



FACULTÉ DE
**GESTION,
ÉCONOMIE
& SCIENCES**



Éviter le legacy en optimisant la maintenance applicative.

Mémoire de recherche présenté en vue de l'obtention du Master 2 Cyber (2022 / 2023)

Anthony Quéré

Tuteur : Lucien Mousin

REMERCIEMENTS

Je tiens à remercier toutes les personnes m'ayant aidé dans la rédaction de ce mémoire.

Dans un premier temps je voudrais remercier mon tuteur de mémoire, Lucien Mousin, pour ces précieux conseils et sa disponibilité qui m'ont permis de travailler efficacement.

Je souhaiterais aussi remercier l'ensemble du DavLab de Davidson SI Nord pour leur accompagnement et leurs retours d'expériences qui m'ont permis d'agrandir mon socle de connaissance et d'apporter de nouveaux éléments pour enrichir ce mémoire.

Je tiens à témoigner toute ma reconnaissance à Jules Spicht et Sylvain Lavazais, mes tuteurs d'alternance, pour s'être rendus disponibles pour les interviews m'aidant ainsi à compléter la recherche de données pour ce mémoire.

Je tiens également à remercier les nombreuses personnes m'ayant aidé à relire et corriger ce travail de recherche.

Enfin, je remercie mes proches, en particulier ma famille et mes amis pour leur soutien au quotidien.

ATTESTATION PLAGIAT

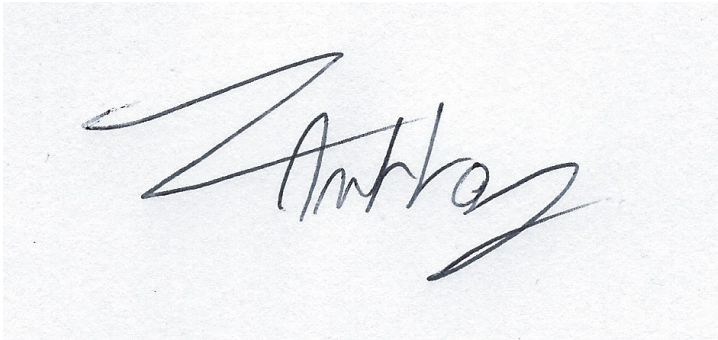
Je soussigné, Quéré Anthony, étudiant à la FGES, durant l'année universitaire « 2022/2023 » certifie que le présent mémoire de Master « titre du document », est strictement le fruit de mon travail personnel, de synthèse et d'analyse.

Toute citation (articles, livres, mémoires, documents d'entreprises, sources Internet, ...) est formellement notée comme telle, explicitée et référencée dans le corps du texte et en bibliographie. Tout tableau ou modèle (photos et illustrations diverses) est dûment cité s'il est emprunté à un auteur ou cité en source s'il est adapté.

Tout manquement à cette Charte de non-plagiat entraînera la suspension de l'évaluation du mémoire, une notation égale à 0, et la convocation devant le conseil de discipline de l'école.

Fait, à Lille le 28/04/2023

Signature

A handwritten signature in black ink on a light gray background. The signature is stylized, starting with a large, sweeping 'Z' shape that loops around the first part of the name 'Anthony'. The name 'Anthony' is written in a cursive, flowing script.

ATTESTATION PLAGIAT.....	1
INTRODUCTION.....	4
Une tour qui part dans tous les sens.....	4
Langage de programmation.....	4
Entité.....	5
Framework.....	5
Paradigme de programmation.....	5
Programmation procédurale.....	6
Programmation orientée objet.....	6
Programmation fonctionnelle.....	6
REVUE DE LA LITTÉRATURE.....	8
Clean Code : La méthode de Martin.....	8
SOLID.....	8
SRP : Single Responsibility Principle.....	8
OCP : Open-Closed Principle.....	8
LSP : Liskov Substitution Principle.....	9
ISP : Interface Segregation Principle.....	10
DIP : Dependency Inversion Principle.....	10
Un code de bonne qualité : Nommage des entités.....	12
Une réflexion plus moderne.....	13
Anti-pattern : Les interfaces de classe.....	13
Les effets de bord en programmation orienté objet.....	14
Architecture hexagonale.....	16
Architecture.....	17
Adapters.....	18
Inspirations.....	18
Au-delà du code.....	19
You build it, you run it.....	19
Normes et définitions.....	20
MÉTHODOLOGIE DE RECUEIL DES DONNÉES.....	22
Cible des entretiens.....	22
Déroulé des entretiens.....	22
Introduction.....	22
Les applications dites legacy.....	23
Clean Code.....	24
Agilité.....	24
You build it, you run it.....	24
ANALYSE DES RÉSULTATS.....	26

La documentation.....	26
L'équipe.....	26
Maintenance opérationnelle et maintenance applicative.....	27
Une rigueur automatisée.....	28
Conception de l'application.....	30
DISCUSSION.....	31
La documentation.....	31
L'équipe.....	32
Maintenance Opérationnelle.....	34
Une rigueur automatisée.....	35
Conception de l'application.....	36
CONCLUSION.....	38

INTRODUCTION

Depuis les débuts de l'informatique moderne, les évolutions en termes de technologies et de bonnes pratiques n'ont cessé de s'accélérer de façon exponentielle. Dans ce contexte en constante évolution, il est en parallèle devenu important que les applications gagnent en fonctionnalités. Le problème majeur rencontré est que l'ajout des fonctionnalités d'une application est souvent complexe car une nouvelle fonctionnalité peut changer (directement ou indirectement) le comportement des autres créant ainsi des applications de plus en plus complexes à étendre. La plus simple des fonctionnalités qui aurait pu prendre quelques heures en début de projet finit par prendre plusieurs semaines à développer dans une application avec un plus fort historique.

Une tour qui part dans tous les sens

Voyez ici la construction d'une application comme une tour d'habitation dont l'achèvement ne sera jamais terminé. Vous ne savez pas s'il faudra qu'elle grandisse en largeur ou en hauteur, sur combien d'étages, quel type de fenêtre... Il est alors nécessaire d'établir une solution dans laquelle il sera simple d'ajouter ou de modifier les éléments sans voir tout s'effondrer et devoir recommencer à zéro. Ainsi vous pourrez facilement vous adapter aux nouveaux changements.

Avant de nous intéresser à comment rendre notre code plus adaptable aux modifications, il est important de préciser la nature de notre code.

Langage de programmation

L'internaute définit la notion de langage de programmation par « Un langage de programmation est un ensemble de règles et de symboles permettant de formuler des algorithmes et de produire des programmes ». Dans la fin des années 1940 furent créés les langages assembleurs permettant aux développeurs de ne plus devoir transcrire leur code en binaire. Puis furent créés d'autres langages de plus en plus simples d'utilisation comme le A0 puis le Fortran, le COBOL, le C... Jusqu'à aujourd'hui avec des langages

créés il n'y a même pas dix ans comme le Go ou le Rust (*Clean Architecture de Robert C. Martin, introduction de la deuxième partie*).

Entité

La notion d'entité représente tout élément manipulable. Une entité peut être une variable, une fonction, une classe, un groupement de classe (package ou composants par exemple) voire même des applications (dans le cas d'une architecture microservice, donc composée de plusieurs applications, par exemple).

Framework

La notion de Framework n'est pas approfondie dans ce mémoire de recherche mais elle peut aider à comprendre certains passages.

Des millions de programmes existent, nombreux sont ceux qui ont du code en commun. Pour épargner aux développeurs la tâche de devoir toujours tout refaire à la main, des outils ont été créés comme par exemple des bibliothèques permettant d'ajouter des entités externes dans les programmes sans avoir à réécrire le code. Au fur et à mesure certains outils ont commencé à être utilisés ensemble et ces ensembles furent appelés des frameworks.

Paradigme de programmation

Nous comptons aujourd'hui plusieurs méthodes de programmation principales que l'on appelle "paradigmes". Les paradigmes sont utilisés par des langages ou frameworks et permettent de garder une cohérence dans la logique d'un langage à un autre. Certains langages sont dit multi-paradigm, ce qui signifie qu'on peut les utiliser avec un ou plusieurs paradigmes comme le Javascript ou le Python. Nous pouvons identifier 3 paradigmes de programmation principaux.

Programmation procédurale

Premièrement la programmation procédurale, comme décrite dans l'article *Programmation procédurale - Définition et Explications de **Techno-Science.net*** est un

paradigme de développement basé sur l'utilisation des procédures. Une procédure est une suite d'instruction ordonnée, « incluant d'autres procédures, voire la procédure elle-même », nous parlons alors de récursivité. C'est un paradigme très commun dans les langages C ou Go par exemple.

Programmation orientée objet

Ensuite nous avons la programmation orientée objet souvent abrégée en OOP (Object-Oriented Programming) qui place l'objet au centre. Un objet est une brique logique représentant « un concept, une idée ou toute entité du monde physique, comme une voiture, une personne ou encore une page d'un livre. » *Programmation orientée objet - Définition et Explications, **Techno-Science.net***. Nous retrouvons très souvent ce paradigme en Java, en C++ ou en Dart par exemple.

Programmation fonctionnelle

Le paradigme de programmation fonctionnelle, décrit entre autres dans l'article *Programmation fonctionnelle - Définition et Explications de **Techno-Science.net*** dans lequel l'élément central est la fonction qui ici reprend la logique d'une des fonctions mathématiques. Il impose le principe d'immutabilité et rejette les changements d'état. La programmation fonctionnelle a été imaginée avant même le début de la programmation elle-même et est fortement basée par le lambda-calcul inventé par Alonzo Church dans les années 1930 comme décrit par **Robert C. Martin** dans le *chapitre 6 de Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Nous retrouvons ce paradigme dans des langages comme Clojure ou Haskell.

Dans ce contexte de nécessité d'évolution constante de nos applications, nous nous demanderons quels éléments permettent d'assurer la maintenabilité au long terme d'une application.

REVUE DE LA LITTÉRATURE

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.

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 (**Robert 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.”

Object Oriented Software Construction, Bertrand Meyer, 1988, chapitre 4

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. (*Clean Architecture: A Craftsman's Guide to Software Structure and Design, Robert 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). (*Do the SOLID principles apply to Functional Programming, 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. (**Robert 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 :

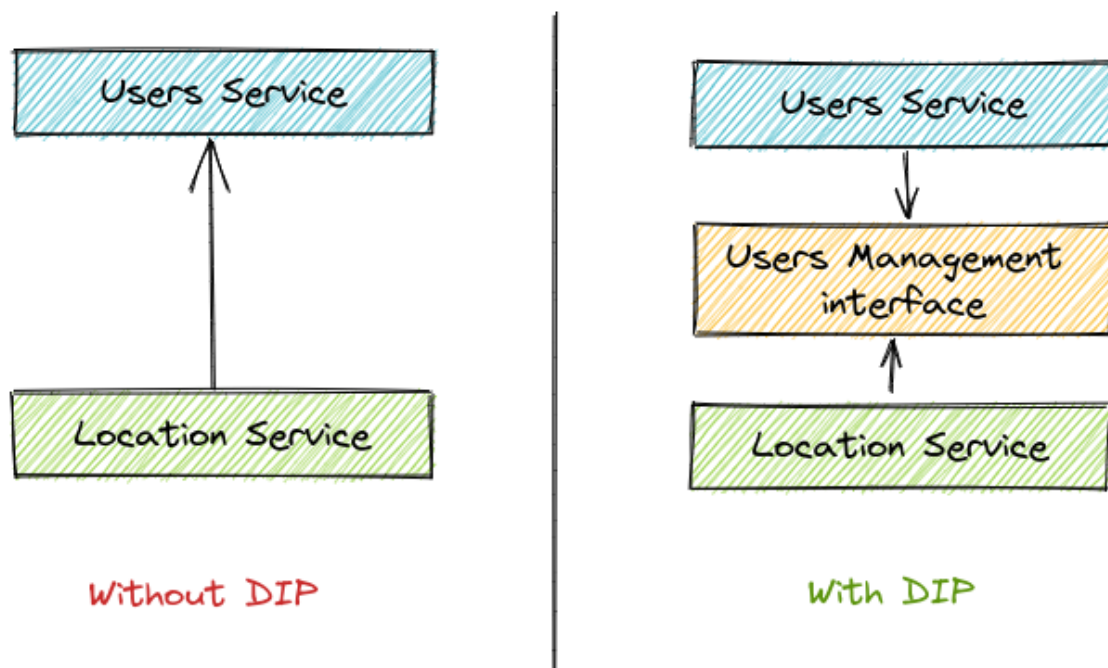


fig.1 - Exemple d'utilisation du Dependency Inversion Principle

Dans l'exemple de fig.1, 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".

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: A Handbook of Agile Software Craftsmanship* de **Robert C. Martin** traite un grand nombre de sujets mais nous prendrons le nommage des entités comme exemple car c'est le plus générique.

Le deuxième chapitre de l'ouvrage est concentré sur le bon nommage des entités dans le code. Ici quand il est question d'entité, il est question de 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 rôle de l'entité et ne pas porter à confusion
- Deux noms doivent être distinguables 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 entités. 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 de *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

Clean Code: A Handbook of Agile Software Craftsmanship, **Robert C. Martin**, 2008, Chapitre 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.

Anti-pattern : Les interfaces de classe

D'après **Scott W. Ambler** dans *Process patterns: building large-scale systems using object technology*, un *anti-pattern* est une solution à un problème récurrent 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 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.

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 en cours d'exécution. 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 orientée 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 l'attribut `age`. En Java, cela pourrait donner :

```

class User {
    private String name;
    private int age;

    public void birthday() {
        System.out.println("Happy Birthday " + name + " !");
        this.age += 1;
    }
}

```

fig.2 - Exemple de classe avec setter en Java

La situation présentée en fig.2 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 **birthday** n'est pas un *mutateur*.

Un *mutateur* ou *setter* en programmation orientée objet est une méthode dont le rôle 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*.

Ici **birthday** n'est pas un *mutateur* or il modifie l'attribut **age**. La méthode contient donc un effet de bord.

Le problème engendré par les effets de bord est qu'il est impossible pour l'utilisateur de la méthode de savoir ce qu'elle fait à moins de regarder le code de la méthode.

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* :

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.

It's probably time to stop recommending Clean Code, Qntm, 2020

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.

Robert C. Martin, 2016

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.

Architecture hexagonale

Nous avons abordé de nombreux éléments concernant le code tel qu'il est écrit, cependant une grande partie de la réflexion autour de la création d'une base de code concerne son architecture.

Le concept d'architecture hexagonal provient l'article *Hexagonal architecture* écrit en 2005 par **Alistair Cockburn**. Il commence par décrire le problème initial à savoir le déplacement de code métier dans des parties de l'application qui ne devraient

pas en avoir comme l'interface utilisateur. À cette solution, l'auteur propose d'isoler toute la logique métier d'une application du reste des éléments d'une application.

Architecture

La partie contenant la logique métier de l'application est appelée "Application" dans l'article mais le terme de "noyau de l'application" ("Application Core") existe aussi dans d'autres articles et évite ainsi une confusion entre l'application au global et l'application au sens de logique métier. Ce noyau de l'application communique avec le reste de l'application via des interfaces appelées "Ports". Chaque port peut être implémenté par des "Adapter".

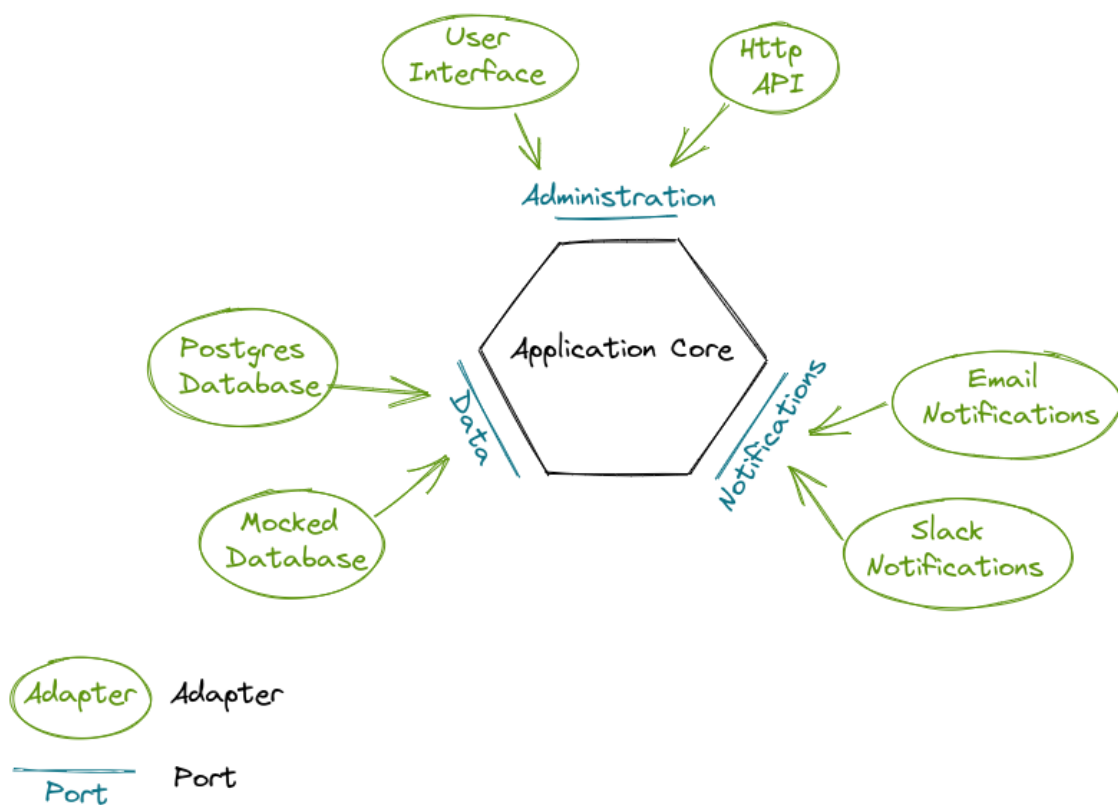


Fig.3 Exemple d'application utilisant une architecture hexagonale

Dans la figure ci-dessus (fig.3), notre application est composée d'un noyau contenant la logique métier de notre application, de trois ports concernant une couche de gestion de la donnée, des notifications et de l'administration de l'application. Chaque

port a ici deux adapters implémentant la même interface. Ces adapters sont interchangeables car leurs comportements ne sont pas connus du noyau de l'application.

Adapters

Le terme d'"Adapter" n'est pas anodin, il fait directement référence au design pattern du même nom.

“Convert the interface of a class into another interface clients expect.” The ports-and-adapters pattern is a particular use of the “Adapter” pattern.

Hexagonal architecture, Alistair Cockburn, 2005

D'après l'article concernant les Adapters sur le site Refactoring Guru, il décrit un objet dont le rôle est de faire communiquer deux interfaces incompatibles. L'exemple évoqué dans l'article présente une situation dans laquelle l'application récupère des données en XML et doit les fournir à une librairie sous format JSON. Les langages JSON et XML n'étant pas compatibles, il est nécessaire d'effectuer une conversion de l'XML vers le JSON, c'est le rôle de l'Adapter.

Dans le contexte d'une architecture hexagonale, l'Adapter va interfacer le service externe avec le Port correspondant. Par exemple, interfacer une base de données Postgres à un Port dédié à la gestion de la donnée.

Inspirations

L'article d'**Alistair Cockburn** peut directement être mis en parallèle avec les principes SOLID décrits par entre autres **Robert C. Martin** et notamment le principe d'inversion de dépendance. En effet ce principe est cité par l'auteur dans les références et il ajoute même une courte partie dédiée à explique le lien

Bob Martin's Dependency Inversion Principle (also called Dependency Injection by Martin Fowler) states that “High-level modules should not depend on

low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.” The “Dependency Injection” pattern by Martin Fowler gives some implementations. These show how to create swappable secondary actor adapters.

Hexagonal architecture, Alistair Cockburn, 2005

Dans cette citation, l'auteur explique que l'utilisation du principe d'inversion de dépendance permet de créer des Adapters remplaçables dans notre application.

Au-delà du code

Nous avons abordé différentes méthodes et points d'attention concernant les bonnes pratiques d'écriture de code dans la maintenabilité d'un projet applicatif cependant la maintenabilité d'un projet ne s'y limite pas.

You build it, you run it

Giving developers operational responsibilities has greatly enhanced the quality of the services, both from a customer and a technology point of view. The traditional model is that you take your software to the wall that separates development and operations and throw it over and then forget about it. Not at Amazon. You build it, you run it. This brings developers into contact with the day-to-day operation of their software. It also brings them into day-to-day contact with the customer. This customer feedback loop is essential for improving the quality of the service.

Werner Vogels, 2006

C'est lors d'une interview de **Werner Vogles en 2006**, CTO d' Amazon que né l'expression "You build it, you run it", un modèle de développement et de maintenance

d'application dans lequel les développeur de l'application sont les mêmes développeurs qui la maintiennent (*Is 'you build it, you run it' living up to the hype?*, **Atlassian**). Le développement d'une application est souvent composé de deux parties, la partie développement ou le "build" et la partie maintenance ou le "run".

La notion de maintenance d'application peut être découpée en deux parties. D'un côté la maintenance applicative et d'un autre la partie opérationnelle comportant entre autres la partie déploiements de l'application et santé de l'application en production. Ici le model "You build it, you run it" concerne essentiellement la partie opérationnelle de la maintenance.

L'avantage de cette méthode est que les développeurs étant fortement impliqués dans la maintenance de l'application et notamment pendant le moment d'astreinte, ils font plus attention à la robustesse de l'application. Le développeur ayant créé l'application est de plus la personne la plus à même de résoudre ses problèmes en production. Et à l'inverse avoir la connaissance sur la méthode de déploiement de l'application permet d'anticiper dès la conception les problématiques éventuelles de production.

Cependant l'application de ce modèle est parfois complexe dans certaines équipes car cela implique de changer leurs structures et les modes de communications. (*Is 'you build it, you run it' living up to the hype?*, **Atlassian**).

Normes et définitions

La *norme ISO 25010* définit la maintenabilité comme étant le degré d'efficacité et de rendement dans l'évolution d'un produit ou d'un système dans un objectif d'amélioration, de correction ou d'adaptation aux changements d'environnement, et de besoin. Cinq éléments sont listés pour caractériser de la maintenabilité :

- **La modularité** : capacité d'un système ou d'un programme à être composé de composants de tel sorte que des changements apportés à un composant aient un impact minimal sur les autres composants.
- **La réutilisabilité** : capacité d'un élément à être utilisable dans plus d'un système
- **L'analysabilité** : degré d'efficacité et de rendement
 - de l'analyse de l'impact d'une modification sur une ou plusieurs parties d'un système.
 - du diagnostic des causes des problèmes rencontrés par le système.
 - de l'identification des éléments de l'application à modifier.
- **La modifiabilité** : capacité d'un produit ou d'un système à être aisément modifiable sans introduire de déficiences ou de régressions
- **La testabilité** : aisance à établir des cas de test d'un système et à leur mise en place

Cette norme ISO répond aux attentes déjà évoquées dans les parties précédentes, mais il reste à voir la problématique suivante : Quels éléments permettent de favoriser la maintenabilité à long terme d'une application ?

En effet, si la norme fixe un cadre, la gestion des détails permet une plus grande latitude au professionnel. Il sera donc de la responsabilité du professionnel de s'assurer que son application est pérenne sur le long terme, et surtout qu'elle puisse être reprise par d'autres personnes sans avoir besoin d'analyser toutes les lignes du code ou d'avoir connaissance de l'historique du projet.

MÉTHODOLOGIE DE RECUEIL DES DONNÉES

Une des problématique engendrée par les questions de maintenance est la difficulté d'avoir un résultat chiffré de la qualité de maintenance d'une application. Certains outils comme Sonar Cloud proposent une note de maintenabilité en se basant sur les "Code Smells" (les mauvaises pratiques identifiées par l'outil) et le temps nécessaire à leurs résolutions. Cependant la notion de "Code Smell" a été ajoutée arbitrairement par l'outil et certains cas sont assez largement débatables.

Il est donc complexe d'obtenir une analyse quantitative de ce qui favorise la maintenabilité d'une application. Je me suis donc dirigé vers une approche qualitative avec des entretiens destinés à établir des éléments récurrents entre différentes équipes ayant favorisé ou non la maintenance de leur application.

Cible des entretiens

J'ai privilégié des profils de Leader Techniques ayant pu travailler dans différentes équipes et différentes entreprises afin qu'ils puissent voir un maximum de projets et d'organisations différentes et donc avoir un bon recul sur le sujet. J'ai également privilégié des consultants en ESN pour ces mêmes raisons. Étant moi-même en ESN, j'ai pu profiter de mes connaissances afin d'obtenir les entretiens.

Déroulé des entretiens

Introduction

Quels sont pour vous les éléments les plus importants quand vous développez/travaillez sur une application ?

Afin d'introduire l'entretien, une première question générale est posée afin de démarrer le questionnement et de noter les différents points qui pourront être utilisés par la suite.

Les applications dites legacy

Les applications legacy (ou héritées) sont des applications informatiques n'étant plus destinées à être utilisées dans un système soit suite à une perte d'un besoin soit à la suite d'un remplacement.

Le legacy faisant partie intégrante du métier de nombreux développeurs et étant le signe de la fin de maintenance d'une application, il est souvent lié à un problème dans cette dite maintenance. Il est donc intéressant d'en tirer des leçons à appliquer ou non sur de nouveaux projets afin qu'ils soient utilisés à plus long terme.

Est ce que vous avez en tête une expérience marquante en lien avec la maintenance d'une application ?

Ici, il est question d'ouvrir cette partie sur le legacy en évoquant un événement marquant en lien avec la maintenance d'une application et de commencer à s'interroger ce qui aurait pu être différent pour améliorer l'efficacité de cette maintenance.

Quand vous travaillez sur de vieilles applications ou que vous reprenez le code d'une autre équipe, quels éléments ont tendance à complexifier votre travail de reprise ? Quels éléments auriez-vous mis en place pendant le développement afin d'éviter ces problèmes ?

Cette question permet d'ouvrir la discussion sur des éléments récurrents retrouvés dans du code legacy afin de déterminer leur impact sur la maintenabilité et des moyens de les éviter si besoin.

Clean Code

Connaissez-vous Clean-Code de Robert C. Martin ?

Cette question vise à établir si l'interlocuteur est sensibilisé aux ouvrages de Robert C. Martin qui sont très largement cités sur le sujet. En cas de réponse négative, nous pouvons directement passer à la question suivante.

Agilité

Quels corps de métiers sont selon vous responsables de la maintenance d'une application ?

Cette question permet d'introduire la partie agile dans laquelle le rôle des membres non-développeurs de l'équipe sont impliqués. J'attends des résultats comme Développeur, Quality Analyst, Product Owner ou DevOps mais c'est aussi l'occasion d'explorer le rôle d'autres métiers.

*Êtes-vous familier des méthodes agiles ? (Framework Scrum, Kanban)
Comment se déroule habituellement la gestion de projet dans vos équipes ?
Pensez-vous que les méthodes agiles sont adaptées à la maintenance d'une application ? Pourquoi ?*

Ces questions permettent d'établir un profil type d'équipe et d'ouvrir la discussion sur les limitations éventuelles de l'organisation sur la maintenance applicative ou au contraire des atouts qu'elle peut apporter.

You build it, you run it

Connaissez-vous le model "You build it, you run it" ? Si oui, l'appliquez-vous ?

Cette partie de l'interview se concentre sur la méthodologie You build it, you run it et le rôle des développeurs dans la maintenance applicative. Bien que l'on puisse différencier la maintenance applicative concernant les évolutions des applications de la maintenance opérationnelle qui sera plus en lien avec le suivi et le monitoring de l'application en production; ces deux parties sont très souvent liées.

En effet, un comportement non-souhaité présent dans une application peut avoir un impact sur son comportement en production nécessitant plus de maintenance opérationnelle. Une application bien maintenue permet de réduire ces événements.

Dans le cas où la personne interviewée ne connaîtrait pas le sujet, je lui présente en décrivant le système dans lequel l'équipe de développement d'une application est aussi responsable de son état en production.

Que pensez-vous de l'implication des développeurs dans les phases de maintenance applicative et opérationnelle d'un projet ?

Cette question permet d'avoir un retour d'expérience pour cette thématique et d'ouvrir la discussion sur le sujet.

ANALYSE DES RÉSULTATS

Deux interviews ont été réalisées dans le cadre de ce mémoire suivant le protocole exprimé précédemment. La première avec Sylvain Lavazais, Technical Leader chez Davidson Consulting, il a beaucoup d'expérience en développement d'application, notamment autour de l'écosystème JAVA. Le deuxième avec Jules Spicht, également Technical Leader à Davidson Consulting.

La documentation

La documentation a été évoquée à multiple reprise dans les deux interviews. Présenté comme point central dès l'introduction. Elle permet aussi de communiquer sur le projet de permettre la passation aux futurs membres de l'équipe. Jules Spicht précise toutefois que la documentation peut se trouver aussi bien dans le projet qu'à l'extérieur en fonction de la pertinence et de la cible de l'information documentée.

L'équipe

Une partie importante des interviews concerne la part de la maintenance applicative dans la vie de l'équipe. Sylvain Lavazais souligne qu'une des sources de ce qui rend un projet "legacy" est parfois le manque d'accompagnement des jeunes développeurs sur un projet. Dans les deux cas, chaque membre de l'équipe est considéré comme important dans le travail de maintenance. Que ce soit le Product Owner (ou PO), comme expliqué par Sylvain Lavazais par exemple, qui a pour charge d'identifier les sources de problèmes et de partager au mieux la vision du projet à l'équipe afin d'éviter les erreurs dues à l'incompréhension des demandée par les développeurs. Ou bien les développeurs qui concrétisent la vision du Product Owner à travers la réalisation des différentes tâches du projet.

Le rôle du QA, "Quality Analyst" ou "Quality Agent" en fonction des sources, a été beaucoup mentionné par Sylvain Lavazais au cours de son interview. Ce corps de métier est très présent dans les grandes entreprises, notamment Décathlon qu'il cite. Le rôle du QA est d'assurer la qualité autour d'une application, d'après Sylvain Lavazais, les équipes ont tendance à trop se reposer sur les QA alors que leur travail pourrait être

effectué plus efficacement par les développeurs. Ainsi, ce rôle est plutôt présenté comme celui d'un accompagnant de début de projet dont la finalité est de disparaître, une fois l'équipe autonome niveau qualité.

L'agilité comme point central de la gestion de projet informatique moderne, elle a bien sûr été mentionnée par les interviewés. Sylvain Lavazais décrit les deux frameworks principaux de l'agilité à savoir le Scrum et le Kanban mais définit le Kanban comme plus adapté à la maintenance applicative. Dans la vision Scrum, toute l'activité de l'équipe est découpée en périodes de durée fixe appelées "sprint" et au début de chaque sprint on définit les tâches à accomplir. À l'inverse la vision Kanban est de prendre les tâches au fil de l'eau sans avoir cette restriction du sprint. Ceci implique que les différentes tâches sont proposées avec notamment l'utilisation d'une "Fastlane", regroupant le suivi des tâches les plus importantes du projet comme la résolution de failles de sécurité. Le framework Kanban permet plus de réactivité que dans le Scrum ce qui permet d'identifier plus rapidement les problèmes et donc de les résoudre plus rapidement. Néanmoins, le Scrum reste plus répandu que le Kanban dans la plupart des équipes.

Maintenance opérationnelle et maintenance applicative

Étant deux sujets extrêmement liés, il a été demandé aux interviewés de parler du lien entre maintenance opérationnelle et maintenance applicatives. À plusieurs reprises dans les entretiens, l'importance du déploiement continu a été demandée. Cela fait partie de la "CI/CD" (Continuous Integration, Continuous deployment) qui décrit toute l'automatisation entre le moment où le code a été envoyé vers un gestionnaire de version comme Git au moment où il est déployé en production. Le déploiement continu est d'ailleurs un des premiers éléments évoqué par Jules Spicht dans son interview.

Sylvain Lavazais a parlé du programme "Accelerate" créé par Google pour identifier les écarts de performance entre différentes équipes via un questionnaire. Ce programme se base essentiellement sur le temps nécessaire au développement d'une fonctionnalité ainsi que du temps de résolution d'un problème présent en production. Le déploiement continu permet de grandement favoriser ces deux points en réduisant au

maximum le temps entre la résolution d'une tâche et son déploiement en production, permettant ainsi aux équipes de déployer leurs applications plusieurs fois par semaine voire même par jour.

La méthodologie "You build it, you run it" décrit un système dans lequel les développeurs s'occupent aussi de la maintenance opérationnelle. Cela peut se faire par exemple par la mise en place d'un déploiement continu comme vu précédemment. Sylvain Lavazais décrit cette méthodologie comme plus stimulante pour les développeurs qui sont ainsi forcés à s'impliquer dans ce que va devenir leur code en production. Jules Spicht ajoute à ces éléments que les problèmes en production sont plus faciles à résoudre, car les développeurs s'occupant de la production sont les mêmes qui pourront résoudre le problème, qu'il soit applicatif ou non. Même si Sylvain Lavazais comme Jules Spicht déclarent que ce n'est pas appliqué dans la majorité des entreprises du Nord de la France, c'est tout de même appliqué dans de grosses entreprises comme Adéo ou Norauto. Cette méthodologie est aussi intéressante dans l'autre sens car les développeurs connaissent mieux que quiconque leurs applications et donc sont plus à même de connaître leurs besoins en production comme décrit par Jules Spicht.

Une rigueur automatisée

La notion de rigueur s'est imposée comme point central de la maintenabilité applicative. Les interviewés ont décrit cette rigueur comme centrale afin d'éviter que l'application ne devienne "legacy". Les points les plus importants évoqués par les deux participants sont:

- Les tests et plus généralement la couverture de code permettant à la fois d'éviter les régressions dans le code mais aussi de simplifier la reprise du code par les nouveaux développeurs car une lecture des tests unitaires ou End-to-end d'une application permet souvent de mieux comprendre le comportement et le rôle des différents éléments de code.
- L'utilisation d'un outils de linting (comme Sonar Cube) pour assurer le respect des convention et de maintenir une homogénéité dans les règles utilisées dans l'écriture du code

- Le suivi d'un guide comme Clean Code, à utiliser comme référence pour trouver des pistes de résolution de problème et d'homogénéiser les bonnes pratiques de développement.
- Utiliser un outil de versioning comme Git
- La mise en place d'une CI/CD si possible
- Penser à l'architecture de l'application et savoir comment l'application va se mettre à l'échelle si elle est fortement utilisée.

À ces différents outils, Sylvains Lavazais en ajoute :

- Le besoin d'une convention de commit. Cela permet d'avoir un historique de commit plus lisible et d'ainsi pouvoir automatiser la gestion de version de l'application, toujours en lien avec le programme Accelerate, évoqué dans la partie sur "You build-it, you run it".
- La mise en place d'une Definition of Done permettant de définir les éléments nécessaires à la validation d'une tâche.
- La mise en place d'une Definition of Ready définissant les éléments nécessaires à valider pour qu'une tâche puisse être effectuée par un développeur.
- La mise en place d'une convention de co-travail permettant de définir comment les membres de l'équipe vont travailler ensemble. Par exemple, si une tâche ne doit être effectuée que par une seule personne ou si du Peer-Programming peut être effectué dans certains cas.

Enfin Jules Spicht ajoute d'autres pratiques :

- Avoir un référent technique connaissant bien la stack technique du projet, permettant ainsi de mieux anticiper le travail nécessaire aux différentes tâches.
- Avoir des outils de monitoring afin de contrôler l'application en production.

Bien que ces pratiques permettent de favoriser la rigueur, il ne faut pas aller dans l'extrême. Par exemple, Jules Spicht prend l'exemple de "Git Hooks", ce sont des scripts qui se lancent à chaque "Commit" sur l'ordinateur du développeur. Ils servent généralement à faire une vérification du bon fonctionnement des tests, de l'application du linter ou de la forme du message de commit. Le problème c'est que le développeur

peut se sentir frustré, de ne pas pouvoir faire ce qu'il veut en local. Il faut donc laisser une marge de manœuvre pour que le développeur soit rigoureux sans pour autant affecter ses performances.

Sylvain Lavazais ajoute la notion de "rigueur automatisée". La rigueur ne devrait pas être contrôlée par les autres développeurs mais par des automatisations. Ainsi, le développeur peut avoir un résultat plus fiable plus rapidement et donc être proactif si besoin. Cette automatisation peut se faire avec des outils de CI/CD comme GitHub Actions ou bien CircleCI.

Conception de l'application

Un point revenu à multiples reprises dans les deux interviews est l'importance de l'architecture des applications. Sylvain Lavazais parle d'homogénéité dans les choix d'architecture d'un projet. Jules Spicht considère que l'architecture est à réfléchir avant même le début des développements. Pris au sens large dans son entretien, l'architecture désigne ici aussi bien la structure de l'application, que la manière dont elle va être déployée mais aussi la façon dont elle va communiquer avec les applications. Il précise l'importance de juger la pertinence d'une application au global et de son intégration avec les autres applications de son écosystème afin d'éviter les redites. Cette question peut parfois amener à se poser la question du découpage de l'application en plusieurs micro-services la rendant ainsi accessible aux autres applications ayant besoin de ses fonctionnalités. Il a également parlé de la manière dont l'application doit être mise à l'échelle, en effet une application recevant beaucoup de requête se doit de se mettre à l'échelle et si l'architecture ne permet pas de le faire assez efficacement, des développements supplémentaires vont être nécessaires et donc affecter la maintenance applicative.

DISCUSSION

Les résultats précédemment présentés permettent d'identifier divers domaines permettant de répondre à la problématique.

La documentation

Largement mentionnée durant les entretiens, la documentation d'un projet fait partie des piliers les plus importants dans la réalisation d'une application. Il est important de distinguer les différents types de documentation et notamment la documentation utilisateur de la documentation technique car elles doivent être pensées différemment. Dans le contexte de la maintenance applicative, la documentation technique est celle qui aura le plus d'impact. Elle doit permettre à un développeur ayant peu ou pas de connaissance sur le projet de commencer à développer dans de bonnes conditions en comprenant les choix architecturaux et de technologies qui ont été réalisés. De plus, cette documentation peut recenser les schémas d'architecture de l'application une fois déployée, permettant ainsi de la même manière de faciliter le déploiement et donc de gagner du temps.

Comme expliqué par Jules Spicht durant son entretien, cette documentation peut aussi bien se trouver dans le code source qu'à l'extérieur. J'ajouterais néanmoins l'importance d'avoir une homogénéité de gestion de la documentation entre les différents projets d'une organisation. Cela permet de repérer plus les similarités et différences entre les différents projets et de profiter de l'expérience acquise durant le développement d'un autre projet sans nécessairement avoir de développeurs en commun. Ainsi, si une équipe développe une fonctionnalité et la documente, une autre équipe souhaitant développer une fonctionnalité similaire pourra s'inspirer de l'expérience de la première équipe pour ne pas reproduire les mêmes erreurs voire apporter des suggestions ou des questionnements de cette équipe. C'est d'autant plus vrai concernant la documentation de l'architecture de l'application déployée car dans une organisation, une homogénéité des méthodes de déploiement est souvent privilégiée afin de simplifier la maintenance opérationnelle.

L'équipe

L'équipe de développement et son organisation a également été un sujet mis en évidence. Afin de créer une application facilement maintenable, il est primordial que son organisation favorise ces principes :

- Favoriser la communication entre le Product Owner et le reste de l'équipe. Un des rôles principaux du Product Owner, comme le dit Sylvain Lavazais dans son interview, est de concrétiser le besoin client et de le transmettre au reste de l'équipe de développement. Si dans cette transmission, le besoin est altéré, la production de l'équipe ne sera pas en accord avec la volonté du Product Owner et du Client, des bugs peuvent alors être créés nécessitant donc une plus grande maintenance.
- Donner du temps aux développeurs. Développer en favorisant la qualité de l'application est un processus chronophage ne devant pas être négligé. Il est hélas courant de voir des projets où l'on considère une tâche terminée à partir du moment où elle est remplie fonctionnellement. Afin de favoriser la maintenance applicative, une tâche doit être prise dans son intégralité. Cela comporte sa réalisation fonctionnelle mais aussi l'application de toutes les contraintes que l'équipe a définie (dans une Definition Of Ready, par exemple)
- Penser l'organisation sur la durée. La maintenance d'une application est un élément à prendre en compte dès le début du projet pour des effets souvent visibles après beaucoup de temps. Il est de l'affaire de chaque membre de l'équipe de penser au futur de l'application et une application maintenable est une application qui a moins de chance d'être refaite de zéro dans quelques années.

Comme évoqué plus haut, la mise en place d'une Definition Of Done est un bon moyen de définir les éléments nécessaires pour considérer une tâche comme terminée. Cette charte peut comprendre aussi bien des éléments de qualité comme la réalisation de tests unitaires mais aussi des tâches de documentation. Il est important qu'elle soit connue, comprise et acceptée par l'ensemble de l'équipe afin de favoriser sa mise en place et son application. Il est cependant important, comme Jules Spicht et Sylvain Lavazais le mentionnent, de ne pas trop restreindre les développeurs. Les éléments

composants la Definition Of Done doivent laisser suffisamment de marge pour les développeurs afin qu'ils ne se sentent pas frustrés et qu'ils puissent être créatifs. L'exemple des hooks de pre-commit mentionné par Jules Spicht représente particulièrement cette situation dans laquelle, étant trop contraints, les développeurs désactivent les outils censés améliorer leur travail.

L'agilité au centre du développement moderne, c'est un outil très utile pour favoriser la maintenance de nos applications. Les deux frameworks agiles les plus présents sont le Scrum et le Kanban. Même si le Scrum est bien plus répandu et est le framework de référence quand il est question d'agilité, le framework Kanban semble être bien plus adapté pour favoriser une bonne maintenance applicative. En effet, le Scrum oriente le développement autour de différents sprints d'un temps défini. Au début de ce sprint, la composition de l'équipe de développement ainsi que les tâches sont définies.

Les développeurs sont donc limités à cette période de temps pour effectuer leurs tâches et bien souvent les rôles déterminant la finalité d'une tâche n'ont pas la vision technique du produit et donc se basent sur la partie purement fonctionnelle. Un cas que j'observe régulièrement est celui où un développeur doit développer une fonctionnalité, la fin du sprint arrive, il faut donc rendre compte de cette tâche. Étant fonctionnellement réalisée, elle est validée et déployée mais la partie test n'a pas été réalisée. Nous sommes donc dans une situation où la fonctionnalité est déployée mais n'a pas un niveau de qualité suffisant pour être facilement maintenable.

Par la suite, il y a trois possibilités :

- Une tâche dédiée est ajoutée au sprint suivant mais elle n'apporte pas de valeur ajoutée.
- L'équipe décide de ne pas réaliser ces tests dans l'immédiat et de le faire si un développeur n'a plus de tâche à la suite d'un sprint.
- Les tests ne seront pas réalisés tant que le code de la fonctionnalité ne sera pas de nouveau repris.

Pour pallier ça, l'équipe peut avoir la rigueur de ne pas déclarer la tâche comme terminée et de la reporter au second sprint mais cela demande une rigueur quasi-utopique aux équipes car la tâche n'apporte pas plus de valeur ajoutée au produit à la suite du sprint suivant. Un argument souvent évoqué suite à cette situation est la meilleure prévision du temps nécessaire à la réalisation des tâches. Cependant, l'agilité doit répondre aux imprévus d'un projet, hors le moindre imprévu peut décaler le temps de travail d'un développeur qui n'aura donc pas le temps de terminer sa tâche.

Le framework Kanban en revanche n'a pas cette notion de sprint. L'accent n'est plus sur le temps mais sur la priorisation des tâches. Ainsi, un développeur peut consacrer tout le temps nécessaire à sa tâche. Pour la même raison, le kanban est bien plus propice au programme Accelerate qui incite les équipes à déployer leurs applications en production plus souvent. Le framework Scrum impliquant un déploiement en production en fin de sprint, il est restreint par rapport au framework Kanban qui peut permettre un déploiement à la fin de chaque tâche.

Bien sûr, comme précisé par Sylvain Lavazais, il ne faut pas utiliser les frameworks agiles tel quel. Les préceptes suivis par l'équipe doivent s'adapter à leurs besoins. C'est la raison pour laquelle il est souvent question de ScrumBan, mélange entre Scrum et Kanban, pour parler de la gestion agile de certaines équipes. Cette capacité d'adaptation des équipes est clef pour favoriser sa productivité. Chaque équipe a ses problématiques, ses contraintes et sa maturation, c'est pourquoi il n'est pas pertinent que toutes les équipes d'une organisation soit organisée de la même manière point de vue agilité.

Maintenance Opérationnelle

La maintenance opérationnelle et la maintenance applicative sont des thématiques extrêmement proches. En effet, une application difficilement maintenable aura une plus grande chance d'avoir des bugs en production et donc d'impacter la maintenance opérationnelle. Une manière efficace de sensibiliser les développeurs aux enjeux de la maintenance applicative est selon moi l'application de la méthodologie

“You build it, you run it». En suivant cette méthodologie, l’équipe de développement est aussi impliquée dans la maintenance opérationnelle de l’application et notamment dans les phases d’astreinte. C’est la métaphore du bâton et de la carotte, une application difficilement maintenable applicativement créera potentiellement plus de problèmes en production et donc plus de charge aux développeurs. En faisant plus attention à la maintenabilité de l’application, le développeur aura moins de charge de maintenance opérationnelle et sera donc plus disponible.

Une rigueur automatisée

Sûrement le point le plus important de la maintenance d’une application, la rigueur est ce qui permettra de mettre en place tous les éléments favorisant la maintenance et assurera le suivi. Cette rigueur doit permettre de garantir un niveau de qualité suffisant. Pour ce faire, il existe de nombreux outils. Premièrement la rédaction de tests sur l’application. Aussi bien unitaires pour tester les composants de l’application indépendamment du reste de l’application. Mais aussi des tests d’intégration vérifiant le comportement de l’application entière en l’isolant du reste de son écosystème. Et enfin les plus importants, les tests End-To-End (ou e2e) où l’application tourne dans les mêmes conditions que l’environnement de production et où le comportement de l’utilisateur est simulé. Pour garantir une rédaction efficace des tests unitaires, la méthodologie du Test Driven Development (TDD) peut être appliquée. Cette méthodologie contraint les développeurs à écrire leurs tests avant de commencer à développer les fonctionnalités. C’est une méthodologie assez largement détaillée dans *Clean Code* de **Robert C. Martin** mais elle n’impacte pas en tant que telle la maintenabilité de l’application. En effet, si le TDD n’est pas appliqué dans une équipe mais que les tests sont effectués après le développement des fonctionnalités, toujours en étant inclus dans les tâches, le résultat côté maintenance est identique. Il n’est d’ailleurs pas rare de voir des équipes dans lesquelles le Test Driven Development est appliqué par seulement une partie des développeurs, ça n’a alors aucun impact sur le reste du projet.

Afin d’éviter des problèmes dues à de mauvaises pratiques de code, des outils appelé Linter peuvent être mis en place. Ils ont d’ailleurs été largement mentionnés dans

les interviews, en particulier Sonar Cloud qui est fortement implanté dans la région Lilloise. Un Linter est un outil qui va réaliser une analyse de code et remonter les lignes de codes ne respectant pas des règles définies dans la configuration de l'outil. Par exemple, une vérification du bon nommage des variables peut être appliquée. Il existe de nombreux Linter comme Sonar Cloud ou ESLint (pour le JavaScript), certains mêmes assemblent plusieurs Linter pour pouvoir assimiler une plus grande quantité de règles comme Super Linter. En plus de ces outils, il est utile que les développeurs partagent une base de références quant aux bonnes pratiques de développement. C'est là que peuvent intervenir des ouvrages tels que *Clean Code* de **Robert C. Martin** par exemple. Ainsi, devant une problématique, les développeurs ont une référence similaire, permettant d'accomplir leurs tâches de façon plus homogène.

Ce qui la rend réellement cette rigueur efficace c'est son automatisation. Nous avons discuté plus haut de la Definition Of Done et de son rôle dans une équipe de développement. Afin d'assurer l'application des «guidelines» dictées par cette charte, leur vérification peut être vérifiée. L'idée est simplement de «cocher des cases» concernant la position de la tâche par rapport à cette Definition Of Done. Très souvent cela passe par une CI/CD (Continuous Integration, Continuous Deployment). Un CI/CD est un outil qui va la plupart du temps consister en une suite d'étapes liées ou non entre elles s'activant suite à une action du gestionnaire de version utilisé. Typiquement lors du push d'un commit sur un répertoire Git. Cette suite d'action va notamment comporter l'ensemble des vérifications à effectuer comme la validité des tests unitaires, le respect des bonnes pratiques de développement ou la non-régression de la fonctionnalité. Mais aussi la manière dont l'application va être déployée, évitant ainsi toute erreur humaine lors du déploiement de l'application.

Conception de l'application

La maintenance d'une application se prépare dès la conception de l'application. En effet, une mauvaise décision à ce moment peut drastiquement augmenter la charge de maintenance à l'avenir. Un élément évoqué par Jules Spicht pendant son interview concerne la reprise d'un code Legacy avec des technologies dépréciées ou plus maintenues. L'informatique se développant exponentiellement, il est fréquent qu'une

technologie soit mise de côté pour une autre soit par la communauté, soit par les contributeurs. Une application sur une technologie vieillissante risque à terme de ne plus pouvoir être mise à jour, que ce soit suite à une fin de support ou bien à la difficulté de trouver des développeurs utilisant cette technologie. Il est impossible de prédire avec certitude l'avenir d'une technologie tant les facteurs pouvant la faire monter ou non en popularité sont importants. Il est cependant intéressant de s'intéresser à la tendance d'utilisation de la technologie. Des sites web comme npmjs.com permettent de se renseigner sur l'utilisation d'une technologie via l'évolution de son nombre de téléchargement. Cela peut donner un indice sur l'avenir de la technologie et la capacité de trouver des développeurs y étant formés.

L'architecture de l'application au sens interne du code est également un facteur important à prendre en compte. Comme le dit Sylvain Lavazais dans son interview, il est important d'avoir une certaine homogénéité dans le code source d'une application. Son exemple dans lequel l'accès à la base de données d'une application est réalisée via deux méthodes différentes est un exemple typique. Il est fondamental qu'un type d'action soit toujours réalisé de la même manière dans une application. Cela permet de faciliter l'arrivée de nouveaux développeurs sur le projet et de simplifier les diagnostics en cas de bug.

Il est également important que l'application soit pensée dans son contexte et donc par rapport aux autres applications de son organisation. Par exemple, si elle doit aborder des fonctionnalités similaires à d'autres applications, il est sûrement plus pertinent d'utiliser cette application pour cette fonctionnalité. De la même manière, si une fonctionnalité de l'application peut être utile pour d'autres applications, une étude peut amener à traiter cette fonctionnalité dans une application à part et ainsi éviter la duplication de code entre les applications d'une organisation.

CONCLUSION

La maintenance applicative fait partie intégrante du métier de développeur, ce n'est cependant pas toujours la plus simple et la plus agréable. Afin de la simplifier, il est intéressant de se pencher sur les éléments permettant de réduire ou de simplifier la charge de travail.

Nous l'avons vu précédemment, cela demande une rigueur importante impliquant un travail humain de formation et de sensibilisation dans un premier temps. L'humain au cœur de la réalisation d'un projet applicatif, il est le principal concerné du travail de maintenance applicative. C'est à lui de prendre les décisions en se concentrant sur l'avenir de l'application. Dans le cas d'une méthodologie trop frustrante pour lui, il risque de la contourner et d'être contre-productif. C'est pourquoi il est important de penser à l'humain à chaque décision du projet.

Un travail au niveau de l'organisation et de la gestion de l'équipe est aussi important. L'agilité montante dans les entreprises a montré des évolutions quant aux problématiques de maintenance. L'organisation et la gestion de l'équipe se doivent de prendre en compte les problématiques de maintenance dès la conception de l'application. Rendre au plus simple l'intégration et la formation de nouveaux arrivants est capital et du temps doit être consacré afin que l'équipe de développement puisse rester productive et homogène dans sa manière de travailler.

Enfin, un travail d'automatisation est également nécessaire afin d'assurer le respect des bonnes pratiques et de la maintenabilité du code. L'automatisation permet de décharger l'équipe de développement d'une partie du travail de qualité et de renforcer la détection de régression entre les versions d'une application.

Les recherches effectuées dans le cadre de ce mémoire sont limitées à la région Lilloise. Toutefois, dans une autre région où la culture du travail diffère, les problématiques autour de la maintenance d'application pourraient être différentes.

Afin de compléter les différents aspects de ce mémoire, il serait pertinent d'étendre cette étude au niveau international.

BIBLIOGRAPHIE

Définition de l'Internaute : Langage de programmation

<https://www.linternaute.fr/dictionnaire/fr/definition/langage-de-programmation/#:~:text=Un%20langage%20de%20programmation%20est,informatiques%20qui%20applique nt%20ces%20algorithmes>

Programmation procédurale - Définition et Explications de Techno-Science.net

<https://www.techno-science.net/definition/11446.html>

Programmation orientée objet - Définition et Explications de Techno-Sciences.net

<https://www.techno-science.net/glossaire-definition/Programmation-orientee-objet.html>

Programmation fonctionnelle - Définition et Explications de Techno-Science.net

<https://www.techno-science.net/glossaire-definition/Programmation-fonctionnelle.html>

Clean Architecture: A Craftsman's Guide to Software Structure and Design, Robert C. Martin (2012)

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

Object Oriented Software Construction, Bertrand Meyer, Prentice Hall, 1988, chapitre 4

Do the SOLID principles apply to Functional Programming, Patricio Ferraggi - 2020

<https://dev.to/patferraggi/do-the-solid-principles-apply-to-functional-programming-56lm>

Ambler, Scott W. (1998). Process patterns: building large-scale systems using object technology

Explicit interface per class' antipattern, Marek Dec

<https://marekdec.wordpress.com/2011/12/06/explicit-interface-per-class-antipattern>

It's probably time to stop recommending Clean Code, Qntm, 2020

<https://qntm.org/clean>

Hexagonal architecture, Alistair Cockburn, 2005

<https://web.archive.org/web/20180822100852/alistair.cockburn.us/Hexagonal+architecture>

Adapter, Refactoring Guru

<https://refactoring.guru/design-patterns/adapter>

Interview de Werner Vogles en 2006

<https://queue.acm.org/detail.cfm?id=1142065>

Is ‘you build it, you run it’ living up to the hype?, Atlassian

<https://www.atlassian.com/incident-management/devops/you-built-it-you-run-it>

Norme ISO 25010

<https://www.iso.org/fr/standard/35733.html>