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* (*test-driven development*)

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 devoir 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 de cette façon de faire sont multiples. Tout d'abord il y a de forte chance que la personne se soit écartée 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 réalisées. Si les tests ne sont que manuels, il n'y aura rien qui facilite la validation du fonctionnement des fonctionnalités dans le futur.

Pour palier à ces problèmes le développeur peut adopter l'approche *TDD* qui consiste à, dans un premier temps, écrire un test puis, ensuite, à écrire l'implémentation associée. Cela à pour objectif de décrire ce que l'on veut avant de réaliser l'implémentation y répondant exactement. De cette manière l'on écrit uniquement le code nécessaire à faire passer le test et pas plus.

B. Fonctionnement

* Écrire un test -> il doit être rouge * Écrire l'implémentation minimum -> le test passe au vert * Refactorer l'implémentation * Au fure et à mesure que des tests vont être écrit d'autres vont apparaître dans l'esprit du développeur, * Processus organique

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. Pair programming

A. Concept

Le pair programming consiste à écrire du code en étant deux personnes. Une ayant le rôle de conducteur, celui qui écrit les lignes de code. Et l'autre ayant le rôle de navigateur, celui qui doit diriger l'autre. Ces rôles sont échangés à un intervalle plus ou moins régulier. Il faut comprendre que le pair programming s'adapte à ses utilisateurs. Donc il est rare de trouver des règles absolues sur le déroulement.

Cependant la règle qui pourrait être qualifiée la plus importante est celle de l'échange entre les deux personnes. Cet échange permet de synchroniser le groupe. Lorsque vous regardez quelqu'un coder vous aurez tendance à lâcher le fil. Sauf si ce codeur vous indiques chaque chose qu'il va entreprendre et vous demande de vous assurer qu'il va au bon endroit. Très vite une inertie se crée et on aura même tendance à oublier de prendre une pause.

En étant toujours concentrer sur le code on devient vite fatigué. Alors il ne faut s'arrêter 2-4 minutes, penser à autre chose, manger un fruit, ou faire un peu d'exercice. C'est très important et ça rejoint l'aspect de ne pas faire d'heure supplémentaire dans le monde agile. Sinon on arrive trop vite au burnout.

Il faut toujours que les deux personnes sachent où elles vont. Nous venons de le voir sur une courte portée, mais aussi avec une portée plus grande. Lorsqu'on commence à travailler on définit une tâche global qui est notre but (en général elle nous prendra 1h-2h). Le navigateur doit d'autant plus bien garder en mémoire ce but. Si le codeur commence à écrire du code qui n'est pas en rapport avec la tâche finale, alors c'est au navigateur de le lui l'indiquer. Une fois la tâche finale définie, il faut définir une petite tâche qui doit prendre entre 5 à 10 minutes. Cela contribue à garder la synchronisation entre les deux membres.

Si le navigateur remarque que le codeur écrire une fonction non optimale, il ne doit pas déranger tout de suite le conducteur. Il faut laisser au conducteur le temps de finir sa fonction, puis ensuite on lui indique qu'il y a une erreur ou que la fonction n'est pas complète. Cela a pour but de ne pas arrêter en cours de créativité la personne qui code.

Le pair programming coute en heures 15% de plus, et fait 15% de bug en moins[p.86]. Cependant il est fait plus rapidement et les bugs peuvent couter cher. Donc si on regarde non pas juste le cout de développement, mais le cout du programme une fois livré, on remarque que le pair programming est bénéfique. Voici un tableau résumant les couts :

	Individual	**Collaborators**
Hours	2'000 hours	2'300 hours
Developpement Time (T)	2'000 hours (12 months)	1'150 hours (7 months)
Development Cost (I)	$100'000	$115'000
Defect in Field (DF)	293	249
	Discovery bugs by Year: T + Year 1: 169; T + Year 2: 81; T + Year 3: 35	Discovery bugs by Year: T + Year 1: 143; T + Year 2: 68; T + Year 3: 30
Operation Cost(M)	(169*33*40)/1.10 + (81*33*40)/1.102 + (35*33*40)/1.103 = 325,874	(143*33*40)/1.10 + (68*33*40)/1.102 + (30*33*40)/1.103 = 275,534
Discount Rate (d)	10% (or 0.8% monthly)	10% (or 0.8% monthly)
Present Value of Lifetime Costs (PVC)	325874/1.00812 + 100,000 =$396,158	275534/1.0087 + 115,000 =$375,586
Difference		$20,572

Donc dans ce cas on voit qu'il est bénéfique d'utiliser le pair programming. Il faut retenir que cet exemple utilise toujours des moyennes prisent dans des statistiques américaines. Si par exemple votre but n'est pas de maintenir les bugs une fois votre programme fini, le pair programming vous couteras plus cher.

Le pair programming aide également pour l'échange du savoir. Lorsque qu'on regarde quelqu'un coder on voit tout de suite ce qu'on ferrait différemment. Par exemple si votre collègue n'utilise pas un raccourci clavier pratique ou alors qu'il fait des boucles for au lieu des foreach. Parfois ça peut être un détail mais d'autre fois on remarque que l'autre personne a une grosse lacune et elle peut tout de suite apprendre et progresser via vos remarques.

Faire des remarques est très délicat. Lorsqu'on fait du pair programming il faut savoir mettre son égo de coté. Mais il faut aussi savoir faire des remarques constructives et non pas juste rabaisser votre collègue. Certaines personnes se sont déjà fait virer pour un manque de tact.

Pour l'expérience des pairs, nous pouvons avoir 3 cas.

— Junior - Junior
— Senior - Junior
— Senior - Senior

La pair Senior-Junior est celles où le transfert de savoir est la plus importante. S'il n'est pas possible de faire ce type de pair, alors on peut faire les deux restantes. Mais il ne faut pas oublier qu'un junior peut apprendre à un senior. Parfois certain senior rouillent.

B. Varation des pairs

blog.pivotal.io/pivotal-labs/labs/pair-programming-matrix Il est bien de varier les paires. Faire le maximum de combinaisons possibles avec chaque membre de l'équipe. Cela optimise le transfert de savoir de toute l'équipe. Mais souvent avec le temps il y a des affinités qui se créer et on va préférer un collègue plutôt qu'un autre. Une manière de résoudre ce problème et d'avoir une matrice triangulaire. L'on peut y voir rapidement si les pairs sont déséquilibré ou non. Un problème surgit quand l'équipe de développement est trop grande, cela devient un peu trop complexe. Alors c'est un bon signe pour faire des équipes plus petites.

Pour ce qui est de la cadence de rotation, il existe plusieurs avis. Certains disent qu'il faut changer tous les jours, d'autre chaque semaine et voir certain à chaque fin de projet. Il faut trouver la cadence qui convient au mieux à celle de l'équipe.

C. Review programming

Le concept du review programming est de demander à une ou plusieurs personnes de relire votre code et de donner des critiques constructives dessus. C'est une alternative si votre manager n'est pas convaincu par le pair programming. Mais il se peut aussi que les reviewers ne soient pas très implique et ne vous donne qu'un avis général. Le review programming a aussi une sorte de latence. Lorsqu'on fait des critiques après que le code soit écrit, cela peut être moins bien perçu par le codeur. Surtout quand c'est votre patron. Alors qu'en pair programming, on fait les remarques à chaud, ce qui diminue l'impact de « tu n'as pas bien fait les choses ». Un des avantage de review, c'est qu'on a pas besoin d'être dans la même pièce pour faire la review correctement.

Les deux méthodes ont un point faible, la motivation. Si un des développeurs n'est pas du tout intéressé par le pair ou review programming, alors il est difficile d'en sortir

quelque chose de bon.Même si l'initiateur tente de motiver son collègue. Ces techniques sont puissantes grâce à leur aspect social, mais c'est aussi leur point faible.

D. Remote programming

Lorsqu'on travaille sur des projets avec des développeurs qui vivent dans d'autres pays, il est difficile de se retrouver dans la même pièce pour programmer ensemble. La solution est de passer par internet. Il faut un logiciel de voix sur IP(VoIP) et un logiciel de partage d'écran. Ces deux technologies impliquent une bonne connexion internet pour ne pas avoir de latence lorsqu'on code où qu'on regarde l'autre personne coder.

Voici une liste de logiciel VoIP (non exhaustive) :

- Skype
- Hangouts (Google+)
- TeamSpeak
- Mumble

Pour ce qui est du partage d'écran nous avons trois choix possible : partager le bureau et son contrôle, partager juste le bureau et partager juste l'IDE. Les solutions sont de la plus gourmande en connexion à la moins gourmande.

Partage controle	Partage bureau	Partage IDE
Screenhero*	Skype	Cloud9
TeamViewer	Hang outs	Tmux (vim, emacs)
VNC		Floobits (sublimetext, IntelliJ)
		Motepair (Atome)
		Madeye

E. Problème des métriques

https ://github.com/therubymug/hitch

https ://github.com/glg/git-pairing

Le pair programming pose des problèmes lorsqu'on a un environnement qui calcule des métriques. Souvent ces derniers ne sont pas adaptés pour des pairs. Il existe néant moins des programmes communautaires qui tentent de pallier ce problème. Par exemple git-pairing permet d'indiquer qu'elles pairs étaient sur le commit. Cela fonctionne mais il y a peu d'utilisateur donc il faut être attentif si jamais il y a des bugs.

F. Pair programming dans l'éducation

http ://www.realsearchgroup.org/pairlearning/index.php

Deux universités ont fait des statistiques sur leurs étudiants qui commencer dans leurs études. La North Carolina State University (NCSU) et la University of California Santa Cruz (UCSC). L'étude portait sur l'impact du pair programming dans les cours de programmation. Elle s'est déroulée sur 1200 élèves sur une période de 3 ans (bachelor).

Les étudiants ayant fait du pair programming avait de meilleures notes en travaux pratiques et écrits. Il y a eu plus d'étudiants qui réussissaient leur examen de première avec une note de C ou meilleur (équivalent au 5 suisse). Lorsqu'ils étaient en 2 ème année, les étudiants avaient tendance à mieux maintenir voir à améliorer leurs moyennes du cours de programmation de l'année précédente. 77% des étudiants qui avaient fait du pair programming choisissaient en deuxième année la suite du cours(contre 62%). Et avaient tendences à mieux réussir le cours de deuxième année que ceux qui avait programmé seul. Et finalement plus d'étudiants choisissaient l'informatique comme matière principale (NCSU : 57% vs. 34% ; UCSC : 25% vs. 11%).

Finalement le mythe que les professeurs peuvent avoir sur « seulement un seul étudiant apprend lorsqu'il y a du pair programming » semble être faux. Les Etats-Unis manquent de programmeurs, et donc ils ont fait une campagne pour promouvoir la programmation dans les écoles. Cette campagne demandait à beaucoup de célébrités de promouvoir la programmation (Barack Obama, Mark Zuckerberg, Shakira, ...). Et certains professeur trouvait plus ludique pour de jeune écolier d'apprendre la programmation en pair plutôt que seul.

https ://www.youtube.com/watch ?v=vgkahOzFH2Q

V. 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

VI. 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.