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 :

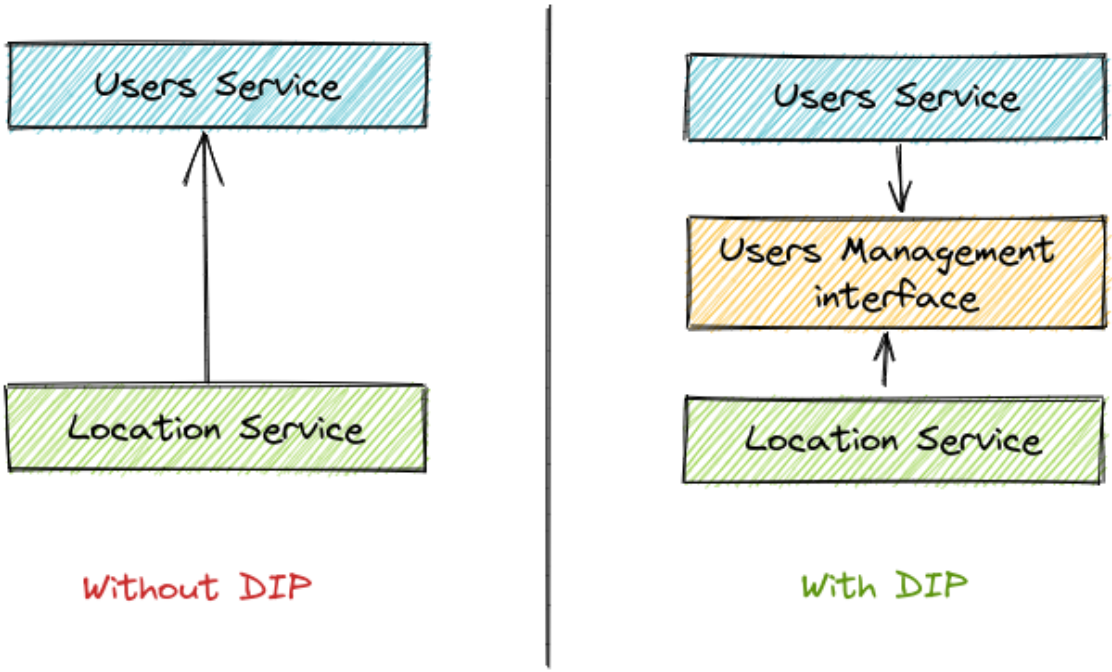

fig.3 - Exemple d'utilisation du Dependency Inversion Principle

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

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.

Nommage des entités 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.