Le clean Code

Alexis Vachard

Chapitre 1 - Introduction

Je suis développeur Java Backend, mais je travaille aussi de plus en plus avec le langage Kotlin. Il m'arrive également de travailler avec du javascript de manière occasionnelle. Je travaille actuellement dans une société de conseils qui s'appelle Carbon IT. Nous sommes un peu plus d'une centaine de consultants avec des profils très variés, mais un point commun nous rassemble tous : nous adorons le développement. Via Carbon IT nous avons la chance de travailler pour de nombreux clients et sur toutes tailles de projets. Nous proposons également des formations pour les personnes qui nous rejoignent, qui peuvent s'étaler sur plusieurs semaines, afin de pouvoir être envoyé chez nos clients le plus sereinement possible.

Objectifs du cours

Avant toute chose, ce cours risque de paraître obscur à certains d'entre vous, mais rassurez-vous, plus nous avancerons plus cela deviendra concret. Le clean code, avant d'être une façon de travailler, c'est un ensemble de codes et de règles que l'on s'applique parfois sans y penser dans nos métiers de développeurs. Mais la plupart de ces concepts viennent avec l'expérience. Ce cours a donc vocation à vous apporter ces concepts afin que vous puissiez les mettre en pratique dans vos missions de développement, que vous soyez stagiaire, alternant, déjà en poste, ou bien dans tous vos projets personnels.

Ce cours n'est pas non plus une vérité absolue. Le monde du développement est extrêmement vaste, les projets et les équipes tous différents. De ce fait, certains concepts peuvent ne pas être applicables partout. Cependant, il s'agit là d'un regroupement de connaissances utiles, à connaître mais surtout à appliquer de manière intelligente. N'appliquons jamais quelque chose parce que nous avons vu qu'il fallait l'appliquer, car tous les sujets ne s'y prêtent pas forcément.

Cependant pendant ce cours, vous aurez l'occasion d'aborder de nombreux concepts, qui vous seront utiles durant toute votre carrière de développeur.

Présentation du plan de cours

Ce cours est divisé en plusieurs partie. Les deux premières seront majoritairement orientées sur des définitions de différents concepts avec, pour certains, des exemples. Les parties nommage et fonctions quant à elles, s'attaqueront à du code en nous permettant de revoir quelques notions de base de techniques développement.

Une fois les quatre premières traitées nous enchaînerons sur un aspect très important de la programmation : les effets de bord. Nous aurons l'occasion d'aborder plusieurs pièges que nous rencontrons fréquemment dans nos métiers, et des solutions pour les éviter correctement et surtout, simplement avec un peu de rigueur.

Le chapitre dédié au travail à plusieurs quant à lui, reviendra sur trois principes et techniques très importants pour le développement à plusieurs. À savoir, la revue de code, le pair programming et enfin un dernier principe moins utilisé mais pourtant intéressant : le mob programming.

Nous allons également passer un moment sur les tests de nos développements dans le chapitre huit. Nous irons redécouvrir les tests unitaires, mais où vous pourrez peut-être découvrir de nouveaux types de tests comme les tests d'intégration ou encore le mutation testing. Nous passerons également une partie de ce chapitre à évoquer le Test Driven Development, une technique d'écriture de code dans laquelle les tests sont mis en avant.

Enfin, nous aborderons les principes SOLID, qui sont eux aussi très importants pour éviter certains problèmes courants que les développeurs rencontrent. Nous traiterons également de l'architecture hexagonale, dont l'utilisation prend de l'ampleur des dernières années. Cela nous permettra de mettre en pratique les principes SOLID beaucoup plus facilement grâce à son cadre rigoureux. Nous terminerons par d'autres notions qui méritent d'être connues, même si vous allez le voir, nous les appliquerons bien souvent inconsciemment dans les chapitres traités avant.

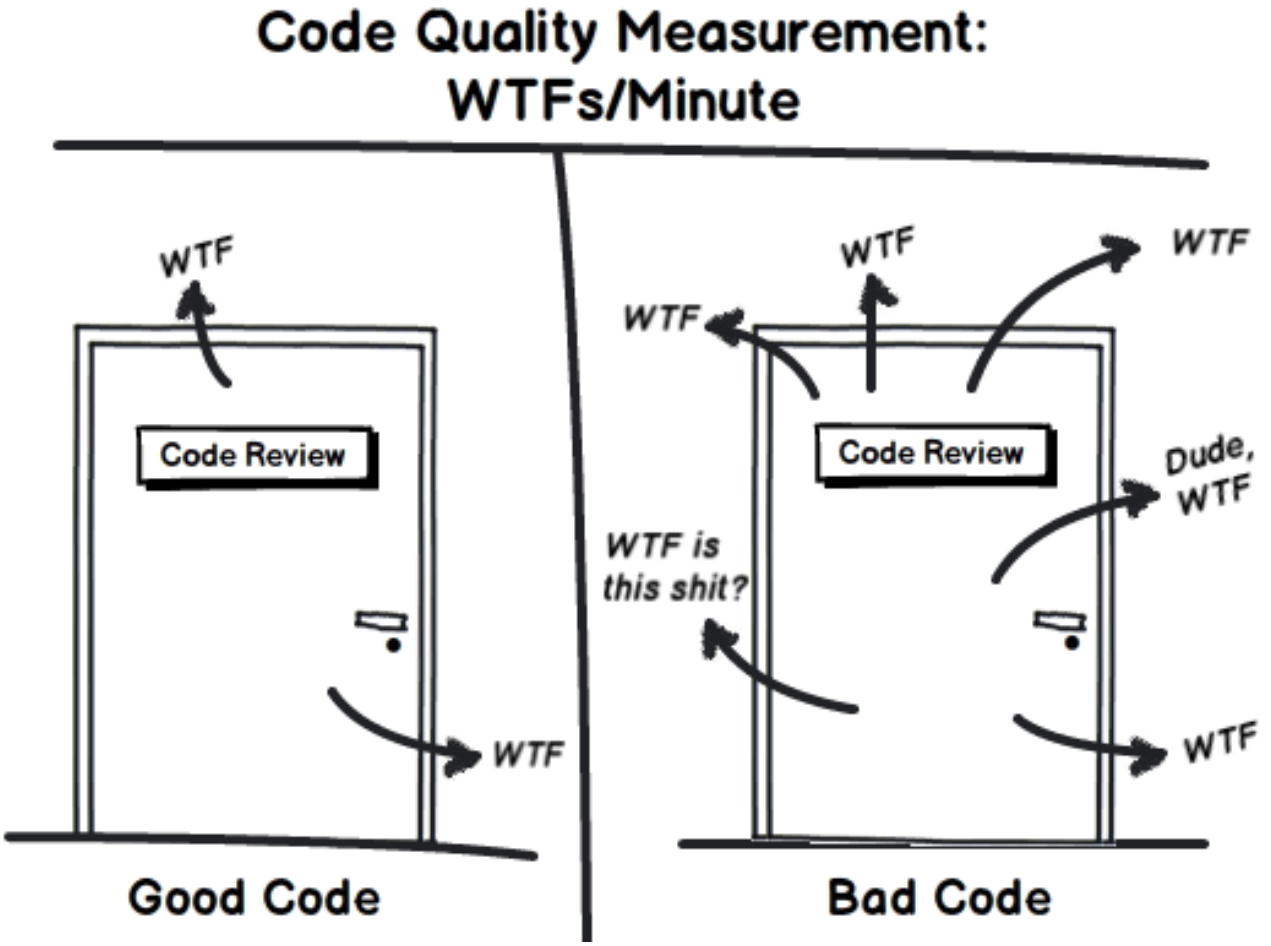
Chapitre 2 - Définitions du clean code

Lorsque l'on parle de clean code, les développeurs, comme sur beaucoup d'autres sujets, ont énormément de mal à s'accorder sur une définition qui fasse consensus. Cela vient principalement de la chance que nous avons d'avoir dans ce domaine de nombreuses cultures qui cohabitent. Par exemple, le libre et l'open source. Il y a aussi de grands courants de pensée dans nos professions, qui évoluent au fur et à mesure que notre discipline s'améliore. Ce ne sont bien sûr que des exemples, et ce cours n'a pas vocation à s'attarder sur ces sujets, qui mériteraient à eux seuls une étude qui serait, à mon sens, très intéressante.

D'après le livre « Clean Code: A Handbook of Agile Software Craftsmanship », que je vous invite fortement à lire, il y aura toujours du code. Je dois avouer que je suis d'accord avec ce postulat. En effet, il est de plus en plus souvent suggéré que la fin du code est proche d'après l'auteur, nous le voyons de plus en plus avec l'émergence du no code. Mais quoi qu'il arrive, nous écrirons toujours du code. L'auteur nous dit aussi que pour de plus en plus de personnes, le code sera bientôt généré au lieu d'être écrit, que notre profession sera bientôt inutile car les sociétés vont générer leurs programmes à partir de spécifications. Ne nous voilons pas la face et rassurons nous d'emblée, c'est très peu probable. D'après l'auteur, nous ne nous débarrasserons jamais du code pour la simple raison qu'il représente le détails des exigences. Et à un moment donné, pour qu'on programme fonctionne, il faut bien les préciser. Or préciser des exigences à un niveau de détail qui permet à une machine de les exécuter s'appelle programmer. La spécification devient le code. Toujours d'après l'auteur, le niveau d'abstraction de nos langages va continuer d'augmenter, ainsi que l'augmentation du nombre de langages spécifiques à certains besoins. C'est une bonne chose, mais cela ne signifie pas la fin du code, car les spécifications écrites dans ces langages de plus haut niveau et spécifiques à un domaine resterons du code. Code qui devra toujours être rigoureux, précis, formel et détaillé afin qu'une machine l'exécute. En bref, il y aura toujours du code, simplement le code évoluera toujours.

Maintenant que nous savons que nous aurons toujours affaire à du code mais pas toujours de la même manière, parlons de clean code ou de code propre. Quand on parle de clean code, soit du code propre, cela implique un opposé, le mauvais code, le code sale. Afin d'avoir une première approche du code propre, attardons nous sur le mauvais code. Prenons un exemple concret pour imager un mauvais code. Nous sommes une société qui développe un produit révolutionnaire, que tous le monde s'arrache, et dont la base d'utilisateurs devient immense. Faire évoluer ce produit devient une nécessité afin de continuer à le vendre. Donc nous allons développer de nouvelles fonctionnalités pour nos utilisateurs. Avec un code mauvais, les cycles de livraison vont inévitablement commencer à s'étirer. La correction des bugs va devenir une tâche titanesque, les performances du produit vont baisser jusqu'à parfois provoquer des arrêts non voulus. Les utilisateurs finiront alors par arrêter d'utiliser notre produit, et nous risquerons de faire faillite. Tout développeur avec une expérience quelconque à déjà dû faire face à du mauvais code. Et quand on y fait face, nous entrons dans un processus durant lequel nous sommes en difficulté, qui pourrait s'apparenter au verbe « patauger ». Et bien dans du mauvais code on patauge, on avance avec difficulté, et nous faisons face à de nombreux problèmes souvent bien cachés. Le code peut paraître n'avoir aucun sens au fur et à mesure du processus. Mais pourquoi le mauvais code existe ? Nous y avons tous déjà été exposé, et parfois même c'est nous qui l'avons écrit. Mais pourquoi avoir écrit du mauvais code ? Pour aller vite ? Pour satisfaire des délais intenables si on souhaite faire un travail de qualité ? Peut être aussi par envie d'en finir au plus vite avec une tâche complexe sur laquelle nous travaillons depuis un moment. Soyons honnêtes, ça nous est tous arrivé, et ça nous arrivera sûrement encore si on ne fait pas attention. Mais le risque c'est d'en arriver à se dire : cela fonctionne, certes ce n'est pas propre mais cela fonctionne et c'est mieux que rien, alors laissons comme ça pour l'instant et revenons-y plus tard. Grave erreur, car en règle générale plus tard signifie jamais. Le coût engendré par le désordre que représente le mauvais code sera toujours plus élevé que les bénéfices à court-terme qu'il apporte. Simplement les coûts ne se font ressentir qu'au bout d'un certain moment, et bien souvent, quand on ne peut plus faire machine arrière facilement.

Alors comment définir le clean code maintenant que nous savons ce qu'est un mauvais code? Notre travaille consiste à écrire des détails dans un langage qu'une machine peut comprendre. Néanmoins, dans la vie, être compris par les machines ne suffit pas. En effet, dans l'immense majorité des projets sur lesquels nous développeurs, sommes amenés à intervenir, un même code est manipulé par plusieurs personnes. Et c'est là que cela devient compliqué, car il est bien plus difficile de se faire comprendre par un humain que par une machine. Donc pour qu'un programme puisse évoluer correctement, il doit impérativement être compréhensible pour les machines qui l'exécutent, mais aussi par les personnes qui travaillent dessus. Si ce n'est pas le cas, il est absolument évident que les coûts pour faire évoluer le projet vont augmenter significativement dans le temps. Une technique simple pour définir si un code est propre peut se résumer en une simple image nommée : « le nombre de what the fuck à la minute ».



Cette image au delà d'avoir une petite touche d'humour résume parfaitement le clean code, plus les développeurs se posent de questions moins le code est propre, tout simplement. Alors qu'est-ce qui définit qu'un code est propre ? Avec quelques principes simples :

1. **Simplicité :** la première caractéristique d'un code propre est sa simplicité. Le simple est l'ami du propre. Il faut toujours se demander si l'implémentation est bien la plus simple pour répondre au besoin énoncé. Plus votre code est simple, plus il devient lisible, et plus il devient lisible, plus il devient compréhensible pour les autres humains, et donc plus il sera maintenable. Pour la simplicité souvenez vous de l'acronyme KISS (Keep It Simple Stupid), grâce auquel on ne fera pas de surconfigurabilité ni de code « au cas où ».
2. **Ciblé :** un bout de code ne doit avoir qu'une unique responsabilité. Par exemple, vous n'allez pas demander à votre cafetière de changer la roue de votre voiture. Et bien ne le demandez pas à votre code. Ce principe est valable à tous les niveaux d'abstraction dans votre code :

les packages, les interfaces, les classes ou encore les méthodes. Cette notion, nous le verrons, s'apparente beaucoup aux principes SOLID que nous aborderons plus loin dans ce cours.

3. **Testable :** un code simple et qui ne fait qu'une chose est un code qu'il est facile de tester. De plus l'exécution des tests sera bien plus rapide. Les tests constituent la meilleure des sécurités pour assurer un code sans bug et qui ne régresse pas. Mais ce sont aussi les tests qui font office de documentation pour les autres développeurs. Un test doit être FIRST, c'est à dire : Fast, Independent, Repeatable, Self-Validating, Thorough. En d'autres termes, un test unitaire doit être rapide, indépendant, reproductible, auto-validant, et enfin approfondit.

En bref, je finirai par dire que selon moi, le clean code représente toutes les pratiques, les techniques et les connaissances mises en place afin de rendre le code que l'on produit, sûr, simple, compréhensible par ses pairs et évolutif.

Chapitre 3 - Quelques concepts

Maintenant que nous avons défini ce qu'est et surtout ce que n'est pas un code propre, regardons quelques autres concepts et définitions.

La maintenabilité

En informatique, la maintenabilité représente la capacité pour des programmes à être maintenus de manière cohérente et à moindre coût, en état de fonctionnement. On pourrait résumer cette définition par la capacité d'un système à être réparé rapidement pour diminuer les temps et les coûts d'intervention. On calcule souvent la maintenabilité par le temps moyen d'intervention. Et l'une des solutions pour améliorer la maintenabilité d'un programme est de le concevoir par sous-ensembles simples à démonter et à inter-changer. Une solution de plus en plus couramment utilisée est l'architecture hexagonale Sur laquelle nous reviendrons plus en détail par la suite.

L'évolutivité

Un code est dit évolutif, lorsque des modifications qui sont à apporter au programme sont réalisées de la manière la plus simple possible et impliquant le moins de contraintes possibles. De nombreuses méthodes permettent de rendre son code évolutif, comme le choix d'une architecture logicielle adaptée par exemple, mais la rigueur des développeurs reste le facteur le plus important à prendre en compte.

La dette technique

Il s'agit là d'un concept inventé dans les années 90, qui vient d'une métaphore, inspirée du concept de dette en finance, que l'on applique en informatique. La dette technique fait est corrélée avec la complexité : si je produis une première version d'un code, c'est comme si je venais de m'endetter. Une petite dette accélère le développement à partir du moment où je la rembourse rapidement par une réécriture de mon code. Le danger arrive donc si je ne rembourse pas ma dette. Chaque instant passé sur un code endetté, pas tout à fait correct, est un intérêt sur cette dette. Ainsi, plus je passe de temps à travailler avec un code endetté, plus je vais payer cher le jour où je vais devoir rembourser cette dette. La dette technique est donc un aspect très important de la programmation. Et comme toute dette, il faudra un jour ou l'autre la rembourser. Il est donc pertinent de la rembourser le plus tôt possible avant d'en perdre le contrôle et de se retrouver insolvable.

En d'autres termes, un développement négligé induit des coûts, les fameux intérêts, que l'on va rembourser par des temps de développement plus longs, une complexité grandissante, et donc des bugs de plus en plus fréquents. Cependant, il faut bien garder à l'esprit que la dette technique est inhérente au développement, nous en aurons toujours, il convient donc de la gérer correctement. Une dette technique peut également être intentionnelle ou non. Si une dette n'est pas intentionnelle, elle est due à des malfaçons comme par exemple le non respect de la conception ou des règles de développement. Ce type de dette ne peut apporter aucun bénéfice. À l'inverse, un dette technique intentionnelle est calculée à l'avance. En effet, favoriser la qualité de conception augmente la charge des développeurs, on peut alors jouer sur cette mesure pour gagner du temps et en tirer des bénéfices. Mais dans tous les cas, l'intention derrière cette dette volontaire doit être le remboursement de cette dernière à court terme.

La couverture de code

La couverture de code, encore un concept contraignant ou bien quelque chose d'absolument bénéfique ? La réponse à cette question est : les deux. La couverture de code, ou code coverage, est souvent une métrique. Une métrique souvent incomprise. Vous tenez là un

excellent sujet débat pour tout l'open space. Mais la couverture de code c'est un concept terriblement simple : cela représente le taux de code exécuté dans un programme lorsque l'on lance une suite de tests dessus. Ce taux allant de 0 à 100%. Mais comment interpréter ce taux ? Doit-on absolument viser les 100% ou bien une couverture de 40% est grave ? À partir de quel pourcentage mon code est correctement testé ? Face à toutes ces questions, deux camps ressortent souvent, deux extrêmes, ceux qui pensent qu'il faut tendre le 100% de couverture, et ceux qui pense que la couverture ne vaut rien. Alors qui a raison et qui a tort ? Et bien c'est un paradoxe assez intéressant car en réalité, les deux camps ont raison, ou plutôt, n'ont pas totalement tort.

Pourquoi ce paradoxe ? Et bien le taux de couverture ne mesure que le code exécuté par nos tests, ni plus ni moins. Or, un test peut très bien exécuter du code sans le tester. Exemple : mon test appelle une fonction, mais ne vérifie pas le retour de cette fonction. Je vous l'accorde cet exemple peut sembler extrême, mais c'est justement pour ça qu'il est pertinent. Dans ce cas la méthode appelée est bien exécutée et rentre dans la couverture de code mais n'est absolument pas testée. Et ce taux peut encore plus devenir abstrait quand on sait que certains outils de mesure de couverture de code permettent d'exclure des parties de code dans le calcul du taux. Mon avis sur ce dernier point est que cette fonctionnalité ne devrait pas être utilisée, et donc ne devrait pas exister, car elle fait augmenter le taux de couverture artificiellement.

D'un autre côté, et malgré tout, la couverture de code reste un excellent indicateur. La valeur du taux à un moment donné n'a pas tant de valeur en soit, en revanche l'évolution de la valeur dans le temps, elle, est très intéressante et plus pertinente. Dans un monde parfait, un taux de couverture devrait a minima rester stable et dans l'idéal augmenter, mais ne devrait pas baisser de manière significative.

La simplicité

Nous en avons déjà parlé, mais il convient à mon avis d'en reparler. Un code est dit simple, lorsqu'il répond au besoin exprimé le plus simplement possible. Et pour obtenir un code simple, il faut réaliser plusieurs conceptions différentes afin de choisir laquelle de ces conceptions est la plus simple tout en respectant le besoin. N'oubliez pas l'acronyme KISS dont nous avons déjà parlé précédemment : Keep It Simple Stupid.

La règle du boy scoot

La règle du boy scoot est une règle que tout développeur devrait selon moi garder dans un coin de sa tête lorsqu'il écrit du code en s'appuyant sur un code déjà existant. Et cette règle est simple : si je vois du code qui peut-être amélioré, alors je l'améliore. Ce code peut pouvoir être amélioré pour plusieurs raisons : parce qu'il est trop complexe, parce qu'il n'est pas correctement testé, parce qu'il ne respecte pas les règles définies au moment de le produire, ou bien parce qu'elles ont évolué. Cela peut paraître contre-productif dans le cadre de notre travail, car nous avons une mission qui consiste à répondre à un besoin, mais rappelez vous tout ce que nous avons pu voir jusque là. Vous comprendrez ainsi que la règle du boy scoot est très importante, que cela soit pour la maintenabilité, l'évolutivité, la couverture de code ou encore la dette technique, si je peux le faire alors je le fais. Cependant, si cette règle n'est pas applicable car bien trop chronophage ou trop complexe, alors il convient d'en parler à son équipe rapidement et de mettre en place un plan permettant de retravailler le code concerné au plus vite, en créant un ticket par exemple. Mais en créant ce ticket, il faut bien prendre garde à détailler le problème du mieux possible, et d'y décrire la manière de rectifier le code concerné, car ce n'est pas forcément nous qui allons le faire, peut-être qu'une autre personne fera ce travail de « refactoring ».

La qualité

La qualité logicielle est une appréciation globale d'un logiciel basée sur plusieurs indicateurs, permettant de définir si oui on non, le code qui constitue le logiciel est de qualité. Contrairement à du matériel informatique, les logiciels ne possèdent pas une fiabilité prévisible, et

le logiciel ne peut pas s'user avec le temps. Une anomalie dans un logiciel peut très bien survenir comme elle peut ne pas survenir dans son exécution, cependant l'anomalie est présente de manière latente dans le code. La qualité d'un programme dépend entièrement de deux choses : sa construction et les processus utilisés pour son développement. Et elle regroupe les facteurs extérieurs, ceux identifiables par les utilisateurs, ainsi que les facteurs intérieurs, ceux identifiables par les développeurs.

Quelques indicateurs de qualité logiciel :

- La capacité fonctionnelle : est-ce que mon programme répond correctement aux exigences et aux besoins des utilisateurs ?
- La facilité d'utilisation : l'effort nécessaire pour apprendre à utiliser le logiciel.
- La fiabilité : fournir des résultats corrects, peu importent les conditions d'exploitation.
- La performance : le rapport entre la quantité de ressources utilisées et la quantité de résultats délivrés. Les ressources utilisées peuvent aussi bien être les moyens matériels que la mémoire, la durée nécessaire pour rendre un résultat, mais également le personnel nécéssaire.
- La maintenabilité : qui mesure l'effort nécessaire pour maintenir ou faire évoluer le logiciel.
- La portabilité : l'aptitude que possède mon logiciel à s'exécuter dans un environnement matériel ou logiciel différent de son environnement initial.

Chapitre 4 - Le nommage

Tout comme nous nommons nos enfants à leur naissance, nous nommons nos programmes et les différents éléments qui les composent. Abordons le nommage dans un code propre.

Tout est nommage

Avant d'aller plus loin, il est crucial de comprendre et d'accepter que les noms dans les logiciels sont abondants, il y en a partout. Nous nommons nos variables, nos fonctions, nos classes, nos arguments et nos packages. Nous nommons également les fichiers, donc puisque que les noms sont omniprésents, il est très important de bien les choisir.

Quelques règles de nommage

Maintenant qu'il est établit que tout est nommage en programmation, voyons quelques règles basiques que l'on peut appliquer dans nos développements.

Des noms révélateurs des intentions

A priori c'est une tâche simple de dire que les noms doivent révéler les intentions, mais il faut bien comprendre que ce point est absolument essentiel. Choisir de bons noms est un processus qui prend du temps, qui demande de la réflexion. Mais ce processus permet de gagner un temps précieux par la suite. Il faut donc faire attention lorsque nous donnons des noms, et les changer dès lors que l'on en trouve des meilleurs. Ceux qui lisent notre code, y compris nous-même, nous en seront reconnaissants. Le nom d'une variable, d'une fonction ou d'une classe doit répondre à certaines grandes questions : la raison de son existence, son rôle et son utilisation. Si votre nommage exige un commentaire pour l'expliquer, c'est que ce nommage n'est pas bon.

```
private final ZonedDateTime date; // Date de création du compte
```

Dans cet exemple, le nom date, ne révèle absolument rien, à part que c'est une date. Mais à quoi peut bien correspondre cette date ? La réponse est dans le commentaire, la date de création du compte. Nous devons donc choisir un nom plus adapté pour décrire cette variable. Par exemple :

```
private final ZonedDateTime accountCreationDate;
private final ZonedDateTime creationDate;
```

En choisissant des noms révélateurs, il sera sans aucun doute plus simple de lire, de comprendre, de maintenir et de faire évoluer le code.

Éviter la désinformation

En tant que développeur, nous ne devons jamais laisser de faux indices à nos pairs et à nous même dans notre code, car ces faux indices cachent la signification du code. Il faut également éviter les mots dont le sens établi varie du sens voulu. Par exemple : si l'on souhaite représenter une addition, nous n'allons pas utiliser le mot add, car ce dernier n'est pas parfaitement représentatif, et en plus peut porter à confusion. De même si nous souhaitons représenter un groupe de comptes, nous n'appellerons pas ce groupe accountList sauf si notre structure de données est une liste. Si tel n'est pas le cas il vaut mieux trouver un nommage plus adapté car pour un développeur une liste est quelques chose de très précis. Par exemple : accountGroup, ou tout simplement accounts. Un autre exemple parfait de chose à ne pas faire est de nommer des variables avec la lettre « l » ou bien le la lettre majuscule « o ». D'autant plus lorsqu'elles sont combinées, car elles ressemblent trop aux chiffres 1 et 0 -> l & 1 -> O & 0.

Faire des distinctions significatives

Quand on développe, on se crée souvent des problèmes si on écrit du code uniquement pour faire plaisir au compilateur ou à l'interpréteur. Un exemple simple : il est impossible de nommer deux variables avec le même nom dans la même portée, bien que ces deux variables fassent deux choses complètement différentes. Certains développeurs peuvent être tentés d'introduire par exemple une faute d'orthographe dans l'un des noms pour que cela fonctionne. Mais ce n'est pas une bonne chose, car si un pair se rend compte de la faute, il va vouloir la corriger, puis se rendre compte que cela ne fonctionnera pas. Si les noms doivent être différents, il doivent alors représenter quelque chose de différent et vice versa.

```
//Un mauvais nommage
public static void copyChars(char[] c1, char[] c2) {
    For (int i = 0; i < c1.length; I++) {
        c2[i] = c1[i];
    }
}

// devrait plutôt se présenter ainsi
public static void copyChars(char[] source, char[] destination) {
    For (int i = 0; i < source.length; I++) {
        destination[i] = source[i];
    }
}
```

Choisir des noms compatibles avec une recherche

En programmation, les noms composés d'une seule lettre ou les constantes numériques par exemple, présentent un problème particulier : ils sont difficiles à localiser dans le code. Il est facile de rechercher une constante nommée BLUE_COLOR, mais ce n'est pas la même chose si on recherche le chiffre 7 par exemple. Les recherches peuvent s'arrêter sur plusieurs résultats, par exemple les noms de fichiers, de variables, de fonctions, de classes, de commentaires, où la valeur recherchée peut être utilisée avec un objectif différent. De même, la lettre e n'est pas une

bonne solution pour nommer quelque chose en programmation, pour la simple et bonne raison qu'il s'agit de la lettre la plus utilisée dans les langues anglaise et française… Nous avons potentiellement un e sur chaque ligne de code. Ainsi, on remarque qu'on nom long présente généralement plus d'avantages qu'un nom court. Or, tout nom facilitant la recherche est préférable dans du code. Dans une certaine mesure bien sûr.

Cependant les noms d'une seule lettre peuvent tout de même êtres utilisés dans certains cas. Par exemples pour des variables locales, à l'intérieur de méthodes courtes. De ce fait on peut tirer une nouvelle règle de nommage : la longueur d'un nom doit être lié à la taille de sa portée. C'est à dire que si une variable ou une constante peut être visible ou utilisée à plusieurs endroits de notre code, il est impératif de lui donner un nom compatible avec une recherche.

Noms des classes

Pour les classes et les objets, il est impératif de choisir des noms ou des groupes nominaux clairs et sans verbe. Par exemple : Account, Customer, User, Bet. Les termes comme Manager, Processor, ou encore Info sont à éviter au maximum car bien trop généralistes et donc, ne représentent pas l'intention de manière claire.

Noms des méthodes

Concernant les méthodes, nous devons choisir des verbes ou des groupes verbaux comme par exemple : deleteAccount(), registerAccount(), postPayment(), save(). Les accesseurs, les mutateurs ainsi que les prédicats doivent être nommés d'après leur valeur et préfixés par get, set, et is.

```
String customerName = customer.getName();
customer.setName("John");
If (customer.isConnected()) …
```

Conclusion sur le nommage

Le choix de bons noms est un processus long et parfois complexe, car il demande une aptitude à la description ainsi qu'une culture générale partagée. C'est là un problème d'enseignement et non un problème technique, c'est pourquoi de nombreuses personnes n'apprennent pas à le faire correctement.

Enfin, nous avons souvent peur de renommer les choses, de crainte que d'autres développeurs ne soumettent des objections. Ne partagez pas cette crainte, ne l'entretenez pas, soyez plutôt reconnaissants lorsque cela arrive. Le code n'en sera que plus lisible.

Chapitre 5 - Les fonctions

Contrairement à l'époque ou le développement logiciel venait d'être inventé, tous les programmes modernes s'organisent autour des fonctions. Voyons donc comment bien les écrire.

Des fonctions concises

Plus une fonction est longue, plus elle est difficile à comprendre, à maintenir et à tester. De plus, elle risque de faire plus d'une seule chose, ce qui n'est absolument pas bénéfique en programmation. Cela peut être très difficile de justifier cette déclaration, mais prenons un exemple simple. Nous avons deux fonctions dont l'objectif est le même. Cependant la première des deux fait tout en une seule fois et compte 300 lignes de code. À l'inverse, notre seconde est bien découpée, nous avons plusieurs fonctions très spécifiques pour répondre à un problème global. Je vous mets au défi de comprendre la fonction qui fait 300 lignes plus rapidement que l'autre.

Enfin, une fonction ne doit faire qu'une seule chose et la faire bien. Une bonne façon de s'assurer qu'une méthode ne fait qu'une seule chose est de regarder si une partie de cette dernière ne peut pas être extraite dans une autre fonction dont le nom n'est pas une simple reformulation de notre fonction initiale. Pour être certain qu'une fonction ne fait qu'une seule chose, on doit également vérifier que les instructions qu'elle contient se trouvent au même niveau d'abstraction. Plus on va mélanger de niveaux d'abstraction dans une fonction plus cette dernière sera déroutante. Alors le lecteur ne sera pas toujours en mesure d'en déduire l'objectif, ni si une expression est un concept essentiel ou bien un détail. C'est pire encore, dès lors que des détails sont mélangés avec des concepts essentiels, de plus en plus de détails ont tendance à se présenter à l'intérieur de la fonction. Vous l'aurez compris, il s'agit d'une sorte de cercle vicieux.

De plus, il faut que le code puisse se lire du début à la fin comme un roman ou une nouvelle. On part du haut pour arriver en bas. Ainsi chaque fonction est suivie des fonctions de niveau d'abstraction inférieur afin que nous puissions lire le programme en descendant d'un niveau d'abstraction à la fois. Uncle Bob appelle cela la règle de la décroissance (Stepdown Rule). Nous sommes nombreux parmi les développeurs à avoir du mal à apprendre cette manière d'écrire du code, aussi, voyons cela comme le fait d'écrire différents paragraphes dans un livre. Chaque paragraphe représentant un niveau d'abstraction, et la suite logique du paragraphe précédent.

Parlons enfin du nommage des fonctions. Le principe de Ward nous indique ceci : « Vous savez que vous travaillez avec du code propre lorsque chaque fonction que vous lisez correspond presque parfaitement à ce que vous attendiez ». La moitié du travail qui mène à ce principe réside dans le choix du nom de fonctions qui soient courtes et ne fassent qu'une seule chose. En effet, plus une fonction est courte et précise, et plus il est facile de lui trouver un nom descriptif. Aussi, nous ne devons pas avoir peur de créer des noms longs tant qu'ils restent descriptifs. Ce genre de nommage vaut mieux qu'un nom trop court nécessitant un commentaire pour expliquer l'objectif de la fonction.

Les arguments

D'après Uncle Bob, le nombre d'arguments d'une fonction devrait idéalement être égal à zéro. On appelle cela les fonctions niladiques. Ensuite viennent les fonctions à un argument (monadiques ou unaires), puis à deux arguments (dyadiques). Toujours d'après Uncle Bob les fonction à trois arguments (triadiques) doivent être évitées autant que possible. Enfin, il estime que les fonctions à plus de trois arguments (polyadiques) exigent une très bonne raison ou alors devraient ne pas exister.

En effet, les arguments sont complexes, et possèdent une puissance conceptuelle très importante. De plus, les arguments sont encore plus pénibles à gérer du point de vue des tests.

Cela vient de la complexité que cela représente de rédiger tous les cas de tests qui permettent de vérifier que les différentes combinaisons des arguments fonctionnent correctement. En partant de ce principe, une méthode sans argument devient très simple à tester. Un fonction unaire l'est presque tout autant. Mais à partir de deux arguments, les choses peuvent devenir plus complexes. Avec plus de deux arguments cela peut vite devenir décourageant.

Les arguments indicateurs sont à proscrire. Par exemple, passer un booléen à une fonction est une pratique épouvantable. Cela complique la signature de la méthode, et proclame d'emblée que cette méthode fait plusieurs choses, par exemple : réaliser l'action A si true, sinon réaliser l'action B.

Lorsqu'une fonction semble avoir besoin de plus de deux ou trois arguments, il est probable que certains d'entre eux puissent être enveloppés dans une classe. Par exemple examinons le code suivant :

```
public Circle makeCircle(double x, double y, double radius);
public Circle makeCircle(Point center, double radius);
```

D'aucun pourrait croire qu'il s'agit là d'une simple ruse que de créer des objets pour réduire le nombre d'arguments, mais ce n'est pas le cas. Lorsque des groupes d'arguments sont passés ensemble, il est fort probable qu'ils fassent partie d'un concept qui mérite son propre nom. Notre argument devient ainsi descriptif via le nom de la classe.

Séparation des commandes et des demandes

Les fonctions que nous écrivons doivent, soit faire quelque chose, soit répondre à quelque chose, jamais les deux. Par exemple, une fonction doit modifier l'état d'un objet, ou retourner des informations concernant cet objet, mais pas les deux. Car en faisant les deux, elle apporte une confusion. Prenons l'exemple suivant :

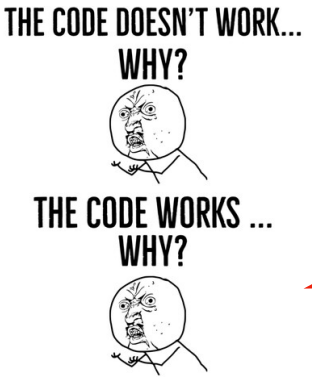
```
public boolean set(String attribute, String value);
```

Elle fixe la valeur d'un attribut nommé et retourne true en cas de succès et false si l'attribut n'existe pas. Cela va donc nous conduire à des instructions étranges :

```
if (set("username", "John")) …
```

Que signifie cette instruction ? Est-ce qu'elle demande si l'attribut username peut prendre la valeur John ou bien met-elle à jour l'attribut avec la valeur John ? Il est difficile de déduire l'objectif à partir de l'appel car le terme set n'est pas clair. Dans ce cas il vaudrait mieux avoir deux fonctions, une pour vérifier si l'attribut existe, et une autre pour définir la valeur de l'attribut. Exemple :

```
if (attributeExists(usernameAttribute)) {
  setAttribute(usernameAttribute, newUsernameValue);
}
```



Ce que l'on souhaite éviter à tout prix comme situation

Chapitre 6 - Les effets de bord

Définition

Les effets de bord, aussi appelés effets secondaires, sont des mensonges. Si une fonction promet de réaliser une tâche alors qu'elle fait des choses supplémentaires sans nous en informer, on dit que nous avons un effet de bord. L'un des effets de bord les plus répandus est la modification par référence à l'intérieur d'une fonction. C'est à dire modifier un objet, sans en informer l'appelant et sans retourner le nouvel état de l'objet. Ce genre de fonctions est absolument à bannir, car très préjudiciable et conduisant à des couplages temporels très étranges. Plus vous aurez d'effets de bord plus on pourra qualifier votre code de plat de spaghettis, soit un entrelacement sans aucune structure fiable des différentes briques de votre programme. Mais ce n'est pas le seul type d'effet de bord. Saurez-vous trouver l'effet de bord présent dans cette fonction ?

```
public class AuthenticationValidator {

  private final UserRepository userRepository;

  public boolean canAuthenticate(String username, String password) {
    final User inDbUser = userRepository.findByUsername(username);

    If (user != null && password.equals(user.getPassword())) {
      Session.initialize();
      return true;
    }

    return false;
  }
}
```

L'effet de bord se trouve dans l'appel à Session.initialize(). Nous avons ici un nom de fonction qui sous entend qu'elle va nous dire si la personne peut se connecter sur la base de son nom d'utilisateur et de son mot de passe. Or, à aucun moment elle ne dit qu'elle va démarrer une nouvelle session, et en plus prendre le risque d'écraser une session qui existe peut-être déjà. Cet effet de bord induit un couplage temporel. Autrement dit, on ne peut appeler cette méthode qu'à un certain moment si on ne veut pas prendre le risque de perdre les données présentes dans la session en cours. Les couplages temporels sont extrêmement déroutants, donc si vous en avez vraiment besoin, indiquez le dans le nom de la méthode : checkAuthenticationAndInitializeSession(). Mais cela ne respecte pas le principe SRP.

Ainsi, nous pourrons résumer les effets de bord de la manière suivante : une fonction est dite à effet de bord, si elle modifie un état en dehors de son environnement local, c'est-à-dire si il y a une interaction observable par le monde extérieur autre que la valeur de retour.

Pourquoi et comment les éviter

Pour éviter les effets de bord, il faut appliquer les règles que nous avons vu dans le chapitre traitant des fonctions. Mais nous allons voir que d'autres solutions existent, nous

permettant de réduire les risques d'effet de bord. La solution la plus intéressante s'appelle la programmation fonctionnelle.

La programmation fonctionnelle (PF) est un paradigme de développement de type déclaratif qui considère le calcul en tant qu'évaluation de fonctions mathématiques. Comme le changement d'état et la mutation des données ne peuvent pas être représentés par des évaluations de fonctions, la programmation fonctionnelle ne les admet pas. Au contraire elle met en avant l'application des fonctions, contrairement à la programmation impérative qui elle met en avant les changements d'état. L'origine de la programmation fonctionnelle peut être retrouvée dans **lambda-calcul**, un système formel inventé par Alonzo Church dans les années 1930. Une très bonne conférence au sujet de lambda-calcul a d'ailleurs été donnée à Devoxx France, je vous invite à aller la voir.

Des fonctions transparentes et pures

Une fonction est dite transparente si elle retourne toujours le même résultat pour une entrée donnée. Une fonction transparente est donc assimilable à un calcul mathématique. Exemple :

```
// a + b = c
public int sum(int a, int b) {
  return a + b;
}
```

Cette fonction est bien transparente, si je lui donne a = 1 et b = 2, elle me retournera toujours la valeur 3.

Mais alors qu'est ce qu'une fonction pure ? Et bien une fonction est dite pure lorsqu'elle ne fait appel qu'à des fonctions qui sont transparentes du début à la fin de la pile d'appels. On parle de fonction, mais les notions de transparence et de pureté s'appliquent également aux instructions plus généralement. À noter qu'en programmation fonctionnelle, il est possible de donner des fonctions en arguments de fonctions.

L'immuabilité

L'immuabilité, souvent appelée immutabilité par anglicisme, est un principe selon lequel tout objet instancié ne peut pas voir son état modifié. En d'autre terme, à partir du moment où un objet est instancié, les valeurs qui le composent ne peuvent plus être modifiées. Cela apporte plusieurs avantages :

- Les objets immuables sont thread-safe et n'ont pas besoin d'être synchronisés;
- Les méthodes equals et hashcode sont beaucoup plus fiables sur des objets immuables;
- Les objets immuables fonctionnent de pair avec les expressions pures et donc les expressions transparentes.

Mais alors comment faire si je dois absolument modifier un objet immuable ? Je vais créer un nouvel objet à partir de mon objet initial, et modifier la ou les valeurs souhaitées au moment d'instancier mon nouvel objet. Ainsi, si l'instance de l'objet initial est toujours utilisée dans une autre portée de mon programme, je n'ai aucun risque d'induire un effet de bord ou bien un couplage temporel. Mon code devient ainsi bien mieux segmenté, plus déclaratif, plus propre. Exemple de solution pour instancier un objet à partir d'un autre en modifiant une valeur :

```java
public final class User {

  private final String username;
  private final String emailAddress;

  public User(String username, String emailAddress) {
    this.username = username;
    this.emailAddress = emailAddress;
  }

  public String getUsername() {
    return username;
  }

  public String getEmailAddress() {
    return emailAddress;
  }

  public User withUsername(String newUsername) {
    return new User(newUsername, this.emailAddress);
  }
}

// Ou alors plus simple avec l'utilisation de Lombok

@Value
public final class User {

  @With
  private final String username;
  private final String emailAddress;
}
```

Chapitre 7 - Travailler à plusieurs

Tous les développeurs professionnels sont amenés à travailler en équipe, mais cela implique certaines complexités d'ordre organisationnel et de connaissances. Afin de réduire ces complexités il est possible de mettre en place un certain nombre d'actions ou de processus, afin d'assurer une organisation fluide, un partage des connaissances et ainsi gagner en qualité de code. Ce chapitre va s'attarder sur trois concepts différents : la revue de code, le pair programming et enfin le mob programming.

La revue de code

La revue de code, ou code review en anglais, est une pratique courante dans le milieu du développement. D'après quelques recherches, cette pratique aurait été formalisée dans les années 1970 chez IBM. Avec l'évolution des outils que nous utilisons depuis cette période, la revue de code a bien changé, mais le résultat est le même, faire relire son code par d'autres développeurs.

La revue de code, s'appuie sur les possibilités que nous offrent les outils de gestion de version comme git, en les couplant avec des plateformes comme GitHub, GitLab ou bien Bitbucket pour les plus connues. L'idée est de créer une nouvelle branche à partir de la branche commune de l'équipe, de réaliser ses développements dessus avant de créer ce que l'on appelle une demande de fusion de branches. La demande de fusion de branches est souvent appelée merge request ou encore pull request. Une fois une demande de fusion créée, les plateformes comme GitHub permettent aux autres développeurs de voir de manière très simple et complète, tous les changements qu'on a réalisé dans le code. Mais quel est le but d'une revue de code ? Le but est de détecter les défauts rapidement au plus tôt, dans le code soumis à revue. C'est une pratique très intéressante car plus un défaut est détecté tôt, plus il est simple à corriger. Néanmoins, une revue de code peut également porter sur autre chose que du code, comme par exemple de la documentation. Mais au delà de cet objectif, les revues créent une suite d'étapes où l'on est poussé à améliorer son code et donc à s'améliorer individuellement et collectivement.
Voici quelques une de ces étapes :

- **L'auto-critique :** lors de cette étape, il faut décrire le code qui est soumis à validation. Sur toutes les plateformes une zone de texte est mise à disposition pour cela. Trop peu de développeurs remplissent cette zone de texte par gain de temps pour eux, mais les collègues eux, risquent bien souvent de passer plus de temps sur votre revue si vous ne leur fournissez pas quelques informations essentielles. Aussi, dans cette zone de texte, soyez critiques envers vous-même et votre code, si une partie de celui-ci, ou une partie de votre raisonnement ne vous satisfait pas pleinement, indiquez-le. Cela permettra aux reviewers de faire plus attention aux points que vous leur soumettez.
- **L'auto-revue :** quand vous étiez en primaire, au collège ou au lycée, vos professeurs ont sans doute dû vous conseiller de toujours vous relire avant de rendre un travail. En programmation ce conseil s'applique également. Ainsi, avant de soumettre votre revue de code sur la plateforme, vous avez la possibilité de relire tous les changements que vous avez effectués avant de valider la création de la revue. Le fait de voir les changements dans un autre cadre que votre IDE permet souvent de détecter des défauts, et cela fera gagner du temps à tout le monde.
- **L'automatisation :** l'avantage d'utiliser un outil de gestion de version du code est de pouvoir automatiser de nombreuses choses. Ainsi, nous pouvons mettre en place une CI (Continuous Integration) qui va s'occuper au moment de la création de la revue, de compiler le code, d'exécuter les différents tests ou encore de lancer des analyses statiques de code par exemple. Cela permet de s'assurer que le code créé pour cette revue s'intègre parfaitement, qu'il ne présente aucun bug détecté par les tests, et donc permet de réduire la charge de travail des reviewers.
- **La relecture par les pairs :** c'est la dernière étape d'une bonne revue de code. Les autres développeurs vont relire le code que vous proposez, et vous faire leurs éventuels retours. C'est l'occasion pour chacun de partager son savoir, d'identifier des défauts que nous

n'avions pas vu avant mais également d'acquérir de la connaissance sur les travaux des autres. C'est un véritable moment de communication entre développeurs, propice à l'amélioration et à l'émergence de nouveautés.

Finalement, la revue de code est devenue avec le temps un processus standardisé, éprouvé mais surtout essentiel dans le développement logiciel, elle apporte la garantie d'une meilleure qualité, favorise la collaboration, pousse les développeurs à apprendre, et enfin elle permet de trouver des défauts bien avant que ces derniers n'apparaissent à l'usage. Aussi je vais conclure cette partie par un avis : la revue de code est essentielle dans tout projet en équipe, elle doit être utilisée autant que possible et doit amener les développeurs à réfléchir aux solutions qu'ils proposent en prenant du recul.

Le pair programming

Le pair programming, ou programmation en binôme, comme son nom l'indique, est une méthode de travail dans laquelle deux développeurs vont travailler ensemble sur un même poste de travail. On distingue deux rôles lors de cet exercice : le driver (conducteur), qui écrit le code ainsi que l'observer (observateur), qui suit ce que rédige le driver, l'assiste en décelant les imperfections. L'observer s'assure également que le code implémente correctement le design et suggère des alternatives de développement. Afin de bien répartir les tâches, les deux développeurs échangent régulièrement leurs rôles pendant la séance de pair programming, le but étant que chacun ait écrit du code et ait soutenu son collègue. Avant de parler des bénéfices induits par cette pratique, essayons dans un premier temps les indicateurs de non-performance de la méthode.

Un indicateur de non-performance pour le pair programming est un élément qui permet de dire que la méthode n'apporte pas les bénéfices escomptés ou alors pas assez. Il en existe d'ailleurs plusieurs. Tout d'abord, la plupart des développeurs sont conditionnés pour travailler seul et l'adoption du pair programming est parfois compliquée à faire passer. Même si la transition se fait en général avec succès, certains binômes ne parviennent pas à fonctionner avec cette méthode. Pour déterminer si un binôme ne fonctionne pas avec le pair programming nous avons heureusement quelques indicateurs :

- **Le silence :** programmer en binôme implique de communiquer à voix haute et de partager son point de vue avec son partenaire. Un silence persistant en session de pair programming indique un manque de collaboration, la méthode perdant de l'intérêt.
- **Le désengagement :** l'un des membre du binôme se détourne du projet, est lassé, et préfère faire autre chose durant la session de pair programming. Cette situation revient pour celui qui s'investit à développer seul.
- **L'effacement :** si l'un des deux membre du binôme est plus expérimenté que l'autre, le plus novice risque de se contenter d'observer le confirmé réaliser la majeure partie du travail. Là encore, la méthode perd de son intérêt.
- **Les problèmes relationnels :** les membres du binôme ne s'entendent pas et ne souhaitent pas travailler ensemble. Le pair programming est bien sûr contre-indiqué dans ce cas.

Voyons maintenant les bénéfices qu'apportent le pair programming.

Qualité et productivité

Certaines entreprises sont encore réticentes à l'adoption du pair programming, cela est dû à l'idée que deux personnes travaillant sur le même sujet constituent un gaspillage de ressources. Ainsi, plusieurs études ont essayé de s'intéresser à la productivité des binômes comparés à un développeur seul. Une mesure, l'effort relatif consenti par un binôme, de l'anglais relative effort afforded by pairs et abrégée REAP, a été définie de la façon suivante :

$\frac{(TempsPasséADeux) \times 2 - (TempsPasséSeul)}{TempsPasséSeul}$	• **TempsPasséADeux :** le temps passé par un binôme • **TempsPasséSeul :** le temps passé par un développeur seul

Avec cette formule, une valeur nulle indique qu'un binôme utilise exactement la moitié du temps nécessaire pour un seul développeur pour réaliser une même tâche. Une valeur de REAP comprise entre 0 et 100 % indique qu'un binôme réalise une tâche plus rapidement mais nécessite davantage d'heure-homme. Une étude de 1998 indiquait qu'à la mise en place du pair programming le REAP moyen était d'environ 60% mais une autre étude complète ce résultat en précisant qu'après une période d'ajustements ce taux descend aux alentours de 15%. On peut ainsi déduire de ces valeurs que le pair programming est un formidable outil pour accélérer la mise sur le marché d'un logiciel.

Le pair programming permet également de gagner en qualité de code grâce à une détection plus rapide des défauts et des bugs, et n'empêche bien sûr pas la revue de code. De manière générale on peut dire qu'un binôme conçoit du code de meilleure qualité qu'un développeur seul. Cette différence de qualité s'explique en partie par la nécessité pour un binôme de s'accorder au préalable sur la conception et de la justifier. On se retrouve ainsi avec des applications bien mieux testées mais aussi avec moins de lignes de code. Or en programmation, moins il y a de lignes de code, moins le risque de bugs est grand. Économiquement le pair programming induit un surcoût en personnel, mais il est vite compensé par la hausse de la qualité. Plus un défaut est détecté tardivement, plus il sera couteux à corriger. Les déceler le plus tôt possible représente donc une économie potentiellement importante pour les entreprises.

Communication

Développeur est un métier social, mais le pair programming est une activité encore plus sociale qui oblige à la communication et d'apprendre à travailler ensemble. Cela permet aux membres d'une équipe d'échanger beaucoup plus naturellement et simplement. De plus, le pair programming permet d'améliorer les compétences de communications qui sont généralement peu travaillées pendant l'apprentissage de l'informatique. Enfin cette pratique a également pour effet de renforcer les liens entre les développeurs et de créer un esprit d'équipe.

Apprentissage

Au sein d'un projet professionnel, le pair programming encourage le transfert de connaissances techniques entre les membres de l'équipe, notamment lorsqu'un développeur novice travaille avec un expert. Le novice peut non seulement apprendre de nouvelles technologies grâce à l'expert, mais aussi apprendre de bonnes pratiques ou découvrir de nouveaux logiciels utiles au développement.

Le mob programming

Le mob programming, ou la programmation en groupe, est une méthode de développement dans laquelle toute l'équipe travaille sur un même sujet, en même temps, dans le même lieu, sur le même ordinateur et est une extension du pair programming. La notion de Mob Programming a été créée par Roy Zuill (a.k.a. Woody) et son équipe. Woody est un coach agile, un manager, et un développeur depuis près de 30ans. Il est le porte-parole de cette méthodologie. Woody explique que cette méthode s'est imposée progressivement. Dans un premier temps l'équipe travaillait de manière plus classique, chacun à son poste. Puis, comme dans de nombreux projets, le besoin de se réunir en équipe dans une salle de réunion s'est fait ressentir lors de problèmes complexes, touchant le cœur du projet et nécessitant les connaissances de l'ensemble des membres de l'équipe. Woody, passionné d'agilité, a donc

organisé des Coding Dojo en y ajoutant des notions de la méthode de pair-programming “Driver/ Observer”. L’équipe a alors immédiatement réalisé la puissance de cet embryon de Mob Programming et pris l’habitude d’en réaliser régulièrement. Ainsi, une réalité s’est imposée : non seulement les développeurs appréciaient d’avantage leurs journées, mais en plus, le nombre de tâches réalisées (et donc leur productivité) était significativement amélioré.

Le Mob Programming se base sur le fait qu’un développeur passe plus de temps à réfléchir, conceptualiser et communiquer avec son équipe qu’à écrire du code. Il est évident que si vous calculez l’efficacité d’une équipe par le nombre de lignes codées par minutes, le Mob Programming ne va pas vous sembler très avantageux. Ceci étant dit, comment expliquer qu’en faisant du Mob Programming une équipe peut être plus productive ? Nous pouvons imaginer trois axes de réponse :

- **Une équipe sur une seule tâche**
  - Il arrive parfois qu’un développeur passe beaucoup de temps sur une tâche avant qu’on ne se rende compte que ça ne correspondait pas à ce qu’il fallait. Ce genre d’erreur de compréhension peut avoir des effets dramatiques. Dans le Mob Programming, les erreurs de compréhension entre les membres internes de l’équipe sont nulles. En effet, à la moindre dérive c’est l’ensemble de l’équipe qui va pouvoir remettre le développement dans la bonne direction. De même, le product-owner est invité à participer aux séances de développement. Ainsi, lorsqu’il établit des scénarios, tous les membres de l’équipe vont les analyser. A la moindre imprécision ou ambiguïté l’équipe va toujours chercher à comprendre clairement ce qui est attendu.
  - Une des plus grandes forces du Mob Programming est le « continuous learning ». Tout d’abord, un nouveau membre sera directement intégré au développement. Il pourra apprendre très naturellement avec beaucoup d’explication orale. Ensuite, les profils juniors vont pouvoir énormément profiter des connaissances techniques des profils seniors, alors qu’inversement, les profils juniors n’hésiteront pas à parler des dernières nouveautés à la mode. Le niveau d’échange est très important, que ce soit au sujet des bonnes pratiques, ou de raccourcis clavier ou encore d’autres fonctionnalités des outils utilisés. Le niveau de l’ensemble de l’équipe va perpétuellement s’améliorer.
- **Une équipe soudée face à son environnement**
  - S’il y a des problèmes liés à l’environnement externe, comme par exemple du bruit, c’est toute l’équipe qui sera mobilisée pour remédier au problème, et non juste un seul développeur qui se sera retrouvé au mauvais endroit. De plus, si toute l’équipe est interrompue, au moins un des membres de l’équipe se souviendra précisément où l’équipe s’était arrêtée, et elle pourra donc rapidement reprendre où elle s’était arrêtée.
  - Il sera plus facile pour l’équipe de faire avancer son avis sur des questions fonctionnelles, comme par exemple supprimer des fonctionnalités pour aller à l’essentiel. Face aux product-owners, ou aux supérieurs, toute l’équipe pourra argumenter sur ce qu’elle estime être la meilleure chose à faire.
- **Le collectif avant l’individualisme**
  - En informatique, il n’est pas rare de voir des développeurs qui arrivent à bien se vendre sans pour autant être très compétents. Lorsque l’on fait du Mob Programming on est exposé, on ne peut plus mentir sur ses capacités. À l’inverse, une personne beaucoup plus modeste sur ses capacités pourra être reconnue à sa juste valeur.
  - Enfin, contrairement à des méthodologies plus classiques où un développeur s’attacherait uniquement à sa partie, ici tout le monde travaille pour l’ensemble de la solution. Il n’y aura donc pas de place pour la politique, chaque développeur va faire de son mieux pour que la solution soit complète.

Comme pour toutes les méthodes, il faut en identifier les points forts et les points faibles. Le mob programming ne déroge pas à cette règle. Effectivement, certains environnement n’y sont pas propices, certains éléments doivent donc être pris en considération avant de se lancer :

- Cette méthodologie n’a de sens que si la qualité du code est une priorité principale. Faire du Mob programming en mettant de côté tout le reste comme le clean code ou l’agilité, alors le mob programming ne résoudra rien car il complète ces autres techniques.
- La mise en place de cette technique peut nécessiter un investissement dans du matériel adapté à cette pratique. Par exemple, une salle dédiée avec suffisamment de chaises ainsi qu’une disposition d’affichage optimale avec des projecteurs.

- Il faut les bonnes personnes, car travailler ensemble constamment nécessite une très bonne entente entre tous les développeurs. Il faut donc faire attention aux relations qu'entretiennent les membres de l'équipe entre eux et aux personnes qui pourraient être trop introverties.

Comme nous venons de le voir lors de ce chapitre sur le travail à plusieurs, de nombreuses méthodes, techniques et outils essentiels sont à notre disposition. La revue de code représente le strict minimum d'un travail en équipe efficient et de qualité, le pair programming est un outil incroyable pour mettre l'accent sur la qualité des développements et gagner en efficacité. Enfin, le mob programming est également une excellente pratique, cependant elle est à mon sens à utiliser avec parcimonie, comme par exemple pour traiter les sujets les plus complexes, et ne doit pas devenir la norme mais bien une solution aux problèmes complexes.

Chapitre 8 - Tester son code

Définition

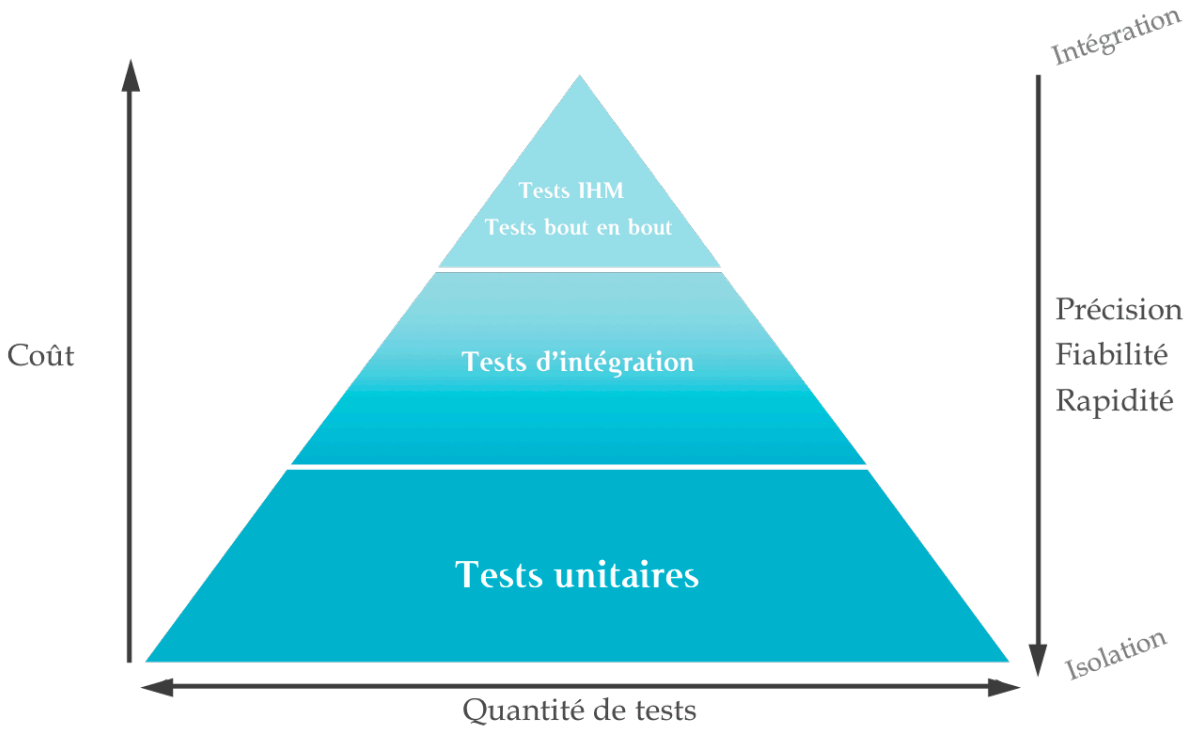
En informatique, et plus particulièrement dans le milieu du développement, un test est une procédure de validation partielle d'un système. L'objectif principal est d'identifier un nombre maximal de comportements problématiques du logiciel. Cela permet ainsi, dès lors que les problèmes identifiés seront corrigés, d'en augmenter la qualité. Dans le monde du développement, lorsque l'on parle de tests, on en distingue majoritairement 3 types mais il en existe un quatrième.

Les tests unitaires (TU) : du code écrit par un développeur qui va tester du code écrit par un développeur.

Les tests d'intégration (TI) : ils complètent les tests unitaires mais à l'inverse de ces derniers, ils vont assembler les différentes parties d'un logiciel pour tester une fonctionnalité dans un ensemble plus complet.

Les tests de bout en bout (E2E) : ils complètent les tests d'intégration et ont pour but de valider si oui ou non une application se comporte comme prévu du début à la fin.

Les tests unitaires représentent le premier rempart, et doivent toujours être réalisés en investissant massivement dessus. Les tests d'intégration nécessitant souvent la mise en place de scénarios doivent être favorisés sur certaines partie d'une application comme par exemple l'intégration avec un moteur de base de données, et rester dans le cadre interne de l'application. Les tests E2E quand à eux sont les plus couteux à mettre en place, mais également les plus difficiles à maintenir. Ainsi nous les réserverons à ce que l'on nomme les « golden paths », soit les parcours ayant le plus de valeur pour l'entreprise. Ces différentes topologies de tests viennent d'un concept que l'on nomme la pyramide des tests à cause de la représentation que nous en faisons.



La dernière topologie de tests, souvent oubliée et très peu mise en place est **le mutation testing**. Cette technique permet de s'assurer que nos tests unitaires sont de bonne qualité et testent réellement notre code, mais nous y reviendrons plus tard dans ce cours.

Pourquoi tester son code ?

Lorsque nous écrivons du code, de nombreux facteurs peuvent faire en sorte que nous introduisions des bugs. L'inattention et le manque de conception en font partie. Tester son code permet de s'assurer que le code produit répond parfaitement au besoin (le cas passant) mais est également résistant à une utilisation détournée comme l'envoi d'arguments invalides par exemple. Cependant il ne faut pas oublier que la présence de tests ne prouve en aucun cas l'absence totale de bugs. Ils permettent simplement de s'assurer que les potentiels cas que le code va rencontrer sont traité de la façon voulue. Néanmoins tous les cas possibles ne sont pas forcément testés. Les tests permettent donc de détecter les bugs de manière précoce, bien avant que le programme soit disponible à l'usage. Les tests assurent également la maintenance du programme. Lors d'une modification du code les tests et surtout les TU vont échouer si une potentielle régression est détectée, ainsi le développeur peut soit modifier les tests car ils ne sont plus à jour, soit réparer l'erreur qu'il a introduite dans le code. Enfin les tests permettent de documenter le code. En effet, lire les tests d'une méthode est un processus utile pour comprendre comment utiliser la méthode testée. Les tests unitaires dans tout projet sont nécessaires, et le coût qu'ils représentent est bien inférieur aux coûts induits par leur absence. De plus, écrire des tests n'est absolument pas une raison pour arrêter de s'appliquer une rigueur de travail en arrêtant de faire du code propre. Le clean code et les tests sont indissociables.

Tests unitaires

Les tests unitaires, dont nous avons parlé précédemment, sont le premier rempart contre une majorité de problèmes, ils coûtent peu cher à mettre en place, sont rapides et testent des parties limitées d'un programme. L'écriture de tests unitaires dans le développement moderne est

une nécessité, pour ne pas dire une obligation. Que le projet soit scolaire, professionnel ou encore personnel, aucun développeur ne devrait écrire du code qui n'est pas testé par des tests unitaires.

Cependant, comment faire pour tester des morceaux de code sans être pollué par les dépendances de ce code ? Nous avons pour cela quelques notions importantes à apprendre : les affirmations, les simulacres, les bouchons et enfin les espions. Les exemples donnés durant ce chapitre sont tous basés sur le langage java, la librairie de tests JUnit 5, sur la librairie de mocks Mockito et sur la librairie d'affirmations AssertJ.

Affirmations (assertions)

Les affirmations dans un test (unitaire, intégration E2E) représentent la partie la plus importante d'un test, c'est grâce aux affirmations que l'on s'assure que le code testé réagit correctement. Prenons le code suivant qui permet de convertir des chiffres romains en chiffres arabes :

```
public interface RomanToIntegerConverter1 {

    Map<Character, Integer> ROMAN_INTEGER_EQUIVALENTS = Map(
            'I', 1,
            'V', 5,
            'X', 10,
            'L', 50,
            'C', 100,
            'D', 500,
            'M', 1000);

    Map<String, String> ROMAN_SUBTRACTION_CHEATS = Map(
            "IV", "IIII",
            "IX", "VIIII",
            "XL", "XXXX",
            "XC", "LXXXX",
            "CD", "CCCC",
            "CM", "DCCCC");

    static int convertRomanToInteger(String roman) {
        if (roman == null || roman.isBlank()) {
            throw new IllegalArgumentException("Please provide a valid roman number");
        }

        final var romanWithoutSubtractions = replaceSubtractionsWithCheats(roman.toUpperCase());
        return List.ofAll(romanWithoutSubtractions.toCharArray())
                .map(ROMAN_INTEGER_EQUIVALENTS::get)
                .map(maybeEquivalent -> maybeEquivalent.getOrElseThrow(() ->
                        new IllegalArgumentException("The given string does not respect the roman notation")))
                .reduce(Integer::sum);
    }

    private static String replaceSubtractionsWithCheats(String roman) {
        for (Tuple2<String, String> cheat : ROMAN_SUBTRACTION_CHEATS) {
            roman = roman.replace(cheat._1, cheat._2);
        }
        return roman;
    }
}
```

Dans notre exemple, la méthode **convertRomanToInteger()** prend en paramètre d'entrée une chaîne de caractères contenant des chiffres romains. A partir d'un chiffre en notation romaine, elle retourne son équivalent en entier. Pour s'assurer que notre méthode convertit bien un chiffre romain en entier nous allons utiliser une affirmation, aussi appelée assertion en anglais. Exemple :

```java
@Test
void should_convert_roman_to_integer() {
  final var roman = "XLII";
  final var actual = RomanToIntegerConverter1.convertRomanToInteger(roman);
  assertThat(actual).isEqualTo(42);
}
```

Dans ce test nous faisons bien un appel à la méthode convertRomanToInteger et nous mettons la valeur de retour dans la variable actual. La dernière ligne de notre test est notre affirmation, et elle vérifie que la valeur de actual est bien égale à 42. Cette affirmation est extrêmement simple. Vérifier que deux entiers sont identiques n'est pas une chose très compliquée, mais les affirmations permettent de faire des choses bien plus complexes comme par exemple comparer les éléments d'une collection comme une liste avec ceux d'une autre liste de manière ordonnée ou non. On peut aussi comparer des objets entre eux, mais également les comparer récursivement entre eux c'est à dire champs par champs. Ce ne sont bien sûr que des exemples car en réalité les affirmations permettent de vérifier tout type de retour avec une valeur attendue.

Simulacres (mocks)

Les simulacres, ou mocks en anglais, sont des objets dont nous déléguons la création à une solution de mocking comme Mockito en java. Cela nous permet de simuler le comportement d'objets dont dépend le code que l'on cherche à tester. Ainsi, si une méthode A d'un objet Foo est appelé dans notre code, on peut définir grâce à notre mock le comportement que doit avoir la méthode A quand elle est appelée. On appel cela un bouchon ou un Stub en anglais. En java avec Junit5 et Mockito il existe deux manière de déclarer des Mocks. Prenons l'exemple d'un service dont la mission est de créer de nouveaux permis de conduire dans un système et dont voici le code :

```java
public class DrivingLicenceCreatorService implements DrivingLicenceCreatorApi {

  private final DrivingLicencePersistenceSpi spi;

  @Override
  public Either<ApplicationError, DrivingLicence> create(DrivingLicence drivingLicence) {
    return SocialSecurityNumberValidator.validate(drivingLicence)
        .toEither()
        .peekLeft(
            error -> log.error("An error occurred while validating driving licence : {}", error))
        .flatMap(spi::save);
  }
}
```

Nous voyons bien que la classe DrivingLicenceCreatorService dépend d'une autre classe nommée DrivingLicencePersistenceSpi, et que la méthode save de cette dernière est utilisée dans la méthode create de notre classe que nous souhaitons tester. Afin de ne pas avoir à gérer tout l'arbre de dépendance de la classe à tester nous créons un mock de notre dépendance. Il existe pour cela deux façons de faire :

```
class DrivingLicenceCreatorServiceTest {
  private DrivingLicenceCreatorService service;
  private DrivingLicencePersistenceSpi spi;

  @BeforeEach
  void init() {
    spi = mock(DrivingLicencePersistenceSpi.class);
    service = new DrivingLicenceCreatorService(spi);
  }
}
```

Voici la première méthode qui est aussi la plus ancienne et qui tend à disparaitre. Dans cette façon de faire, nous réalisons nous même la gestion de la classe mockée.

```
@ExtendWith(MockitoExtension.class)
class DrivingLicenceCreatorServiceTest {

  @InjectMocks
  private DrivingLicenceCreatorService service;

  @Mock
  private DrivingLicencePersistenceSpi spi;
}
```

Voici la seconde méthode, plus concise et qui est à favoriser. Grâce à JUnit5 il est possible de déléguer une partie du code à des extensions et nous utilisons ici l'extension fournie par la librairie de mocking Mockito. Cet exemple est absolument identique au premier en terme de fonctionnement, seulement il présente l'avantage d'être plus lisible et plus simple. En clean code, on préfère toujours ce qui est plus lisible et plus simple.

Bouchons (stubs)

Maintenant que nous savons mocker des objets dont dépendent nos classes testées, voyons comment simuler les comportements lors des interactions avec ces derniers. Pour cela, nous l'avons évoqué dans la partie Mocks, nous allons utiliser des bouchons ou stubs en anglais. En repartant du même exemple que dans la partie mocks, voici un exemple de stub :

```
@Test
void should_create_licence() {
  val given = DrivingLicence.builder().driverSSNumber("123456789098765").build();
  when(spi.save(given)).thenReturn(Right(given));

  val actual = service.create(given);
  assertThat(actual).containsRightSame(given);
}
```

La ligne 2 de notre test définie un Stub. Ici, elle indique que lorsque la méthode **save()** de la classe mockée est appelée avec un certain paramètre, alors elle retournera ce que l'on indique dans l'instruction **thenReturn().**

Capteurs (captors)

Les capteurs ou captors en anglais, sont des types d'objets que l'on peut utiliser sur nos méthodes mockées. Il permettent de vérifier la valeur d'un paramètre donné à une méthode mockée quand depuis le test nous ne sommes pas capable de l'avoir directement. En d'autres termes, les capteurs permettent de nous assurer que les arguments fournis et qui sont créés à l'intérieur de méthode testée sont bien ceux auxquels nous nous attendons.

```
@InjectMocks
private DrivingLicenceDatabaseAdapter adapter;

@Mock
private DrivingLicenceRepository repository;

@Captor
private ArgumentCaptor<DrivingLicenceEntity> entityCaptor;

@Test
void should_save() {
  val licence = DrivingLicence.builder().build();
  val entity = DrivingLicenceEntityMapper.fromDomain(licence);

  when(repository.save(any(DrivingLicenceEntity.class))).thenReturn(entity);

  val actual = adapter.save(licence);

  verify(repository).save(entityCaptor.capture());
  verifyNoMoreInteractions(repository);

  VavrAssertions.assertThat(actual).isRight().containsRightInstanceOf(DrivingLicence.class);
  assertThat(actual.get()).usingRecursiveComparison().isEqualTo(licence);
  assertThat(entityCaptor.getValue()).usingRecursiveComparison().isEqualTo(entity);
}
```

Les tests unitaires paramétrés

Les tests unitaires paramétrés sont des outils extrêmement puissant dans le sens où ils permettent de factoriser un seul test pour réaliser plusieurs vérifications avec des paramètres d'entrés différents. Pour le dire d'une autre manière, les tests paramétrés permettent de tester plusieurs cas de tests en n'écrivant qu'un seul test unitaire. De plus, les tests paramétrés peuvent

aussi fonctionner avec des mocks, des stubs ainsi que des captors. Voici un exemple de test unitaire paramétré qui teste un méthode validant le format d'un numéro de sécurité sociale :

```
@ParameterizedTest
@NullAndEmptySource
@ValueSource(strings = {"lorem ipsum", "azertyuiopmlkjh", "098765432109876543210", "098654"})
void should_not_validate(String invalidSSNumber) {
  val actual = validate(DrivingLicence.builder().driverSSNumber(invalidSSNumber).build());
  assertThat(actual).containsInvalidInstanceOf(ApplicationError.class);
}
```

Dans cet exemple, nous allons exécuter 6 fois le même test mais avec des valeurs d'entrée différentes, ce qui est bien plus rapide que d'écrire six tests identiques mais également beaucoup plus simple et lisible dans notre cas. Attention toutefois, les tests paramétrés peuvent s'avérer parfois très complexes à relire, il faut donc savoir les utiliser avec parcimonie.

Tests d'intégrations

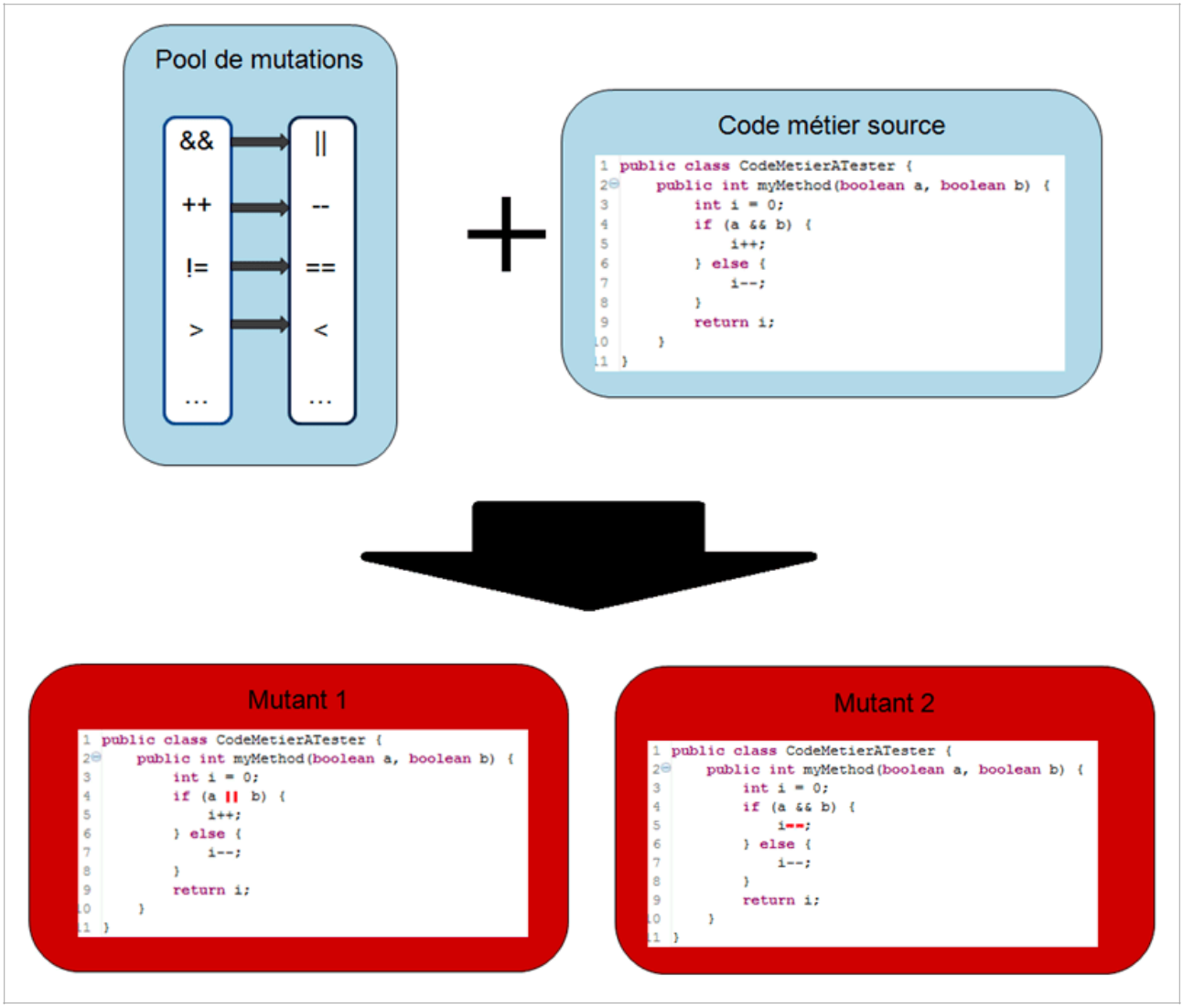
Les tests d'intégration sont les tests qui arrivent juste derrière les tests unitaires. Ils permettent de vérifier une partie bien précise d'un programme comme par exemple les interactions avec une base de données. Ce genre de test est bien plus poussé que les tests unitaires car il demande plus de code. Dans notre exemple des interactions avec une base de données, il faudra absolument mettre à disposition de nos tests d'intégration une base de données dont le moteur est le même que celui utilisé lors de l'utilisation de notre programme. Nous ne nous attarderons pas plus sur ce sujet dans ce cours, mais voici une librairie Java qui permet de réaliser des tests d'intégrations au moyen d'images docker, et qui fonctionne parfaitement avec JUnit : testcontainers.

Tests de mutations

Les tests de mutations ou Mutation Testing sont souvent mis de côté par la communauté des développeurs et sont même très peu souvent enseignés. Pourtant ils apportent certains avantages intéressants à notre code. Le mutation testing a été inventé en 1971 par Richard Lipton, mais l'usage de cette technique était alors très limité car il nécessitait des ressources matérielles assez importantes pour l'époque. Aujourd'hui les machines modernes sont tout à fait adaptées à cette pratique. Dans un processus de mutation testing on distingue deux étapes : la génération des mutants et la chasse de mutants.

La génération des mutants permet de générer des classes à partir des classes testées, dans lesquelles on introduit des variations au moyen d'un pool de mutations. Par exemple, on va remplacer l'opérateur **>** par l'opérateur **<**. Pour générer ces mutants, on parcourt toutes les instructions présentes dans le code, pour chaque instruction on regarde si une mutation est applicable depuis le pool de mutations, et si une mutation est possible on donne naissance à un mutant. Ainsi, plus il y aura d'instructions, plus le nombres de mutants pourra être élevé et plus les ressources nécessaires au processus seront importantes.

Exemple de pool de mutation appliqué à une classe :



Maintenant que nos mutants sont générés, passons à la chasse de ces derniers et surtout à leur abattage. Cette phase à pour but de tuer le maximum de mutants, et les armes utilisées pour cela sont les tests unitaires. En d'autres termes, nous exécutons nos tests unitaires sur les mutants générés. Une fois l'exécution terminée, un mutant peut avoir deux statuts : **SURVIVED** ou **KILLED**. En temps normal, nos tests unitaires doivent être verts une fois exécuté, dans le cadre du mutation testing il doivent être rouges et donc en erreur. C'est l'erreur des tests unitaires qui atteste de la mort des mutants. Si au minimum un test unitaire est en erreur sur un code testé, alors la détection des modifications est considérée comme valide et nos tests unitaires testent correctement notre code. À l'inverse, si tous les tests restent verts cela signifie que si l'on modifie le code testé, les tests continueront de passer. Leur utilité est donc très limitée, il faudra penser à les revoir. Il est important de garder en tête qu'un taux de 100% de mutants tués dans un cas concret est presque impossible, et nous reviendront sur ce sujet juste après au sujet des mutants équivalents.

Le mutation testing possède quelques limitations car le principe est simple mais l'analyse des résultats peut être très longue et complexe à cause du nombre de mutants générés. Par exemple, si nous générons 8000 mutants avec un taux de 95% de mutants tués, il reste 400 mutations à analyser. Cette analyse doit être faite à la main par un développeur, et certains mutants peuvent devenir complexes à comprendre. De plus, un mutant qui survit n'est pas forcément dû à de mauvais tests unitaires, car il existe ce que l'on appelle les mutants équivalents.

Les mutants équivalents sont des mutants qui modifient la syntaxe mais pas la sémantique du code testé. De ce fait, aucun test unitaire au monde ne peut détecter ce genre de cas. Voici un exemple de mutation équivalente :

```
// métier
int index = 0;
while(...) {
  ...;
  index++;
  if (index == 10) break;
}
```

```
// mutant
int index = 0;
while(...) {
  ...;
  index++;
  if (index >= 10) break;
}
```

En Java il existe deux outils pour faire du mutation testing : Javalanche et PiTest. Ma préférence va plutôt vers PiTest.

Test Driven Development (TDD)

Le test driven development (TDD) ou développement piloté par les tests en français, est une méthode de développement logiciel par laquelle on conçoit un logiciel de manière progressive par une série de petites étapes et dans laquelle on écrit toujours les tests avant d'écrire le code source. L'un des avantages du TTD, est que nous allons devoir nous efforcer d'écrire du code source simple, qui satisfasse les attentes de nos tests unitaires. Autrement dit, nous devons écrire les solutions les plus basiques possibles pour faire passer nos tests au vert. Une fois que les tests sont au vert, on ne touche plus à rien et nous passons aux tests suivants.

D'après Robert C. Martin alias Uncle Bob, l'auteur de l'ouvrage sur le clean code mais aussi l'une des figures de tête du mouvement TDD, la pratique du TDD doit répondre à trois règles :

- **On doit écrire des tests qui échouent avant d'écrire le code lui même**
- **On ne doit pas écrire un test plus compliqué que nécessaire**
- **On ne doit pas écrire plus de code que nécessaire, soit juste assez pour que le test passe au vert**

Ces trois règles fonctionnent ensemble et font partie d'un processus que l'on appelle red-green-refactor qui est un cycle de développement à suivre si l'on souhaite faire du TDD.

- **Red :** écrire un test simple qui échoue et qui ne compilera probablement pas.
- **Green :** écrire le code source le plus simple et basique possible pour faire passer le test au vert.
- **Refactor :** le code que l'on vient d'écrire est peut-être trop simplifié ou illogique, il faut maintenant le remanier (de l'anglais to refactor) afin d'obtenir un code propre.

Mais alors quel est l'intérêt d'utiliser le TDD dans un processus de développement d'un programme ? Les tests tels qu'ils sont mis à profit dans le TDD permettent d'explorer et de préciser le besoin, puis de spécifier le comportement souhaité du logiciel en fonction de son utilisation, avant chaque étape de développement. Le logiciel ainsi produit est tout à la fois pensé pour répondre avec justesse au besoin et conçu pour le faire avec une complexité minimale. On obtient donc un programme mieux conçu, mieux testé et donc plus fiable, c'est à dire de meilleure qualité. Le TDD s'insère donc parfaitement dans le clean code. Mais le TDD présente un autre intérêt : quand les tests sont écrits après le code source, comme c'est le cas traditionnellement, les choix d'implémentation contraignent l'écriture des tests : les tests sont écrits en fonction du code, et si certaines parties du code ne sont pas testables, elles ne seront pas testées, ou alors il faudra remanier tout le code déjà écrit pour le rendre testable. Au

contraire, en testant avant d'écrire le code source, on utilise le code avant son implémentation, de sorte que les contraintes définies par les tests s'imposent à l'implémentation : le code est écrit en fonction des tests. Le fait d'écrire les tests avant le code conduit donc à des implémentations testables à 100% et facilement. Or la testabilité du code favorise une meilleure conception par un couplage faible et une cohésion forte, ce qui évite des erreurs de conception courantes. Enfin la pratique du TDD s'intègre parfaitement dans le cadre du pair et du mob programming.

Chapitre 9 - Les principes SOLID

Impossible de faire un cours sur le clean code sans parler des principes SOLID. Il s'agit là d'un acronyme mnémotechnique qui représente 5 principes de conception en programmation orientée objet qui permettent de produire des architectures logicielles plus compréhensibles, plus flexibles et surtout plus maintenables. Ils sont d'ailleurs un sous ensemble des principes de programmation promus par Uncle Bob. La théorie des principes SOLID a d'ailleurs été introduite par Uncle Bob dans son article « *Design Principles and Design Patterns* » paru en 2000. Mais l'acronyme SOLID, lui, n'est apparu qu'un peu plus tard.

Au premier abord, les concepts objets ne sont pas simples à appréhender. Il est en effet plus facile de raisonner de manière linéaire et concrète que de manière abstraite comme en orienté objet avec des notions comme l'héritage ou encore le polymorphisme. Afin de bien comprendre l'orienté objet, il faut un certain recul et une capacité d'abstraction pas totalement intuitive au départ. Mais la conception objet offre d'un autre côté des mécanismes uniques de représentation de notre environnement en mixant à la fois données et comportements, et en déclinant des notions génériques en plusieurs notions spécialisées. En plus de cela, la POO offre des mécanismes de tolérance aux changements incroyablement puissants. Cette tolérance est un des facteurs clés du développement logiciel, la maintenance demandant souvent des efforts croissants au fil du temps et de l'évolution du logiciel. Mais comment caractériser l'intolérance au changement ? Voici une liste non exhaustive des symptômes de l'intolérance aux changements :

- **La rigidité :** « Chaque changement cause une cascade de modifications dans les modules dépendants. » Une intervention simple au premier abord se révèle être un véritable cauchemar, à la manière d'une pelote de laine qui n'a pas de fin.
- **La fragilité :** « Tendance d'un logiciel à casser en plusieurs endroits à chaque modification. ». La différence avec la rigidité réside dans le fait que les interventions sur le code peuvent avoir des répercussions sur des modules n'ayant aucune relation avec le code modifié. Toute intervention est soumise à un éventuel changement de comportement dans un autre module.
- **L'opacité :** correspond à la lisibilité et la simplicité de compréhension du code. La situation la plus courante est un code qui n'exprime pas sa fonction première. Ainsi, plus un code est difficile à lire et à comprendre et plus nous allons considérer que ce dernier est opaque.

Mais heureusement quand on applique les principes SOLID, on peut anéantir la résistance aux changements.

Single responsibility principle

« UNE CLASSE NE DOIT CHANGER QUE POUR UNE SEULE RAISON » (*A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE*)

Comme son nom l'indique, ce principe signifie qu'une classe, une méthode, une variable ne doit posséder qu'une et une seule responsabilité. Si une classe a plus d'une responsabilité, ces dernières se retrouveront liées. Les modifications apportées à une responsabilité impacteront l'autre, augmentant la rigidité et la fragilité du code, et plus longtemps le code restera comme cela, plus les problèmes deviendront importants et complexes à résoudre. Le **SRP** ou **principe de la responsabilité unique** est probablement le plus connu et le plus simple à comprendre, des principes SOLID. Le SRP est aux principes orientés-objet ce que le Singleton est aux patrons de conception. Pour savoir si une classe respecte le SRP, il faut dire: « La classe X fait … » en étant le plus spécifique possible. Si la phrase ci-dessus contient des *et* ou des *ou,* alors votre classe a plus d'une responsabilité. De façon plus subtile, si elle contient des mots génériques comme *gère* ou *objet (*exemple: gère les utilisateurs, valide les objets), alors vous devriez avoir la puce à l'oreille. Exemple :

Cette classe encapsule les données d'un utilisateur

```
public class User {
  private String firstName;
  private String lastName;
  private String phoneNumber;

  public void setFirstName(String firstName) {
    if(firstName == null) throw new Exception("Invalid first name");
    this.firstName = firstName;
  }

  public void setLastName(String lastName) {
    if(lastName == null) throw new Exception("Invalid last name");
    this.lastName = lastName;
  }

  public void setPhoneNumber(String phoneNumber) {
    if(phoneNumber == null || phoneNumber.lenght() != 10) throw new Exception("Invalid phone
numeth}is.phoneNumber = phoneNumber;
  }
}
```

Elle ne respecte pas le principe SRP car les setters possèdent une logique de vérification des données fournies et lèvent des exceptions en cas de valeurs non admissibles.

Afin de respecter le SRP il faut avoir une classe sans aucune logique, sont but est uniquement d'encapsuler les données d'un utilisateur :

```
public class User {
  private String firstName;
  private String lastName;
  private String phoneNumber;

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public void setPhoneNumber(String phoneNumber) {
    this.phoneNumber = phoneNumber;
  }
}
```

Note : les objets immuables sont à favoriser, ceci n'est qu'un exemple simple afin d'illustrer le SRP.

Open Close principle

« LES CLASSES, LES MÉTHODES, DOIVENT ÊTRE OUVERTES À L'EXTENSION MAIS FERMÉES AUX MODIFICATION » (CLASSES, METHODS SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR MODIFICATIONS)

Une classe, une méthode, doit pouvoir être étendue, supporter différentes implémentations (Open for extension) sans pour cela devoir être modifiée (closed for modification). Les instanciations conditionnelles dans un constructeur sont de bons exemples de non respect de ce principe. Une nouvelle implémentation aura pour impact l'ajout d'une condition dans le constructeur. Exemple :

```
public class Car {
  private final Engine engine;

  public Car (EngineType engineType) {
    if(engineType == FUEL) {
      this.engine = new FuelEngine();
    } else if (engineType == ELECTRIC) {
      this.engine = new ElectricEngine();
    }
  }
}
```

Dans cet exemple nous voyons que l'ajout d'un type de moteur va entraîner une modification du constructeur : nous sommes en violation avec la deuxième partie du principe *Closed for modification*. Plusieurs solutions s'offrent à nous pour respecter les deux parties du principe : utiliser l'injection de dépendance dans sa forme la plus simple ou encore utiliser une fabrique d'objets. Ce ne sont bien sûr par les seules solutions mais ce sont les plus courantes. Exemples :

```
public class Car {
  private final Engine engine;

  public Car (Engine engine) {
    this.engine = engine;
  }
}
```

```
public class Car {
  private final Engine engine;

  public Car (EngineType engineType) {
    this.engine = EngineFactory.getEngine(engineType);
  }
}
```

Liskov substitution principle

« LES SOUS-TYPES DOIVENT POUVOIR ÊTRE SUBSTITUÉS À LEURS TYPES DE BASE » (SUBTYPES MUST BE SUBSTITUABLE FOR THEIR BASE TYPES)

Le principe de substitution de Liskov est une définition particulière de la notion de sous-types. Selon ce principe les sous classes doivent pouvoir être substituées à leur classe de base sans altérer le comportement de ses utilisateurs. Dit autrement, un utilisateur de la classe de base doit pouvoir continuer de fonctionner correctement si une classe dérivée de la classe principale lui est fournie à la place. Une autre manière de résumer ce principe pourrait être : si B sous type de A alors tout objet de type A peut être remplacé par tout objet de type B sans altérer les propriétés désirables du programme. Les cas de violation du principe de Liskov ne sont pas si fréquents en réalité et nous concevons en général des modèles qui ne violent pas ce principe. Cela signifie, entre autre, qu'il ne faut pas lever d'exception imprévue (comme UnsupportedOperationException par exemple), ou modifier les valeurs des attributs de la classe principale d'une manière inadaptée, ne respectant pas le contrat défini par la méthode. Cependant, cela peut se produire par précipitation ou méconnaissance des détails d'implémentation des classes de base, et sa détection est généralement difficile. Il faut donc toujours être vigilant quand on étend une classe. Exemple :

```
public class User {
  private String emailAddress;

  public String getEmailAddress() {
    return this.emailAddress;
  }
}

public class AnonymousUser extends User {
  public String getEmailAddress() {
    throw new RuntimeException("Anonymous users don't have an email address.");
  }
}
```

Ici, le fait de lancer une exception à l'appel d'une méthode héritée est une violation du principe. Il va de même pour les méthodes héritées d'une interface ou d'une classe abstraite qui ne ferait que lancer une exception. En général, c'est un signe que le modèle de données est mal pensé. Dans notre exemple il est vrai qu'un utilisateur anonyme est un utilisateur, mais de ce fait un utilisateur anonyme n'a pas d'adresse mail. Donc tous les types d'utilisateurs n'ont pas d'adresse mail, et donc la classe parente User ne doit pas avoir de champ pour cela. Voici comment régler ce problème :

```
public abstract class User {

}

public class RegisteredUser extends User{
  private String emailAddress;

  public String getEmailAddress() {
    return this.emailAddress;
  }
}

public class AnonymousUser extends User {}
```

Interface segregation principle

« LES UTILISATEURS NE DOIVENT PAS ÊTRE OBLIGÉS DE DÉPENDRE DE MÉTHODES QU'ILS N'UTILISENT PAS » (CLIENTS SHOULD NOT BE FORCED TO DEPEND ON METHODS THAT THEY DO NOT USE)

Le but de ce principe est d'utiliser les interfaces pour définir des contrats, des ensembles de fonctionnalités répondant à un besoin fonctionnel, plutôt que de se contenter d'apporter de l'abstraction à nos classes. Il en découle une réduction du couplage, les clients dépendant uniquement des services qu'ils utilisent. En d'autres termes, le couplage est un principe orienté-objet qui se quantifie en déterminant les cas d'utilisation des classes. On dit de deux classes qu'elles ont un couplage fort lorsqu'elles s'utilisent. Afin de diminuer le couplage, il est possible de dépendre d'une abstraction, cachant ainsi l'implémentation à l'appelant. Le problème avec l'ISP, c'est lorsqu'une interface ou une classe abstraite devient trop volumineuse et expose trop d'information via son API. Une mise en garde cependant : un des travers de ce principe peut être de multiplier les interfaces. En poussant cette idée à l'extrême, nous pouvons imaginer une interface avec une méthode par client. Bien entendu, l'expérience, le pragmatisme et le bon sens sont nos meilleurs alliés dans ce domaine. Dans l'exemple ci-dessous, la classe Vehicle a deux comportements. Nous pouvons démarrer la voiture ou allumer les phares.

```java
public class Vehicle {
  public void startEngine() {
    System.out.println("Engine started");
  }
  public void turnLightsOn() {
    System.out.println("Lights turned on");
  }
}
```

En supposant qu'une deuxième classe ait la responsabilité d'allumer les phares du véhicule :

```java
class VehicleLightTurner
{
  public Vehicle vehicle;

  public void turnOnTheLights() {
    vehicle.turnLightsOn();
  }
}
```

Afin de respecter au maximum l'ISP, il faut se demander pourquoi une classe responsable d'allumer les phares devrait savoir qu'un véhicule peut être démarré. On peut alors extraire une interface afin que notre classe ne connaisse que ce dont elle a besoin. Exemple :

```java
public interface VehicleWithLights {
  void turnLightsOn();
}

public class Vehicle implements VehicleWithLights {
  public void startEngine() {
    System.out.println("Engine started");
  }
  public void turnLightsOn() {
    System.out.println("Lights turned on");
  }
}

class VehicleLightTurner {
  public VehicleWithLights vehicle;

  public void turnOnTheLights() {
    vehicle.turnLightsOn();
  }
}
```

Dependency inversion principle

Ce principe fait référence à une forme spécifique de découplage des modules logiciels. En suivant ce principe, la relation de dépendance conventionnelle que les modules de haut niveau ont, par rapport aux modules de bas niveau, est inversée dans le but de rendre les premiers indépendants des seconds. Le terme inversion est le plus important de ce principe.
Le DIP comporte deux règles :

1. Les modules de haut niveau ne doivent pas dépendre des modules de bas niveau. Les deux doivent dépendre d'abstractions.
2. Les abstractions ne doivent pas dépendre des détails. Les détails doivent dépendre des abstractions.

Autrement dit, nous allons devoir utiliser des interfaces au lieu d'utiliser directement les implémentations. Et puisqu'une classe ne doit pas connaitre l'implémentation de ses dépendances nous allons devoir mettre en place l'injection de dépendances, qui est un patron de conception. L'injection de dépendances peut se faire de deux façons : via le constructeur ou via les setters. Et si vous avez bien suivi ce cours jusque là, il n'est pas nécessaire d'indiquer que l'injection de dépendance doit se faire via le constructeur et non par un setter. Supposons que nous ayons une classe dont la responsabilité est d'écrire des lois dans la console, et qu'un service ait besoin de cette classe :

```
class Logger {
  public void log(String msg) {
    System.out.println(msg);
  }
}

class MyService {
  private Logger logger;

  public SomeService() {
    this.logger = new Logger();
  }

  public void myMethod() {
    this.logger.log("Hi!");
  }
}
```

Dans cet exemple, nos classes Logger et MyService sont fortement couplées. Cela est dû aux constructions de la classe MyService qui construit une instance de la classe Logger. Mais comment faire si nous devons logger ailleurs que dans la console ? Et bien cela ne se passera pas bien, car il faudra remanier tout le code ou la classe Logger est utilisée… Cette façon de faire n'est donc pas du tout en adéquation avec le clean code.

Pour régler ce problème nous allons donc utiliser une interface, comme ceci :

```
interface Logger {
  void log(String msg);
}

class ConsoleLogger implements Logger {
  public void log(String msg) {
    System.out.println(msg);
  }
}

class SomeService {
  private Logger logger;

  public SomeService() {
    this.logger = new ConsoleLogger();
  }

  public void someMethod() {
    this.logger.log("Hi!");
  }
}
```

Malheureusement même avec cette conception, nos deux classes restent bien trop fortement couplées, car la construction du logger se passe toujours dans le constructeur de notre service.

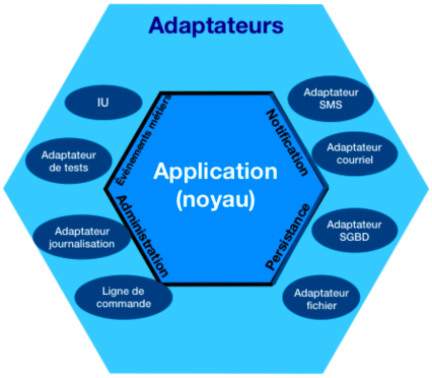
Nous allons donc avoir besoin d'injecter un Logger au moment de créer une instance de notre service :

```
interface Logger {
  void log(String msg);
}

class ConsoleLogger implements Logger {
  public void log(String msg) {
    System.out.println(msg);
  }
}

class MyService {
  private Logger logger;

  public MyService(Logger logger) {
    this.logger = logger;
  }

  public void myMethod() {
    this.logger.log("Hi!");
  }
}
```

Et nous voilà avec un code non couplé, grâce auquel notre classe MyService n'a aucune connaissance de l'implémentation qui se cache derrière l'interface Logger. On peut donc substituer le Logger par n'importe laquelle de ses implémentations sans casser le fonctionnement de MyService ni en modifier le code. L'injection de dépendance est une notion fondamentale en programmation orientée objet en cela qu'elle représente une façon de penser son code très puissante. Cependant c'est un concept qui peut paraître obscur et il faut donc faire attention à bien la mettre en place et la comprendre, surtout si l'on rajoute des frameworks comme Spring dans notre code.

Chapitre 10 - L'architecture hexagonale

L'architecture Hexagonale (Archi Hexa) a été documentée pour la première fois par Alistair Cockburn. Utilisée jusqu'en 2008, elle a perdu de sa notoriété avant de revenir sur le devant de la scène en 2015 avec l'avènement des micro-services. En effet, cette architecture (et ses dérivées) est présente absolument partout ces derniers temps. Et c'est à mon avis une très bonne chose. Elle cherche à nous éviter une chose, les pièges courants de la programmation orientée objet. Son nom vient de la représentation que nous en faisons sous forme d'hexagone mais elle est également utilisée sous le nom d'architecture Ports and Adapters.



Trois grands principes

L'architecture hexagonale nous dicte trois grand principes :

1. **Séparer les responsabilités :** en isolant le code métier. Le code métier est ce qui fait toute la logique d'un programme, ce qui a le plus de valeur, car c'est lui qui répond au besoin fonctionnel.
2. **Les dépendances vont vers la logique métier :** le code métier est agnostique au code technique. On va ainsi éviter au maximum d'intégrer des frameworks dans le code métier car ce sont les adapters, qui comportent tout le code technique, qui vont les intégrer.
3. **Isoler avec des ports et des adapters :** dépendre d'interfaces plutôt que d'implémentations. On limite ainsi le couplage entre nos classes et surtout entre le code métier et le code technique.

Pour résumer, nous cherchons grâce à cette architecture, à créer des composants logiciel faiblement couplés tout en mettant le code métier à l'honneur. La technique se retrouve ainsi réellement au service du métier.

Des composants logiciels faiblement couplés

Si vous avez bien suivi le chapitre précédent, le titre de cette partie devrait normalement vous rappeler quelques chose, principalement grâce au mot couplage. En effet, les principes SOLID visent à permettre aux développeurs d'écrire un code avec le minimum de couplage possible. Et bien l'architecture hexagonale est une architecture logicielle qui permet de mettre en place les principes SOLID de manière simple, dans un cadre strict qui empêche toute transgression aux principes SOLID, ou plutôt qui encourage leur application. Limiter au maximum le couplage permet d'avoir une maintenabilité accrue ainsi qu'une évolutivité préservée à bien plus long terme. Mais on y gagne également un code plus simple, plus maintenable, et aussi bien plus facilement testable.

Isoler le code métier

Le fait d'isoler le code métier consiste a rendre le code métier agnostique à la plupart des frameworks non essentiels à l'écriture du code métier, ainsi que de le tester de manière isolée. Mais cela n'implique pas l'absence totale de librairies tierces, nous allons simplement nous restreindre à l'essentiel. Grâce à tout cela, notre code métier sera bien plus léger et les tests plus simples mais également plus rapides à s'exécuter.

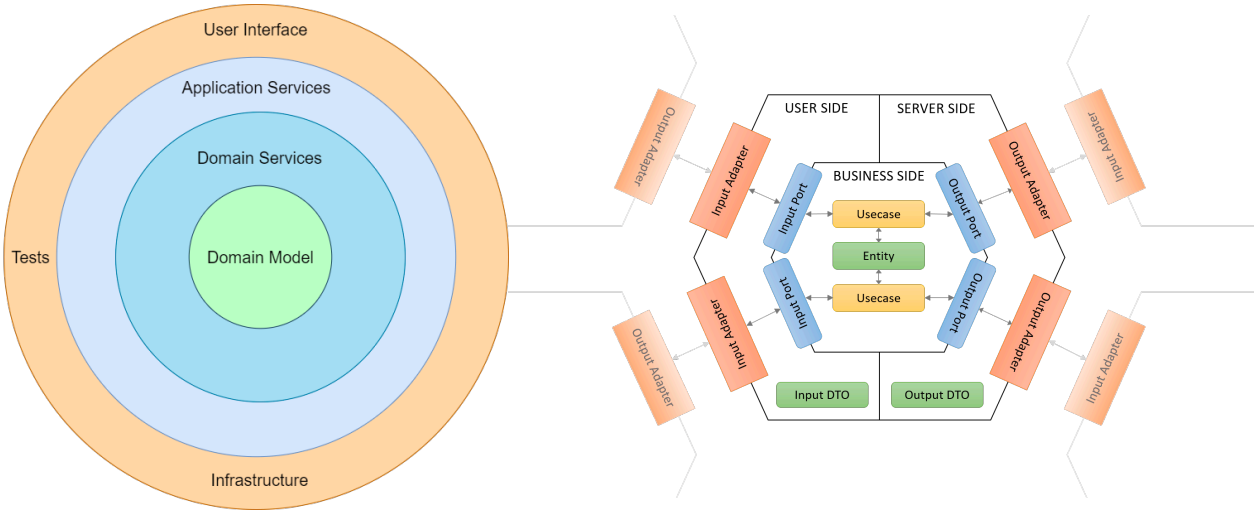
Quelques précisions sur cette architecture :

- **Elle demande énormément de rigueur :** c'est une architecture qui peut paraître assez complexe, voire obscure si on ne maitrise pas bien les principes SOLID et la programmation orientée objet, même si nous n'avons que quelques règles à respecter. Cela est dû au fait que les règles de l'archi hexa limitent nos possibilités (en bien) pour nous « imposer » une conception propre de notre code, contrairement à une architecture plus classique comme l'architecture en couches par exemple. En revanche, cette architecture apporte un bienfait incroyable, la réduction de la résistance aux changements.
- **Plusieurs solutions d'implémentation :** on peut implémenter une architecture hexagonale de deux grandes manières : en modules ou en packages. L'implémentation en packages est la plus simple et la plus rapide, mais rend difficile l'application précise des règles de l'architecture. L'implémentation en modules quant à elle, est plus lourde en terme de compilation, mais apporte la garantie de ne pas dépendre de ce dont le domain n'a pas besoin. Une implémentation en module est selon moi la meilleure solution.
- **Elle est testable :** étant donné que notre architecture impose des règles claires, il est possible de s'assurer du respect de ces règles par des tests unitaires avec des librairies comme ArchUnit en Java.
- **Apporte plusieurs avantages :** séparation des différents types de code (métier et technique), réduction du code spaghetti, un code maintenable et testable facilement le mettant réellement au service du métier. Et surtout, les problématiques techniques ne viennent plus impacter les problématiques métier. Enfin, elle permet un travail à plusieurs bien plus efficient et permet de créer de la valeur plus vite et de manière plus durable.
- **Le revers de la médaille :** cette architecture apporte deux problématiques principales. La première est l'augmentation importante du mapping d'objets entre les différents codes, ce qui représente un travail supplémentaire de la part des développeurs. Enfin, comme évoqué juste avant, la multiplication des modules entraîne une augmentation des temps de compilation.
- **Quand ne pas l'utiliser :** deux cas possibles :
  - Lorsque la logique métier est inexistante, comme dans une simple application CRUD par exemple.
  - Lorsque la logique métier est technique, comme lorsque l'on écrit un framework par exemple.

Enfin, il existe deux autres architectures qui sont similaires à l'architecture hexagonale :

- **L'architecture en oignon** de Jeffrey Palermo (2008)
- **L'architecture clean** de Uncle Bob (2012)

Mais en réalité, ces deux dernières ne font que reprendre les principes de l'architecture hexagonale en présentant les choses de manière un peu différentes ou alors en rajoutant un petit quelque chose. Donc si vous entendez parler d'hexagonale, d'oignon ou de clean, vous pouvez ne retenir que ceci : c'est globalement la même chose.



Exemple d'implémentation

https://github.com/Bad-Pop/hexagonal-architecture-in-java

Chapitre 11 - Pour aller plus loin

Les principes KISS & DRY

Les principes KISS (Keep It Simple Stupid) et DRY (Don't Repeat Yourself) sont des notions implicites au clean code. En suivant les différents préceptes abordés durant ce cours, nous appliquons sans forcément le savoir ces principes.

Dans le cas de KISS, nous cherchons en effet a garder le code le plus simple possible, afin de bénéficier de tous les avantages évoqués durant ce cours quant à la simplicité du code.
Dans le cas de DRY nous cherchons simplement à profiter des bénéfices d'un code sans couplage fort afin de pouvoir réutiliser la majeure partie du code écrit.

La loi Déméter

La loi Déméter ou principe de connaissance minimale, est une règle de conception pour développer un programme, particulièrement adapté à la programmation orientée objet. Elle a été inventée en 1987 à Northeastern University et peut être résumée en : ne parlez qu'à vos amis immédiats. La notion fondamentale est qu'un objet devrait faire aussi peu d'hypothèses que possible à propos de la structure de quoi que ce soit d'autre, y compris ses propres sous-composants.

Appliquée à la programmation orientée objet, la loi de Déméter peut être appelée plus précisément la **loi de Déméter pour les fonctions et les méthodes**. Dans ce cas, un objet A peut requérir un service (appeler une méthode) d'un objet B, mais A ne peut pas utiliser B pour accéder à un troisième objet et requérir ses services. Faire cela signifierait que A a une connaissance plus grande que nécessaire de la structure interne de B. Au lieu de cela, B pourrait être modifié si nécessaire pour que A puisse faire la requête directement à B, et B propagera la requête au composant ou sous-composant approprié. Si la loi est appliquée, seul B connait sa propre structure interne.
Plus formellement, la Loi de Déméter pour les fonctions requiert que toute méthode M d'un objet O peut simplement invoquer les méthodes des types suivants d'objets :

1. O lui-même
2. les paramètres de M
3. les objets que M crée/instancie
4. les objets membres de O

En particulier, un objet doit éviter d'invoquer des méthodes d'un objet retourné par une autre méthode.
L'avantage de suivre la règle de Déméter est que le logiciel est plus maintenable et plus adaptable. Puisque les objets sont moins dépendants de la structure interne des autres objets, ceux-ci peuvent être changés sans changer le code de leurs appelants. Un désavantage de la règle de Déméter est qu'elle requiert l'écriture d'un grand nombre de petites méthodes "wrapper" pour propager les appels de méthodes à leurs composants. Cela peut augmenter le temps de développement initial, et augmenter l'espace mémoire utilisé.
La loi de Déméter trouve donc tout particulièrement sa place dans ce cours sur le clean code.