu et une fin, de ne pas passer sans arrêt du coq à l'âne et de ne pas noyer l'auditeur avec une foule de détails insignifiants, sous peine de le perdre rapidement. Ces préoccupations doivent aussi s'appliquer au code pour en faciliter la lisibilité et la compréhension globale. D'une certaine manière, le code doit raconter une histoire, et si possible en accord avec le problème que le code est supposé résoudre.

Les compétences-clés pour y parvenir consistent à bien découper le code et, dans la foulée, à bien l'organiser. En fin de compte, le code doit se parcourir comme une table des matières, afin d'avoir la vision d'ensemble, avec la possibilité de naviguer dans les détails (facilitée par les IDE modernes) lorsqu'on a besoin d'en savoir plus.

◆ *De la vision globale aux points de détails*

On trouve trop souvent dans le code des fonctions qui résolvent un problème complexe en plusieurs dizaines, voire centaines (et, même si c'est heureusement beaucoup plus rare, parfois milliers) de lignes de code. De telles fonctions sont difficiles à comprendre, parce qu'elles nécessitent de comprendre tous les tenants et aboutissants du problème dans ses moindres détails, et se retrouvent systématiquement qualifiées de « code spaghetti ». Le code se retrouve dans un tel état lorsqu'on n'a pas pris le temps de décomposer le problème principal en problèmes plus petits (selon l'approche « diviser pour mieux régner »).

2.3.5 Découper les fonctions

Les fonctions (ou méthodes dans les langages orientés objet) doivent rester petites, en se limitant à quelques lignes de code. Si on ressent le besoin de faire plus long, c'est qu'on peut certainement découper en sous-fonctions. La signature d'une fonction doit se focaliser sur le service rendu par la fonction, c'est une sorte de contrat qui ne tient pas compte des détails d'implémentation (qui eux sont dans le

corps de la fonction). Autrement dit, on sépare le « quoi » ou le « pourquoi » avec la signature (et le nommage est une fois encore crucial) du « comment » présent dans le corps de la fonction, de façon à renforcer l'encapsulation.

Une fonction prend peu de paramètres. Lorsque le traitement nécessite beaucoup de données, on cherche à regrouper ces données dans des types plus complexes afin de limiter le nombre de paramètres.

Le découpage en fonctions et sous-fonctions aide à hiérarchiser les niveaux d'abstraction. Les fonctions les plus hautes sont très proches de la problématique métier à traiter, alors que les détails techniques sont présents dans les fonctions les plus profondes. On cherche aussi à faire en sorte que, dans une fonction donnée, l'ensemble des lignes de code se situent au même niveau d'abstraction (en d'autres termes, on ne mélange pas les considérations métiers avec des calculs plus techniques). Quand ce n'est pas le cas, cela nous donne un bon indice pour extraire des sous-fonctions.

Enfin, il est important de bien séparer les responsabilités : lorsqu'une fonction traite plusieurs problématiques d'un coup, on va chercher à traiter chacun des problèmes dans une sous-fonction dédiée.

En résumé, un découpage approprié vise à bien séparer ce qu'on cherche à faire (porté par le nom de la fonction) de comment on le fait (le corps de la fonction). Les détails d'implémentation sont ainsi bien enfouis, tout en restant accessibles lorsque c'est nécessaire, et ne perturbent donc pas la compréhension du code.

◆ *Granularité des fonctions*

Nous donnons ici quelques conseils permettant d'évaluer si la taille d'une fonction est adéquate. Tout d'abord, une fonction ne doit faire qu'une seule chose comme fournir un calcul simple, modifier une valeur, vérifier une règle métier, ou encore agréger des calculs simples fournis par d'autre fonctions dans un calcul plus complexe.

◆ *Séparation des niveaux d'abstraction*

Elle doit ensuite, dans la mesure du possible, se situer à un seul niveau d'abstraction, c'est-à-dire que tous les appels qui sont faits sont sur le même niveau d'abstraction. On ne mélange pas par exemple des appels liés à une règle métier avec des appels pour faire des traitements plus techniques (comme de la manipulation de chaînes de caractères) ou avec un algorithme.

Prenons comme exemple les conditions pour « être électeur » en France, définies par l'article 3 de la Constitution comme « *tous les nationaux français majeurs des deux sexes, jouissant de leurs droits civils et politiques* ».

Un code naïf et peu propre serait :

```
if (DateUtils.yearsCount(now(), person.birthDate) >= 18
  && person.hasFrenchNationality()
  && !blockList.contains(person) && !listeElectorale.contains(person)) {
   // peut voter...
}
```

Mais ce code mélange les niveaux d'abstraction, par exemple « plus de 18 ans entre la date de naissance et maintenant », « être majeur » et « être électeur ». Si on sépare les niveaux d'abstractions, alors cela devient par exemple :

```
// --- niveau d'abstraction du métier tel que décrit par la loi

public boolean estElecteur(person){
    return estMajeur(person)
        && aNationalitéFrancaise(person)
        && aTousSesDroits(person)
        && estInscritSurListeElectorale(person);
}

// --- niveau d'abstraction inférieur
public boolean estMajeur(person){
    return age(person) >= 18;
}

// --- niveau d'abstraction encore inférieur
private boolean age(person){
    return DateUtils.yearsCount(now(), person.birthDate);
}

// ... reste du code omis par souci de concision
```

Garder les niveaux d'abstraction en tête et les matérialiser aide à raisonner sur une quantité réduite de code, donc à moindre effort.

◆ *Simplicité contractuelle*

Un moyen efficace de limiter la complexité des fonctions consiste à les forcer à voyager léger, ce qui s'obtient lorsque les paramètres d'entrée et de sortie respectent les règles suivantes :

- ✓ Un seul paramètre en sortie : la plupart des langages ne prennent qu'une seule valeur de retour, et ce n'est pas une raison de contourner cette limitation en utilisant des paramètres d'entrée comme des paramètres de sortie. Au contraire, on devrait considérer que tous les paramètres d'entrée sont immuables en dehors du contexte de la fonction.
- ✓ Si la valeur retournée est complexe, on l'encapsule dans un type dédié.
- ✓ De la même manière, si les données en entrée sont nombreuses, on essaye de les regrouper dans un type dédié (qui, au passage, amènera davantage de cohésion).
- ✓ Se restreindre à un faible nombre de variables, et les définir au bon endroit, au plus près de leur utilisation, ce qui permet de diminuer de façon notable la complexité.

◆ *Organisation du code*

Le découpage depuis le « grand » vers le plus « petit » ne constitue qu'un aspect de la structuration du code. Une fois que nous avons uniquement des petits morceaux, il est important de bien les organiser. Idéalement, les fonctions traitant le problème principal devraient se situer en tête du fichier source, et les fonctions secondaires gérant un sous-problème se trouver plus bas dans le code source. Ainsi le code va se lire comme une table des matières, du plus global, en haut, vers le plus spécifique, en bas : le lecteur perçoit en premier lieu l'idée générale, au plus haut niveau d'abstraction ; s'il a besoin d'en savoir plus sur les points de détails, il lui suffira de naviguer à l'endroit souhaité.

On évite de polluer le cheminement global en y entrelaçant des détails d'implémentation. La visibilité (ou portée) des fonctions peut être un indicateur : les fonctions publiques (utilisées par d'autres composants) seront plus haut dans le code que les fonctions privées (dont l'usage est exclusivement interne au composant qui les définit). Attention, parfois la visibilité est définie à tort, et inversement la position d'une fonction dans le code peut aider à en raffiner la visibilité. On essaye également de regrouper les fonctions se situant au même niveau d'abstraction.

Cette règle simple peut aider à y parvenir : aucune fonction ne devrait être définie plus haut qu'une fonction qui l'appelle (les appels récursifs constituent bien entendu une exception à cette règle).

◆ *Aspect visuel du code*

Avec l'habitude, un œil exercé est capable de distinguer rapidement si du code est relativement propre, juste en se basant sur sa forme générale (sa graphie). Un code propre présente les caractéristiques suivantes :

- ✓ Il est bien formaté et respecte un standard défini par l'équipe ; pour plus de facilité, ce standard est importé dans l'IDE et ce dernier est configuré pour reformater automatiquement le code à chaque modification.
- ✓ Le niveau maximum d'indentation est limité, on ne rencontre pas plus d'un ou deux niveaux par fonction ; lorsque les niveaux d'indentation sont trop nombreux, c'est qu'il y a certainement des opportunités de faire plus de découpage.
- ✓ Les lignes de code ne sont pas très larges et de fait n'obligent pas le lecteur à faire défiler le texte horizontalement. Pour obtenir ce confort dans la navigation du code, il convient de se restreindre le plus possible à une largeur maximum de ligne (les IDE bien conçus indiquent cette largeur maximale par un trait vertical) en utilisant sans modération les retours à la ligne et en limitant le niveau d'indentation.

Normer le code

Il est important de se mettre d'accord avec le reste de l'équipe sur un ensemble de règles de formatage :

- Définir un encodage de caractères commun (par exemple UTF-8).
- Faire un choix pour la représentation des retours chariot, notamment dans Git, ainsi que des indentations (au risque de l'éternel débat espaces contre tabulations, il n'y a pas vraiment de solution meilleure que l'autre, le plus important étant que tout le monde choisisse la même représentation).
- Décider ensemble des règles de formatage (position des espaces, accolades systématiques pour tous les `if`…).
- Choisir une politique précise pour les imports : ordre des imports, utilisation des imports `*`, etc.

Ces règles, une fois configurées directement dans l'IDE (par exemple formatage et imports automatiques), ne nécessitent plus aucun effort de la part des développeurs. Le fait qu'elles soient communes à l'équipe offre un avantage énorme pour la gestion des versions : les différences ne sont plus polluées par des modifications de forme insignifiantes et permettent à chacun de se concentrer sur les modifications essentielles lors des revues de code. De la même manière, pour se faciliter la vie lors de la navigation dans un historique de code, il est rentable de définir et d'appliquer des règles sur le format des commentaires de commit.

2. 4 DES COMMENTAIRES À CONSOMMER AVEC MODÉRATION

Une idée très largement répandue dans le monde du développement voudrait nous faire croire que la lisibilité et la qualité d'un code sont proportionnelles à la quantité de commentaires présents. De la même manière, on a tendance à considérer, à tort, que du code exempt de commentaires serait forcément du code de mauvaise qualité. Ces idées reçues sont bien évidemment fausses et pour plusieurs raisons.

La raison majeure tient à la nature documentaire des commentaires : par essence, les commentaires peuvent être obsolètes, en décalage avec la réalité du code, voire totalement faux. Le lecteur du code peut être induit en erreur par les commentaires, ce qui amène à un bon conseil lorsqu'on lit du code : considérer les commentaires avec méfiance, seul le code exécutable contient la vérité. L'autre bonne raison devrait être évidente si vous avez lu ce chapitre avec attention : il existe suffisamment de moyens d'exprimer l'intention par le code pour pouvoir se passer des commentaires dans la grande majorité des cas.

Parfois, on pourrait être tenté d'ajouter des commentaires lorsque du code est obscur. Mais, pour Brian Kernighan (dans *The Elements of Programming Style*), il est beaucoup plus utile de réécrire le code pour refléter notre nouvelle compréhension :

« *Don't comment bad code – rewrite it.* »

2.4.1 Les commentaires à éviter

Commençons par partir de la réalité suivante : la très grande majorité des commentaires dans le code est inutile. Dans un certain nombre de cas, les commentaires n'apportent pas d'information supplémentaire par rapport à ce qui est explicite dans le code. Par exemple :

```
public class Person {
    private final String firstname; // the person's first name
}
```

Ici, le commentaire paraphrase clairement le code, on désigne cet anti-pattern sous le terme moyennement élogieux de *commentaire de la honte* (*shameful comment* en anglais). Non seulement ce commentaire n'apporte rien, mais il ajoute du bruit et une charge de lecture supplémentaire.

Dans d'autres cas, on rencontre ce genre de formulation :

```
  private String str; // the description of the item
```

Si vous avez lu attentivement la partie dédiée au nommage, vous aurez reconnu le *smell*[6] qui consiste à nommer un élément en fonction de son type (ou en abréviation). Et vous savez alors que la correction naturelle est la suivante :

```
  private String itemDescription;
```

En procédant ainsi, la connaissance initialement fournie dans le commentaire devient auto-portée par la variable elle-même. L'intention reste donc véhiculée par le code lui-même : le code, quand il est bien écrit, possède une valeur documentaire.

Voici un troisième exemple, un peu plus complexe :

```
  ... // some code
  // looking for adult persons
  List<Person> matchingPersons = new ArrayList<Person>;
  for(Person person : persons) {
      if (person.age >= 18) {
          matchingPersons.add(person)
      }
  }
  ... // some other code
```

Si on procède à un habile découpage en remplaçant ce code par une méthode dédiée `List<Person> findAdults(List<Person> persons)`, le message initialement inscrit dans le commentaire devient explicite avec la signature de la méthode.

---

6. Un *smell* (ou « mauvaise odeur ») de code est une caractéristique dans le code source d'un programme qui indique un défaut de conception et qui peut être révélatrice de problèmes plus profonds.

L'objectif de ces exemples est de montrer qu'avec un peu d'astuce et d'espièglerie on peut rendre explicite dans le code ce qu'on avait l'habitude d'indiquer dans des commentaires. Ceci amène les considérations suivantes :

- ✓ Dès qu'on veut ajouter un commentaire, on doit se demander comment exprimer la même connaissance en utilisant uniquement du code.
- ✓ Si on sent que le commentaire est vraiment nécessaire (à part quelques cas exceptionnels présentés plus loin), on peut le considérer comme un indice sur un souci dans la propreté du code ou au niveau du design.
- ✓ Les commentaires ne sont pas vraiment du code, il s'agit plutôt de documentation textuelle co-localisée avec le code et, en tant que tels, ils souffrent de tous les travers inhérents de la documentation (décalage avec le réel et obsolescence notamment).

On trouve de nombreuses situations pour lesquelles les commentaires sont contre-productifs et donc évitables :

- ✓ lorsqu'on donne des informations qui peuvent être fournies par l'outillage : par exemple des informations d'historique du code ou liées aux auteurs, alors qu'elles sont directement accessibles avec l'outil de versionnage ;
- ✓ lorsqu'on commente du code qui n'est plus utilisé au lieu de le supprimer, « au cas où » on en aurait besoin plus tard : en pratique le « au cas où » n'arrive que très rarement, et le risque est que personne n'ose supprimer ce code mort par la suite. Par conséquent il vaut mieux le supprimer tout de suite, d'autant que si besoin on peut le retrouver dans l'historique des versions.

2.4.2 Les commentaires réellement utiles

Si, comme nous venons de le voir, les commentaires sont si peu utiles, pourquoi sont-ils encore présents dans les langages de programmation ? De fait, les commentaires n'apportent rien dans la plupart des cas, mais il existe encore quelques situations pour lesquelles on ne peut pas encore s'en passer. Le fait de ne s'autoriser que les commentaires réellement utiles donne à chacun d'entre eux une plus grande valeur : lorsqu'on les rencontre, on y fera d'autant plus attention que nous ne sommes pas supposés en rencontrer.

◆ *Signaler une subtilité*

Parfois la complexité de l'information à communiquer est telle que le code ne suffit plus, ce qui nous contraint à employer les grands moyens et à expliciter clairement les choses. Un commentaire détaillé peut se montrer pertinent lorsqu'une modification non éclairée du code alentour pourrait entraîner des conséquences malencontreuses. La description d'une optimisation subtile, d'un effet de bord pouvant être source d'anomalie, ou le balisage d'un contournement lié à l'utilisation d'un composant externe sont autant de situations légitimant l'utilisation de commentaires.

Dans tous les cas, il est de notre responsabilité de signaler clairement toutes les chausse-trappes auxquelles nos successeurs pourraient être confrontés, et pourquoi ils doivent y réfléchir à deux fois avant de modifier notre code.

◆ *Marquer des problèmes à résoudre dans le code*

Lorsqu'on détecte des problèmes dans le code mais qu'on manque de temps, qu'on veut conserver le focus sur la tâche en cours ou tout simplement qu'on a la flemme, il est peu coûteux et pertinent de les indiquer aux endroits appropriés sous forme de TODO ou de FIXME. Ainsi les problèmes sont signalés clairement et ne tombent pas dans l'oubli.

◆ *Apposer des mentions légales*

Dans certaines situations, le code doit comporter des informations légales, ou indiquer les auteurs. C'est notamment le cas dans le monde de l'open source, qui impose de protéger juridiquement son code en décrivant les modalités de réutilisation, de partage ou de modification.

Pour aller plus loin...

Les principes énoncés dans cette section se bornent au fait de limiter les commentaires en les exprimant directement dans le code. Au travers de techniques plus avancées, le code peut porter davantage de valeur documentaire (en exploitant habilement les annotations, par exemple). Si le sujet vous intéresse, n'hésitez pas à consulter l'excellent ouvrage *Living Documentation*[1] de Cyrille Martraire (co-auteur de ce livre).

2.5 NE PAS SE RÉPÉTER (DRY)

Quand on écrit du code et qu'on souhaite aller plus vite, il est tentant de faire appel massivement au copier-coller : on repère une portion de code qui fait à peu près l'affaire, on le récupère ni vu ni connu et on modifie une ou deux lignes de code pour coller à notre besoin. Nombre de développeurs procèdent malheureusement de la sorte, adoptant ainsi une programmation *par mimétisme*. C'est un style qui peut sembler efficace sur le court terme, mais le design global ne s'en trouve pas grandi : le code est évidemment moins lisible, mais surtout la moindre évolution va nécessiter de nombreuses modifications dans le code.

Soyons clairs tout de suite : l'utilisation du copier-coller pour écrire du code est manifestement une mauvaise idée. Cette prescription est représentée par l'acronyme DRY : *Don't Repeat Yourself*.

On pourrait traduire ceci par « Ne vous plagiez pas ! » Or forcément, quand il y a copier-coller, et même si on adapte la copie, il y a répétition et donc duplication. Le démon du copier-coller est une addiction dont vous devez absolument vous débarrasser pour produire du code de bonne qualité.

---

7. Martraire, C. *Living Documentation*, Addison Wesley, 2018.

Lorsqu'on rencontre une duplication, on doit évaluer la possibilité de faire émerger une abstraction.

Bien sûr, comme la plupart des principes liés au Clean Code, celui de réduction de duplication n'est pas absolu : dans certains cas, il est préférable de s'en abstenir. La duplication offre des avantages, principalement la possibilité de faire évoluer un morceau de code librement, indépendamment du reste (les autres morceaux de code lui ressemblant). Lorsqu'on rassemble plusieurs morceaux de code en apparence semblables en une unique abstraction, on crée un couplage entre eux : chaque modification impactera ensuite chaque utilisateur de cette abstraction. C'est la question qu'on doit se poser lorsqu'on veut éliminer une duplication : est-ce que les morceaux de code sont liés à la même problématique ? Est-ce qu'ils ont toutes les raisons du monde d'évoluer de concert ? Dans ce cas, on réduira la duplication. Sinon, il est préférable de laisser la duplication pour permettre à chacun d'évoluer en toute liberté.

Comme souvent, tout est une question de conserver un bon équilibre entre la liberté d'évoluer individuellement liée à la duplication, mais à laquelle s'associe plus de difficulté pour maintenir le code, et le risque d'introduire du couplage. Certaines heuristiques peuvent aider à faire le choix de conserver ou non la duplication :

- ✓ Quelle est la nature de la connaissance portée par le morceau de code ? Si c'est de la connaissance métier, on cherchera à retirer la duplication : en effet, chaque connaissance métier ne devrait apparaître qu'une seule fois dans le code.
- ✓ Sur quel périmètre s'étend la duplication ? Est-elle limitée à une seule classe, ou une seule méthode ? À l'inverse, est-elle étalée sur plusieurs composants ou modules ? Plus le périmètre est restreint, et plus il est conseillé de réduire la duplication, parce que la portée du couplage introduit reste limitée. Il est généralement conseillé de réduire la duplication lorsque le périmètre est petit, et de la conserver quand le périmètre est étendu.

2. 6 LA PROPRETÉ DE CODE AUSSI POUR LES TESTS

On a souvent tendance à mettre beaucoup d'efforts dans la qualité du code qui va s'exécuter en production, et à laisser le code des tests en jachère. C'est une erreur, pour de multiples raisons. Les tests constituent, pour beaucoup de nouveaux arrivants sur un projet, le point d'entrée sur le code : ils décrivent (sous réserve d'être suffisamment exhaustifs) le comportement attendu pour les différentes fonctionnalités de l'application. De la même manière, nombre de développeurs abordant une revue de code commencent par lire les tests pour essayer de comprendre de quoi il retourne.

Vus sous cet angle, les tests sont la vitrine du code, et leur qualité se doit d'être exemplaire. Il est donc vital de mettre autant d'effort, voire plus, sur la propreté du code de test que sur celle du code de production. N'oublions pas que le rôle des tests n'est pas seulement de vérifier le comportement de l'application, ils ont aussi la responsabilité d'en fournir une documentation sous forme exécutable. L'intention doit donc être exprimée avec clarté dans les tests, en bénéficiant des techniques que

nous venons de voir : nommage, organisation du code, réduction de la duplication, etc. Il faut également s'assurer que les tests seront faciles à maintenir lorsque les fonctionnalités sous-jacentes nécessiteront des évolutions.

2. 7 CONCLUSION

Dans ce chapitre nous avons précisé l'importance d'écrire du code de qualité, facilement lisible et maintenable par les autres développeurs. Nous avons également souhaité montrer que ce ne sont pas tant les techniques qui importent que le fait de se poser les bonnes questions sur la façon de révéler au mieux l'intention, notamment à travers les axes du nommage approprié et de la bonne organisation du code.

Le chapitre 5, « L'importance des techniques de refactoring », traite de techniques plus poussées pour améliorer le design du code de façon continue, par transformations successives. En effet, l'écriture de code de qualité ne se fait pas en un seul jet, c'est un processus itératif par la pratique du refactoring.