Clean code

Table des matières

Introduction

Un peu de contexte

Pour beaucoup de développeur.se.s, la notion de clean code est assez floue et semble être d'une importance secondaire. Nombreux sont ceux qui privilégient le "quick and dirty" pour répondre à la demande du client qui attend toujours plus de fonctionnalités, toujours plus rapidement. Il est difficile de ne pas succomber aux solutions court-termistes face à la pression constante pour faire évoluer le produit et augmenter sa rentabilité. Caricaturalement, il est bien plus rapide de construire une cabane en carton qu'une cabane en bois ou en béton. Pour autant, il y a un coût à ce choix.

Les choix court-termistes permettent d'apporter une réponse rapide à une problématique mais ce ne sont pas nécessairement des solutions pérennes ou viables à l'échelle de la vie d'un projet. Dit autrement, le choix d'une solution faite en quelques heures plutôt qu'une solution nécessitant plusieurs jours de travail n'est pas toujours la meilleure option pour un projet durant plusieurs années. Ces décisions mènent souvent à du code fortement couplé, difficilement lisible et peu adaptable. Ainsi le problème n'apparaît pas immédiatement, mais deviendra flagrant et très impactant lors des futurs développements.

Cette notion d'impact a posteriori se comprend très bien lorsque l'on évoque la lecture du code. En moyenne, un.e développeur.se va lire 10 fois plus de code qu'il n'en écrit. Ce qui veut dire que la lisibilité du code est 10 fois plus importante que son écriture. Donc l'impact d'un code facile à écrire mais difficile à lire sera 10 fois plus grave qu'un code difficile à écrire mais facile à lire.

Le clean code est à la fois un cadre de développement et une rigueur que l'on s'applique à soi-même pour maximiser la qualité du code. Cela va permettre d'écrire le meilleur code qu'il nous est possible de faire à l'instant T en favorisant la lisibilité et la compréhension du code. De cette manière, les prochains développements seront plus simples et rapides.

Contenu du cours

L'objectif du cours est de fournir un ensemble de pratiques et outils essentiels permettant de favoriser l'application du clean code. Pour cela, nous aborderons le sujet par des définitions formelles et des bonnes pratiques de développement. Bien que simples, il est très facile d'oublier ces définitions et de bafouer les pratiques qui seront évoquées. Il est donc essentiel de les avoir toujours en tête et d'y apporter une attention constante.

Ensuite nous aborderons ensuite la stratégie de test et les bonnes pratiques associées. Le testing permet de garantir le bon fonctionnement du code à tout instant. Ainsi, c'est l'outil privilégié pour prévenir des régressions et favoriser la maintenabilité du code. Nous en profiterons pour nous pencher sur le Test Driven Development, pratique de développement mettant en avant les tests.

Puis nous discuterons des pratiques de développement en équipe. Certaines d'entre elles favorisent l'écriture d'un code de qualité et un fort partage de connaissances. Bien qu'ayant fait leurs preuves, elles sont souvent sous-estimées par les développeur.se.s et incomprises des non-développeur.se.s. Les comprendre permet de les appliquer et de les transmettre pour en propager l'usage.

Enfin, nous nous pencherons sur des problématiques architecturales au travers des principes SOLID, du Domain Driven Development et de l'architecture Hexagonale. Cela nous permettra d'aborder le clean code à l'échelle d'un projet et d'appliquer le clean code dans le design même de l'architecture logicielle.

Chapitre 1 - Définitions

Le clean code est une notion assez difficile à cerner. Chaque développeur.se aura sa propre définition de part ses expériences et ses influences. Ainsi, pour commencer à comprendre ce que représente le clean code, le “code propre”, il est intéressant de se poser la question de ce que n’est pas le clean code. Parlons donc du “mauvais” code. Il est fréquent d’entendre plusieurs termes à ce propos, dont les plus récurrents sont “code legacy” et “dette technique”.

Code legacy

De manière très formelle, nous pourrions traduire cette expression par la formulation “code historique”. En effet, legacy signifie héritage et dans notre cas nous parlons du code qui a été écrit par le passé et que nous recevons de nos prédécesseurs au travers des outils de versionning.

À ce moment, un biais de compréhension a souvent lieu sur le terme “prédécesseur”. On visualise souvent les personnes qui étaient sur le projet avant nous et qui sont parties depuis un moment. Ils font effectivement partie de ces prédécesseurs, mais nous ne parlons pas que de ces personnes. Nous parlons ici de toute personne ayant travaillé sur un morceau de code, cela peut donc être le “vous” du passé. Il n’est en effet pas rare de s’énerver contre l’auteur d’un code que l’on ne comprend pas pour finir par se rendre compte que nous l’avons nous-même écrit il y a quelques mois.

Maintenant que nous avons la bonne signification du terme “prédécesseur”, on se rend vite compte que du code devient legacy dès lors qu’il est versionné. Ainsi, tout code écrit, aussi récent soit-il, est du code legacy. C’est le code que l’on hérite du passé ou que l’on va transmettre aux futures personnes travaillant sur le sujet. Malgré tout, cette expression est souvent utilisée pour décrire le code historique difficile à maintenir ou à comprendre. Dès lors, l’usage de l’expression “dette technique” est souvent plus approprié.

Dette technique

La dette technique regroupe 2 types de dettes. Celle qui apparaît naturellement et celle que l’on contracte sciemment. Mais que ce soit l’une ou l’autre, à l’instar d’une dette bancaire, il y a des intérêts qui rendent le remboursement de cette dette de plus en plus cher au fil du temps.

La dette “naturelle” ne peut pas être évitée. Elle provient de l'obsolescence des technologies que l’on utilise. Le monde de l’informatique et du développement est toujours à la recherche du progrès. Ainsi, toute technologie finit par devenir obsolète et doit être mise à jour ou remplacée. Par exemple, même si aujourd’hui le Fortran ou le Pascal sont des langages décriés, ils ont été à la pointe de la technologie en leur temps. Pour contenir et rembourser ce type de dette, il faut régulièrement mettre à jour les technologies que l’on utilise afin de garantir la plus faible obsolescence possible.

La dette que l'on contracte sciemment peut être évitée, mais elle est beaucoup plus difficile à contenir et ses impacts peuvent être dramatiques. Ce genre de dette est souvent issu d'un compromis ou d'une contrainte. Par exemple, il faut développer une nouvelle fonctionnalité mais pour avoir une architecture scalable, il faudrait refondre une grosse partie du code. On fait donc un compromis entre la quantité de refonte et la contrainte de temps. Elle peut également venir d'un manque de validation par les pairs. Une personne peu expérimentée va avoir tendance à produire du code moins maintenable et si elle n'est pas accompagnée, la dette ainsi générée peut s'accumuler.

Cette dette devient rapidement très coûteuse si elle n'est pas gérée voire résorbée. Le développement de nouvelles fonctionnalités devient lui aussi très coûteux, voire impossible, car le code est fortement couplé ou difficilement modifiable. Aussi, du code endetté aura tendance à générer des bugs ce qui va nuire à la qualité de l'application, donc à la satisfaction client et, in fine, à la rentabilité de l'application. Mais le coût de gestion ou de résorption de cette dette est également important pour les mêmes raisons. Ainsi, il est compliqué de déconstruire l'existant pour le refondre de manière plus maintenable.

Par conséquent, le plus simple est de contracter le moins de dette technique possible et ce grâce au clean code.

Clean code

Le clean code est une expression permettant de décrire l'ensemble des pratiques, outils et designs permettant de produire du code qui sera lisible, maintenable et robuste. Cette expression peut aussi servir à décrire le code écrit en suivant les pratiques et designs évoqués ci-dessus.

De nombreuses pratiques tombent dans le périmètre du clean code et en faire une liste exhaustive serait impossible. Cependant, elles se regroupent toutes autour de quelques principes clés :

- **Simplicité :** il est important de systématiquement s'assurer que le code que l'on produit est le plus simple pour répondre à la demande. Plus le code est simple, plus il sera facile à comprendre, adapter et modifier. Pour cela, 2 expressions nous guident dans la bonne direction : KISS (Keep It Stupid Simple) et YAGNI (You Ain't Gonna Need It). Garder le code stupidement simple en ne faisant rien de superflu.
- **Spécificité :** Comme le dit Robert C. Martin dans son ouvrage "Clean Code: A Handbook of Agile Software Craftsmanship" :
  « Functions should do one thing. They should do it well. They should do it only »
  Dit autrement, un code ne doit avoir qu'un seul et unique objectif. Nous en reparlerons avec les principes SOLID.
- **Testabilité :** Le code que vous écrivez doit être testé afin de valider et garantir son comportement. Cela a un fort impact sur la maintenabilité car un code testé a de très faibles risques de subir une régression. En cas de changement, si une erreur est introduite, au moins un test sera invalide.

Respecter ces critères n'est pas sans coût. En effet, produire un code de qualité nécessite du temps et du travail. Mais, comme évoqué en introduction, la lecture du code représente près de 90% de votre travail en tant que développeur.se. Ainsi, même si vous passez plus de temps à écrire votre code, les gains sur le long terme de par la simplicité de lecture et de par la maintenabilité seront 10 fois plus important. En somme, le clean code revient à l'expression "reculer pour mieux sauter", c'est-à-dire prendre le temps d'aller vite en faisant de la qualité en continu pour ne pas avoir à gérer de dette technique plus tard.

Chapitre 2 - Le nommage

Comme évoqué dans la définition du clean code, en tant que développeur.se.s nous passons le plus clair de notre temps à lire et comprendre du code. Partant de ce constat, il faut écrire du code pour l'humain et non pour la machine.

À l'image d'un langage verbal, un langage de programmation a pour vocation de permettre à ses locuteurs de se comprendre entre eux. Ainsi, nous parlons bien d'un outil de communication entre les développeur.se.s et non avec la machine. Caricaturalement, si nous ne cherchions à communiquer qu'avec la machine, nous ne ferions que de l'Assembleur et non un langage de haut niveau comme le Java qui ajoute de nombreuses surcouches.

Afin que les locuteurs de notre langage, les développeur.se.s, puissent se comprendre, ils ont besoin d'un vocabulaire commun. Ce vocabulaire se présente sous différentes formes, par exemple les mots clés du langage (*if, for, class,* etc …).

Dans ce chapitre nous allons nous intéresser au nommage des variables, soit les noms de notre langage. À l'instar du français, ces noms doivent suivre un certain nombre de règles et de conventions pour que tous les développeur.se.s puissent les comprendre. Grâce à ces règles, notre code pourra être expressif et révélateur de ce que nous sommes en train de manipuler.

Nommage qui révèle l'intention

Comme nous venons de l'expliquer, l'objectif du code est d'expliquer au prochain.e développeur.se ce que fait notre code. Un code incompréhensible est un code qu'on ne peut pas modifier. Il est donc essentiel que ce que nous écrivons puisse être compris par n'importe quel.le développeur.se et que notre code explicite son comportement ainsi que son intention.

Pour cela, nous devons trouver un nommage porteur de sens. C'est probablement l'une des tâches les plus compliquées du développement : trouver le bon nom. La première étape est de disposer d'un vocabulaire métier commun avec les autres développeur.se.s, nous aborderons cela plus en détail dans le *Chapitre 7 - Domain Driven Development*. Ensuite, il faut être sûr de ce que l'on cherche à faire. Si nous ne sommes pas au clair sur notre objectif et les données que nous manipulons, il sera très compliqué de trouver le nommage adéquat.

Prenons l'exemple suivant :

```
private final ZonedDateTime date1;
private final ZonedDateTime date2;
```

De quelles dates s'agit-il ? Le nom *date1* et *date2* ne révèlent absolument pas la signification de la variable, ce qu'elle représente. Le lecteur doit soit se référer à une

documentation extérieure soit faire l'effort de trouver les usages pour essayer de comprendre ce qu'il en est. Nous pouvons améliorer notre exemple comme suit :

```
private final ZonedDateTime creationDate;
private final ZonedDateTime lastUpdateDate;
```

Ici, aucun doute ne persiste. La première date représente la date de création de l'objet et la seconde représente la date de dernière mise à jour.

Éviter la désinformation

Afin de garantir la clarté de notre code, il faut éviter toute ambiguïté dans notre nommage. Les mots déjà chargés sémantiquement, trop génériques ou trop proches de mots déjà présents dans le langage sont donc à bannir. Autrement, on prend le risque que notre lecteur interprète ces mots avec un sens différent du nôtre.

En ce qui concerne la charge sémantique, il faut être attentif à ne pas tomber dans la facilité des mots qui nous viennent spontanément. Ces mots sont souvent employés très régulièrement au quotidien et peuvent avoir une signification légèrement différente de notre besoin du moment. Par exemple, le mot *Factory* peut être adapté à notre besoin mais il est déjà fortement chargé sémantiquement. Tout le monde pensera au design pattern Factory, ce qui n'est pas nécessairement ce que nous sommes actuellement en train de faire. À l'inverse, des mots comme *data* ou *item* n'ont absolument aucune charge sémantique, même lorsqu'ils sont contextualisés. Cela revient à dire *donnée* ou *élément* ce qui ne nous aide pas à savoir de quoi il s'agit.

Enfin, les mots trop proches de l'implémentation rendent notre nommage très fortement couplé à notre implémentation et, en cas de changement, nous nous retrouvons avec un nommage incohérent. Par exemple :

```
private final Set userList;
```

Ici, la *userList* n'est pas une List mais un Set. Même si le comportement est proche, il existe des différences non négligeables entre ces deux structures de données. On arrive donc dans le cas où notre nommage nous induit en erreur. Dans un tel cas, il aurait été préférable d'appeler notre groupement d'utilisateur simplement *users* : le nommage est indépendant de l'implémentation et ne risque plus de nous amener sur la mauvaise piste.

Éviter les littéraux

Un littéral est une valeur en dur dans le code qui est utilisée telle quelle. Pour les décrire, on entend souvent parler de "Magic String". Même si ces valeurs peuvent permettre de faire fonctionner notre code, elles ne fournissent aucun indice sur leur signification ou sur le pourquoi elles sont utilisées à cet endroit. Par exemple :

```
new Player(5, 12);
```

Ici, nous n'avons aucune idée de ce que signifie le 5 ou le 12. Pour résoudre le problème, nous pouvons utiliser des constantes ou des énumérations afin de donner une signification à ces valeurs qui semblent arbitraires :

```
new Player(INITIAL_MOVE_POINTS, INITIAL_HP_MAX);
```

De cette manière, les valeurs gagnent en signification et cela facilite le travail en cas de changement de ces valeurs.

Éviter les schémas mentaux

Un schéma mental est une association entre un élément et sa signification. Cela arrive fréquemment lorsqu'une variable n'est nommée que par une unique lettre. Par exemple :

```
notifications.stream()
    .forEach(n -> {/* do things */})
```

Ici, à la lecture du *forEach*, on comprend que la variable *n* représente une notification. Mais dès lors que l'on lira l'implémentation contenu dans le *forEach* nous nous retrouvons obligés de nous souvenir que *n* représente une notification. Ce problème est simplement résolu en nommant les variables dans leur intégralité, c'est-à-dire sans utiliser d'abréviation. Dans notre exemple, remplacer *n* par *notification* résout le problème.

Des noms prononçables

D'une manière générale, les développeur.se.s travaillent en équipe. Ainsi, ils.elles ont besoin de communiquer et d'échanger sur leur travail. Pour que cela puisse être fait de manière fluide, il est essentiel que les noms utilisés dans le code puissent être prononcés à l'oral. Autrement, comment pouvons-nous aller voir un collègue et lui demander de l'aide efficacement ? Par exemple :

```
private Integer hCrdnt;
```

En plus d'être difficilement compréhensible et de demander un schéma mental, cette variable est imprononçable à l'oral. Si l'on devait essayer cela donnerait "*hachecreudéneté*". Il est probable que si l'un de vos collègues vous demande de vous expliquer ce qu'est cette valeur en la nommant oralement vous soyez pris d'une certaine hilarité. Ainsi, pour fluidifier le travail à plusieurs, il faut éviter les abréviations et avoir des variables prononçables, telles que :

```
private Integer horizontalCoordinate;
```

Des noms recherchables

La très grosse majorité des applications modernes contiennent plusieurs centaines de fichiers, comportant eux-mêmes plusieurs dizaines voire centaines de lignes. Aussi bon qu'un.e développeur.se puisse être, cela fait trop pour le cerveau humain. Ainsi, que ce soit pour de la lecture ou pour des modifications, nous avons besoin de nous appuyer sur les IDE pour rechercher des informations dans cette masse de code. Le problème est que les IDE ne prennent pas en compte le contexte ou l'intention d'une recherche. De fait, les noms très récurrents de variables (ex.: *item*) ou très courts (ex.: *n*) sont très difficiles à localiser.

Prenons l'exemple de la variable *item*, c'est un nom qui revient très souvent lorsque l'on manipule un élément d'un groupe (ex.: itération). Si je devais rechercher "*item*" dans tout le code d'une application, la probabilité que l'ensemble des résultats correspondent à mon besoin est très faible. Inversement, si je cherche "*notification"* la probabilité de trouver des variables concernant les notifications est significativement plus élevée.

Quelques conventions

En plus de ces règles, il existe des conventions qui permettent de simplifier encore plus la lecture de code avec des standards partagés entre tou.te.s les développeur.se.s :

Les noms de variables sont des noms au singulier représentatifs de leur contenu écrit en lower camel case : *notification*.

Les noms de groupes de variables (ex.: List) sont des noms au pluriel représentatifs de leur contenu, écrits en lower camel case : *notifications*.

Les noms de variables booléennes commencent par un *is* : *isConnected*. Selon l'usage il est possible d'utiliser *has*, *should* ou *could* comme préfixe.

Les noms de fonctions sont des verbes représentatifs de l'action réalisée associés à un nom représentatif de la variable sur laquelle l'action est faite, écrits en lower camel case : *discardNotification*.

Les noms de classe sont des noms du vocabulaire métier représentatif de leur contenu, écrits en upper camel case : *Notification*.

Les noms de constantes sont des noms représentatifs de leur contenu, écrits en screaming snake case : *NOTIFICATION_TIMEOUT*.

Conclusion

Le nommage est un processus long et parfois laborieux. Cela demande d'avoir une bonne vision de ce que l'on cherche à faire et des standards d'équipe. Cela demande également de se poser plusieurs questions sur l'usage, la portée et la signification de ce que

nous sommes en train de nommer. Cependant, ce travail est nécessaire pour garantir un code lisible et abordable par n'importe qui.

Enfin, nous avons souvent peur de renommer les choses. Que ce soit par crainte de l'ampleur du travail ou par crainte des objections de nos collègues. Mais si vous trouvez un nom ambigu, imprononçable voire incompréhensible, soyez certain que vous n'êtes pas le premier et vous ne serez pas le dernier. Ainsi, prenez votre courage à deux mains et changez-le ! Cela rendra le code plus clair et plus compréhensible. Et même si votre changement n'est pas accepté, au moins vous aurez provoqué des discussions pour essayer de trouver un meilleur nom.

Chapitre 3 - Les fonctions

Maintenant que notre code a un nommage expressif permettant de comprendre la donnée que nous sommes en train de manipuler, il nous faut clarifier les traitements qui sont appliqués à cette donnée. Dans la très grosse majorité des cas, ce traitement est décrit dans l'implémentation d'une ou plusieurs fonctions. En continuant le parallèle avec le langage verbal, les fonctions sont nos verbes et leurs arguments en sont la conjugaison.

Cependant, l'usage d'un verbe ambigu ou mal conjugué peut porter à confusion voire provoquer un contresens. De la même manière, une fonction trop complexe ou mal utilisée peut rendre difficile l'interprétation du code et la lecture de son intention. Ainsi, il existe quelques règles et bonnes pratiques à respecter pour favoriser une simplicité de lecture.

La concision

À l'image d'une phrase remplie de subordonnés, une fonction trop longue est difficile à comprendre. De fait, elle sera compliquée à maintenir et à tester. En effet, plus une fonction est longue, plus le nombre d'actions réalisées est grand. Il est facile d'imaginer qu'une fonction faisant plusieurs centaines de lignes ne se contente pas de faire une simple addition. Cette taille induit trois problèmes : la charge mentale, la testabilité et la responsabilité multiple. La charge mentale est assez simple à deviner. Une fonction très longue demande de se souvenir de beaucoup d'informations pour comprendre son fonctionnement de la première à la dernière ligne. Ainsi, la prise en main d'une telle fonction demande beaucoup d'énergie et d'attention.

Plus une fonction est longue, plus elle fait d'actions. Plus elle fait d'actions, plus le nombre de cas possible est grand. Plus le nombre de cas possible est grand, plus le nombre de tests à réaliser pour valider le comportement de ladite fonction est grand. Cet argument peut sembler fallacieux car le nombre de cas possible devrait rester le même, que nous utilisions une fonction de 500 lignes ou 50 fonctions de 10 lignes. Cependant, il ne faut pas oublier les notions de combinatoire et de dépendance. Lorsque nous testons une fonction, il est nécessaire d'envisager la combinaison de tous les cas possibles afin de garantir une bonne couverture de test. Aussi, il est fréquent de simuler nos dépendances. Seulement, dans le cadre d'une fonction de 500 lignes, la combinatoire et le nombre de dépendances sera drastiquement plus élevé que pour des fonctions d'une dizaine de lignes. L'augmentation de la combinatoire peut faire apparaître des cas qui n'auraient pas été possibles avec des fonctions plus petites. En outre, la gestion des dépendances peut être très complexe de part les potentielles intéractions entre elles et la mise en place de jeu de données pour traiter tous nos cas. En réduisant la taille de nos fonctions, nous limitons directement le nombre de cas possible et la complexité des simulacres de dépendances. Ainsi, plusieurs petites fonctions seront significativement plus simples à tester qu'une grosse fonction faisant tout le travail.

Plus une fonction est longue, plus elle fait d'actions. Une fonction de seulement quelques dizaines de lignes peut rapidement faire 3 à 4 actions différentes. Elle est donc responsable de plusieurs comportements à la fois. C'est une violation directe du *Single Responsibility Principle*. Nous rentrerons dans le détail de ce principe au *Chapitre 6 - Les principes SOLID* mais nous pouvons dès maintenant en voir l'impact : la spécialisation et le couplage. Une fonction faisant plusieurs actions induit un couplage fort entre chacune d'elles. C'est-à-dire qu'une action donnée est co-dépendante des comportements des actions précédentes et suivantes. Dit autrement, cette action est implémentée de telle manière qu'elle ne peut pas exister par elle-même. Comme les actions ne peuvent pas être déclenchées de manière indépendante, notre fonction décrit un comportement complet mais très spécifique. Cette spécificité rend le code très "rigide" car chaque comportement est défini de manière extrêmement précise. Le problème est que cette rigidité ne permet pas la réutilisation du code dans le cadre de l'implémentation de nouveaux comportements. Ainsi, nous serons forcés de réimplémenter certaines parties de notre code pour créer de nouveaux comportements. Des fonctions plus petites ne faisant qu'une seule action deviennent comme des briques de Lego que nous pouvons assembler à notre convenance pour créer le comportement que nous souhaitons sans réimplémenter quoique ce soit. Cela rend le code plus modulaire et réutilisable.

Nous voyons donc qu'écrire des fonctions trop longues nuit fortement à la compréhension du code, sa testabilité et sa modularité. Ainsi, écrire des fonctions concises permet de faciliter la lecture, limiter le nombre de cas de test et favoriser la maintenabilité et la modularité du code. En d'autres termes, écrire des fonctions courtes permet une meilleure maintenabilité. Mais cela a un autre avantage que nous n'avons pas encore évoqué, la lecture du code appelant ces fonctions. Passer d'une fonction de 100 lignes à une fonction appelant d'autres fonctions faisant chacune une dizaine de lignes permet de s'abstraire des détails d'implémentation. Comme chacune des sous-fonctions aura un nom descriptif de l'action réalisée (cf.: *Chapitre 2 - Nommage*), lire la fonction appelante sera comme lire un roman où chaque action est décrite. Ainsi, la compréhension du comportement global ne se fait plus par la lecture des détails d'implémentation mais par la combinaison de plusieurs actions indépendantes.

Les arguments

Une fonction prend de 0 à X arguments, le nombre d'arguments d'une fonction s'appelle l'arité. Une fonction à 1 argument aura une arité de 1, une fonction à 5 arguments aura une arité de 5. Bien que rarement évoquée, l'arité est un facteur important dans la maintenabilité d'une fonction car cela influe directement sur le nombre de cas possible d'une fonction. Schématiquement, une fonction d'arité 0 ne peut avoir qu'un seul et unique comportement, nous n'avons donc besoin de tester qu'un seul cas. Si notre fonction passe à une arité de 1, alors il y aura N valeurs possibles pour cet argument, il faudra donc tester N cas. Pour une arité de 2, il faudra tester la combinatoire des 2 arguments soit N*N cas donc N^2 tests. Pour une arité de 3, N^3 tests. On constate ici que l'arité influe exponentiellement sur le nombre de cas possible à tester pour valider le comportement de notre fonction, tel que $N^{arité}$ cas. Ainsi, il est facile de comprendre qu'en matière d'arité, il faut appliquer le principe "less is more". Idéalement, une fonction ne dispose d'aucun argument, soit une arité de 0. De cette manière, il n'y a qu'un unique cas à tester. Seulement, dans la majorité des cas, il

est impossible de maintenir une arité à 0. Il faut donc trouver des astuces pour réduire au maximum le nombre d'arguments de notre fonction.

Il existe plusieurs types d'argument. Certains ne peuvent pas être transformés mais d'autres peuvent être utilisés de manière différente pour réduire l'arité de la fonction que nous sommes en train d'écrire. Le type d'argument le plus simple à modifier sont les flags. Les flags sont des arguments booléens permettant de transmettre un choix entre 2 possibilités. Par exemple :

```
compute(isTestEnv);
```

Ici, nous avons une fonction permettant de faire un traitement. Ce traitement est différent selon si l'environnement dans lequel il est déclenché est un environnement de test ou non. Autrement dit, nous avons besoin de faire 2 fois plus de tests. L'ensemble de nos tests de comportement pour le cas où *isTestEnv=true* et à nouveau pour le cas où *isTestEnv=false*. Pour obtenir une arité de 0, plutôt que le test sur le booléen soit fait dans l'implémentation de *compute*, faisons deux fonctions différentes. Ce sera à l'appelant de savoir laquelle utiliser :

```
if (isTestEnv) {
    compute();
} else {
    computeForTest();
}
```

Nous pouvons aussi supprimer un argument d'une fonction en utilisant les membres de la classe dans laquelle nous nous trouvons. Par exemple :

```
List<Tool> tools;

class Player {
    public boolean hasKey(List<Tool> tools) { /*...*/}
}

player.hasKey(tools)
```

Ici, la liste d'outils est stockée de manière indépendante du joueur. De fait, pour savoir si notre joueur dispose d'une clé, il est nécessaire de lui transmettre la liste d'outils en paramètre. Seulement, dans notre usage, la liste d'outils est indissociable du joueur. Nous pouvons donc déplacer cette variable en tant que membre de la classe *Player*. De cette

manière, la fonction *hasKey* a accès à la liste d'outils et n'a plus besoin de recevoir de paramètre.

```
class Player {
    private List<Tool> tools;
    public boolean hasKey() {/*...*/};
}

player.hasKey()
```

Pour finir, il est possible de réduire l'arité d'une fonction grâce aux arguments "regroupables". C'est-à-dire qu'il faut identifier les arguments représentant une même information. Par exemple :

```
Circle (int x, int y, float radius);
```

Ici, nous créons un cercle avec le centre positionné en *[x;y]* de rayon *radius*. On peut remarquer que les coordonnées *x* et *y* sont indissociables l'une de l'autre car il nous faut les deux informations pour positionner le centre du cercle. Aussi, ces deux informations sont en fait des parties constitutives d'une information plus globale : le centre du cercle. Ainsi, on peut regrouper ces deux arguments en un seul :

```
Circle (Point center, float radius);
```

En plus de réduire l'arité de la fonction, cela permet aussi de renforcer l'expressivité du code en passant de *x* et *y* à *center*.

Les fonctions pures

Une fonction pure est une fonction sans effet de bord. Ces fonctions sont plus compréhensibles et plus maintenables. Mais pour le comprendre, il faut déjà savoir ce qu'est un effet de bord.

On appelle effet de bord tout impact extérieur au scope de la fonction effectuant le traitement. Dit autrement, lorsqu'une fonction effectue un traitement qui modifie une valeur extérieure à cette même fonction, c'est un effet de bord. Ainsi, l'interaction avec une base de données, l'affichage d'informations dans le terminal ou la modification d'un argument sont des effets de bord. Les fonctions pures sont donc des fonctions prenant des arguments pour faire un traitement interne et retourner un résultat sans interagir avec quoique ce soit d'autre que leur propre valeur de scope.

Le problème des effets de bord est que les impacts sont difficiles à identifier et à détecter. Ainsi, une fonction qui semble isolée peut avoir un impact sur un traitement complètement différent car elle peut partager une valeur avec une autre fonction. Cela amène un risque très fort en cas de modification car l'impact d'un effet de bord est incertain. De fait, la suppression ou la modification non maîtrisée d'un effet de bord peut avoir des conséquences très lourdes sur l'ensemble de l'application. Cela amène également des difficultés supplémentaires dans le cadre des tests. En présence d'effet de bord, soit nous simulons la dépendance avec un mock soit nous devons instancier les consommateurs de cet effet de bord pour s'assurer du bon fonctionnement. Dans un tel cas, la solution du mock est souvent choisie. Seulement le mock retire la réelle implémentation.De fait, en cas de changement nous ne pouvons pas garantir que la consommation de l'effet de bord est correcte.

De part les conséquences et le risque que cela amène, nous pourrions nous dire que seules les fonctions pures sont acceptables. Cependant, même si cette conclusion est vraie en théorie, en pratique il n'est pas possible de se passer des effets de bord. L'interaction avec une base de données ou une interaction HTTP sont des effets de bord incontournables pour n'importe quelle application d'entreprise. Il nous faut donc faire un compromis pour avoir un maximum de fonctions pures et maîtriser au mieux les effets de bord. Pour cela, il faut isoler les comportements à effet de bord afin de concentrer cette responsabilité à un seul endroit. Par exemple :

```
public void updateEventsValidity(List<Event> events) {
    events.stream().forEach(event -> {
        if (!isEventValid(event)) {
            event.isValid = false;
            database.connect();
            database.updateEntity(event);
            database.disconnect();
        }
    })
}
```

Ici, on constate qu'il y a deux potentiels effets de bord : le changement de valeur du *isValid* et l'écriture en base de données. Ces changements sont explicitement inscrits dans les détails d'implémentation de la fonction. Dit autrement, on ne peut pas la tester en isolation et les comportements à effet de bord ne peuvent pas être testés indépendamment les uns des autres. Aussi, l'implémentation du stockage en base est définie localement donc si nous avons besoin de le faire à un autre endroit, l'implémentation sera à refaire et présentera peut être des différences qui amèneront à des bugs. Ainsi, pour simplifier la testabilité, la réutilisabilité et la fiabilité du code, nous allons isoler les effets de bord :

```java
public void updateEventsValidity(List<Event> events) {
    events.stream().forEach(event -> {
        if (!isEventValid(event)) {
            event.setIsValid(false);
            eventRepository.update(event);
        }
    })
}
```

Grâce à ces changements, les effets de bord et le détail d'implémentation associé sont isolés. De cette manière, il est possible de tester et ré-utiliser les comportements à effets de bord. Aussi, en cas de changement du comportement de la fonction, l'implémentation du changement de la valeur *isValid* et de la mise à jour de la base de données ne seront pas impactés. De fait, ces traitements à effet de bord auront un comportement consistant entre tous les usages. Enfin, et pour renforcer le fait qu'un effet de bord va avoir lieu, le nom de la fonction indique qu'une modification va être effectuée avec des mots clés comme *set*, *update* ou *save*.

Chapitre 4 - Le testing

Pourquoi tester ?

Toute application est susceptible de présenter des bugs. Ces bugs, bien que souvent mineurs, ne sont pas à négliger. En effet, chaque correction de bug consomme du temps de développement qui ne peut pas être investi sur le développement de nouvelle feature ou la résorption de la dette technique. Aussi, des bugs majeurs peuvent avoir un impact direct sur la satisfaction des utilisateurs. Ainsi, les bugs sont un réel coût pour l'entreprise. Il est donc essentiel d'en limiter la création au maximum. Les tests automatisés sont l'outil de choix pour cela.

Les tests automatisés sont une part importante du Clean Code. Cela permet de valider le comportement du code que l'on est en train d'écrire. Ainsi, nous nous assurons que le code fait bien ce qu'on attend de lui sans provoquer de bug. Cela permet également de favoriser la maintenabilité du code en garantissant qu'en cas de changement, le comportement reste le même. Si le comportement change, certains tests seront en erreur. Ainsi, les tests réduisent significativement le risque de régression et donc la création de nouveaux bugs. Pour finir, les tests décrivent le comportement attendu. Nous pouvons donc les voir comme une sorte de documentation de ce que fait chaque morceau de code testé. De cette manière, il est plus simple de comprendre pourquoi un code agit de telle ou telle façon.

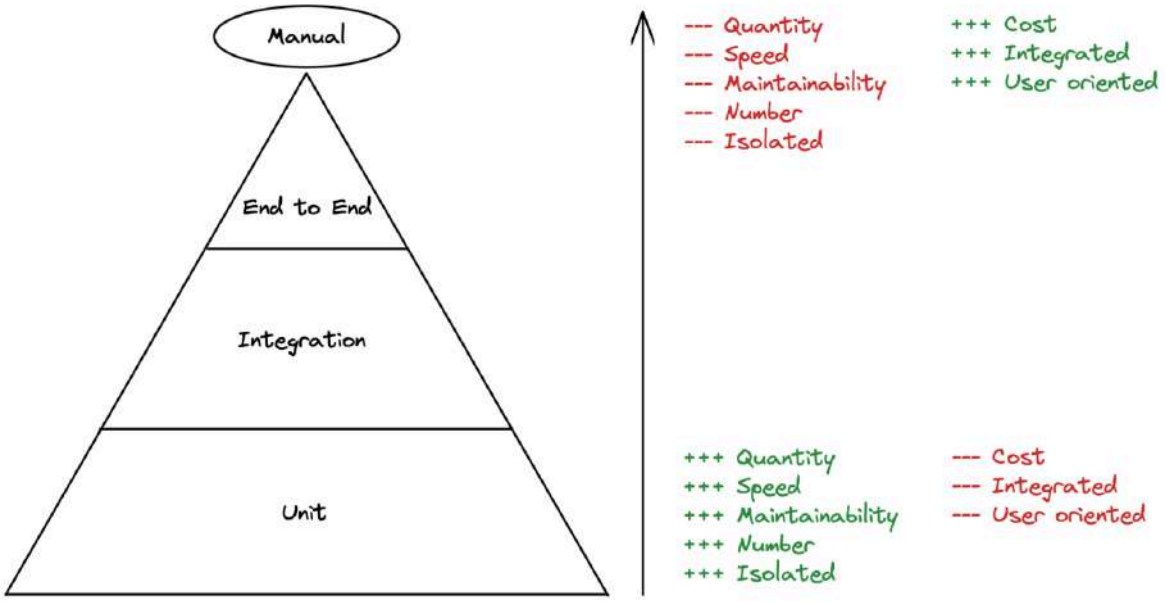
Mais l'écriture des tests peut être coûteuse à l'instant T, chaque test écrit limite la création de bug sur la totalité de la vie de l'application. Autrement dit, les tests sont un investissement sur le long terme. Le développement sera un peu ralenti à court terme, mais au fur et à mesure que l'application va grandir ils seront un excellent garde-fou contre les régressions. De fait, le temps passé sur les bugs sera restreint et la majorité des développements pourront être concentrés sur l'amélioration de l'application.

Pour quantifier l'efficacité de nos tests, il existe plusieurs métriques. La plus utilisée est la couverture de test. Elle représente le pourcentage de la base de code étant effectivement exécuté et validé par des tests. Schématiquement, une couverture de test de 80% indique que 80% de notre code et des cas possibles ont été testés automatiquement. D'une manière générale, on considère une couverture de test satisfaisante à partir de 70-75% et très bonne lorsqu'elle atteint les 90%. Cependant, il existe différentes typologies de test et pour chaque type de test on aura une attente différente quant à la couverture de test.

Pyramide de test - les différentes typologies

Les différents types de tests sont souvent représentés sous la forme d'une pyramide. La base de la pyramide représente les tests les plus rapides, isolés et nombreux mais aussi

les moins coûteux. Le haut de la pyramide représente les tests les plus lents, intégrés et coûteux.



Tests unitaires

Les tests unitaires sont les plus nombreux, les plus rapides et les moins coûteux. Ce sont eux qui vont représenter le plus gros volume de nos tests. Leur but est de tester en isolation les différentes briques de notre code. C'est-à-dire que chaque fonction devra disposer de son propre jeu de tests unitaires pour valider son comportement.

La notion d'isolation est assez importante ici, l'objectif est de tester nos fonctions indépendamment de leurs dépendances. C'est-à-dire qu'il va falloir simuler les appels aux éléments extérieurs à notre application (ex.: appel à la base de données, appel HTTP, etc…). Cela va permettre de garantir le comportement de notre application sans être soumis au bon fonctionnement d'éléments que nous ne pouvons pas contrôler. Nous verrons plus loin dans le chapitre comment gérer les dépendances.

Pour écrire des tests unitaires efficaces, il faut tester les comportements en boîte noire (ou boîte grise dans le cas de gestion de dépendances). C'est-à-dire que nous devons écrire nos tests comme si nous ne connaissions pas les détails d'implémentation de notre système. En cas de dépendances, il ne faut prendre en compte que les détails d'implémentation directement liés à cette dépendance. Dit autrement, la seule chose que nous contrôlons sont les arguments de notre fonction et la seule chose que nous pouvons vérifier est son résultat. De cette manière, on s'assure de tester le comportement et pas l'implémentation. Ce qui veut dire que quelle que soit l'implémentation, si le comportement reste le même, les tests doivent rester valides.

Une fois la mécanique de "boîte noire" appliquée, il faut se poser la question des cas à tester. La réponse est assez simple, il faut tester tous les cas. C'est-à-dire que pour

garantir une bonne couverture de test, il faut que chaque cas dispose d'un test dédié, même si le cas testé n'est pas censé arriver.

Tests d'intégration

Les tests d'intégration sont relativement nombreux et rapides mais, à l'inverse des tests unitaires, ils n'ont pas pour vocation de tester la totalité des cas possibles. Le but de ces tests est de valider l'interaction entre 2 briques techniques. Par exemple, un test d'intégration va s'assurer que le contrat d'interface de notre API HTTP est bien respecté et que notre serveur va correctement répondre. Ainsi, nous n'avons pas besoin de tester l'intégralité des cas. Il est plus intéressant de tester les "types" de cas : succès, échecs.

Le cas d'une API HTTP est assez simple à comprendre. Cependant, les tests d'intégrations peuvent tester l'interaction entre deux modules d'une même application. À ce moment, la frontière entre le test d'intégration et le test unitaire devient assez floue. Il n'y a malheureusement pas de bonne réponse car la notion d'isolation et de "unitaire" peut légèrement varier d'une personne à l'autre. Ainsi, il convient de faire la différence sur ce que l'on cherche à vérifier plutôt que la manière dont on s'y prend. En d'autres termes, si l'objectif du test est de vérifier l'intégralité des cas possibles, nous sommes dans le cas de tests unitaires. Au contraire, si l'objectif du test est de s'assurer que le code consomme correctement une dépendance, nous sommes dans le cas de tests d'intégration.

Tests de bout en bout

Les tests de bout en bout, ou end-to-end, sont relativement peu nombreux et longs à exécuter. Leur objectif est de simuler le comportement d'un utilisateur en étant exécutés sur une application entièrement instanciée (frontend + backend + base de données). Par exemple, dans le cadre d'une application web, un test end-to-end interagit avec le navigateur comme un utilisateur en cliquant sur les éléments de l'interface. Cela permet de tester des parcours utilisateur complets et de valider le comportement intégral de l'application.

Bien que très efficaces pour détecter les régressions, ces tests sont très longs à exécuter et coûteux à maintenir. La lenteur d'exécution se répercute directement sur la vitesse de développement. Ainsi, un workflow demandant l'exécution trop fréquente de ce type de tests amène un ralentissement de l'équipe. En ce qui concerne le coût de maintenance, le problème vient du fait que ces tests sont "trop" efficaces pour détecter les régressions. C'est-à-dire que des changements mineurs peuvent amener à l'échec d'un test. Dès lors, il est nécessaire de mettre à jour ces tests très régulièrement. Comme ils représentent des scénarios complets, la maintenance est assez lente car le volume de code pour un test est assez grand et le temps d'exécution est long.

Pour limiter cette contrepartie, il convient de n'écrire qu'un nombre limité de ces tests. C'est-à-dire que nous n'allons écrire de tests end-to-end que pour les cas passants des scénarios critiques et importants de notre application. Il est envisageable de tester des scénarios secondaires mais il est important d'avoir en tête le rapport coût / sécurité que ces tests nous apportent afin de ne pas se retrouver noyés par leur maintenance.

Tests manuels

Ces tests sont les plus coûteux et les plus lents. En effet, ils doivent être exécutés par des humains autant de fois que nécessaire. Comparé à une machine, l'humain sera toujours plus lent et plus cher. Cependant, ces tests ont un intérêt dans le cadre de tests exploratoires.

Les tests exploratoires sont des tests réalisés manuellement sur des scénarios que l'on ne peut pas ou ne veut pas automatiser. Par exemple, des clics aléatoires sur la machine ou les cas en erreurs ne peuvent pas ou ne doivent pas être implémentés dans des tests end-to-end. Ces tests sont idéaux lors de la création d'une nouvelle fonctionnalité pour s'assurer qu'elle n'apporte pas de comportement inconsistant. Il est également possible de répéter ce genre de test afin de garantir la robustesse de l'application face à des comportements imprévus.

Stratégie de test

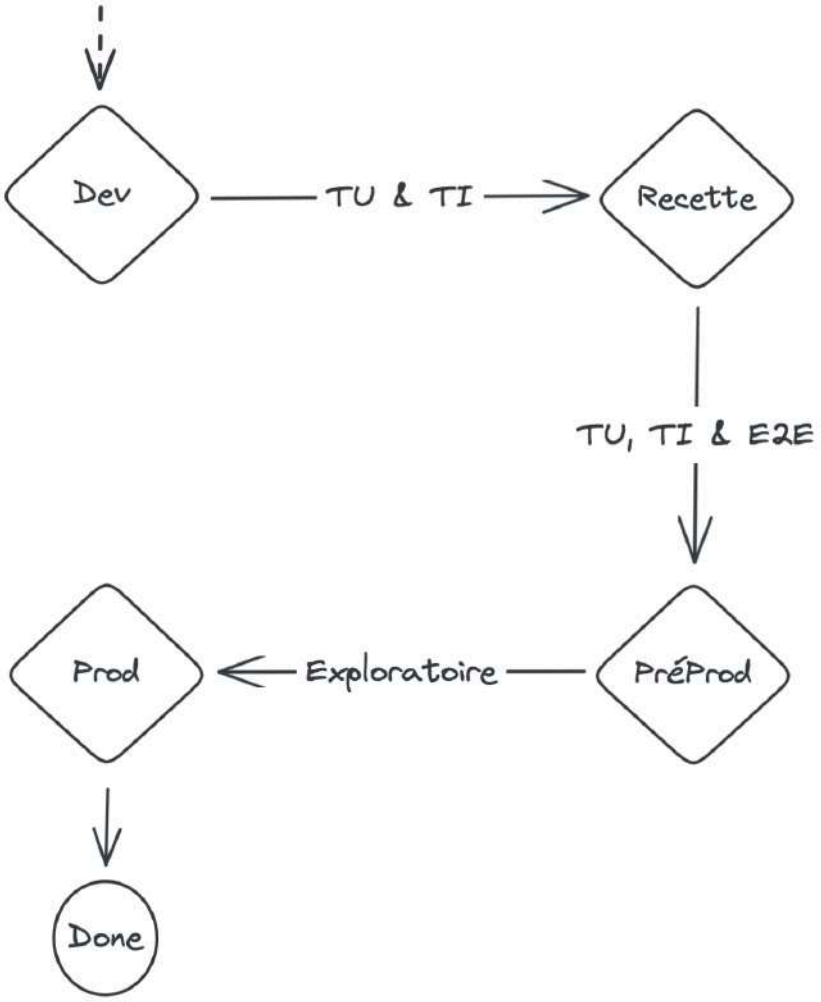
La pyramide de test est une base sur laquelle s'appuyer pour développer une stratégie de test cohérente avec le produit concerné. Le but de la stratégie de test est de déterminer quelles sont les étapes de test qu'un produit doit valider pour passer d'un environnement à l'autre. Cela permet également d'indiquer quels rôles sont responsables de chaque type de test. Une telle stratégie est extrêmement importante car cela permet de matérialiser un cycle de vie du produit et les tests associés pour garantir son bon fonctionnement.

Prenons un exemple que l'on croise fréquemment en entreprise. Avant de déployer un produit en production, il est nécessaire de passer par plusieurs environnements de validation. Le premier environnement est généralement l'environnement de développement (Dev). Il est généralement à la main des développeurs, ils peuvent l'utiliser comme ils le désirent. Il n'est pas nécessaire d'avoir de tests valides ici car nous sommes sur un environnement de travail.

Ensuite, le produit doit être déployé sur l'environnement de recette pour qu'il puisse être recetté. A ce moment, il faut disposer d'un produit stable et fiable. Ainsi, les tests unitaires et d'intégration doivent être validés avant le déploiement. Une fois en recette, le produit se trouve dans un environnement "viable", c'est-à-dire que des jeux de données sont disponibles pour pouvoir y jouer des cas réels. Dans cet environnement sont exécutés les tests end-to-end. Malheureusement, il est fréquent qu'une entreprise n'ait pas automatisé ces tests. Ils sont donc déroulés manuellement. La validation de ces scénarios (automatiquement ou non) permet de s'assurer du bon fonctionnement de l'application. Il sera donc possible de déployer le produit en pré-production.

Une fois en pré-production, le produit sera dans un environnement très similaire à la production. Il sera donc possible de le mettre dans des conditions de stress réel. Ce stress est représenté de deux manières. D'abord, les tests exploratoires pour simuler le comportement non prédictible de certains utilisateurs. Ils permettent de s'assurer que l'application est robuste. Ensuite, les tests de charges pour simuler un grand nombre

d'utilisateurs simultanés. Ces tests ne sont pas dans la pyramide de test car celle-ci s'adresse surtout aux développeurs alors que les tests de charge sont souvent gérés par des rôles tels que les DevOps. Une fois ces tests validés, le déploiement en production peut se faire sereinement.



En plus de donner un séquencement des étapes de validation, la stratégie de test permet d'indiquer qui est responsable de la bonne qualité et de l'exécution de chacun des tests. Dans l'idéal, la répartition des responsabilités devraient être telle que :

- les développeurs sont garants de la qualité de code
    - tests unitaires
    - tests d'intégration
- le Product Owner est garant de la qualité du produit
    - tests end-to-end
- les designers sont garants du parcours utilisateur
    - tests end-to-end
- les testeurs (QA) sont garants de la robustesse du produit
    - tests exploratoires

Attention cependant, être garant et responsable d'un type de test ne signifie pas que c'est le rôle de cette personne de les développer. Sa responsabilité consiste à s'assurer que les tests concernés permettent bien de valider ce qu'on attend de lui. Par exemple, c'est le rôle du Product Owner de fournir aux développeurs des scénarios utilisateurs cohérents pour développer des tests end-to-end pertinents.

La stratégie doit s'adapter au contexte de l'entreprise, au produit et aux équipes travaillant dessus. Dit autrement, une stratégie extrêmement sécurisante ne serait pas adaptée à un projet "Proof Of Concept". De la même manière, une stratégie de test très légère ne serait pas adaptée à un produit bancaire car cela nécessite un très haut niveau de confiance avant un déploiement. Il convient donc de créer une stratégie de test pour chaque produit voire pour chaque brique technique d'un produit (ex.: front et back).

Principe F.I.R.S.T.

Afin d'écrire des tests de qualité, nous pouvons appliquer le principe FIRST. Ce principe donne 5 critères que doivent respecter les tests pour être réellement utiles à notre quotidien de développeur.se.

- Fast

  Les tests doivent être le plus rapides possible à exécuter. L'objectif est d'avoir un résultat le plus tôt possible lors de la phase de développement pour s'assurer que rien ne casse. Des tests rapides pourront être exécutés fréquemment et éviter les erreurs dès le départ.

- Independent

  Les tests doivent être indépendants les uns des autres. Autrement dit, le succès d'un test ne doit pas dépendre des autres tests. Une telle dépendance rend les tests difficiles à corriger. Ils ne permettent donc pas d'identifier rapidement la cause de l'erreur. Pour ce faire, il convient de mettre en place une phase de setup (mise en place) et de teardown (nettoyage) pour que chaque test soit exécuté dans un environnement cohérent et que l'environnement revienne à un état consistant après chaque test.

- Repeatable

  Les tests doivent pouvoir être exécutés autant de fois que l'on veut dans chaque environnement sans avoir à faire d'action particulière. Si ce n'est pas le cas, cela va rendre complexe la maintenance et l'exécution des tests car des actions manuelles seront nécessaires pour garder l'environnement dans une état cohérent. Ainsi, notre garde-fou n'est plus systématique et donc moins efficace.

- Self-validating

  Un test doit pouvoir de lui-même exprimer son résultat : être passant ou non passant.

Il ne doit pas y avoir d'action humaine (ex.: validation d'une string à la main) pour s'assurer de la validité d'un test. Autrement, la validité d'un test serait subjective et coûteuse.

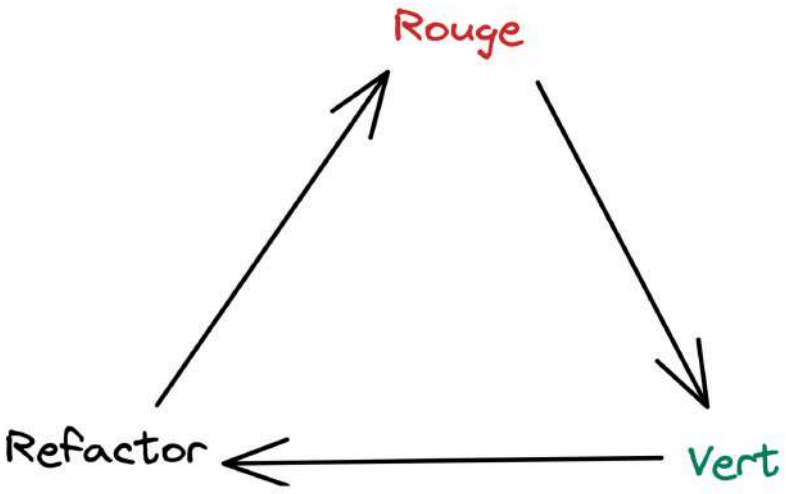
- Timely

  Les tests doivent être écrits "just in time" c'est-à-dire au plus proche de l'écriture du code testé. L'idéal est de les écrire juste avant le code de production (TDD). Mais il faut a minima les écrire juste après le code de production. Cela permet d'avoir un garde fou dès le départ et de s'assurer que le nouveau code est bien testable.

Test Driven Development

Le Test Driven Development, ou TDD, est un bon moyen de garantir que les tests écrits respectent le principe FIRST. Le TDD se concentre principalement sur le critère "Timely", mais pour en faire il est nécessaire que les autres critères soient respectés. Ainsi, s'imposer cette rigueur facilite grandement l'écriture de tests de qualité.

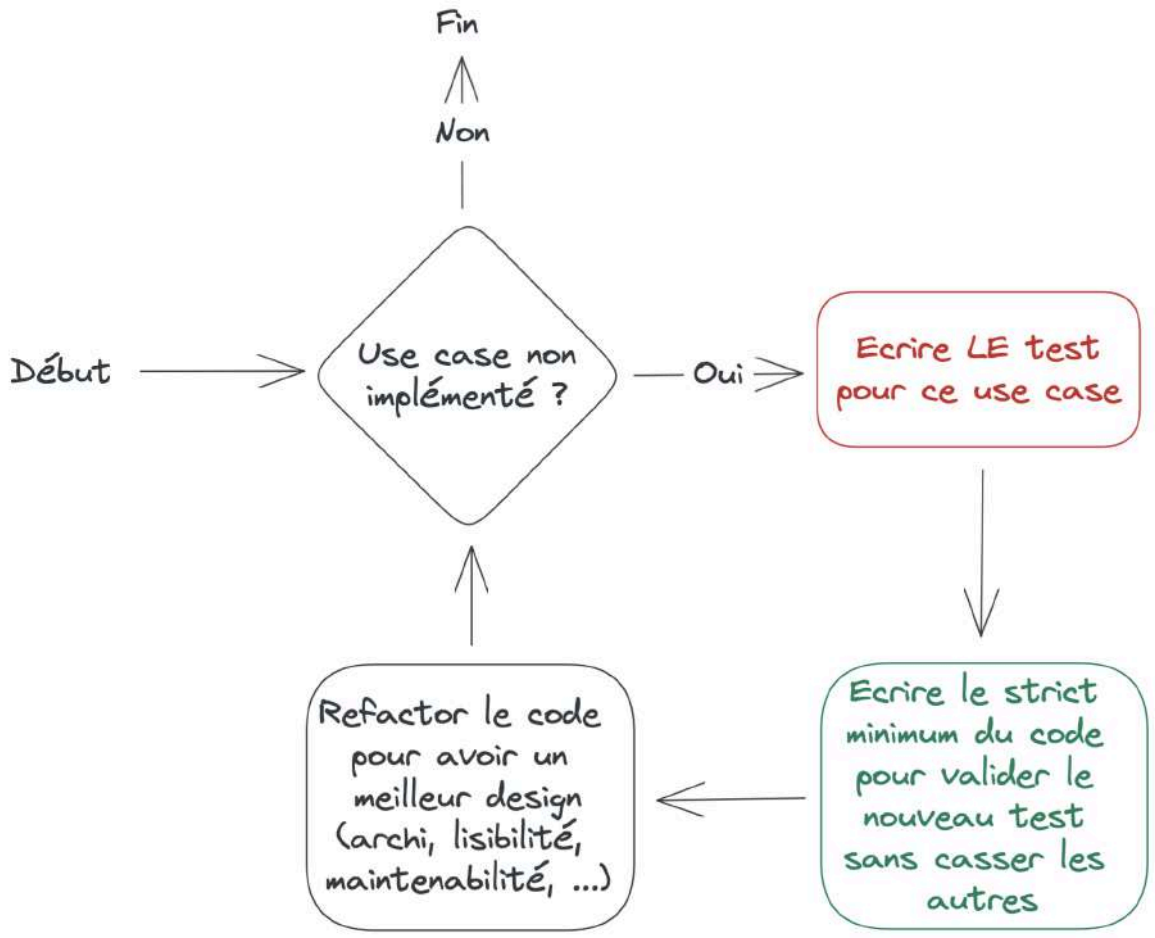
L'objectif du TDD est de développer en se concentrant sur les tests et non sur le code de production. C'est-à-dire que notre point de départ n'est pas le code mais le test. Nous allons donc commencer par écrire les scénarios de tests que nous souhaitons valider et seulement ensuite nous écrirons le code respectant ces tests. Cette approche, qui peut paraître contre-intuitive, doit suivre un rythme bien précis pour être efficace : "red, green, refactor".



Le rouge représente l'échec d'un ou plusieurs tests, le vert représente le succès de l'ensemble des tests et le refactor représente la phase d'amélioration du code de production. Entrons dans le détail de ce rythme.

La première étape est d'écrire un test représentant un cas que notre fonction doit remplir. N'ayant pas de code de production associé pour répondre à ce cas, le test sera en

échec, soit rouge. Une fois que le cas est décrit par le test et que le test est en échec, nous pouvons nous attaquer au code de production. Il faut écrire le strict minimum de code sans modifier le test pour le faire passer sans casser les autres tests existants. Ici, il faut savoir se restreindre car il est facile de vouloir faire plus que nécessaire. L'important est que notre test soit valide, donc vert, pas de faire du code parfait et exhaustif. Maintenant que nous avons fait le strict minimum pour que notre test soit vert, nous pouvons nous attaquer au refactoring. C'est-à-dire que nous allons pouvoir modifier notre code pour qu'il soit plus lisible, plus générique, plus adaptable, etc… Un moyen efficace pour savoir si notre code a besoin de refactoring dans cette phase est la mise en évidence de répétition. Si une opération est répétée plusieurs fois dans le code, c'est probablement qu'il faut le refactor. Cependant, nous avons une contrainte. Il ne faut pas simplement refactor, il faut le faire sans qu'aucun test ne passe à rouge. Grâce à cette contrainte, nous nous assurons que le code modifié continue de répondre au besoin. Maintenant que notre code est de meilleure qualité tout en respectant les cas tests, nous allons pouvoir ajouter un nouveau cas en créant un nouveau test et recommencer le cycle. De manière graphique, le TDD respecte ce diagramme de décision :



Ce fonctionnement peut paraître contre-intuitif et difficile à mettre en place. Cependant, les apports de cette pratique sont avérés et reconnus. Le premier apport est de forcer le respect du principe FIRST. C'est-à-dire que les tests sont écrits au plus proche du code, rapides à exécuter, peuvent être exécutés dans n'importe quelle condition autant de

fois qu'on le souhaite en donnant automatiquement un résultat. Le second apport est de garantir que tout nouveau code est accompagné de ses tests. Ainsi, en plus d'assurer une bonne couverture de test, cela permet de garantir une très bonne fiabilité et maintenabilité du code. En effet, un code testé automatiquement est un code dont on peut s'assurer de son fonctionnement à tout instant, même après modification. Aussi, cette pratique permet d'écrire du code au plus proche du besoin car nous sommes forcés de l'écrire selon les cas d'usages et non une compréhension partielle ou une anticipation inutile. Dit autrement, étant dirigés par les cas d'usages, on réduit le risque d'"over engineering". On respecte donc le principe YAGNI ("You ain't gonna need it"). Enfin, le dernier avantage du TDD est l'émergence de design. Étant donné que l'on fait le strict nécessaire pour répondre au besoin, et ce de manière incrémentale, nous allons pouvoir architecturer notre code au plus proche du besoin. Cela mène souvent à des design que nous n'avions pas envisagé initialement mais qui sont bien plus efficaces.

Vocabulaire et définitions

Nous savons désormais pourquoi les tests sont importants, les différents types de tests à utiliser, les contraintes que doit respecter un bon test et comment tester efficacement le code dès son écriture. Cependant, nous ne savons pas comment écrire un test à proprement parler. Pour cela, il nous faut quelques outils et pratiques.

Les affirmations (assertions)

Les affirmations ou assertions dans un test représentent l'action de validation du résultat. C'est la partie la plus importante du test car c'est cette action qui permet de valider que le résultat de notre code correspond bien à l'attendu de notre scénario. Les assertions se présentent toujours de la manière suivante :

```
assertThat(result).isEqualTo(expected)
```

Cette ligne peut être lue telle que : "je m'attends à ce que mon résultat soit égal à mon attendu". Ici, nous avons une assertion d'égalité mais il existe d'autres types d'assertion. Par exemple, il est possible de vérifier qu'une liste contient bien un élément donné. Ces assertions vont être interprétées par la librairie de test pour générer le rapport et indiquer quels tests sont en succès et quels tests échouent.

Les simulacres (mocks)

Dans le cadre des tests unitaires ou d'intégration, il est fréquent de vouloir tester notre code en l'isolant complètement ou partiellement. Pour cela, nous avons besoin de simuler tout ou partie de nos dépendances afin d'en contrôler le comportement. Ces simulations sont nommées des simulacres ou mocks. L'objectif est de remplacer les dépendances simulées par des implémentations plus simples et prédictibles afin d'exécuter nos tests dans un environnement contrôlé et obtenir un résultat fiable et identique à chaque exécution. Les mocks sont très pratiques pour avoir des comportements identiques à

chaque exécution tout en conservant un certain dynamisme dans la réaction des dépendances simulées en fonction des paramètres transmis.

Les bouchons (stubs)

Les bouchons ou stubs répondent à la même problématique que les mocks. Cependant, ici on ne parle pas de remplacer l'implémentation mais de remplacer le résultat. Dit autrement, un mock va être une nouvelle fonction ou un nouveau module qui va remplacer notre dépendance. Ainsi, il est possible d'avoir des comportements dynamiques selon les paramètres. Un stub est simplement un résultat qui remplace toute l'exécution et le résultat des dépendances simulées. Par exemple, le stub de la fonction d'addition pourrait être le chiffre 5. Ainsi, à chaque appel à la dépendance, quels que soient les paramètres, c'est le stub qui sera retourné. Bien que peu adaptable, ce comportement est très utile lorsque le résultat de la dépendance n'a pas besoin de changer au fil de l'exécution de notre test et il est plus simple que la mise en place d'un mock.

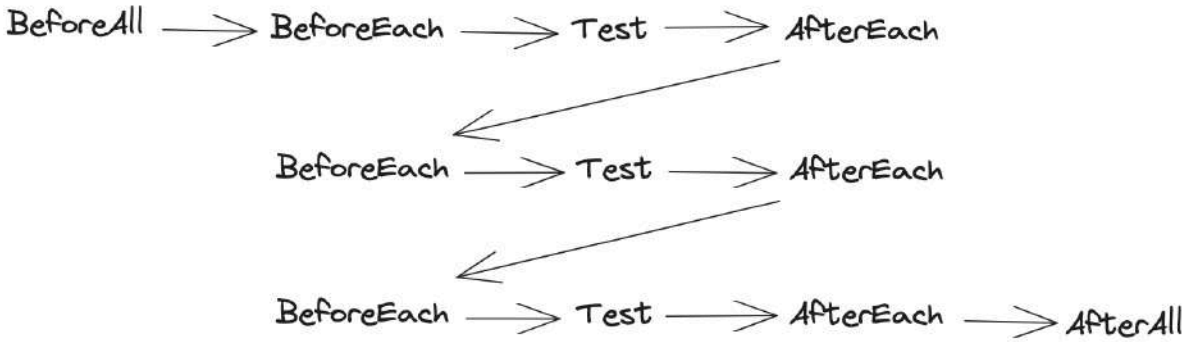
Les suites de tests (test suite)

Une bonne pratique pour les tests automatisés est de faire un test par scénario ou use case. C'est-à-dire qu'une fonction disposant de 10 use cases aura 10 tests associés. Si les tests ne sont pas regroupés, il sera difficile de comprendre quels tests correspondent à quelles fonctions. Pour cela nous utilisons des suites de tests (test suite en anglais). Une suite de tests est un regroupement de tests par brique technique ou fonctionnalité. Ce regroupement n'est pas seulement pour la lecture, mais les librairies de tests en tiennent compte pour exécuter les tests d'une même suite en même temps et afficher les résultats suite par suite dans le rapport.

Une suite de test permet également de définir des phases d'initialisation (setup) et des phases de nettoyage (teardown). Il existe quatres phases :

- *BeforeAll* : cette phase permet d'exécuter un traitement une seule fois avant l'ensemble des tests d'une même suite. Elle est souvent utilisée pour initialiser les dépendances de tests (ex.: base de données de tests).
- *BeforeEach* : cette phase permet d'exécuter un traitement avant chacun des tests d'une même suite. Elle est souvent utilisée pour initialiser les mocks et les stubs.
- *AfterEach* : cette phase permet d'exécuter un traitement après chacun des tests d'une même suite. Elle est souvent utilisée pour annuler les effets de bord qui auraient pu avoir lieu lors des tests (ex.: nettoyer les jeux de données).
- *AfterAll* : cette phase permet d'exécuter un traitement une seule fois après l'ensemble des tests d'une même suite. Elle est souvent utilisée pour nettoyer l'écosystème de test (ex.: éteindre la base de données de test).

Ces phases sont très utiles pour limiter la répétition des fixtures de setup et teardown. Elles permettent aussi d'améliorer la fiabilité des tests en garantissant l'état de l'écosystème d'exécution (ex.: toujours les mêmes données). Enfin, elles permettent de garantir l'indépendance des tests en les exécutant systématiquement dans un environnement vierge de toute autre modification. Ainsi, une suite de tests est exécutée telle que :


BeforeAll
BeforeEach
Test
AfterEach
BeforeEach
Test
AfterEach
BeforeEach
Test
AfterEach
AfterAll

Chapitre 5 - Travailler en équipe

Nous avons vu des pratiques, des réflexes et des outils que nous pouvons utiliser pour favoriser un code de meilleure qualité. Mais comment s'assurer que ces pratiques, réflexes et outils sont effectivement en place ? Malheureusement, il n'est pas toujours possible d'automatiser, ainsi nous devons faire appel à l'humain et plus particulièrement à l'équipe projet pour en faire des standards appliqués systématiquement.

Tout.e développeur.se dans le monde professionnel est amené à travailler en équipe. Comme nous l'avons suggéré plus haut, ce travail en équipe peut être un levier pour produire du code de qualité. Aussi, rappelons-nous que les développeur.se.s passent le plus clair de leur temps à lire du code, le leur ou celui de leurs collègues. Ainsi, le travail en équipe est un élément du clean code à part entière. Nous allons voir 3 pratiques d'équipes qui permettent de valider qu'un code est de qualité ou de produire un code de qualité.

Revue de code

La revue de code, ou code review en anglais, est la pratique d'équipe la plus répandue pour s'assurer de la qualité du code. L'objectif de cette revue est d'avoir la validation d'un ou plusieurs collègues quant à la qualité de notre code avant de fusionner notre modification dans la base de code partagée à l'équipe. Elle a généralement lieu à la fin du développement d'une fonctionnalité et consiste en une relecture du code pour s'assurer du respect des standards d'équipe, des bonnes pratiques et de la maintenabilité du code. Souvent, cette relecture provoque des demandes de modification ou d'ajustement pour améliorer le code que le.a développeur.se pourra prendre en compte. Cela permet donc de produire un code de meilleure qualité en tant qu'équipe. En outre, de part les échanges induits, cela amène à une montée en compétence des membres de l'équipe.

Pair programming

Le pair programming est une pratique de développement à plusieurs permettant que deux personnes travaillent sur le même ordinateur. L'un va écrire le code en se concentrant sur la qualité du code qui est écrit, c'est le **driver**. L'autre va réfléchir à la conception et à l'architecture de la solution pour s'assurer que le code écrit s'inscrit correctement dans l'architecture de l'application, c'est le **navigator**. Dans le cas où le driver et le navigator sont de même niveau d'expérience, il est intéressant d'intervertir régulièrement les rôles pour éviter la lassitude. En revanche, si l'un d'eux est moins expérimenté, il est préférable qu'il soit le driver pour que le développement aille à son rythme et qu'il puisse apprendre de la conception de l'autre.

Cela amène de gros avantages quant à la qualité du code et au partage de connaissance. D'abord, le pair programming permet d'avoir une sorte de revue de code au fil de l'eau. C'est-à-dire que le code écrit sera d'emblée lu par deux personnes. À l'image d'un correcteur orthographique, cette double lecture va significativement minimiser les erreurs car elles seront immédiatement détectées et corrigées. Cette logique s'applique

également à la conception et l'architecture du code car le driver pourra challenger les idées du navigator.

Par ailleurs, le pair programming permet de favoriser le partage de connaissance. Que ce soit pour faire monter en compétence un junior ou découvrir une partie du code, être accompagné simplifie grandement le développement. En effet, cela permet d'apporter des explications et des réponses immédiatement sans avoir à solliciter quelqu'un ou explorer le code par soi-même. Cette logique s'applique également au nouveau code qui est créé de cette manière.

Enfin, cela permet d'augmenter la résilience de l'équipe. Le partage de connaissance autour du code permet de limiter le risque de perte de connaissance. Ainsi, plusieurs personnes maîtrisent un sujet donné. De cette manière, si l'une d'elle n'est pas disponible, l'autre peut toujours intervenir en cas de besoin. Aussi, le travail en pair favorise le rapprochement des développeur.se.s et le partage de pratiques. Ainsi, la communication entre les membres de l'équipe est plus fluide ce qui limite les risques de quiproquo ou d'incompréhension.

Cette pratique peut sembler être une perte de temps car, naïvement, on peut se dire qu'un développement va coûter deux fois plus cher car 2 personnes travaillent sur le sujet au lieu d'une. Cependant, une étude de 1998 a montré que le pair programming ne rajoute un coût que de 60% par rapport au travail seul et, après une période d'adaptation, la surcharge de coût descend à 15%. C'est-à-dire que pour une équipe habituée, le pair programming n'est que 15% plus cher que le travail individuel. Ce coût doit être comparé aux apports évoqués ci-dessus. Le pair programming est donc un investissement car, même si à court terme cette pratique est plus coûteuse, sur les moyen et long termes cela permet d'avoir une meilleure résilience d'équipe et une meilleure maintenabilité de code.

Mob programming

À l'instar du pair programming, le mob programming est une pratique de développement à plusieurs. Ici, on ne parle pas de seulement de 2 personnes mais d'un groupe entier (au moins 3 personnes) sur le même ordinateur. Cette pratique est très peu répandue car elle est consommatrice de ressources. Même si, comme le pair programming, travailler à plusieurs a de nombreux avantages, écrire du code nécessite un temps minimum incompressible. Ainsi, lorsque l'on s'intéresse aux ressources (en jour/homme par exemple) ce temps minimum est multiplié par le nombre de personnes participant au mob programming. Nous pourrions nous dire que cette technique doit donc être évitée. Cependant, elle a de nombreux avantages lorsqu'elle est utilisée à bon escient.

Tout d'abord, il faut suivre quelques règles pour que cela fonctionne efficacement. D'une manière générale, il est préférable que l'ensemble de l'équipe participe au mob programming. Cela permet un engagement global des membres du cette équipe et cela évite qu'une personne ne se sente exclue. Ensuite, il faut diviser l'équipe en 2, une personne avec le clavier (driver) et les autres (mob). Le groupe "mob" aura pour but de discuter autour de la réalisation de la tâche à faire pour concevoir et designer la solution de la manière la plus efficace possible. De son côté, le driver aura pour seul objectif de traduire

les idées du reste du groupe en code sans participer aux discussions. Ce rôle nécessitant une concentration très forte et une bonne réactivité, il est important de changer régulièrement pour éviter d'épuiser quelqu'un et que le driver puisse avoir l'occasion de s'exprimer en passant dans le groupe "mob".

Cette pratique permet donc de faire travailler l'ensemble d'une équipe sur un seule et unique sujet. C'est donc un outil tout indiqué pour traiter des sujets complexes et fortement impactant. En effet, la présence de tous les membres de l'équipe favorise l'identification et le contournement des potentiels problèmes. Cela favorise également l'apparition de design architecturaux et de solutions au plus proche du besoin, en évitant l'over-engineering. Cette pratique peut également être utilisée lors de la création d'une équipe ou l'arrivée d'un nouveau membre pour mettre en place et partager des standards et pratiques d'équipes. Enfin, lorsque le groupe est peu expérimenté (professionnellement ou sur le projet), il est possible de modifier la pratique pour que le driver soit la personne la plus expérimentée et qu'elle participe au discussion pour transmettre de la connaissance et mettre en évidence le fonctionnement du projet.

Contrairement au pair programming qui peut être utilisé presque systématiquement, le mob programming doit être une pratique plus ponctuelle. Même si elle apporte beaucoup en termes de qualité et de partage de connaissance, elle reste très coûteuse. Ainsi, il est important de sélectionner les bons sujets pour appliquer cette pratique afin de garder un engagement maximal de l'équipe sans la ralentir.

Gestion de l'historique

Le travail en équipe implique le partage du travail de chacun. Pour ce faire, l'outil principal du développeur est son gestionnaire de version. La très grosse majorité des projets utilise Git comme outil de versionning. La force de cet outil est sa gestion de l'historique sous la forme d'un arbre disposant de d'autant de branches que nécessaire. Le problème c'est que cette force peut rapidement se transformer en problème si les branches sont mal gérées.

L'historique d'un projet doit être vu comme un outil de partage de connaissance de la frise chronologique des développements. Ainsi, comme tout outil de partage, plus il sera clair et facile à lire, plus son utilisation sera efficace. Pour simplifier son usage, il existe plusieurs bonnes pratiques. Tout d'abord, il faut s'attaquer à la clarté de notre travail personnel. Pour que notre travail soit correctement historisé, 3 règles doivent être respectées :

- Un titre de commit court et explicite décrivant le contenu
- Un commit doit contenir des modifications cohérentes entre elles (ex.: une modification et les tests associés)
- Un commit doit laisser l'application dans un état fonctionnel

En suivant ces règles, un commit est "auto-porteur" et peut être traité indépendamment des autres. Cela ne sera peut-être pas cohérent d'un point de vue métier mais l'application fonctionnera.

Ensuite, il faut s'attaquer à la clarté de l'historisation à l'échelle de l'équipe. Le plus simple est de définir un workflow d'équipe. Cela permet de standardiser l'usage des branches, des tags, des merge et des rebases. De cette manière, tout le monde travaillera de la même manière, ce qui rendra l'historique uniforme et plus simple à comprendre.

Enfin, il faut choisir le bon workflow. Un workflow Git doit être le reflet du cycle de vie de l'application. C'est-à-dire qu'une application très simple pourra se contenter d'un workflow simpliste. A l'inverse, une application multi-environnement, multi-production devra disposer d'un workflow plus complexe. Aujourd'hui, deux grandes familles de workflow s'opposent :

- Gitflow, ce workflow et ceux qui s'en inspirent sont très complets mais nécessitent une bonne connaissance de Git. Ils sont souvent lourd à manipuler et à comprendre mais permettent de gérer quasiment tous les cas
- Trunkbase, ce workflow et ceux qui s'en inspirent ont pour vocation d'avoir un historique le plus simple possible (historique linéaire). La lecture et la navigation dans un tel historique sont triviales cependant, la mise en place de ce workflow nécessite une bonne maturité de travail pour éviter de provoquer des régressions.

Il est possible de combiner ces deux grands courants pour obtenir un compromis adapté à un projet. Cependant, il faut avoir en tête que plus un workflow est simple à utiliser, plus il a de chances d'être respecté. Ainsi, l'usage d'un workflow complet comme le Gitflow doit être un choix conscient permettant de répondre à un besoin précis.

Chapitre 6 - Les principes SOLID

SOLID est un acronyme regroupant 5 principes de développement. Bien que ces 5 principes soient connoté programmation orientée objet, ils peuvent s'appliquer dans n'importe quel paradigme de programmation.

Ces 5 principes sont reconnus par une très grande partie de la communauté comme étant des principes fondamentaux à respecter pour produire un code robuste, maintenable et de qualité. Cependant, contrairement à ce que nous avons vu jusqu'à présent, ces principes ont un prisme architectural, macroscopique.

Single responsibility principle

Ce principe nous indique que les différentes parties de notre code ne doivent avoir qu'une seule et unique responsabilité. C'est-à-dire que qu'une partie donnée de notre code ne doit gérer qu'une seule action. Comme le dit Robert C. Martin dans son livre "Clean Code" à propos des fonctions :

"Functions should do one thing. They should do it well. They should do it only"

Ce principe peut sembler relativement opaque et difficile à vérifier. Un bon moyen de savoir si notre code respecte ce principe, et pour encore citer Robert C. Martin, est de vérifier que notre code respecte la phrase suivante :

"A [function] should have only one reason to change"

C'est-à-dire que seul un changement bien spécifique dans nos fonctionnalités ou dans notre architecture ne devrait nous amener à modifier notre morceau de code. Par exemple :

```
public void addTodo(Todo todo, User user) {
    if (user.policy == Policy.FREE && user.todos.size() < 10
        || user.policy == Policy.PREMIUM) {
        user.todos.add(todo);
        userRepository.update(user);
    }
```

Ici, la fonction *addTodo* vérifie le plan de tarification et le nombre de todos que l'utilisateur a pour savoir s'il peut ajouter la todo à l'utilisateur pour ensuite mettre à jour la base de données. Dit autrement, notre fonction a pour responsabilité la vérification du plan de tarification, la vérification du nombre de todos, la modification de l'utilisateur et la modification des données en base. De fait, si les restrictions liées au plan de tarification changent, si l'objet *User.todos* ou si le modèle de données change, nous devons mettre à jour notre fonction. Cela fait 3 raisons de changer. Ainsi, notre fonction ne respecte pas le "*single responsibility principle*".

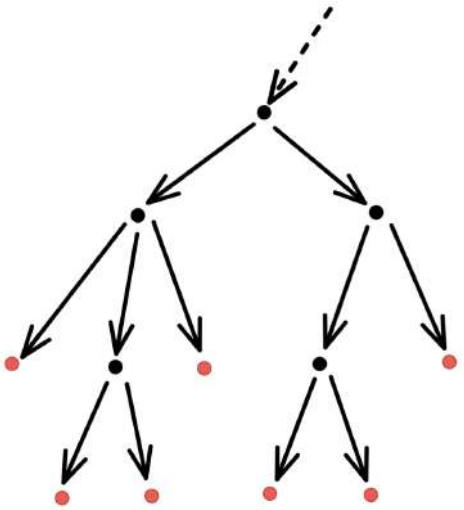
Maintenant, considérons l'exemple suivant :

```
public void addTodo(Todo todo, User user) {
    if (canUserHaveMoreTodos(user)) {
        userService.createTodo(todo, user)
    }
}
```

Ici, notre fonction vérifie si l'utilisateur peut avoir plus de todo pour déclencher ou non le comportement d'ajout de todo à notre utilisateur. C'est-à-dire que notre fonction ne changera que si la restriction n'est plus nécessaire. Le comportement d'ajout d'une todo n'étant pas géré par notre fonction, son changement n'est pas impactant. Ainsi, notre fonction garde exactement le même comportement mais respecte le "*single responsibility principle*".

Cela peut sembler artificiel car si les règles pour ajouter des todos changent, le comportement global de l'ajout de todo sera impacté. Cependant, la gestion et la prise en compte de ces règles n'est plus la responsabilité de la fonction *addTodo*. Grâce à une abstraction, *addTodo* n'a plus connaissance des détails d'implémentation de ces règles. Ainsi, sa seule responsabilité est de déclencher la vérification. La bonne application de ces règles devient la responsabilité de la fonction *canUserHaveMoreTodos*. De la même manière, *addTodo* est responsable du déclenchement de sauvegarde d'une todo mais l'implémentation est la responsabilité de *userService.createTodo*.

Ainsi, notre code sera composé de deux types de fonctions. Des fonctions d'implémentation qui vont déclarer le détail d'implémentation d'un comportement et l'enchaînement des instructions pour le réaliser. Et des fonctions d'agrégation qui vont faire appel à des fonctions d'implémentation pour construire un comportement global.



En représentant notre code sous la forme d'un arbre, les fonctions d'implémentations sont les feuilles et les fonctions d'agrégations sont les nœuds intermédiaires.

Open / close principle

Ce principe nous indique que notre code doit être ouvert à l'extension mais fermé à la modification. C'est-à-dire que nos fonctions, classes et modules doivent permettre l'enrichissement par de nouveaux cas sans avoir à changer leurs implémentations.

Pour mieux le comprendre, prenons un exemple :

```
public float area(Shape shape) {
    if (shape instanceof Square) {
        return Math.pow(shape.length, 2);
    } else if (shape instanceof Circle) {
        return PI * Math.pow(shape.radius, 2);
    }

    return null;
}
```

Ici, la fonction *area* permet de calculer l'aire d'une forme. Si nous regardons de plus près les détails d'implémentation, ce calcul n'est disponible que pour les carrés et les cercles. Pour ajouter un nouveau cas, disons le rectangle, nous serons obligés de modifier l'implémentation de la fonction pour que cela fonctionne. De fait, notre fonction n'est pas fermée à la modification.

Modifions notre code :

```
class Square implements Shape {
    @Override
    public float area() {
        return Math.pow(this.length, 2)
    }
}
```

Ici, la fonction *area* est un membre de l'interface *Shape* qui est surchargée dans la class *Square*. Ainsi, pour ajouter un nouveau cas, le cercle ou le rectangle, nous devrons créer une nouvelle classe. Cependant, si l'interface *Shape* ni la classe *Square* ne doivent être modifiées. De fait, notre code est ouvert à l'extension car il autorise les nouveaux cas, mais fermé à la modification car il n'est pas nécessaire de modifier l'existant pour ajouter ce nouveau cas.

Liskov principle

Le Liskov principle nous indique si q(x) est une propriété démontrable pour tout objet x de type T, alors q(y) est vraie pour tout objet y de type S tel que S est un sous-type de T. Dit autrement, un comportement valide pour une entité donnée sera valide pour toutes entités enfant. Ce principe est très important en programmation objet car elle permet de

créer des comportements se basant sur des interfaces, utilisables pour toutes classes implémentant cette interface.

Prenons l'exemple de la classe "Talkative" permettant à toutes entités de parler :

```
class Talkative {
    public String greet() {
        return "Hi !";
    }
}
```

Nous allons utiliser cette classe pour déclarer une fonction affichant le résultat de la salutation d'un "*Talkative*" :

```
public void sayHi(Talkative entity) {
    System.out.println(entity.greet())
}
```

Grâce au principe de Liskov, nous savons désormais que toutes entités enfant de "*Talkative*" peut être utilisées dans cette fonction :

```
class Humain extends Talkative {
    @Override
    public String greet() {
        return "Hi, I'm a human";
    }
}
sayHi(new Human());
```

Cependant, un changement de comportement dans un des enfants serait une violation du principe de Liskov. Tel que :

```
class Snake extends Talkative {
    @Override
    public String greet() {
        throw new Exception();
    }
}
```

Ici, la fonction ne retourne pas une *String* mais jette une exception. Cette possibilité n'est pas indiquée dans la classe *Talkative*, ainsi nous ne respectons plus le principe de Liskov

Interface segregation principle

Le principe de ségrégation des interfaces indique qu'aucun code ne doit dépendre de fonctions qu'il n'utilise pas. C'est-à-dire que dans le cas où une interface expose des

fonctions que certains enfants n'utilisent pas, alors nous sommes en train de violer ce principe. Dans un tel cas, il faut séparer les fonctions en plusieurs interfaces différentes afin que chaque enfant n'implémente que les fonctions dont il a réellement besoin.

Prenons l'exemple de la géométrie :

```
interface Shape {
    public float area();
    public float volume();
}

class Cube implements Shape {
    /* ... */
}
```

Ici, notre interface *Shape* déclare 2 méthodes : *area* et *volume*. Ainsi, pour notre cube cela ne pose aucun problème.

```
class Square implements Shape {
    /* ... */
}
```

Cependant, dans le cas d'un carré qui est une forme en 2 dimensions, la notion de volume n'a pas de sens. Il serait possible de faire retourner 0 à la fonction volume, mais dans ce cas, nous sommes en violation du principe de ségrégation des interfaces car le carré dépend d'une volume n'ayant pas d'utilité pour lui.

Ainsi, pour respecter ce principe, nous devons séparer notre interface *Shape* en deux interfaces distinctes. Une pour les formes en générale et une spécifique pour les formes tri-dimensionnelles :

```
interface Shape {
    public float area();
}

interface 3DShape {
    public float volume();
}

class Cube implements Shape, 3DShape {
    /* ... */
}


class Square implements Shape {
    /* ... */
}
```

De cette manière, le carré n'a plus besoin de se préoccuper de la notion de volume tandis que le comportement du cube est inchangé.

Dependency inversion principle

Le principe d'inversion de dépendance indique que les entités doivent dépendre des abstractions et non des implémentations. Ainsi les modules de haut niveau ne doivent pas dépendre des modules de bas niveau mais doivent dépendre des abstractions. Dit autrement, les intéractions entre les modules doivent se faire en respectant des contrats d'interface et non les détails d'implémentations spécifiques.

Prenons l'exemple d'un module permettant de se connecter à différents types de base de données :

```
interface DBConnector {
    public void connect();
    public void disconnect();
}

class MySQLConnector implements DBConnector {
    /* ... */
}

class PostgreSQLConnector implements DBConnector {
    /* ... */
}
```

Ici, nous avons une interface décrivant le comportement de connexion et de déconnexion d'un connecteur et nous avons fait 2 implémentations, une pour MySQL et une pour PostgreSQL. Maintenant, créons un service utilisant ces connecteurs :

```
class DBService {
    private MySQLConnector mySQLConnector;
    private PostgreSQLConnector postgreSQLConnector;

    DBService(MySQLConnector mySQLConnector) {
        this.mySQLConnector = mySQLConnector;
        this.postgreSQLConnector = null;
    }

    DBService(PostgreSQLConnector postgreSQLConnector) {
        this.mySQLConnector = null;
        this.postgreSQLConnector = postgreSQLConnector;
    }
}
```

Ici, notre service dispose de 2 membres et 2 constructeurs pour gérer soit le connecteur MySQL soit le connecteur PostgreSQL. Le problème est qu'avec cette façon de faire, notre service dépend de l'implémentation car il a besoin de savoir qu'il y a une différence entre les 2 bases de données. De fait, nous sommes en violation de l'inversion de dépendance.

Pour respecter le principe et éviter de devoir systématiquement faire des tests pour savoir la base actuellement utilisée, nous devons nous abstraire des détails d'implémentation en utilisant une abstraction. Ici, notre abstraction est l'interface *DBConnector* :

```
class DBService {
    private DBConnector dbConnector;

    DBService(DBConnector dbConnector) {
        this.dbConnector = dbConnector;
    }
}
```

En utilisant l'abstraction, notre service n'a plus besoin de savoir quel type de base de données nous sommes en train de manipuler. Etant donné que tous les connecteurs respectent l'interface *DBConnector*, notre service fonctionnera toujours quel que soit le type. Aussi, nous pouvons voir dans cet exemple que le respect de l'inversion de dépendance nécessite de respecter le principe de Liskov. Cela permet également de respecter le "open close principle" car dans le cas d'une nouvelle base de données, le connecteur respectera l'interface *DBConnector*. Ainsi, l'usage d'une nouvelle base sera transparent pour le *DBService*.

Chapitre 7 - Domain Driven Development

Nous l'avons vu dans le *Chapitre 2 - Le nommage*, l'un des plus grands défis de la programmation est de comprendre et se faire comprendre. Jusqu'à présent, nous avons abordé des pratiques, des standards, des outils et des principes permettant de communiquer entre développeur.se.s. Le vecteur principal de communication est le code, mais nous avons aussi vu des manières de documenter notre code (tests automatisés) et de partager la connaissance (pair et mob programming). Cependant, dans la très grande majorité des cas, les développeur.se.s répondent à un besoin émis par le "métier" ou "business". Ainsi, une sorte de dualité se met en place. D'un côté les développeur.se.s qui n'ont pas connaissance de tous les besoins et subtilités métier mais qui connaissent très bien le code. Et de l'autre nous avons les responsables "métiers" qui ont une très bonne connaissance du besoin et de ses subtilités mais qui ne connaissent pas le code. Dit autrement, deux types de profils n'ayant ni la même connaissance ni le même langage doivent communiquer et interagir ensemble pour créer un produit qui répond au besoin tout en étant de qualité sur le plan technique. Comment peuvent-ils comprendre et se faire comprendre de l'autre ? C'est là que le Domain Driven Development (DDD) intervient.

Un langage commun

La première étape est de définir un vocabulaire commun. Quelle que soit la nature du projet, le besoin est exprimé via des spécifications telles que des User Stories, un cahier des charges ou des tâches. Ces spécifications seront écrites en utilisant une terminologie métier. Prenons un exemple financier :

En tant que souscripteur, je veux avoir un contrôle sur l'approbation du contrat pour limiter les risques d'exposition et rejeter les contrats risqués.

Qu'est-ce qu'un souscripteur ? De quel contrat parle-t-on ? Qu'implique son approbation ? Qu'est-ce que l'exposition ? Comment considère-t-on un contrat risqué ? Qu'implique le rejet d'un contrat ? Sans ce vocabulaire, il nous est impossible de comprendre ce qu'on attend de nous en tant que développeur.se. Aussi, nous n'avons pas questionné ce que voulait dire "contrôle sur l'approbation". La terminologie étant proche du langage commun, nous avons déduit sa signification. Seulement, sommes-nous sûrs d'avoir bien compris ? En cas d'ambiguïté, nous sommes peut-être sur le point de développer quelque chose ne répondant absolument pas au besoin.

On constate avec l'exemple ci-dessus qu'établir un langage commun est la base de la communication. En l'absence de vocabulaire et expression partagés, il est très probable que le code ne réponde pas au besoin exprimé. Cependant, ce langage ne doit pas se limiter à la simple transmission de spécifications. Si nous entendons les termes "contrat", "souscripteur" et "exposition" au quotidien, nous voulons aussi voir cette terminologie dans notre code.

Pour bien le comprendre, prenons l'exemple d'un.e nouveau.elle développeur.se sur notre application financière. Après avoir discuté avec les métiers, il sait ce qu'est un

souscripteur. Cependant, il ne trouve aucun mot approchant dans le code. En effet, à la création du projet, les souscripteurs ont été nommés *Validator* dans le code. Ainsi, notre nouveau collègue ne retrouve pas les informations qu'il vient d'apprendre. Cela peut paraître secondaire, mais en propageant cette erreur de nommage à l'ensemble des termes métiers, nous nous retrouvons constamment à faire la traduction. Nous sommes donc en train de maintenir l'écart de langage entre les développeur.se.s et les métiers. A l'inverse, si le terme "souscripteur" avait été traduit par *Underwriter* dans le code, aucune question ou traduction ne serait nécessaire. Aussi, pour citer Robert C. Martin, un bon code est un code "où l'on trouve quelque chose là où on s'attend à le trouver".

Pour nous, développeur.se.s non anglophones, il y a un problème supplémentaire. Le "franglais" et les anglicismes. Dans l'exemple précédent, nous avons suggéré de créer dans le code une entité *Underwriter* pour gérer les souscripteurs". Même si la traduction est valide, elle n'est pas forcément simple pour tout le monde. Ainsi, nous pourrions être tentés de renommer notre entité en *Souscripteur* mais nous allons avoir des mélanges de langues tels que *souscripteur.update()*. Dans un tel cas, il n'y a pas vraiment de bonne solution. Utiliser l'anglais est un standard de développement mais utiliser les termes métiers tels quels est une bonne pratique du DDD. Ainsi, la meilleure chose à faire est souvent de s'accorder sur une pratique en équipe. En revanche, une fois la décision prise, il faut s'y tenir. Si nous nous mettons d'accord pour garder les termes français, alors tous les termes métiers doivent être en français, même si la traduction est simple. De la même manière, si nous choisissons de traduire en anglais, tout doit être traduit, même lorsque cela est compliqué.

La notion de domaine

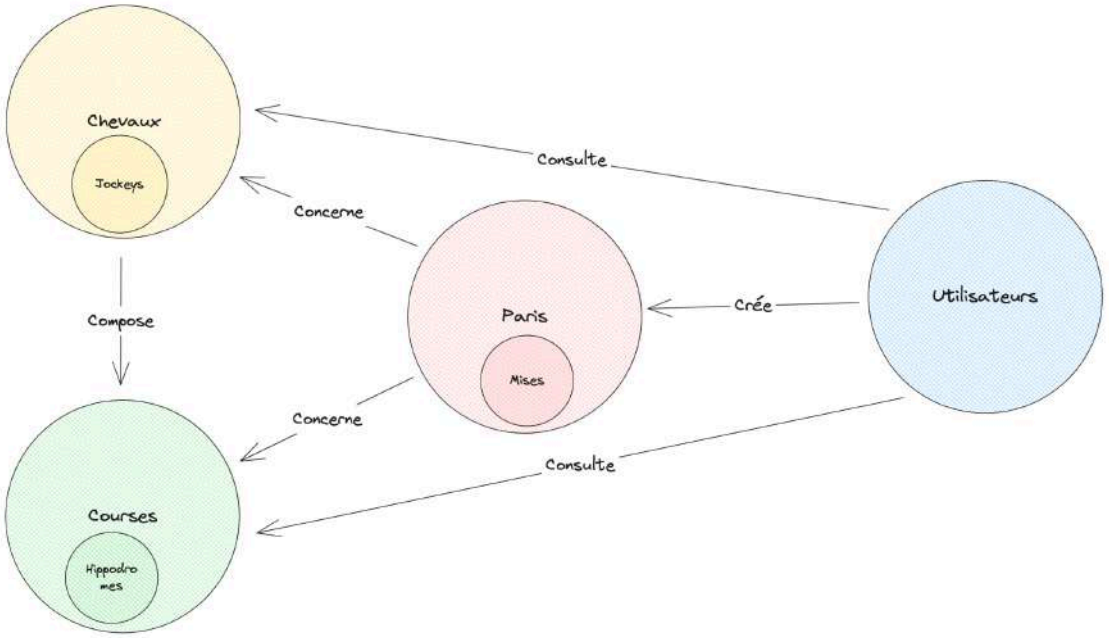
Nous avons désormais un vocabulaire commun entre tous les membres d'une équipe, développeur.se ou non, et ces termes se retrouvent dans le code. Nous pouvons donc facilement interagir ensemble, se comprendre et passer du langage oral au code. Cependant, ce n'est pas aussi simple que ça. Avant de traduire les termes métiers dans notre code, il faut comprendre les entités représentées par ces termes et leurs interactions. C'est là qu'intervient la notion de domaine.

Un domaine est un regroupement d'entités cohérentes et co-dépendantes permettant l'ensemble des actions et comportements associées à celle-ci.

Pour bien comprendre cette définition, prenons l'exemple d'un site de pari hippique en ligne. Penchons nous sur une liste restreinte de termes : course, hippodrome, jockey, cheval, pari, mise, utilisateur. Ces termes représentent les entités que nous allons devoir manipuler. Cependant, nous ne savons pour le moment pas comment ces entités interagissent entre elles. Il faut donc les mettre en scène au travers de spécification :

En tant qu'utilisateur, je souhaite pouvoir analyser les performances des chevaux, montés par des jockeys, participants à une course ayant lieu dans un hippodrome pour afin de parier sur un cheval avec une mise adéquate.

Grâce à cette mise en scène, nous avons plus d'informations sur les interactions entre nos entités. Nous constatons que l'hippodrome est une information d'une course, les chevaux participent à une course, le jockeys est une information liée au cheval, le pari va être pris sur un cheval pour une course donnée, la mise est une information du pari et l'utilisateur va consulter les courses et les chevaux pour créer des paris. Nous allons pouvoir faire des regroupements pour mettre en évidence les interactions :



Nous pouvons voir qu'en représentant notre système sous la forme d'un graphe des groupes apparaissent. Les groupes *Courses*, *Chevaux*, *Paris* et *Utilisateurs* vont représenter nos domaines et les flèches qui les lient seront leurs intéractions. Dit autrement, chaque domaine va contenir l'ensemble des informations et actions liées aux entités qu'il contient tout en exposant aux autres domaines les fonctions nécessaires aux interactions identifiées. Dans notre exemple, le domaine *Courses* expose des fonctions de lecture pour répondre au besoin de consultation du domaine *Utilisateurs*. Également, le domaine *Paris* expose des fonctions de création pour répondre au besoin du domaine *Utilisateurs*.

Cependant, pour les besoins de l'exemple, nous nous sommes contentés d'un unique cas d'usage. Dans le monde réel, il est nécessaire de représenter les entités et leurs interactions dans plus cas d'usage différents voire la totalité. Cette vision plus large permettra de créer des regroupements et des interactions plus proches du besoin réel et donc de créer des domaines plus adaptés.

Séparation des préoccupations

Grâce à la séparation de notre code en différents domaines spécifiques à seulement quelques entités, il est très facile de savoir où chercher pour trouver une information. Mais cela permet aussi d'avoir des morceaux entiers de notre code respectant le Single Responsibility Principle. En effet, un domaine n'aura pour responsabilité que la gestion des quelques entités lui appartenant. De cette manière, il est le seul maître des entités qu'il gère

et de leur capacité. Cependant, comme l'a montré l'exemple précédent, des domaines peuvent avoir besoin de créer ou modifier des entités ne leur appartenant pas. Nous avons donc un problème car le principe de séparation des préoccupations, *separation of concern* en anglais, créer ou modifier une entité d'un autre domaine n'est pas la responsabilité du domaine actuel. Ce n'est donc pas à lui de le faire.

Pour gérer ce genre d'interaction, il est nécessaire de créer une interface permettant d'interagir avec le domaine dans lequel nous souhaitons créer ou modifier l'entité. A l'image d'une API REST, un domaine pourra interagir avec un autre via un contrat d'interface indiquant ce qu'il est possible de faire, avec quels paramètres et pour obtenir quel résultat. Le problème étant que dans le cas du DDD, les domaines font souvent partie de la même base de code. Il est donc facile d'aller faire des actions directement dans un autre domaine. Seulement, nous sommes ici en plein effet de bord car un domaine X modifiant le contenu d'un domaine Y revient à dire que le domaine X à un impact à l'extérieur de son scope. Dans un tel cas, en plus des problèmes que nous avons évoqués dans le *Chapitre 3 - Les fonctions*, les effets de bord peuvent mettre l'application dans un état corrompu. C'est-à-dire que la modification dans un autre domaine est susceptible d'avoir été faite selon des règles invalides. Ainsi, le domaine modifié se trouve dans un état anormal et inutilisable. Il est donc essentiel de n'utiliser que l'interface exposée par le domaine.

Cette interface est une couche anti corruption, anti corruption layer en anglais. Elle permet de garantir que les modifications appliquées seront toujours valides selon les règles du domaine concerné. Cette couche d'abstraction permet également de découpler les domaines. C'est-à-dire qu'ils ne seront dépendants que des contrats d'interfaces et non de l'implémentation. Cela nous permet de respecter le Dependency Inversion Principle car nos domaines deviennent indépendants des détails d'implémentation. Aussi, il est important de fournir une interface générique permettant l'ajout de nouveaux cas sans avoir à modifier le comportement du domaine concerné. De cette manière nous respectons le Open / Close Principle.

Dans le code

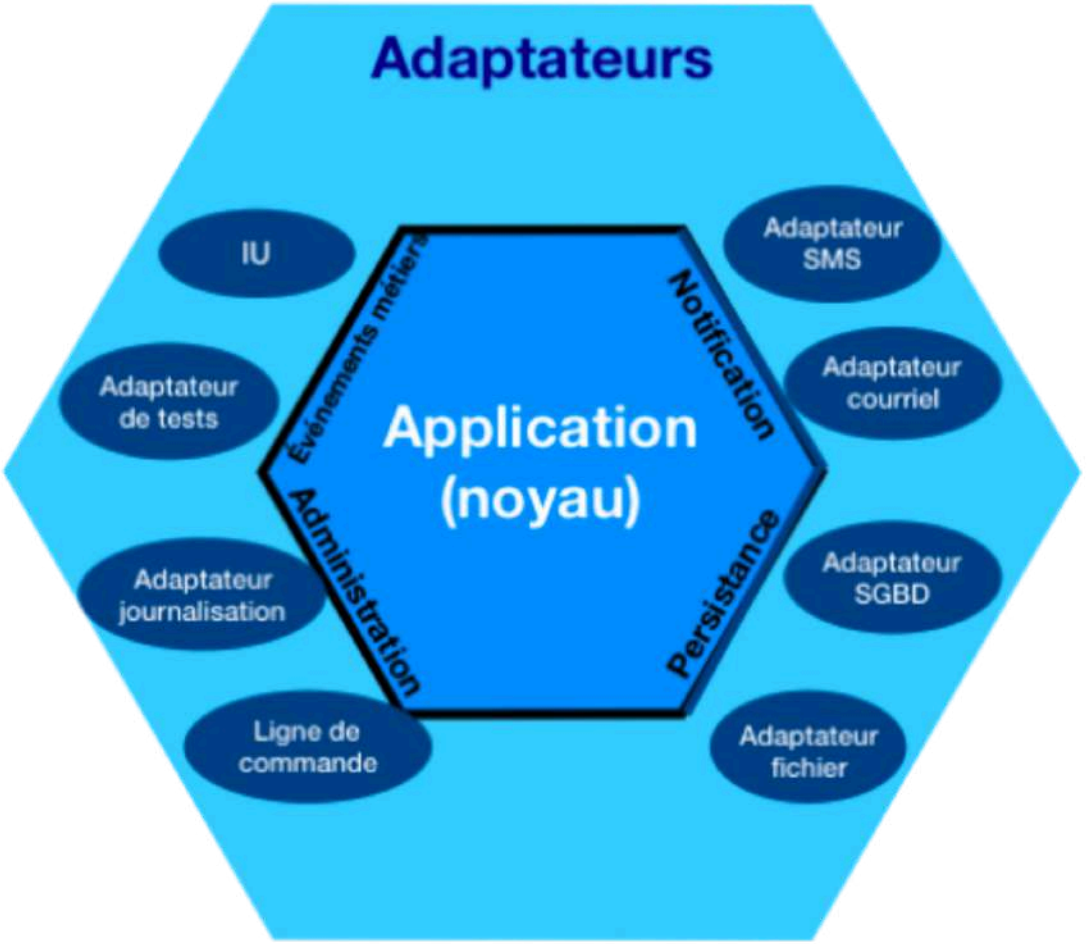
Appliquer le principe du DDD dans le code est relativement simple mais il est nécessaire d'être rigoureux. D'une manière générale, cela se traduira par un dossier ou un package par domaine. Grâce aux modificateurs d'accès ou gestion de package des langages, il sera possible de ne mettre à disposition que certaines fonctions et entités aux autres domaines.

Cependant, même si la logique peut paraître simple, définir des domaines adéquates est un réel défi. Aussi, il n'est pas exclu qu'un changement d'orientation métier ait lieu. Il est donc nécessaire de modifier les domaines, les entités associées et les interactions pour correspondre aux problématiques métier.

Enfin, ce cours n'est qu'une introduction au DDD. Il existe de nombreuses autres règles et considérations à prendre en compte. Le livre *Domain-Driven Design: Tackling Complexity in the Heart of Software*, par Eric Evans est une référence pour apprendre le DDD.

Chapitre 8 - Architecture Hexagonale

L'architecture hexagonale a été inventée par Alistair Cockburn. Ces dernières années, cette architecture a fait son grand retour car elle est très pratique pour mettre en place des micro-services. L'objectif est d'isoler les différentes responsabilités techniques du code. Bien que partant d'un besoin différent, le DDD et l'architecture hexagonale arrivent à un résultat similaire : du code isolé interagissant au travers d'interface. Ainsi, même si elles peuvent être appliquées séparément, ces deux techniques sont complémentaires car l'une aborde la séparation d'un point de vue métier tandis que l'autre aborde la séparation d'un point de vue technique.



Un découpage par couche

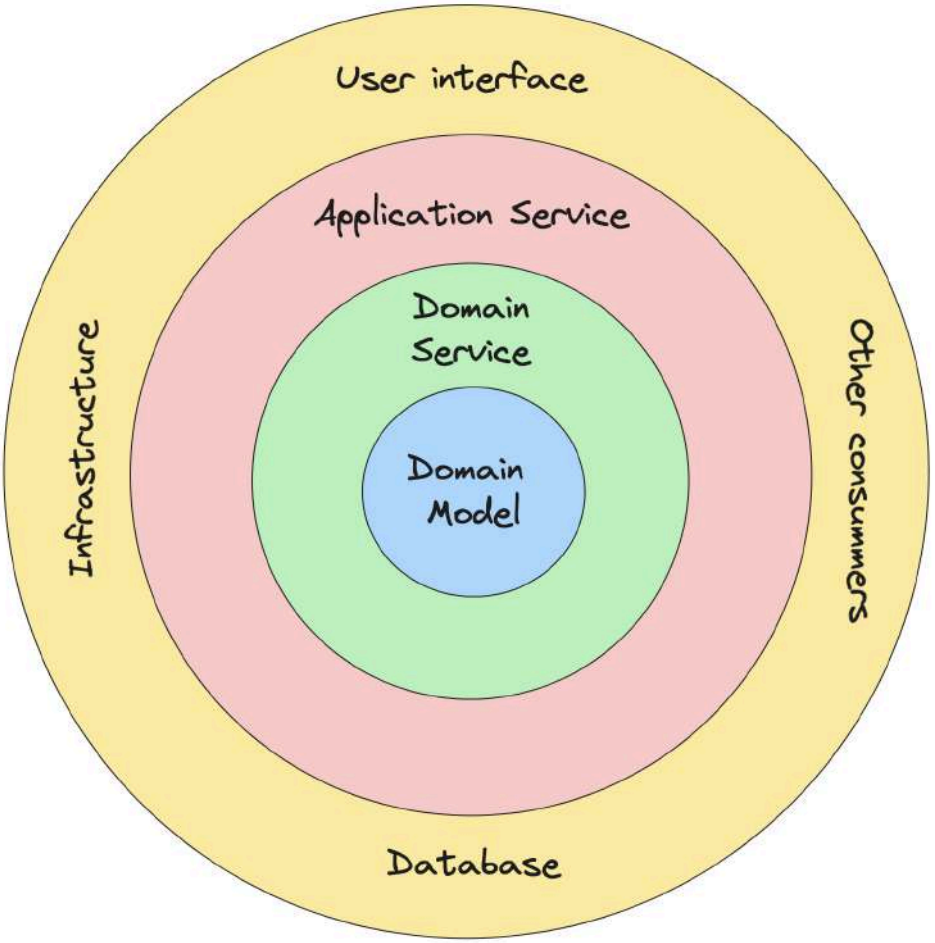
L'architecture hexagonale propose une vision de l'application par couche. A l'image d'un oignon, chaque couche est entièrement contenue dans la couche supérieure. Aussi, pour atteindre une couche inférieure, nous devons traverser l'ensemble des couches qui lui sont supérieures.

Au centre de l'application, le noyau, se trouve la définition du domaine métier de l'application. Y seront définies les entités propres à notre domaine. Par dessus se trouvera la couche de service qui contiendra l'implémentation des règles métiers. Ce noyau applicatif (entités + service) est le cœur de notre application autour duquel tout le reste va graviter. C'est à cet endroit que l'implémentation des fonctionnalités et des comportements se trouvera. Cette couche est la couche de domaine. Idéalement, elle doit être agnostique du framework que nous utilisons. Cela ne veut pas dire qu'on ne peut pas utiliser de bibliothèque, mais l'usage d'un framework est souvent lié aux interactions avec d'autres parties de notre application. Ce n'est pas la responsabilité de cette couche.

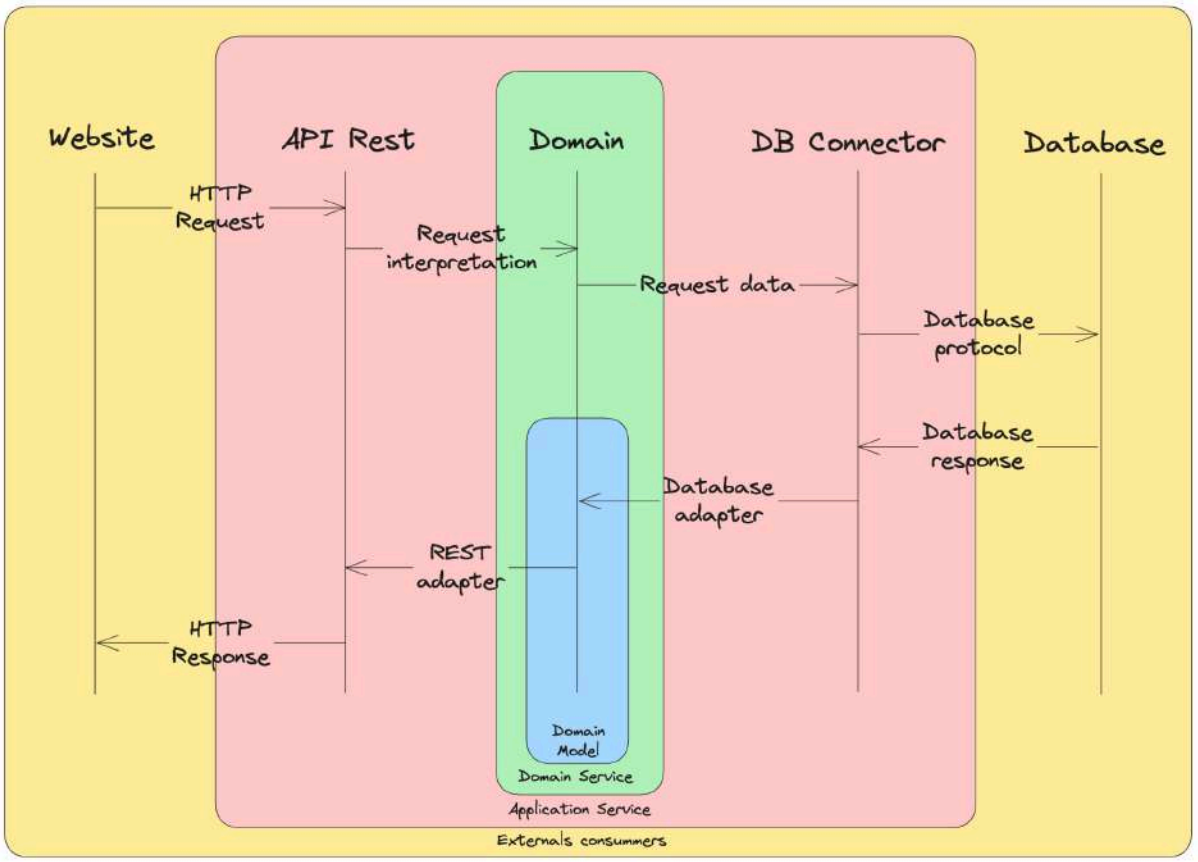
Afin d'interagir avec le reste de l'application, la couche de domaine sera encapsulée dans une couche de service applicatif. C'est ici que les liens entre le domaine et les besoins techniques seront implémentés. C'est-à-dire que nos frameworks interviendront dans cette couche. Aussi, ici seront définis des *adapters*. Le principe des adapters est de transformer des entités extérieures à notre métier en entités compréhensibles par notre métiers et réciproquement. Dit autrement, les adapters sont des traducteurs entre notre métier et l'extérieur. Les adapters permettent de rendre notre métier indépendant des besoins extérieurs. De fait, le métier reste isolé des changements extérieurs et, en cas de changement du métier, les adapters font le travail de traduction pour ne pas impacter l'extérieur. Ici, nous parlons donc de découplage entre le métier et les consommateurs.

Maintenant que nous avons notre métier et nos outils de connexions, il faut permettre la connexion aux entités extérieures. Ces points de connexion font partie de la couche de service applicatif et sont appelés des ports. Les ports vont utiliser les adapters pour transmettre les informations transformées du métier vers l'extérieur et de l'extérieur vers le métier. Lorsque l'on parle d'extérieur, ici nous parlons de n'importe quelle brique technique ne faisant pas partie de notre application. Par exemple, une interface web serait un consommateur externe communication avec l'application en passant par le port de l'API REST.

Enfin, la couche extérieure sera celle des consommateurs. C'est-à-dire que la base de données, l'infrastructure ou les interfaces utilisateurs sont des briques externes qui devra utiliser un port pour communiquer avec notre application. Cette couche doit être anticipée dans l'architecture de notre application mais ce n'est pas à nous de la gérer. Notre seule responsabilité est de fournir un contrat d'interface et un canal de communication (le port) pour interagir avec notre application.



Une fois toutes ces couches en place, il peut être difficile de savoir dans quel sens lire le graphique et d'identifier la manière dont transite la donnée. De manière quasiment systématique, la donnée va suivre les rayons du diagramme. C'est-à-dire qu'elle va soit provenir de la couche extérieure pour aller vers le centre du graphique, soit partir du centre du graphique pour aller vers la couche extérieure. Prenons l'exemple d'une application web interagissant avec un serveur via une API REST. Dans le cadre de la récupération d'une donnée, le web va solliciter la couche de service via le port API REST pour atteindre le domaine qui va interpréter la requête au sens métier pour transmettre l'information à la couche de service pour interroger la base de données. La donnée ainsi sélectionnée fera le chemin inverse pour retourner au web.



Isolation des modules

Ce découpage par couche permet d'isoler notre domaine du reste des applications. Une fois mise en place, cela permet de diminuer significativement les couplages et la résistance aux changements. Le code étant découpé par logique technique, une fois un problème ou une modification identifié, il est facile de l'appliquer au bon endroit. Aussi, les différentes couches étant découplées les unes des autres, une modification n'a que peu voire pas d'impact sur les autres couches. Ce qui permet d'avoir des changements très localisés. De plus, le découpage et le découplage facilite grandement le travail à plusieurs car le risque de conflit est limité. Pour finir, le risque de régression pour les consommateurs est très limité car les adapters servent à transformer la donnée pour que toutes les briques les comprennent.

Cependant, pour pouvoir mettre en place une telle architecture et garantir l'isolation, il faut avoir une bonne maîtrise des principes SOLID. En effet, elle est une réelle contrainte car elle limite nos possibilités et notre liberté. Ainsi, l'architecture hexagonale nécessite une rigueur et une qualité de code élevée pour ne pas subir sa mise en place. Également, le code va s'alourdir car le lien entre les différentes couches doit se faire par des adapters et des interfaces pour éviter tout couplage.

Dans le code

Dans le monde réel, cette architecture va se traduire par la création de module, de dossier ou de package, suivant le langage que nous utilisons. Aussi, chaque couche pourra suivre toutes les règles de clean code que nous avons vu jusqu'à présent. Par exemple, la couche de domaine pourra appliquer le DDD et la couche de service applicatif devra mettre à disposition des fonctions clairement nommées.

Aussi, à l'instar du *Chapitre 7 - Domain Driven Development*, ce chapitre n'est qu'une introduction pour comprendre le principe de l'architecture hexagonale. Pour l'appliquer dans sa totalité, de nombreuses autres règles doivent être prises en compte. Pour ce faire, nous avons trois alternatives :

- L'architecture hexagonale, créée par Alistair Cockburn
- L'architecture en oignon, créée par Jeffrey Palermo
- L'architecture clean, créée par Robert "Uncle Bob" C. Martin

Les deux dernières architectures reprennent le principe de l'hexagonale avec des prismes légèrement différents et quelques ajouts.

Pour finir, cette architecture étant très lourde, elle n'est pas à appliquer de manière dogmatique. Il existe deux cas où il n'est pas nécessaire d'appliquer l'architecture hexagonale ou ses dérivées :

- Il n'y a pas de logique métier. C'est le cas des applications CRUD pures ou des middlewares. Dans un tel cas, l'application n'est qu'un passe-plat.
- La logique métier est technique. C'est le cas des frameworks ou des drivers.

Dans ces deux cas, d'autres architectures doivent être utilisées pour correspondre au mieux au besoin. Malgré tout, même avec une architecture différente, les autres principes du clean code sont applicables.

Chapitre 9 - Aller plus loin

Pour appliquer toutes les règles et pratiques que nous avons vu dans ce cours, certains principes de pragmatisme peuvent facilement être appliqués.

Les mnémotechniques

KISS

“Keep It Simple Stupid”. Le principe est de garder le code le plus simple voire simpliste possible. De cette manière, il est plus facile de comprendre et de modifier le code. Naïvement il est plus simple de faire une addition que de faire une multiplication matricielle avec des nombres complexes. Pour le code, il en va de même. Une instruction directe est plus simple qu’un algorithme récursif.

DRY

“Don’t Repeat Yourself”. Le principe est de minimiser la répétition des instructions. L’objectif est de rendre certaine partie de notre code réutilisable pour éviter de réécrire plusieurs fois la même logique. Grâce à cela, en cas de changement de la logique, il n’y a qu’un endroit à modifier ce qui limite le risque d’erreur.

Attention cependant, trop factoriser le code peut entrer en conflit avec le principe du KISS. Il faut donc faire preuve de discernement pour savoir si la complexité introduite par la factorisation vaut le coût.

YAGNI

“You Ain’t Gonna Need It”. Le principe est de ne rien faire dont nous n’aurions pas besoin immédiatement. L’objectif est d’éviter ce qu’on appelle “l’over-engineering” c’est-à-dire la sur-complexification d’une implémentation pour prendre en compte des cas qui n’existent pas. Il est facile de se rendre compte que nous allons chercher trop loin. Si nous sommes en train de nous dire “et si [cas complexes]” ou des “si jamais [cas]”, c’est que nous sommes en train d’anticiper des cas qui n’existent pas encore. Si ces cas n’existent pas encore, il est probable qu’ils n’existent jamais. Ainsi, écrire du code pour gérer des problématiques qui n’auront jamais lieu est une perte de temps et cela rentre en conflit avec le KISS.

La loi de Déméter

La loi de Déméter ou principe de connaissance minimale nous indique qu’il ne faut parler qu’à nos amis immédiat en faisant le moins de supposition possible. Dans le code cela signifie qu’un morceau code ne doit pas supposer du fonctionnement d’un autre morceau de code.

Formellement, la Loi de Déméter pour les fonctions requiert que toutes méthodes M d'un objet O peut simplement invoquer les méthodes des éléments suivants :

- O lui-même
- les paramètres de M
- les objets que M crée/instancie
- les objets membres de O

Prenons l'exemple d'un objet A qui requiert les services d'un objet B. A peut donc consommer les services de B mais il ne peut pas accéder aux dépendances de B pour les utiliser directement. En cas de changement de fonctionnement de B, notre objet A ne fonctionnerait plus.

La loi de Déméter est donc une extension du principe de séparation des préoccupations. Un objet donné ne doit se préoccuper que de son propre comportement. Il doit donc faire confiance à ses dépendances et ne pas utiliser leurs comportements internes. Cela revient à dire que les dépendances d'un code doivent être utilisées comme des boîtes noires dont nous n'avons aucune connaissance du fonctionnement interne.

Conclusion

En tant que développeur.se, nous passons le plus clair de notre temps à lire et essayer de comprendre du code déjà écrit. Ainsi, pour gagner du temps, nous devons trouver un moyen d'optimiser et de minimiser ce temps de lecture. Pour cela, nous devons y faire attention dès l'écriture pour que notre code soit lisible immédiatement.

Également, l'une des grosses sources de coûts d'une entreprise sont les bugs. Que ce soit pour leur correction ou l'impact utilisateur, ces incidents représentent un fort manque à gagner voire une perte sèche. Ainsi, pour minimiser les coûts et la pression que ces bugs induisent, nous devons trouver un moyen d'en réduire le nombre. Là aussi, nous devons y faire attention dès l'écriture pour que notre code n'est pas de problème et qu'il n'impacte pas l'existant.

Pour cela, nous avons à notre disposition la boîte à outils que représente le clean code. Grâce au clean code nous allons pouvoir produire du code lisible, maintenable et robuste. En d'autres termes, nous allons écrire du code legacy avec le minimum de dette technique possible.

Bien que la lisibilité du code soit un critère relativement subjectif, il existe des standards et des pratiques qui peuvent nous permettre de tendre vers une lisibilité optimale. Cela va passer par un nommage explicite et clair de nos variables et fonctions. Ce nommage devra s'appuyer sur la terminologie métier pour favoriser la communication entre développeur.se.s mais également avec les autres corps de métier. La concision et le faible nombre d'arguments de nos fonctions sont également un facteur déterminant.

La maintenabilité et la robustesse sont toutes deux impactées par les mêmes critères. Tester le code que nous écrivons est la première étape. Cela permet de garantir que notre code répond au besoin et prévient des régressions. Ensuite, le respect des principes SOLID, découpler les différents modules de son code et éviter les effets de bord permettent de réduire le risque de comportement imprévu ou de régressions. Enfin, la mise en place de l'architecture hexagonale et le DDD sont de bons moyens d'éviter le couplage.

Pour finir, la présence de code legacy implique que le projet a déjà quelques mois voire années d'existence. Ainsi, il faut garder en tête que nous ne sommes pas les seuls à travailler dessus. Il est donc important de ne pas être le seul à avoir la connaissance et à appliquer le clean code. De fait, pour partager la connaissance et standardiser le clean code dans une équipe, nous avons à notre disposition des outils comme la revue de code, le pair et le mob programming.

Avec toutes ces techniques, nous sommes en mesure d'écrire du code de qualité et de le faire perdurer dans le temps. Avec l'expérience, l'usage de certaines pratiques pourront évoluer mais leur principe fondamental restera le même : écrire du code pour le.a prochain.e développeur.se qui le modifiera.

Bibliographies

Robert “Uncle Bob” C. Martin (2008), *Clean Code: A Handbook of Agile Software Craftsmanship*

Robert “Uncle Bob” C. Martin (2011), *The Clean Coder: A Code of Conduct for Professional Programmers*

Kent Beck (2003), *Test-Driven Development by example*

Eric Evans (2003), *Domain-Driven Design: Tackling Complexity in the Heart of Software*

Telma Ugonna (2020), *The S.O.L.I.D Principles in Pictures*, Backticks & Tildes