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.**

***Mots-clefs*—*TDD*, *test-driven development*, *pair programming*.**

I. Introduction

La réalisation d'un logiciel est toujours une aventure incertaine durant laquelle beaucoup de choix vont devoir être fais, notamment en terme de *design*, qui vont fortement impacter la qualité finale. Nous présentons ici deux approches permettant de réduire cette incertitude et d'augmenter de manière générale la qualité d'un logiciel.

La première approche proposée est le *TDD* (*test-driven development*). Cela consiste à systématiquement écrire des tests unitaires avant l'implémentation correspondante.

La deuxième approche proposée est le *pair programming* qui consiste à développer par paire en alternant la personne qui écrit le code et la personne qui la guide.

Ces deux approches peuvent être utilisées de manière indépendante ou conjointe.

II. *TDD* (*test-driven development*)

A. Motivations

Lorsqu'un développeur se met au travail et choisit 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 aux 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 fortes chances 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 a pour objectif de décrire ce que l'on veut avant de réaliser l'implémentation y répondant exactement. De cette manière, uniquement le code nécessaire à faire passer le test est écrit, et pas plus.

B. Fonctionnement

Voici les trois étapes du processus.

1) Écriture d'un test unitaire, le test doit échouer (être marqué rouge).
2) Réalisation d'une implémentation minimale afin de faire passer le test au vert.
3) *Refactorer* l'implémentation tout en maintenant le test au vert.

Lors de l'écriture du test il faut se focaliser sur l'interface et non anticiper l'implémentation. Il est également important qu'à la première étape, le test ne passe pas afin d'être sûr que ce que l'on veut tester n'existe pas ou ne fonctionne pas encore.

Ce processus est itératif et va être réitéré pour chaque test jusqu'à aboutir à une implémentation complète. Il est possible que des tests écrits précédemment doivent être modifiés ou deviennent obsolètes au fil de la réalisation des fonctionnalités : dans ces cas il ne faudra pas hésiter à les modifier ou à les supprimer.

Afin d'aider le développeur à structurer ce processus il est fortement conseillé de garder à jour une liste des éléments à tester (*todo list*). Avant d'écrire le premier test, cette liste va être peuplée avec tout ce que l'on pense pertinent à tester, puis va être mise à jour après chaque test.

La phase de *refactoring* est très importante et à ne pas négliger, car elle va permettre d'obtenir un design et un code le plus propre possible au fur et à mesure de la réalisation de tests.

Le code exécuté durant les tests ne doit réaliser aucune entrées-sorties comme, par exemple, accéder au système de fichier, au réseau ou à une base de données. Cela a pour but d'obtenir des tests s'exécutant rapidement et ne dépendant pas de données externes. Afin de simuler une ressource externe il est nécessaire d'utiliser des objets *mocks* qui vont posséder un comportement spécifique à ce que l'on souhaite tester et qui vont vérifier comment la ressource est accédée.

Pour finir, précisons qu'il faut faire en sorte que les tests soient indépendants les uns des autres. Si possible, aucun état global ne doit être modifié par un test qui modifierait le comportement d'un autre.

C. Bénéfices

Les bénéfices de l'approche *TDD* sont multiples.

Premièrement cette approche oblige l'écriture systématique de tests pour chacune des fonctions d'une *API*. Ces tests vont permettre, par la suite, d'éviter au maximum toute régression lors de modifications ultérieures. De plus les tests vont également servir à documenter l'utilisation d'une implémentation : cela se révélera d'une très grande valeur lorsque de nouveaux développeurs seront impliqués dans le projet. Les tests déjà présents jouent également un rôle dans la confiance des développeurs, qu'ils soient novices ou seniors. Lorsque ceux-ci sont amenés à réaliser des modifications ou des ajouts, ils pourront à tout moment exécuter l'ensemble des tests et vérifier que les changements apportés ne cassent rien.

Deuxièmement le design du code va directement dépendre de cette approche en l'orientant vers une meilleure modularité et, naturellement, une meilleure testabilité. La qualité générale des composantes d'une application va tendre à s'améliorer. Cette approche a également pour but de coller au plus proche des spécifications en définissant dans un premier temps ce que l'on veut sous la forme de tests, puis comment on résout notre problème via l'implémentation.

D. Correction de bugs

La correction des bugs 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 échouer afin de mettre en évidence le bug. Le code incriminé va ensuite être corrigé afin que le test passe. Une phase de *refactoring* peut également être appliquée après la correction d'un ou plusieurs bugs.

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 sont une pratique courante dans le monde *agile*, il est alors naturel de vouloir intégrer le *TDD* dans le processus de développement. Dans *Scrum*, par exemple, chaque tâche d'un *sprint* va être implémentée en écrivant, dans un premier temps, les tests automatisés. Ceux-ci s'ajouteront au processus d'intégration afin de renforcer la qualité du produit final. Il ne faut pas oublier de prendre en compte l'écriture systématique de tests lors des estimations car ceux-ci peuvent allonger la durée de développement.

III. *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 le conducteur. 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 de plus importante est celle de la communication 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 indique 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 ils auraient même tendance à oublier de prendre une pause.

En étant toujours concentré sur le code, on devient vite fatigué. Alors il faut s'arrêter 2-3 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 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 globale qui est notre but (en général elle nous prendra entre une et deux heures). 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. Puis après chaque petite tâche, il faut inverser les rôles. C'est un pattern qui s'appelle le *Ping Pong pattern* [5]. Il contribue à garder la synchronisation entre les deux membres.

Si le navigateur remarque que le codeur écrit une fonction non optimale, il ne doit pas le déranger tout de suite. 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 l'arrêter en cours de sa créativité. Vous noterez qu'on ne lui dit pas qu'il fait une erreur, mais qu'il y a une erreur. C'est ce genre de subtilité qui permet de conserver une bonne harmonie au sein de la paire.

Le *pair programming* coûte en heure 15% de plus, et réduit de 15% le nombre de bug [6, p. 86]. Cependant le code est écrit plus rapidement. Si l'on regarde toute la vie d'un logiciel, les bugs peuvent coûter cher. Donc si on regarde non pas juste le coût de développement, mais le coût du programme une fois livré et maintenu, on remarque que le *pair programming* est bénéfique. Le tableau I résume les coûts.

Donc dans ce cas on voit qu'il est bénéfique d'utiliser le *pair programming*. Il faut retenir que cet exemple utilise toujours des chiffres pris dans des statistiques américaines. De plus dans certains cas, il serait moins rentable d'en faire. Par exemple si votre but n'est pas de maintenir les bugs une fois votre programme fini, le *pair programming* risque de vous coûter plus cher.

Le *pair programming* aide également pour l'échange du savoir. Lorsqu'on regarde quelqu'un coder, l'on voit tout

TABLE I
TABLEAU DE COMPARAISON DU *pair programming* CONTRE DES DÉVELOPPEURS SEULS [6, P. 71]

	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 Discovery bugs by Year T + Year 1: 169 T + Year 2: 81 T + Year 3: 35	249 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

de suite ce que l'on ferait différemment. Par exemple si votre collègue n'utilise pas un raccourci clavier ou alors qu'il fait des boucles `for` au lieu de boucles `foreach`. Parfois ça peut être un détail, mais d'autres 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 ego de côté. Mais il faut aussi savoir faire des remarques constructives et non pas juste rabaisser votre collègue. Certaines personnes se sont déjà fait licenciées à cause d'un manque de tact.

Pour l'expérience des paires, nous pouvons avoir trois cas :

— junior - junior
— senior - junior
— senior - senior

La paire senior-junior est celle où le transfert de savoir est le plus important. S'il n'est pas possible de faire ce type de paire, alors on peut faire les deux restantes. Mais il ne faut pas oublier qu'un junior peut apprendre à un senior. Parfois certains seniors rouillent et souvent ont de la peine à mettre leur ego de côté.

B. Variation des pairs

Il est bien de varier les paires et 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 créent et on va préférer un collègue plutôt qu'un autre. Une manière de résoudre ce problème est d'avoir une matrice triangulaire : ainsi on peut voir rapidement si les paires sont déséquilibrées ou non. Un problème survient lorsque l'équipe de développement est trop grande. La matrice devient trop complexe et donc illisible : alors c'est une bonne indication pour faire des équipes plus petites [7].

Pour ce qui est de la cadence de rotation, il existe plusieurs avis. Certains disent qu'il faut changer tous les jours, d'autres chaque semaine, voire éventuellement 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 impliqués et ne vous donnent 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 et 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 avantages de *review*, c'est qu'on n'a pas besoin d'être dans la même pièce pour faire la *review* correctement. De plus c'est beaucoup moins chronophage que du *pair programming*.

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 ou que l'on regarde l'autre personne coder.

Voici une liste de logiciels *VoIP* (non exhaustive) :

— *Skype*
— *Hangouts (Google+)*
— *TeamSpeak*
— *Mumble*

Pour ce qui est du partage d'écran nous avons trois choix possibles : partager le bureau et son contrôle, partager

uniquement le bureau et partager juste l'*IDE*. Les solutions sont de la plus gourmande en connexion à la moins gourmande. La table II montre des exemples de logiciels.

E. Problème des métriques

Le *pair programming* pose des problèmes lorsqu'on a un environnement qui calcule des métriques. Souvent ces derniers ne sont pas adaptés pour des paires. Il existe néanmoins des programmes communautaires qui tentent de pallier ce problème. Par exemple *git-pairing* permet d'indiquer quelles paires étaient sur quel commit. Cela fonctionne mais il y a peu d'utilisateurs : il faut donc être attentif si jamais il y a des bugs [8] [9].

F. Pair programming *dans l'éducation*

Deux universités ont fait des statistiques sur leurs étudiants qui commençaient leurs études. La *North Carolina State University (NCSU)* et *l'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* avaient 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 mieux (équivalant au 5 suisse). Lorsqu'ils étaient en 2^{e} année, les étudiants avaient tendance à mieux maintenir, voire 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 ils avaient tendance à mieux réussir le cours de deuxième année que ceux qui avaient programmé seuls. De plus, un plus grand nombre d'étudiants choisissaient l'informatique comme matière principale (NCSU : 57 % vs. 34 % ; UCSC : 25 % vs. 11 %). [10]

Finalement le mythe selon lequel les professeurs pensent que « seulement un étudiant apprend lorsqu'il y a du *pair programming* » semble être faux. Les États-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 (Barak Obama, Mark Zuckerberg, Shakira, etc.). Et certains professeurs trouvaient plus ludique pour les jeunes écoliers d'apprendre la programmation en paire plutôt que tout seul. Cette campagne a été promue par code.org. [11] [12]

IV. 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 exhaustif et ne montre qu'une introduction à échelle réduite.

L'exemple proposé ici est la réalisation d'un panier d'achat électronique faisant partie d'une librairie online. La section suivante liste le périmètre fonctionnel de ce petit composant.

B. Spécifications fonctionnelles

Il doit être possible de créer des livres et de les ajouter ou de les enlever à un panier puis de lister son contenu. L'utilisateur doit pouvoir demander le prix d'un panier et payer celui : dans ce cas le statut du panier passe à *payé* et ne peut plus être modifié.

De plus la gestion des numéros *ISBN-13* pour les livres est requise, et une validation de l'intégrité de ce numéro doit être réalisée.

Un livre comprend les champs suivants :

— Un titre
— Un auteur
— Un prix
— Une année de parution
— Un numéro *ISBN-13*

Le prix doit pouvoir être défini en dollars (*USD*), en francs suisse (*CHF*) ou en bitcoins (*XBT*). Les taux de change peuvent être modifiés de manière globale.

C. Outils utilisés

Nous avons décidé d'utiliser le langage *F#* qui a les caractéristiques d'être à la fois concis et très expressif. Nous avons utilisé le livre *Testing with F#* [2] pour nous guider dans notre démarche.

Le code a été écrit sous l'environnement de développement *MonoDevelop*. Les bibliothèques *FsUnit* et *nUnit* ont été utilisées pour l'écriture des tests unitaires.

La structure du projet comprend un fichier `Tests.fs` pour l'ensemble des tests, un fichier `Cart.fs` pour la gestion du panier, un fichier `Book.fs` décrivant le type d'un livre et d'un numéro *ISBN-13* et un fichier `Currency.fs` pour la gestion des devises.

D. Démarche

Nous avons strictement suivi les deux approches, c'est à dire que nous écrivons d'abord un test unitaire dans le fichier `Tests.fs` puis son implémentation dans un fichier séparé. Concernant le *pair programming*, une personne jouait le rôle du pilote et écrivait un test pendant que l'autre jouait le rôle du navigateur. Puis les rôles étaient inversés, le navigateur devenait le pilote et écrivait l'implémentation du test précédent et le test suivant. Cela faisait changer les rôles toutes les quinze minutes environ.

La figure 1 montre un extrait des tests concernant la gestion des devises. On peut constater que les manières d'écrire les tests s'approchent le plus possible du langage naturel. L'ensemble du code est fourni en annexe.

E. Résultats

La paire étant du type *senior-junior*, le *pair programming* a permis un transfert de compétences en matière de programmation en *F#*. L'approche *TDD* a très bien fonctionné et a donné lieu à une implémentation robuste ainsi qu'à une batterie de tests automatisés. De plus le paradigme fonctionnel de *F#* s'accorde bien avec le *TDD* car celui-ci prône la création de fonctions indépendantes qui améliorent la testabilité et la modularité.

TABLE II
AN EXAMPLE OF A TABLE 2

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

FIGURE 1. Exemple de tests concernant le type `Money`

```
[<TestFixture>]
type ``Currency`` () =
    [<TestFixtureSetUp>]
    member this.SetUp () =
        USD_CHF_rate <- 0.9M
        USD_XBT_rate <- 0.005M

    [<Test>]
    member this.``A money object should be printable`` () =
        Money(42, USD) |> string |> should equal "42 USD"
        Money(42.33M, USD) |> string |> should equal "42.33 USD"

    [<Test>]
    member this.``A money object should be comparable to another money object`` () =
        Money(3, USD) > Money(1, USD) |> should be True
        Money(2, CHF) > Money(2, USD) |> should be True
        Money(1, XBT) <= Money(200, USD) |> should be True
        Money(1, XBT) <= Money(180, CHF) |> should be True
```

V. CONCLUSION

Souvent les approches *TDD* et *pair programming* sont vues comme coûtant plus cher qu'une approche traditionnelle alors que leurs buts sont justement une meilleure maîtrise du design du logiciel. Ceci induit une augmentation de la qualité mais également, *in fine*, une baisse du coût global.

Malgré cela, ces approches ne sont pas évidentes à mettre en place et nécessitent un apprentissage préliminaire. Par exemple, le *pair programming* peut entraîner des frictions entre développeurs et le *TDD* peut sembler peu productif au départ.

Nous encourageons tout développeur à essayer ces approches et à les adapter à sa manière de travailler.

RÉFÉRENCES

[1] Kent Beck, *Test-Driven Development*, 2003.

[2] Mikael Lundin, *Testing with F#*, 2015.

[3] David Heinemeier Hansson, Kent Beck, Martin Fowler, *Is TDD dead ?*, 2014, http://martinfowler.com/articles/is-tdd-dead/.

[4] Mark Seemann, *Look, No Mock ! Functional TDD with F#*, 2015, http://www.infoq.com/presentations/mock-fsharp-tdd.

[5] Example d'un pattern en pair programming, 04.05.2015, http://c2.com/cgi/wiki?PairProgrammingPingPongPattern.

[6] Laurie Ann Williams, *The collaborative software process*, Department of Computer Science University of Utah , Août 2000.

[7] Exemple de matrices pour le *pair programming*, 04.05.2015, http://blog.pivotal.io/pivotal-labs/labs/pair-programming-matrix.

[8] *Hitch* est un logiciel pour faire du *pair programming* avec Git, 04.05.2015, https://github.com/therubymug/hitch.

[9] *Git-pairing* est un logiciel pour faire du *pair programming* avec Git, 04.05.2015, https://github.com/glg/git-pairing.

[10] Étude concernant le *pair programming* dans des universités américaines, 04.05.2015, http://www.realsearchgroup.org/pairlearning/index.php.

[11] Vidéo d'écolières expliquant le *pair programming*, 04.05.2015, https://www.youtube.com/watch?v=vgkahOzFH2Q.

[12] Vidéo de code.org pour promouvoir le *pair programming* via des célébrités, 04.05.2015, https://www.youtube.com/watch?v=FC5FbmsH4fw.