1 Revue de la Littérature

1.1 Clean Code : La méthode de Martin

Aujourd'hui, quand il est question de bonnes pratiques d'écriture de code, les ouvrages de Robert C Martin (ou Uncle Bob) reviennent systématiquement et plus spécifiquement **Clean Code** Et **Clean Architecture**. Le premier se concentre sur l'écriture du Code en elle même en partant de règles de nommage des entités jusqu'aux scopes des fonctions. Le second a une approche plus haute pour se concentrer sur la structure des différents composants d'une application et les façons dont ils doivent communiquer.

1.1.1 SOLID

Les principes SOLID sont largement détaillés dans **Clean Architecture**. Ils ont pour objectif de créer une architecture de code tolérante au changement, simple à comprendre et utilisable dans n'importe quelle application.

SRP : Single Responsibility Principle Le principe de responsabilité unique impose qu'une entité (une fonction, une classe, un composant, un package…) n'ait qu'une seule responsabilité, ce qui signifie qu'il ne doit y avoir qu'une seule raison pour laquelle cette entité serait modifiée (Rober C. Martin, 2012).

Prenons un composant qui manipule les utilisateurs d'une application. Ce composant ne doit être modifié que si la stratégie de gestion des utilisateurs est modifiée.

OCP : Open-Closed Principle

> A module is said to be open if it is still available for extension. For example, it should be possible to expand its set of operations or add fields to its data structures. A module is said to be closed if it is available for use by other modules.
>
> *Bertrand Meyer, 1988*

Ce principe impose qu'un ajout de fonctionnalité ne doit pas modifier le code existant d'un projet mais doit uniquement en ajouter. Pour ce faire, chaque entité de notre code doit être aisément extensible mais la signature de cette entité (par exemple, les méthodes proposées) ne doit pas être modifiée. Ainsi, on réduit au minimum le risque de changer le comportement d'une entité qui utiliserait le code que l'on modifie. (Rober C. Martin, 2012)

En programmation fonctionnelle, ce principe peut aussi s'appliquer en utilisant la composition de fonction et les "high-order functions" (fonction pouvant prendre une autre fonction comme paramètre et / ou retournant une fonction). (Patricio Ferraggi, 2022)

LSP : Liskov Substitution Principle

> Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

Do the SOLID principles apply to Functional Programming? par Patricio Ferraggi

Le principe de substitution de Liskov dit qu'une entité doit pouvoir être remplacée par une sous-entité sans altérer l'exécution du programme. Bien que le lien avec le polymorphisme de la programmation orienté objet est flagrant, ce principe peut tout à fait s'appliquer en programmation fonctionnelle. Par exemple avec l'utilisation des paramètres génériques, ce qui est très courant en programmation fonctionnelle, les fonctions résultantes de la fonction avec les paramètres génériques se comportent toujours de la même manière sans nécessiter de changement sur le code résultant. (Patricio Ferraggi, 2022)

ISP : Interface Segregation Principle Ce principe énonce qu'il est préférable d'utiliser plusieurs petites interfaces plutôt qu'une grande. Ces petites interfaces sont dites "client-specific" ce qui signifie qu'elles ne sont faites que pour un seul client. Ce qui implique en toute logique que le client définit les interfaces dont il a besoin. (Rober C. Martin, 2012)

En programmation orienté objet, il est courant de pouvoir implémenter plusieurs interfaces dans une même classe. Cependant le concept d'interface n'est pas propre à la programmation orienté objet et ce concept existe aussi dans les autres paradigmes de programmation même s'il n'est pas toujours explicite.

DIP : Dependency Inversion Principle Le principe d'inversion de dépendance dit qu'une entité ne doit jamais dépendre directement d'une autre mais doit toujours passer par une forme d'abstraction. Pour schématiser, prenons :

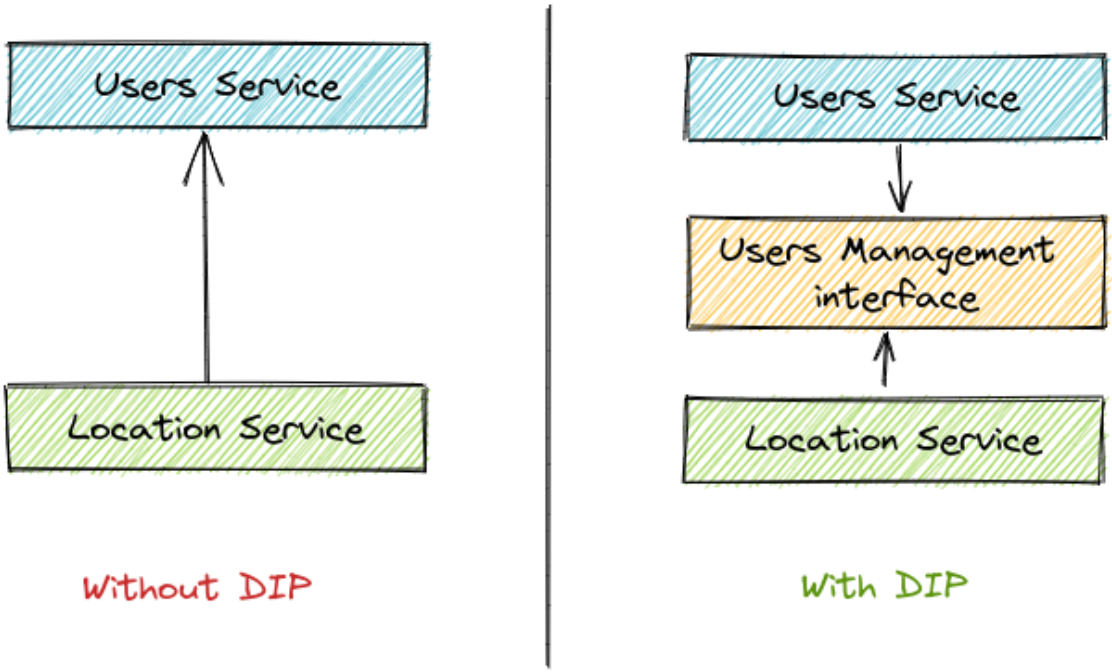

fig.3 - Exemple d'utilisation du Dependency Inversion Principle

Dans l'exemple de fig.3, nous avons deux composants : "Users Service" et "Location Service". "Location Service" a besoin des données des utilisateurs pour fonctionner, il va donc utiliser le "Users Service". Pour respecter le principe d'inversion de dépendance, le Location Service doit passer par une interface ici appelée "Users Management interface" implémentée par le User Service. Cette interface sert de contrat et permet de rendre abstrait la logique du "Users Service".

Ce principe a un grand intérêt car il est très résistant au changement. Mettons que la gestion des utilisateurs soit déportée dans une autre application afin de rendre la gestion des utilisateurs globale au sein d'une entreprise. La logique du "Users Service" dans notre application de départ changerait alors complètement. Il faudrait donc implémenter une nouvelle entité qui serait chargée de communiquer avec cette nouvelle application. Tant que cette nouvelle entité implémente l'interface, il n'y a nul besoin de modifier le "Location Service".

1.1.2 Un code de bonne qualité : Nommage des entités

Après avoir abordé les fondamentaux de l'architecture d'application avec **SOLID**, il est maintenant sujet de prendre nos composants indépendamment et d'étudier les bonnes pratiques de construction. Clean code traite un grand nombre de sujet mais nous prendrons le nommage des entités comme exemple car c'est le plus générique.

Le deuxième chapitre de Clean Code est concentré sur le bon nommage des entités dans le code. Ici quand il est question d'entité, il est question tout ce qui peut avoir un nom dans le code

d'une application. Par exemple une classe, une fonction, une variable, une constante, un attribut ou de manière générale, tout ce que le langage a besoin de nommer. Les règles principales quant aux choix des noms sont peu débattues :

- un nom doit clairement exprimer le role de l'entité et ne as porter à confusion
- deux noms doivent être distinguable de façon logique. Par exemple si on a une entité `Product`, on évitera d'en appeler une autre `ProductData` car ça n'apporte pas d'information sur la différence entre ces deux entities. Par extension, il vaut mieux éviter les mots tels que `Data` ou `Info` dans le nommage des entités car ils n'apportent pas d'informations. Toujours dans ce chapitre, ils sont qualifiés *noise words*, pour signifier leur manque de sens.
- un nom doit être prononçable et avoir du sens une fois prononcé, c'est pourquoi il faut éviter voir exclure les acronymes dans le nommage des entités.
- chaque concept de l'application doit être identifiable par un mot et ne doit être identifiable que par ce mot. Un autre concept ne doit pas être identifié par ce même mot.
- un nom doit être trouvable facilement si cherché dans la globalité du projet. Pour se faire, il ne faut pas hésiter à avoir des longs noms de variable

  The length of a name should correspond to the size of its scope

Robert C. Martin, Clean Code, Chapitre 2.

1.2 Une réflexion plus moderne

Bien que les ouvrages de Robert C. Martin restent des références sur le sujet, certains éléments sont encore source de débat.

1.2.1 Anti-pattern : Les interfaces de classe

D'après sa page Wikipédia, un *anti-pattern* est une solution à un problème récurant qui est souvent inefficace voire contre-productive.

Dans l'article Explicit interface per class *anti-pattern*, l'auteur, Marek Dec, énonce un anti-pattern souvent utilisé dans le Framework Java EE mais pouvant tout aussi bien s'appliquer dans d'autres langages ou Frameworks. Cet anti-pattern est l'utilisation des interfaces explicites de classe ou de façon plus concrète l'utilisation d'interfaces destinées à être implémentées par une seule classe. Il est expliqué que cette pratique remonterait du début de l'injection de dépendance dans laquelle elle était nécessaire pour la phase d'injection.

Nous retrouvons bien sûr la problématique du rôle de l'interface qui doit être définie par l'entité qui en a besoin puis implémentée par les autres entités et non l'inverse. Avoir une interface conçue uniquement pour une classe n'est donc pas utile car la classe pourrait être utilisée seule.

Afin de repérer cet *anti-pattern* et le supprimer, l'article propose plusieurs moyens : - Le nom de l'interface contient le nom d'une technologie : Une interface devant être la plus générique possible, la dissocier des technologies utilisées pour ses implémentations est important car la classe utilisant l'interface n'a pas besoin de savoir quelles technologies sont utilisées derrière la classe. - Les types de retours, de paramètres ou les exceptions sont associés à l'implémentation : Par exemple, si le type d'un paramètre est défini dans la dépendance utilisée dans son implémentation. Les types utilisés se doivent d'être le plus générique possible afin de faciliter la création de nouvelles implémentations. - Le nom de l'implémentation contient les mots « Impl », « Default » ou tout autre mot ne décrivant pas l'implémentation.

Là où l'auteur porte une idée contraire à Robert C. Martin est qu'à plusieurs reprises dans Clean Code, il est fait mention de cette pratique de création d'interface dédiée à une classe. Ce sujet est d'ailleurs une dualité dans les ouvrages de Robert C. Martin. Dans Clean Architecture il est exprimé qu'une interface doit être définie par le module en ayant besoin et non pas dans le module l'implémentant ce qui va dans le sens de l'article de *Marek Dec*. Cependant Clean Code casse à multiple reprise ce principe sans détailler pourquoi.

1.2.2 Les effets de bord en programmation orienté objet

En lisant Clean Code, il est surprenant de voir à quel point le livre se concentre sur la programmation orientée objet en omettant les autres paradigmes de programmation que ce soit pendant les exemples ou les appellations d'entités. Dans cette partie, il sera question de la notion d'effet de bord en programmation orientée objet.

Un effet de bord désigne tout changement d'état dans un programme se produisant en dehors du scope de la méthode ou de la fonction e cours execution. On peut prendre comme exemple d'effet de bord la modification d'une variable globale, d'un fichier ou une modification en base de donnée.

Dans le cadre de la programmation orienté objet, la notion de scope est plus complexe. Prenons une classe `User` avec deux attributs privés `name` et `age`. Si dans cette même classe nous avons une méthode birthday ne prenant aucun paramètre et ajoutant 1 à `age`. En Java, cela pourrait donner :

```
class User {
    private String name;
    private int age;

    public void birthday() {
        this.age += 1;
    }
}
```

fig.7 - Exemple de classe avec setter en Java

Cette situation présente-t-elle un effet de bord ?

Il y a deux possibilités. Soit on considère que l'objet avec ses attributs et méthodes ne forment qu'une seule et même entité donc on exclut les effets de bords, tant que rien n'est modifié en dehors du scope de la classe. Soit on considère que c'est bien un effet de bord comme l'attribut `age` est définie en dehors de la méthode et que `incrementAge` n'est pas un *mutateur*.

Un *mutateur* ou *setter* en programmation orientée objet est une méthode dont le role est de modifier un objet en assignant une nouvelle valeur à un des champs de cet objet. Champ devant être explicité dans le nom du *mutateur*.

Le problème de la deuxième vision est que chaque "setter", même au plus simple devient un effet de bord, ce qui s'oppose à une convention fondamentale de la Programmation orientée objet qui est que chaque attribut d'une fonction ne doit être accédé et modifié que par l'intermédiaire de méthode.

Cependant, cette approche est plus pertinente dans d'autres cas.

```
class Dockerfile {
    private StringBuilder content;

    public Dockerfile() {
    content  = new StringBuilder("");
    }

    public void addFrom(String from) {
        content
            .append("FROM ")
            .append(from)
            .append("\n");
    }

    public void addCommand(String command) {
        content
            .append("CMD ")
            .append(command)
            .append("\n");
    }

    public String render() {
        return content.toString();
    }
}
```

fig.8 - Exemple de classe avec des méthodes que l'on pourrait qualifier d'effet de bord en Java

Dans cet exemple, le contenu de l'attribut content est modifié par les méthodes addFrom et addCommand et lu par la méthode render. Le comportement de la méthode render sera donc lourdement impacté par les méthodes précédentes et le sera encore plus si la classe est héritée par des classes modifiant de nouveau l'attribut. De plus, en cas d'utilisation dans des processus parallèles, il se peut que les méthodes addFrom et addCommand soient appelées en même temps ce qui rendrait imprévisible leur comportement.

Dans It's probably time to stop recommending Clean Code par Qntm, l'auteur écrit en réponse à un code similaire écrit par Robert C. Martin, l'auteur de Clean Architecture: A Craftsman's Guide to Software Structure and Design :

At this point you might reason that maybe Martin's definition of "side effect" doesn't include

member variables of the object whose method we just called. If we take this definition, then the five member variables […] are implicitly passed to every private method call, and they are considered fair game; any private method is free to do anything it likes to any of these variables.

Il ajoute ensuite une citation de Robert C. Martin

« Side effects are lies. Your function promises to do one thing, but it also does other hidden things. Sometimes it will make unexpected changes to the variables of its own class. Sometimes it will make them to the parameters passed into the function or to system globals. In either case they are devious and damaging mistruths that often result in strange temporal couplings and order dependencies. »

Clean Code: A Handbook of Agile Software Craftsmanship, Robert C. Martin, chapitre 3, 2008

Il existe donc une dualité concernant les attributs privés d'un objet. D'un côté certains considèrent qu'ils peuvent être modifiés et d'autres au contraire considèrent qu'ils doivent être constants et que toute modification devrait entraîner la création d'un nouvel objet.