Le TDD comme rempart contre nos biais

Le développement piloté par les tests ou Test Driven Development - TDD - a changé notre relation au code. Profondément. Pour le meilleur. Plus détendus, plus efficaces et plus pertinents, les professionnels que nous sommes devenus le doivent beaucoup à cette manière de coder qui agit comme une force de rappel contre de nombreux biais ou difficultés dont nous sommes parfois victimes en tant que développeurs. Nous allons ici zoomer sur quelques-uns de ces biais qui faisaient partie de notre quotidien de développeur pour pouvoir vous expliquer en quoi la pratique du TDD nous a permis de les dominer.

Bruno BOUCARD - @brunoboucard
À la fois, Coach Agile et Software Craftsman, il enseigne le TDD, le Refactoring, le BDD et DDD. Il est aussi spécialiste sur les technologies Microsoft depuis de nombreuses années. Bruno est aussi président de la société 42 SKILLZ.

Thomas PIERRAIN - @tpierrain
Créateur de la librairie open source NFluent (http://www.n-fluent.net/), architecte technique à la Société Générale et adepte du DDD, des pratiques XP et du Software Craftsmanship, Thomas pratique le TDD depuis plus de 10 ans maintenant.

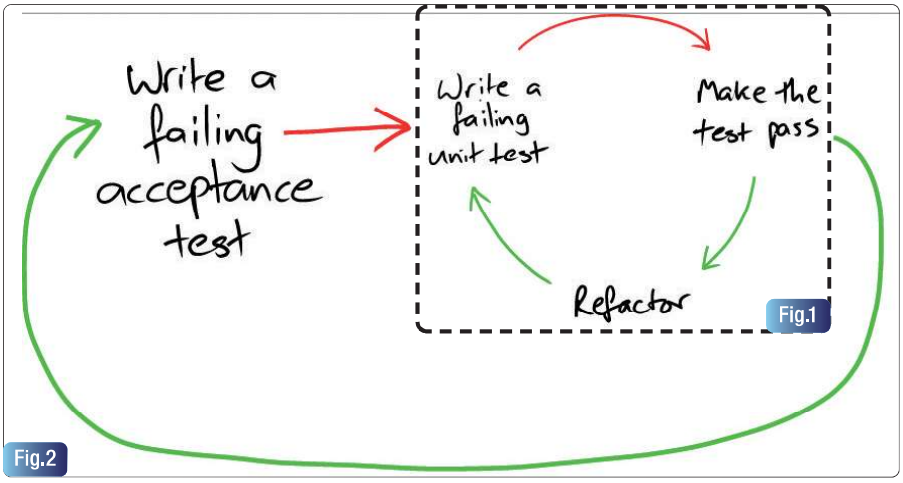


Schéma de la double boucle de l'outside-in TDD (Nat Pryce)

Plus détendus

Thomas : Pour commencer, je dois vous faire une confession : même si je me soigne, j'ai une fâcheuse tendance à la procrastination... Dit autrement, il m'arrive assez souvent de remettre au lendemain ce que je peux faire le jour même. En général cela se produit quand je tombe sur quelque chose qui me dérange (ouvrir des factures, faire les magasins pour les cadeaux de Noël ;-). J'ai alors tendance à tourner autour du pot en m'autorisant parenthèse sur parenthèse, avant de me pencher sur la corvée quand je n'ai vraiment plus le temps d'y échapper. Pour le développeur que j'étais au début de ma carrière, cela se traduisait par de nombreuses digressions que j'opérais lorsque j'étais face à un problème ou à un DEV qui m'apparaissait compliqué à résoudre, compliqué à attaquer. Il m'arrivait alors dans ces moments de tomber dans une situation similaire à l'angoisse de la page blanche : par quel angle dois-je attaquer ce développement ? Qu'est-ce qui est le plus important dans ce contexte ? Ai-je suffisamment fait le tour de la question pour être pertinent dans mon DEV ? Bref, il m'arrivait souvent de tourner autour du pot en googlant sur des sujets connexes et en passant de page en page avant de me retrouver pris par le temps - et d'être contraint d'attaquer le développement, un peu plus stressé du coup (car sans rabe coté délais). Le TDD lui nous décomplexe en nous forçant à attaquer notre développement par **une première fonctionnalité, la plus simple possible** (en fait par le test pour celle-ci). Nous sommes même encouragés à utiliser des valeurs en dur dans les premières étapes de l'implémentation. Cette ambition un peu paralysante de faire les choses au mieux, à la cible dès le début donc, disparaît. Cette façon de développer en commençant par un test et qui nous pousse mécaniquement à faire des petits pas (RED-GREEN-REFACTOR... RED-GREEN-REFACTOR...) me permet du coup de réaliser facilement la chose la plus difficile lorsqu'on est victime de procrastination : COMMEN-CER. Ce premier petit test est tout simple, sans ambition ; il agit sur moi comme un déclencheur. C'est en effet le petit coup derrière l'épaule dont j'ai besoin pour commencer et me mettre en mouvement (oui parce qu'une fois qu'on est lancé il n'y a plus de procrastination qui tienne, c'est le principe).

Bruno : Un autre intérêt de cette approche test-first, c'est le caractère motivant du workflow RED-GREEN-REFACTOR. Pour rappel :

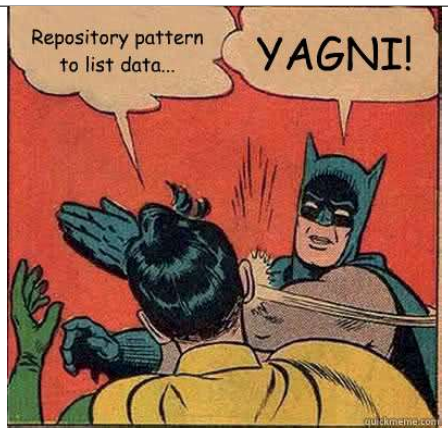
- **RED**: c'est la première étape. On commence par identifier le comportement ou la fonctionnalité qu'on souhaite rajouter et on écrit un test qui illustre celle-ci. Bien sûr, comme l'implémentation à ce stade n'est qu'un "walking skeleton" qui émerge depuis le test (aidé de notre IDE préféré, on crée la structure vide de cette implémentation au moment où l'on définit son usage, c'est à dire depuis le code du test), le test échoue. D'où la couleur rouge. Les Anglo-Saxons résument cette étape par la formule: "Make it fail".
- **GREEN** : c'est la seconde étape qui vise à implémenter le plus rapidement possible (et c'est important de s'y tenir) le code qui est ciblé par notre test pour le faire passer au vert. Pour les Anglo-Saxons, c'est le : "Make it works". Il est important d'aller vite ici, et il est même recommandé de prendre quelques raccourcis comme des valeurs en dur qui nous aideront juste à faire passer le test le plus vite possible avant de passer à l'étape d'après (c'est très perturbant au départ, mais bénéfique, nous allons voir pourquoi en détaillant l'étape suivante).
- **REFACTOR** : c'est la troisième et dernière étape, trop souvent oubliée ou sacrifiée dans un quotidien parfois stressant. Maintenant que nous avons une implémentation et que le test passe, il s'agit ici de revoir la structure interne de notre code pour corriger tous les raccourcis pris lors du "Make it work". C'est d'ailleurs pour cela que les Anglo-Saxons parlent ici de "Make it better". Le fait d'avoir séparé les préoccupations de

coder quelque chose qui fait le job (definition-of-done précisée par le test), et ensuite d'améliorer la qualité de l'implémentation mène à une plus grande efficacité dans notre action (tant il est parfois facile de se perdre dans des optimisations à tiroirs lorsqu'on tente de faire les 2 en même temps). Fig.1.

Bon, on vient de détailler ces 3 étapes, mais en quoi ce workflow est-il encourageant ? Eth bien je ne sais pas pour vous, mais moi je trouve que les journées d'un développeur sont truffées de feedbacks négatifs... Ça commence par notre compilateur qui passe son temps à nous remonter des erreurs, à notre chaîne d'intégration continue ensuite, qui passe au rouge de temps en temps (perturbant notre cadence en nous forçant à la corriger toute affaire cessante), quand ce n'est pas le chef de projet qui se retrouve sur notre dos parce qu'il est stressé (ou parce qu'il a une bonne raison de l'être). Dans ces journées où les feedbacks négatifs sont légion et impactent sans doute plus notre mental qu'on veut bien le reconnaître. Dans ces journées donc, chaque petit test qui passe, chaque loupiotte verte du RED-GREEN-REFACTOR sont autant de petites victoires, de petits signes encourageants qui m'indiquent que ça avance. Je les prends comme des petites friandises qui viennent cadencer mes baby steps et ma journée.

Plus efficaces

Thomas : On vient de le voir, le TDD me permet de démarrer mes développements plus rapidement; enlevant au passage une partie inhérente du stress que je pouvais éprouver à la fin de mon travail et qui était liée à ma procrastination. Un autre bénéfice du TDD est qu'il me permet d'être plus rapide et plus efficace quand je code, ce qui est souvent contre-intuitif pour ceux qui n'ont pas l'habitude de le pratiquer. Comment est-ce possible ? Eh bien c'est lié au fait que le TDD - lorsqu'on est bien équipé - nous permet de ne pas répéter encore et encore les mêmes gestes fastidieux, les mêmes tests à la main, les mêmes sessions de débogage ou d'essais-erreurs comme j'avais l'habitude de le faire par le passé. L'équipement dont je parle ici est un outil comme NCrunch (en .NET) ou infinitest (en Java), même si je connais surtout NCrunch. Le concept : lorsque vous codez, l'outil compile, exécute les tests et analyse le test coverage en tâche de fond pour que vous puissiez avoir un feedback quasi instantané dans votre IDE (sous la forme de puces de couleur pour chaque ligne de code dans la marge: vert quand la ligne est couverte par des tests, rouge quand la ligne est impliquée dans au moins un test qui échoue, et noir lorsque la ligne de code n'est couverte par aucun test). Que vous soyez en train de refactorer du code legacy ou en train de travailler sur un tout nouveau code, cela vous permet d'avoir un feedback immédiat pour chacune de vos interventions. Un must!



Du coup, **les moments où vous avez besoin de lancer votre debugger deviennent de plus en plus rares** ce qui n'est pas pour me déplaire tant le lancement du debugger me paraît lent comparé au feedback rapide de NCrunch (plus proche d'un REPL(1) en termes de feedback).

Quelqu'un a écrit une fois sur Twitter :

Ecrire du code : 1 heure / Fixer les bugs : 4 heures

Ecrire des tests : 1 heure / Ecrire du code : 1 heure / Fixer des bugs : 15 minutes

Ca résume assez bien l'efficacité du TDD je trouve, même si l'écriture des tests et du code sont entrelacés.

Bruno : De mon côté, le TDD, m'a également permis de changer ma vision sur le développement logiciel. Effectivement, le développement traditionnel débute généralement avec une forme de sketching au niveau du code. On comme avec de la pâte à modeler, on recherche une solution technique acceptable. Après une brève prise en compte du besoin utilisateur, le développeur écrit directement quelques classes qui semblent rassurer sa compréhension. Dans les faits, ce mode encourage une forme de design qui part du code pour adhérer au besoin.

L'approche TDD est dirigée par le besoin utilisateur pour déboucher sur une proposition technique **en parfaite adéquation avec le besoin ni plus ni moins.** En d'autres mots, on doit s'interroger en termes de cas d'utilisations portés par les comportements attendus par l'utilisateur avant d'engendrer son code. C'est un point important.

Avec ce workflow RED-GREEN-REFACTOR, le TDD est également l'assurance d'avancer par petit pas en toute quiétude.

Pas d'effet tunnel

Votre code est « commité » régulièrement, il est donc facile de comprendre la démarche d'implémentation a posteriori. Le code est ultra simple, et ne contient aucune astuce, un autre développeur peut le reprendre sans risque d'ambiguïté sur son intention.

Plus pertinents

Thomas : Le TDD s'il est pratiqué en mode "**outside-in**", se révèle diablement efficace **pour construire sans hésitation ni gâchis, le bon système.** Comme je n'ai pas encore eu l'occasion d'introduire ce terme, laissez-moi vous le présenter.

De nos jours, on distingue deux formes majeures de TDD: l'école classique, et l'école dite de Londres (ou "outside-in TDD"). La forme classique est celle introduite par le créateur du TDD: Kent Beck (aussi le créateur de l'eXtreme Programming). Dans cette forme historique également la plus connue, on part en général du centre du système pour construire autour et faire grossir celui-ci par strates successives pour arriver au système final. L'école de Londres elle (appelée ainsi, car elle a été introduite par Steve Freeman et Nat Pryce dans leur ouvrage de référence "Growing Object-Oriented Software Guided by Tests" - le GOOS), propose de considérer le système à construire d'un point de vue extérieur, comme une grosse boîte noire que nous allons définir, remplir et implémenter petit à petit.

La double boucle de l'outside-in TDD

Alors au début bien sûr, cette boîte noire est vide. Et nous allons définir petit à petit ses contours et la façon d'interagir avec elle via des tests "grosse maille" : des tests d'acceptance qui viseront le système dans son ensemble (la boîte noire). Sur le chemin, nous allons aussi être amenés à écrire des tests unitaires qui vont nous aider à coder petit à petit l'intérieur de la boîte en partant des interfaces définies par les tests d'acceptance. On appelle ça la démarche de la "double boucle". Fig.2.

Bruno : Détaillons un peu : tout commence par une 1ere boucle de RED-GREEN-REFACTOR au niveau acceptance (le sujet à l'étude est donc à cet instant le système dans son ensemble). Le RED nous permet de définir, à travers un 1er test d'acceptance, une des fonctionnalités de cette boîte noire. Le test d'acceptance échouant, nous descendons tout de suite d'un niveau d'abstraction pour commencer à itérer sur plein de petits RED-GREEN-REFACTOR au niveau unitaire, pour implémenter ce qu'il faut à l'intérieur de la boîte ; l'objectif étant toujours de faire passer notre 1er test d'acceptance. Une fois que c'est fait, on continue et on réalise l'étape REFACTOR du test d'acceptance initial... Avant

(1) Read–Eval–Print Loop

de repartir pour un nouveau tour de grande boucle en commençant par écrire le second test d'acceptance, qui nous fait redescendre au niveau des petites boucles unitaires pour son implémentation et ainsi de suite...

Une approche minimaliste et YAGNI par essence

Thomas : L'intérêt de l'outside-in est d'**empêcher qu'on se perde en route en implémentant des choses qui n'ont pas un lien direct avec le résultat final** (i.e. notre système pris dans sa globalité). Dès le départ on est piloté par la définition des contours et des interactions avec celui-ci (l'approche classique pouvant nous mener à découvrir trop tard ce qu'on s'est loupé et qu'on n'a pas construit le bon système ou un système pratique à utiliser).

Aujourd'hui, je peux dire que je pratique presque exclusivement l'outside-in TDD (à l'exception de certains petits katas de code où il m'arrive de refaire du TDD classique).

Cela nous ramène à une caractéristique importante du TDD : l'efficacité. Une efficacité liée au minimalisme de la démarche qui n'est rien d'autre que l'application du principe YAGNI (*You Ain't Gonna Need It*) : ici on ne code rien qui ne soit directement lié à un test, et donc si on s'y prend bien, à une fonctionnalité requise pour notre système.

Au final des professionnels plus épanouis, tout simplement

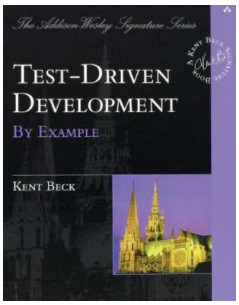
Thomas : M'ayant permis de régler définitivement certains problèmes qui m'empêchaient d'être réellement efficace en tant que développeur (procrastination, debugging intempestif, effet tunnel lors de la réalisation, hors sujets dans certaines sessions de DEV où on finit par se faire plaisir en oubliant l'objectif initial ...), la pratique du TDD a fait de moi un meilleur développeur, mais surtout un professionnel plus aguerri, capable désormais de me concentrer sur ce qui est le plus important pour mes clients ou mes utilisateurs finaux sans me perdre en route.

Vers un développement logiciel en harmonie avec soi même

Bruno : Si vous êtes développeurs, vous avez sans doute une vie ponctuée de périodes calmes où vous développez gentiment les premières briques de votre projet, puis des périodes plus intenses lorsque la date de la première mise en production va se rapprocher, pour finir par des périodes très intenses où les problèmes de productions vont dominer votre quotidien. Ce n'est pas vraiment une vie simple à la fois pour vous, et votre entourage personnel. Vous pouvez être dans le déni et considérer ce mode de vie comme une fatalité, ou bien vous pouvez changer de mode de développement. La pratique du TDD dépasse largement les aspects techniques, car elle permet d'objectiver son métier de manière itérative. À l'instar d'un footing où l'objectif n'est pas de courir le plus vite possible, mais de trouver son rythme, le TDD s'inscrit dans un rythme cadencé par les tests. Thomas en a parlé, on choisira toujours un ordre de difficulté croissant. Les premiers tests seront simples et permettront de « croquer » le début d'implémentation ». Dans ce début de parcours, les tests seront relativement génériques et le code assez spécifique. Au fil du chemin parcouru, les tests seront de plus en plus spécifiques et le code de plus en plus générique, car au fil des implémentations les refactorings successifs, vont apporter leurs lots de simplifications. Ceci pour obtenir un code qui transpire les comportements fonctionnels à travers les noms des tests que vous avez choisis, permettant d'être à la fois lisible et facile à maintenir. Il peut arriver que votre parcours s'étale sur plusieurs jours. Mais comme chaque test développé doit donner lieu à un commit libellé par le comportement que vous avez illustré, vous gagnez une forme d'assurance vis-à-vis de votre travail. Avec le temps, vous gagnerez en confiance sur votre métier et vous serez plus rassurés sur votre code. C'est le début d'une attitude plus sereine et plus en harmonie avec vous-même. Il vous restera même du coup plus d'énergie pour aller vers les autres (vos collègues, vos utilisateurs, vos clients, etc).

Après ce 1er article consacré au "pourquoi" du TDD, nous vous proposerons par la suite d'autres articles (illustrés par du code cette fois) sur les pièges et les écueils à éviter lorsqu'on fait du TDD, parce qu'il est très facile de se louper, et de passer du coup à côté de cette pratique incroyable. Ce serait dommage. Vraiment...

Quelques références sur le TDD



Le premier ouvrage sur le TDD est celui de Ken Beck. Il représente la première incarnation de cette pratique. La communauté qualifie la méthode décrite dans ce livre, de TDD classic. Le second ouvrage est plus abouti, il décrit l'usage du TDD dans toutes les phases de développement d'un projet. C'est de ce livre qu'est tiré le schéma de la double-boucle de l'outside-in. La communauté qualifie la méthode décrite dans ce livre, d'*école de Londres* en hommage aux auteurs du livre.