Test-driven developement et pair programming

Étienne Frank, Grégory Burri

***Résumé*—Cet article à pour but de présenter les approches *TDD* (*test-driven development*) et *pair-programming* dans le domaine du développement logiciel, de mettre en avant leurs atouts ainsi que de les inscrire dans des méthodologies de développement connues.**

***Index Terms*—TDD, test-driven development, pair-programming.**

I. Introduction

CECI est un test.

II. *TDD*

A. Motivations

Lorsqu'un développeur se met au travail et choisi de réaliser une nouvelle fonctionnalité d'un logiciel il va, en premier lieu, réfléchir comment celle-ci va être implémentée et quels changements il va falloir réaliser dans la structure actuelle du programme. Le développeur va ensuite ajouter ou modifier des types et implémenter les fonctions nécessaires pour répondre au spécifications. Puis, finalement, il va faire en sorte que le tout compile et il va lancer l'application afin de tester que ce qu'il vient de réaliser fonctionne bien.

Les problèmes à cette approche sont multiples. Tout d'abord il y a de forte chance que la personne se soit écarté des spécifications ou ait voulu trop en faire en créant plus de types que nécessaire ou trop de couches d'abstraction. De plus, les tests n'étant effectués qu'à la fin de l'implémentation, il y fort à parier que le résultat ne corresponde pas complètement aux spécifications et que des bugs subsistent. Si certains de ces derniers sont liés au design alors il est probable que des modifications assez importantes vont devoir être réalisé. Si les tests ne sont que manuels, aucun test ne pourra garantir le fonctionnement de la fonctionnalité dans le future de manière automatisé.

* Pas de tests

* implémenter ce qui est vraiment nécessaire, ne pas se perdre * définir en premier ce que l'on veut (le quoi) et pas comment on va le réaliser (le comment)

B. Fonctionnement

* Écrire un test -> il doit être rouge * Écrire l'implémentation minimum -> le test passe au vert * Refactorer l'implémentation

C. Bénéfices

* Documentation de l'utilisation de l'API * Meilleur design * Confiance (au moment d'écrire, par rapport au code déjà écrit) * Oblige à avoir des tests pour toutes les fonctionnalités * Meilleure qualité

* Approche pas forcément toujours nécessaire

D. Correction de bugs

La correction du bug peut aussi suivre une forme s'approchant du *TDD*. Lorsqu'un bug doit être corrigé, l'on va d'abord ajouter un ou plusieurs tests qui vont échoués afin de mettre en évidence le bug. Le code incriminé va ensuite être corrigé afin que le test passe. Une phase de *refactoring* peut

Cette approche est à la fois importante pour la documentation du bug corrigé mais également pour éviter toute régression ultérieure.

E. Intégration avec une approche agile

Les tests unitaires -> s'ajoute un processus d'intégration continue

III. Cas pratique

A. But

Afin d'évaluer ces deux approches de manière concrète nous les avons mis en œuvre à l'aide d'un petit cas pratique. Cet exemple ne se veut en aucun cas exhaustive et ne montre qu'une introduction échelle réduite.

B. Outils utilisés

IV. Conclusion

Annexe A
Proof of the First Zonklar Equation

Appendix one text goes here.

Annexe B

Appendix two text goes here.

Acknowledgment

The authors would like to thank...

Références

[1] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England : Addison-Wesley, 1999.

Michael Shell Biography text here.

John Doe Biography text here.

Jane Doe Biography text here.