



CULTURE CODE

SOFTWARE CRAFTSMANSHIP : BETTER PLACES WITH BETTER CODE



CULTURE CODE

SOFTWARE CRAFTSMANSHIP : BETTER PLACES WITH BETTER CODE



Avant donc que d'écrire, apprenez à penser.

*Ce que l'on conçoit bien s'énonce clairement,
Et les mots pour le dire arrivent aisément.*

*Hâitez-vous lentement, et sans perdre courage,
Vingt fois sur le métier remettez votre ouvrage,
Polissez-le sans cesse, et le repolissez,
Ajoutez quelquefois, et souvent effacez.*

Nicolas Boileau, L'Art poétique, chant I (extrait)

PRÉFACE	6
INTRODUCTION	7
BETTER PLACES	10
Chapitre 1 : Transmettre une culture de la qualité logicielle et de l'excellence technique	12
Chapitre 2 : Maintenir la maintenabilité	21
Chapitre 3 : Le Tech Lead, au service de l'équipe	46
Chapitre 4 : Dette technique et non-qualité	58
Interlude : Crissez ! Proliférez !	
BETTER CODE	66
Préambule	68
Chapitre 5 : Écrire du code compréhensible	73
Chapitre 6 : La revue de code	84
Chapitre 7 : Une stratégie efficace de tests automatisés	108
Chapitre 8 : Test Driven Development	125
Chapitre 9 : Récit d'un apprentissage par la pratique	138
BIBLIOGRAPHIE	150
INDEX	154
LE MOT DE LA FIN	156

Préface

Le nombre de lignes de code actuellement en production dans le monde – si tant est qu'il puisse être recensé – dépasse très certainement tout ce que nous pouvons imaginer. Le code est partout : après l'informatique industrielle, de gestion, et aujourd'hui avec celle des interactions digitales, le code automatise toutes les petites routines de nos vies jusqu'aux plus grands engrenages de nos sociétés et de nos économies.

La stratégie digitale des organisations, si elles souhaitent effectivement opérer dans le digital, devra, à un moment donné, se matérialiser quelque part. Ce quelque part, c'est dans *votre code*. C'est lui qui va porter vos ambitions digitales. Votre code, c'est votre écriture, c'est votre signature ; c'est pour cela qu'il faut en prendre soin. Nul doute que ceux qui auront ce livre sous le coude seront mieux préparés.

Ce livre est multiple, il vous offrira de réelles méthodes thérapeutiques quant aux pratiques du code, notamment dans la partie « Maintenir la maintenabilité », avec une adresse au management.

C'est également un livre d'économie du code, car il véhicule une certaine vision des activités de codage qui n'oublie jamais les contraintes économiques. La vision qu'il expose nous emmènera au-delà du simple catalogue d'astuces et vous serez sensibilisés à la mise en œuvre d'une politique des pratiques de développement. C'est peut-être là le tour de force des auteurs : arriver à nous ouvrir les yeux sur le sens et les enjeux d'une économie politique du code, dont les débats récurrents sur l'enseignement de l'informatique à l'école devraient s'inspirer.

C'est aussi un manifeste, qui rend vivant et donne du corps au mouvement *Software Craftsmanship* en valorisant toutes les formes de savoirs individuels et collectifs. Ce livre sera apprécié par ceux qui aiment le code et qui veulent en saisir les enjeux. Je pourrais dire que c'est un livre d'amateur, si j'étais sûr que cela ne soit pas compris comme « pas sérieux » ou « pas professionnel ». Car l'amateur, comme le mot l'indique, est avant tout celui qui s'investit dans son travail parce qu'il l'aime.

Le recours à des exemples de situations vécues témoigne d'une certaine éthique qui n'est pas sans faire écho au texte précurseur sur l'Éthique Hacker¹ où Pekka Himanen écrivait, en citant Linus Torvalds : « Linux a largement été un hobby (mais un sérieux, le meilleur de tous). »

Quelle joie de lire un livre sérieux sur le code !

Christian Fauré, amateur de code, Partner OCTO Technology

1. L'Éthique Hacker et l'esprit de l'ère de l'information, Pekka Himanen, 2001

Introduction

Par Michel Domenjoud

Better Places with Better Code : un environnement de travail propice à du code de qualité. Créer une culture de la qualité prend du temps et nous pensons que l'amélioration de la qualité repose sur la pratique. Le geste qu'on fait au quotidien a un impact à long terme.

Produire du code de qualité est une question de culture, **une culture de la qualité**. Faire émerger cette culture dans une organisation prend du temps, car elle implique non seulement l'individu, qu'il soit développeur ou non, mais aussi l'équipe et l'entreprise dans son ensemble.

L'amélioration de la qualité du code repose sur les **pratiques** des développeurs, ainsi que l'environnement qui favorise ces pratiques. L'essentiel des logiciels est construit en équipe, et la manière dont les individus travaillent ensemble influe sur la qualité de ces logiciels, de même que l'environnement fourni par l'entreprise.

Le **geste** exécuté au quotidien a un impact sur le long terme. Nos pratiques individuelles de développement, notre façon d'échanger à propos du code, et la politique de gestion de la qualité, impactent la maintenabilité du logiciel et donc directement son coût. On peut avoir l'impression d'obtenir le même logiciel avec ou sans tests automatisés, que l'on ait pris le temps ou non de revoir en continu le design et la compréhension du code, mais c'est dans le temps que l'écart se creusera.

La non-qualité du code a également des conséquences économiques concrètes, comme un *Time To Market* plus long en raison de fonctionnalités plus difficiles à ajouter ou à faire évoluer, des régressions plus promptes à apparaître qui vont prolonger les cycles de développement, des applications trop lentes et des risques d'indisponibilité accrus. L'absence de culture de la qualité explique aussi souvent un piètre degré de satisfaction et un désengagement des développeurs, qui semble être l'origine principale des 20 % de turn-over moyen qu'on observe dans le secteur informatique². De plus, le développeur démissionnaire part souvent avec un pan entier de la connaissance du système d'information...

Pourquoi avoir écrit ce livre ? Pour changer les esprits. Nous sommes convaincus que créer des logiciels de qualité ne coûte pas plus cher, au contraire. Malheureusement, la situation dans de beaucoup d'entreprises est **préoccupante**.

2. Nouvelles perspectives pour réduire l'impact du turnover dans l'informatique, Janice Lo, 2014 <http://www.hec.fr/Knowledge/Strategie-et-Management/Management-des-Ressources-Humaines/Nouvelles-perspectives-pour-reduire-l-impact-du-turnover-dans-l-informatique>

De nombreuses équipes surnagent dans un « océan de code legacy³ », car la qualité a été trop souvent négligée. Le problème avec la non-qualité – la dette technique – est un peu le même qu’avec le réchauffement climatique : tout le monde en parle, mais peu de personnes agissent réellement.

C'est d'autant plus critique à l'heure où le logiciel est omniprésent, en train de dévorer la planète comme l'expliquait Marc Andreessen⁴. Nous avons une lourde responsabilité en tant que développeurs, managers, responsables métiers, directeurs techniques, directeurs des systèmes d'information : nous assurer que le code ait la qualité requise pour répondre aux besoins qu'il couvre.

Chez OCTO Technology, nous avons une conviction : le développement de logiciels est un savoir-faire, qui s'acquierte via l'expérience et l'accompagnement de ses pairs, comme dans l'artisanat. Une simple formation n'est pas suffisante : le développement est un métier, il nécessite un **apprentissage** permanent qui passe par la programmation mais aussi par les pratiques de développement associées.

Ce livre s'inscrit dans la lignée des valeurs portées par le mouvement *Software Craftsmanship* (voir encadré). D'excellents ouvrages traitent déjà de la posture et des pratiques individuelles de ce mouvement. Ici, nous souhaitions plutôt explorer en quoi créer les conditions propices pour écrire du code de qualité est aussi une problématique d'équipe et d'entreprise.

C'est un ouvrage collectif, que nous avons construit à partir de nos expériences et à partir d'entretiens : vous y trouverez des éléments concrets activables, comme de nombreux retours d'expérience. Des succès, mais aussi des échecs que nous avons nous-mêmes vécus.

○ À qui s'adresse ce livre ?

Ce livre s'adresse aussi bien :

- Au développeur, au Tech Lead, au coach, et plus généralement à toute personne participant à la réalisation de produit logiciel, qui trouvera des éléments concrets pour emmener son équipe vers une culture de la qualité du code et mettre en œuvre les pratiques associées.
- Au manager, au directeur technique, au DSI, qui trouvera des pistes pour faciliter un changement dans l'entreprise vers une culture de la qualité, mais aussi des éléments pour gérer efficacement la qualité du code (plus particulièrement dans le chapitre Maintenir la maintenabilité).

3. Code legacy : code qui n'est plus supporté ou plus maintenu, et plus précisément tout code qui n'est pas couvert par des tests automatisés.

4. Why Software Is Eating the World, Marc Andreessen (Wall Street Journal, 2011) <http://www.wsj.com/articles/SB1000142405311903480904576512250915629460>

Quelques mots sur le mouvement *Software Craftsmanship*

Le mouvement *Software Craftsmanship*, en français « artisanat logiciel », est une approche du développement logiciel qui met en avant les pratiques de développement pour réaliser des logiciels de qualité, arguant qu'il ne suffit pas qu'un logiciel fonctionne, il faut également qu'il soit bien conçu.

Le mouvement prend forme en 2009 avec la publication d'un manifeste⁵, après que Robert C. Martin, « Uncle Bob », ait proposé l'ajout d'un cinquième pilier au manifeste agile, considérant que celui-ci n'insistait pas assez sur la qualité des réalisations :

- Pas seulement des logiciels opérationnels, mais aussi des **logiciels bien conçus**.
- Pas seulement l'adaptation aux changements, mais aussi l'**ajout constant de valeur**.
- Pas seulement les individus et leurs interactions, mais aussi **une communauté de professionnels**.
- Pas seulement la collaboration avec les clients, mais aussi **des partenariats productifs**.

Il trouve ses origines notamment dans l'ouvrage *The Pragmatic Programmer : From Journeyman To Master* d'Andy Hunt et Dave Thomas, et reprend également de nombreuses pratiques déjà mises en avant par *eXtreme Programming*⁶.

Plusieurs ouvrages traitent déjà avec brio de ce sujet comme *Clean Code* de Robert C. Martin ou plus récemment *The Software Craftsman - Professionalism, Pragmatism, Pride* de Sandro Mancuso.

◉ Better places, Better Code

Ce livre démarre par une approche générale du sujet, que nous creuserons ensuite progressivement. La première partie de ce livre, *Better Places*, traite de la culture de la qualité du code et des moyens de la porter à l'échelle de l'équipe et de l'entreprise. La seconde partie, *Better Code*, s'attache plutôt aux pratiques sur lesquelles une équipe peut s'appuyer, et donne des pistes pour les mettre en œuvre efficacement au quotidien.

5. Manifesto for Software Craftsmanship, 2009, <http://manifesto.softwarecraftsmanship.org/#fr-fr>.

6. Extreme Programming, https://fr.wikipedia.org/wiki/Extreme_programming.



BETTER
PLACES

TRANSMETTRE UNE CULTURE DE LA QUALITÉ LOGICIELLE ET DE L'EXCELLENCE TECHNIQUE



Transmettre une culture de la qualité logicielle et de l'excellence technique

Par Michel Domenjoud

Le manifeste du mouvement *Software Craftsmanship* est sous-titré par *Raising the bar*, soit éléver le niveau. Pour réaliser des produits de haute qualité, il faut s'appuyer sur des personnes compétentes, motivées et même passionnées, et qui cherchent à apprendre et à s'améliorer en continu.

Il ne s'agit pas que de pratiques à mettre en œuvre : c'est une véritable **culture du développement**, qui implique des changements dans les mentalités, dans le fonctionnement des équipes et de l'entreprise.

Mais une culture, ça ne s'impose pas : **comment faire entrer les valeurs du Software Craftsmanship dans l'entreprise afin qu'elles deviennent le standard et non l'exception ?** Nous commencerons à répondre à cette question dans ce chapitre et plus largement dans tout cet ouvrage.

○ Qualité, non-qualité et conséquences

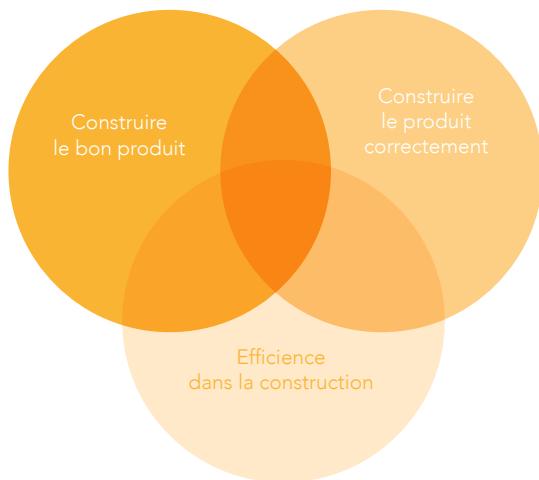
Le terme de « qualité logicielle » peut être porteur de plusieurs interprétations : nous l'utiliserons ici pour parler d'un logiciel bien conçu, qui apporte de la valeur et repose sur un code de qualité.

Pour réaliser un logiciel de qualité, les méthodes agiles que nous utilisons cherchent à répondre à trois enjeux essentiels, mais surtout à faciliter le fragile équilibre entre eux⁷ :

- Réaliser le **bon** produit, celui qui apporte de la valeur et répond aux besoins des utilisateurs.
- **Bien** réaliser le produit, le faire sans défaut, grâce à une qualité technique et une maintenabilité suffisantes pour répondre à des enjeux métiers en perpétuelle évolution.
- **Être** efficient au cours de la réalisation : aboutir à de bons résultats avec le minimum de dépense et d'effort.

7. *Agile Product Ownership in a Nutshell*, Henrik Kniberg. Une présentation du processus agile vu du Product Owner, qui présente également ce difficile équilibre. <http://blog.crisp.se/2012/10/25/henrikkniberg/agile-product-ownership-in-a-nutshell>

Trois enjeux de construction d'un produit de qualité



Construire un logiciel de qualité, c'est chercher cet équilibre, créer un système qui répond à des besoins tout en reposant sur des bases saines.

L'agilité est censée être une pratique devenue courante dans les organisations. Malheureusement, trop de projets continuent d'être compliqués à mener. Il y a toujours trop de bugs, d'utilisateurs insatisfaits, de coûts de réalisation qui explosent sur le long terme. Que s'est-il passé ? Les méthodes agiles ne tiendraient-elles pas leurs promesses ?

La raison de ces échecs est que la qualité technique n'était pas au rendez-vous⁸. Nous

utilisons alors le terme de « non-qualité ».

Cette non-qualité a un coût, qui se paie à différents niveaux :

- **Stratégique** : atteindre un haut niveau de qualité n'est pas un enjeu nouveau, mais il est d'autant plus stratégique dans notre société où le logiciel devient omniprésent. À l'ère du numérique, il faut aller toujours plus vite et satisfaire ses utilisateurs pour ne pas se faire dépasser par la concurrence.

- **Financier** : une application de mauvaise qualité est généralement propice à l'ajout de bugs, de régressions, et l'accumulation de

8. Sandro Mancuso explique cette « gueule de bois » de l'agilité dans son livre *The Software Craftsman : Professionalism, Pragmatism, Pride* (Prentice Hall, 2014). Voir aussi cette interview par InfoQ <http://www.infoq.com/articles/mancuso-software-craftsman>.

défauts de qualité rend plus difficile l'ajout de nouvelles fonctionnalités. Celle-ci sera donc plus coûteuse à maintenir, et les clients qui perdent confiance risquent de ne plus vouloir payer.

- **Humain** : c'est un risque souvent sous-estimé. Travailler dans un cadre où la qualité des réalisations est négligée devient vite démotivant, notamment pour les développeurs, au risque de voir les meilleurs partir dans une autre entreprise ; un développeur préfère investir du temps pour des fonctionnalités futures plutôt qu'éponger les bugs du passé...

Atteindre un haut niveau de qualité, ce n'est pas simplement une question de produire des applications avec du code de qualité : c'est une culture, celle de la qualité logicielle. S'il est important de regarder les artefacts produits et les pratiques qui permettent de les réaliser, il est surtout vital de **s'intéresser aux facteurs qui favorisent cette culture dans nos entreprises.**

○ Tous des Craftsmen ?

Il serait dommage de dire qu'il y a d'un côté des Craftsmen, des sortes de super développeurs qui se regroupent dans les meetups et recherchent toujours la perfection, et de l'autre côté des développeurs lambda qui ne poursuivraient pas le même but ou

n'auraient pas les bonnes compétences.

L'objet du mouvement *Software Craftsmanship* n'est pas de catégoriser les développeurs – ou d'insinuer que les autres sont mauvais : ils n'ont simplement pas eu l'occasion **d'expérimenter** cette culture. **Car le véritable enjeu est là**, dans la culture. On fait malheureusement encore aujourd'hui le constat que beaucoup de pratiques de développement qui contribuent à des réalisations de qualité ne sont tout simplement **pas enseignées à l'école**.

Plus tard, les répercussions dans les équipes sont là : beaucoup pensent que ce n'est pas possible dans leur entreprise et que les choses sont immuables.

Alors quelle réponse apporter à ce problème ? Pour certains il n'y aurait pas d'autre solution que de changer d'entreprise.

Mais que faire si votre nouvel environnement n'est finalement pas idéal non plus ? Si on essayait plutôt de trouver des pistes pour amener du changement ?

Sans rentrer dans le respect stricto sensu des piliers de cette culture, nous ne pouvons que vous répéter que la clé est là : **ces valeurs ont intérêt à être partagées pour le bien de votre entreprise.**



Pendant mes premières années en tant que développeur, je connaissais tout juste l'existence des tests unitaires ou de l'intégration continue. C'est simple, je n'avais pas encore été exposé à ces pratiques, ni à l'école ni dans ma vie professionnelle, et les gens avec qui je travaillais non plus. Pourtant j'essayais de faire de mon mieux, de produire des applications dont j'étais fier. Puis j'ai rencontré un développeur passionné, qui m'a transmis certaines pratiques et exhortait ses collègues à les utiliser. J'ai ensuite participé à un autre projet où cette culture était centrale. Et ainsi de suite, jusqu'à vouloir moi-même transmettre cette culture, convaincu que c'était une bonne manière d'exercer mon métier.

Ø Adopter la culture de l'artisanat logiciel

Lorsque nous parlons de Software Craftsmanship, nous pensons **excellence technique dans la réalisation, professionnalisme du développement logiciel et responsabilité de chacun** dans l'accomplissement de son métier.

Certaines pratiques nous sont vitales pour atteindre ce niveau : **le développement dirigé par les tests** (*Test Driven Development, Behavior Driven Development...*), l'application de **standards de code** propre, le **remaniement**

de code (**refactoring**)⁹ en continu, ou la pratique du design émergent.

Lorsque nous utilisons le terme **professionnalisme**, nous ne désignons pas uniquement l'excellence du développeur :

- Il s'agit également d'adopter une posture pragmatique : on ne cherche pas absolument la perfection, on cherche à faire au mieux en tenant compte du contexte connu.

- Pour agir au mieux dans l'équipe, et collaborer efficacement avec chacun, il est utile de comprendre un minimum le métier de chacun, en acquérant par exemple une certaine sensibilité métier, Ops ou encore UX.

- Rechercher la qualité passe par l'**amélioration continue**, du code mais aussi des pratiques, ce qui implique une certaine sensibilité à la méthodologie utilisée.

Alors, comment continuer à **apprendre, encore et toujours** dans un métier en perpétuelle et rapide évolution ?

- En lisant **des livres, des blogs**.
- En **s'entraînant**, en pratiquant de nouveaux outils, de nouveaux langages.
- En **rencontrant** d'autres développeurs à des conférences, des meetups.
- En trouvant des **mentors**.

9. Refactoring : action de modifier la structure du code, en conservant un comportement strictement identique <http://refactoring.com/>

○ Transmettre ses valeurs et ses pratiques dans l'équipe

Adopter les valeurs « Craft » est une bonne première étape mais les partager avec ses collègues et encore plus aux managers, est la clé pour pouvoir produire du code de qualité au sein de votre entreprise.

Pour diffuser la culture de la qualité au sein de l'équipe, on peut s'appuyer sur plusieurs pratiques :

- Garantir une **propriété collective du code** en organisant des **revues de code**, mais aussi en évitant que chaque développeur ne travaille seul sur son périmètre.
- Apprendre et transmettre les pratiques de développement via le **binômage** (*pair programming*), l'organisation de **Coding Dojos**¹⁰, etc.
- Établir **ensemble** des standards de développement et les faire vivre.
- S'appuyer sur les outils de **l'intégration continue** pour **réduire la boucle de feedback**.

Pour cela, il est nécessaire de rompre avec certaines – mauvaises – habitudes bien ancrées dans l'entreprise.

Le métier de développeur est encore trop souvent perçu comme une activité solitaire, ce qui a deux conséquences :



Sur un précédent projet, nous avions mis en place un créneau de Coding Dojo ritualisé toutes les deux semaines, au cours duquel nous partagions nos pratiques en prenant du recul par rapport au quotidien du projet. Selon les séances, c'était l'occasion d'expérimenter de nouvelles technologies et décider de leur usage futur sur le projet, de partager de nouvelles techniques de code et de faire évoluer nos standards ensemble.

• Les pratiques de développement collectives comme le binômage sont considérées par les acteurs non techniques des projets comme un surcoût conséquent alors que la plupart des études¹¹ tendent à prouver le contraire.

• Les développeurs qui ont été habitués à travailler seuls peuvent avoir des difficultés à travailler à plusieurs, ne serait-ce que pour montrer leur code ou en parler. Sur cet aspect, il est important d'adopter une posture positive, **d'apprendre à donner et recevoir du feedback** sans pointer les autres du doigt. Les principes d'*Egoless Programming*¹² vont dans ce sens.

Favoriser la culture de la qualité à l'échelle de l'équipe n'est pas une chose aisée, c'est une **véritable conduite du changement** à mener. Cela prend du temps.

10. Atelier collectif de programmation visant à s'entraîner sur une pratique ou une technologie.

11. *The costs and benefits of Pair Programming*, Alistair Cockburn, 2000, http://www.cs.pomona.edu/~markk/cs121.f07/supp/williams_prgm.pdf

12. *The Psychology of Computer Programming*, Gerald M. Weinberg, 1971. Les principes sont résumés sur ce blog : <http://blog.codinghorror.com/the-ten-commandments-of-egoless-programming/>



Pour cela, il est utile de s'appuyer sur les personnes qui maîtrisent les pratiques, comme **un Tech Lead qui va aider l'équipe à progresser** jusqu'à ce qu'elle devienne autonome. Si cette personne n'est pas présente au sein de l'équipe, on peut faire appel à un tiers extérieur, qui pourra la sensibiliser, la former et l'accompagner.

➊ Une culture à l'échelle de l'entreprise

On a vu que **le développement de cette culture est favorisé par des rencontres**, par des personnes qui souhaitent partager leur savoir-faire et leur savoir-être, et qui souhaitent continuer à apprendre dans ce métier en perpétuelle évolution.

À l'échelle de l'entreprise, développer cette culture peut sembler une tâche titanique pour quelques individus, c'est pourquoi **il est**

vital que les décideurs permettent à ces rencontres d'avoir lieu.

Plusieurs actions concrètes peuvent être mises en place :

- **Soutenir les démarches d'amélioration continue**, en laissant les équipes s'organiser. Les développeurs sont des professionnels du développement, il est indispensable de leur **accorder une certaine autonomie dans leurs pratiques**.

- Créer des **espaces de rencontre** entre les équipes, favoriser l'émergence de **communautés de pratiques**. Par exemple chez OCTO Technology, des BBLs (*Brown Bag Lunch*¹³) sont organisés presque tous les midis, et le troisième jeudi de chaque mois est réservé à l'organisation des BOFs (*Birds of a Feather*¹⁴), au cours desquels chacun peut partager un retour d'expérience.

- Dédier un **budget à l'apprentissage** dans tout projet, afin que chacun ait un autre choix que prendre sur son temps personnel pour apprendre, et surtout, que les gens puissent apprendre collectivement.

- **Revoir l'organisation de l'espace** et des bureaux pour favoriser un travail collectif.

- **Identifier les leaders d'influence** et leur donner le mandat pour diffuser leurs valeurs et leurs pratiques, en animant la communauté, en accompagnant et en coachant d'autres équipes.

13. BBL - *Brown Bag Lunch* : des moments de partage d'expérience organisés sur l'heure du déjeuner. <http://www.infoq.com/fr/articles/bbl-fr>

14. BOF - *Birds Of a Feather* : littéralement les oiseaux d'une même plume. Moment dédié lors duquel des professionnels d'une même spécialité se retrouvent <http://blog.octo.com/un-jeudi-tres-bof>

Il faut néanmoins être vigilant et **ne pas chercher à passer à l'échelle supérieure trop rapidement** : ce n'est pas parce qu'une équipe a adopté un certain nombre de pratiques qu'il faut chercher à les imposer à toute l'organisation. **Imposer le changement est contre-productif. Il est préférable de l'inspirer, ce à quoi la culture d'entreprise contribue largement.**

○ Points à retenir

Le mouvement *Software Craftsmanship* propose une vision du métier de développeur, et celle-ci ne doit pas seulement considérer les individus, mais apporter une **vision du métier dans l'entreprise**, qui promeut **une culture de la qualité** et de l'apprentissage permanent. La création de logiciel est un métier passionnant et même plutôt fun, pour peu que l'on puisse l'exercer avec application et professionnalisme **dans un environnement favorable**.

Et c'est un enjeu absolument crucial pour l'entreprise : notre société toute entière repose sur des logiciels, et la tendance n'est pas près de s'inverser. **Il est de notre responsabilité d'adopter des valeurs et des pratiques permettant de réaliser des logiciels de qualité.**

○ Par où commencer ?

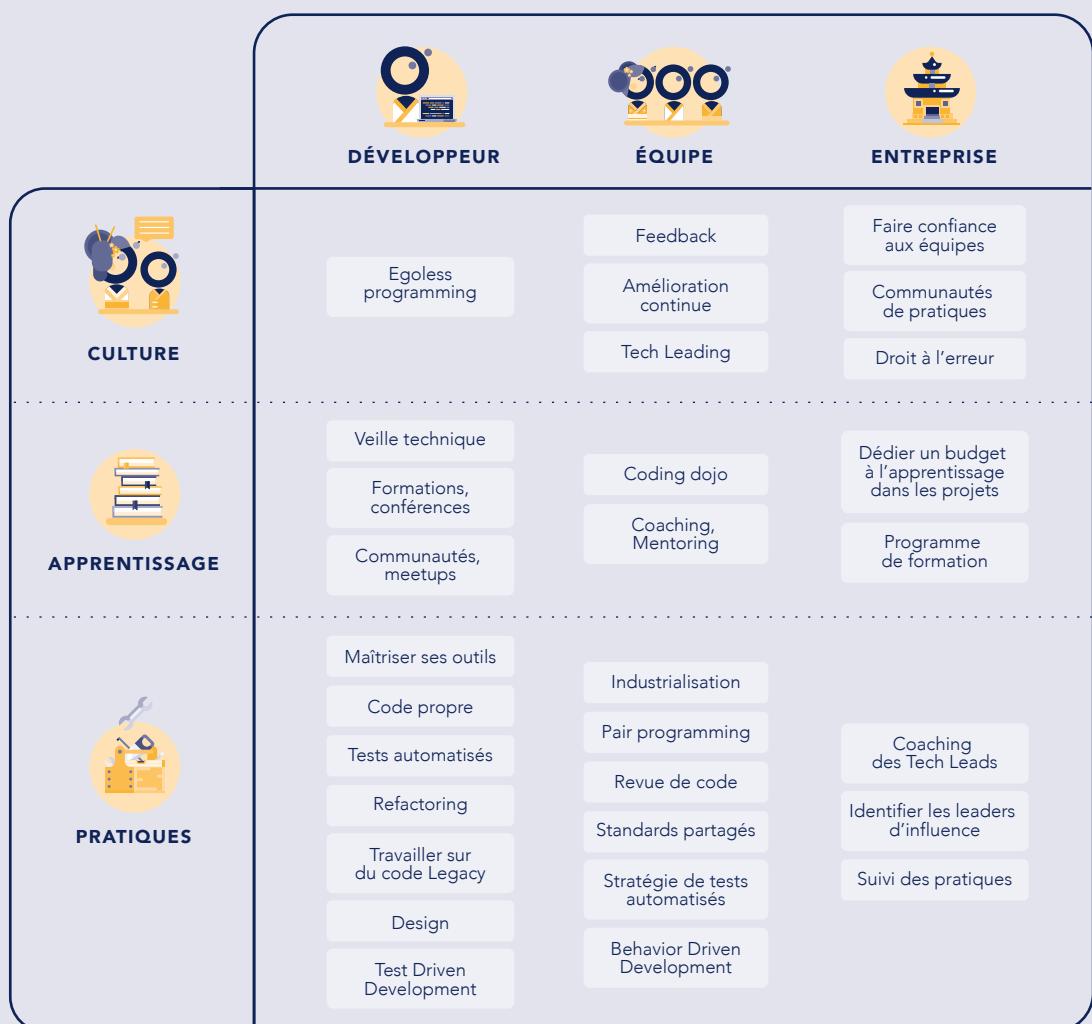
- La prochaine fois que vous rencontrez une difficulté sur votre code, demandez de l'aide à vos collègues et proposez une

séance de binômage ou une revue de code collective (voir chapitre 6 : la revue de code).

- Faites de même la prochaine fois que vous pratiquez TDD (voir chapitre 8 : *Test Driven Development*), ou n'importe quelle pratique de développement ; proposez de binômer à un collègue qui n'a pas l'habitude de cette pratique.
- Prenez le temps de discuter avec des collègues d'un article qui vous a marqué.
- Organisez un *Dojo* sur cette nouvelle librairie que vous souhaitez mettre en place sur votre projet.
- Allez voir d'autres équipes qui travaillent sur des sujets et des technologies proches des vôtres et échangez sur vos pratiques.
- Accompagnez l'intégration des nouveaux développeurs pour les familiariser au plus tôt à ces pratiques et guider leur apprentissage.

Qu'est-il ressorti de ces rencontres ? Qu'avez-vous appris ou transmis ? Ces expérimentations ont-elles été bénéfiques à l'équipe ?

Vue d'ensemble des éléments d'une politique de la qualité du code dans l'entreprise



MAINTENIR LA MAINTENABILITÉ



Maintenir la maintenabilité

Par Christophe Thibaut

Dans ce chapitre, nous examinons la question de la qualité du code en prenant le point de vue de l'organisation et du management.

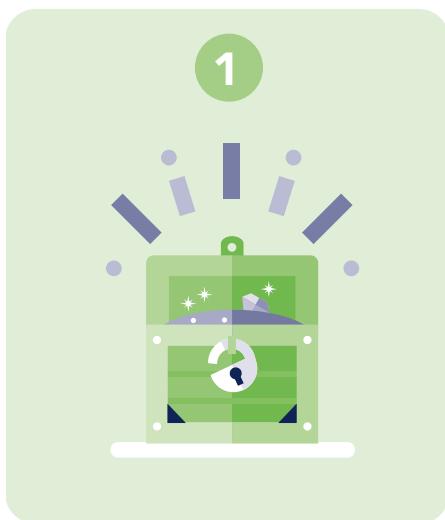
Si vous êtes manager, en quoi la qualité du code est-elle votre affaire ? C'est un **facteur stratégique de la « santé » du système d'information** de l'entreprise, et de sa capacité à innover sur le terrain de ce qu'on appelle aujourd'hui la « révolution numérique ». Par conséquent, la qualité du code que produit votre entreprise est une affaire dont vous devez vous préoccuper. Pour autant, il n'est pas nécessaire de vous former (ou de revenir) à la programmation, aux *Design Patterns* ou bien à TDD. Il est tout à fait possible de mener dans votre entreprise une politique de culture de la qualité du code sans connaître les arcanes du développement. Pour manager des équipes de développement remarquables, il est important de faire siens les principes directeurs qui suivent :

1. Le code est une source de valeur pour l'entreprise, et cette valeur doit être préservée par la mise en place de pratiques de développement de qualité.

2. L'équipe, son « produit » et ses « clients » forment un système, c'est-à-dire un ensemble structuré d'éléments en relation les uns avec les autres, visant un but particulier : livrer de manière efficace, continue et durable, des fonctionnalités ayant de la valeur.

3. Ce système peut être freiné dans ses performances au point qu'il devient impossible pour l'équipe de répondre de manière efficace à la fois aux demandes des clients (fonctionnalités) et aux problèmes de qualité du code.

4. Pour améliorer la qualité du code, il faut observer et intervenir sur le système dans son ensemble, notamment en créant un contexte qui favorise l'amélioration des pratiques.



Comprendre le rôle de la qualité dans le développement

○ Le code source, une valeur pour l'entreprise

Le premier principe du manager responsable d'équipes de développement est de considérer le code développé par ses équipes comme un « actif », au même titre que tout autre moyen de produire de la valeur dans l'entreprise. Prenons par exemple une imprimante.

Exemple de mesures prises afin de préserver la valeur d'un actif de l'entreprise

Rappel : L'imprimante X-ArcEnCiel disponible au 5^e étage, est un outil de travail précieux qui contribue à la valeur de ce que nous produisons. J'ai constaté qu'elle tombait souvent en panne et qu'elle était, de plus, utilisée pour des impressions personnelles. Sa détérioration diminue notre capacité à offrir des services de valeur. J'ai donc pris les mesures suivantes :

- Contrat d'entretien régulier avec le fournisseur X-ArcEnCiel.
- Restriction de son usage aux documents offerts aux clients.
- Contrôle d'accès et traces pour toutes les impressions.

Merci de votre attention.

Il en va exactement de même pour le code d'une application développée dans l'entreprise que pour l'imprimante de notre exemple. Si le manager pouvait déterminer l'état du code des applications de son entreprise, de manière aussi rapide et simple qu'il peut jauger l'état de l'actif *Imprimante*, il prendrait le cas échéant les mesures nécessaires pour que la valeur de cet actif soit préservée au mieux.

Exemples de mesures prises afin de préserver la valeur du code d'une application

Rappel : Le code source du projet XXL maintenu par votre équipe, contribue hautement à la valeur de ce que nous produisons. Sur les 6 mois qui précèdent, j'ai constaté que l'application XXL compte plus de 50 tickets d'anomalies, dont beaucoup sont des régressions suite à des corrections. J'ai, de plus, entendu dire que la maintenance était particulièrement difficile étant donné que le code est devenu « illisible ». La détérioration de ce code diminue notre réactivité et notre capacité à offrir des services de grande valeur. Je vous demande donc de réagir, et de mettre en place les mesures permettant de remettre ce code à l'état de l'art :

- Évaluation du niveau de qualité du code et plan d'amélioration.
- Formation et mise en place des pratiques nécessaires à sa remise en état.
- Mise en place d'un suivi régulier du niveau de qualité du code.

Merci de votre attention.

Pour l'imprimante, les éléments à prendre en compte dans la valeur de l'actif sont la valeur apportée par les impressions effectuées, le coût en énergie, en consommables, en temps passé à l'utiliser, en entretien et en remplacement éventuel.

Pour le code de l'application, les éléments à prendre en compte dans le calcul de sa valeur sont bien plus nombreux et plus complexes que pour une imprimante. On peut les résumer comme suit :

- Coûts de développement, de maintenance, de déploiement et de dépannage de l'application.
- Fonctionnalités de l'application et autres formes de revenus liés à l'application.
- Ergonomie, robustesse, fiabilité, performances et autres qualités perceptibles par l'utilisateur, dites « externes ».
- Lisibilité, évolutivité, modularité, et autres qualités perceptibles par les développeurs, dites « internes ».

Si l'on considère en détail les qualités dites « internes » de l'application, alors on ne peut pas limiter la valeur du code à celle d'un produit logiciel exécutable servant les opérations de l'entreprise.

Le code a aussi une valeur de documentation de la conception de ce produit ainsi que des règles liées aux opérations¹⁵. Enfin, sa valeur tient aussi au fait que **le code est un espace de travail pour les développeurs**, et à ce titre il reflète la culture de leur équipe et de leur entreprise.

Lorsqu'on parle de la valeur du code d'un projet, par exemple en disant « ce code ne vaut pas grand chose », on pourrait tout autant signifier que :

- Le logiciel produit ne fonctionne pas suffisamment correctement pour être utilisable,
- Le code est trop compliqué, fortement couplé ou illisible pour être compris et modifié sans un effort démesuré,
- Le code n'est pas un exemple à suivre pour les autres projets de l'entreprise.

Par conséquent, la qualité du code détermine la valeur du logiciel au moins de trois manières différentes : via les revenus des opérations de l'entreprise, par le coût induit des évolutions futures, et enfin en tant que source d'influence pour d'autres projets du système d'information de l'entreprise.

26

○ Mythe : « la qualité, ça coûte cher »

Une opinion couramment entendue à propos de la qualité du code est l'idée selon laquelle écrire du code de très bonne qualité demande beaucoup de temps. La raison qui fait que cette idée fausse est si largement répandue, est que tout problème de qualité se traduit **en surface** par un problème de temps. Typiquement, le logiciel est livré (rarement dans les délais initialement prévus), des défauts à corriger apparaissent, et cela n'était bien sûr pas compris dans le planning. Par conséquent, lorsqu'on demande à l'équipe d'y remédier, sa réaction spontanée est de demander du temps. D'où la conclusion (hâtive) : pour faire de la qualité, il faut plus de temps.

Il est évident que pour corriger les problèmes de qualité, il faut plus de temps. **Mais c'est donc la non-qualité qui coûte cher, et non la qualité.** Le constat réalisé par Capers Jones corrobore ce fait à partir de données issues d'un très grand nombre de projets :

« *The most effective way of improving software productivity and shortening project schedules is to reduce defect levels¹⁶* ».

15. De fait, la connaissance du domaine métier est « enfouie » dans le logiciel qui met en œuvre les opérations de ce métier. Ce phénomène s'accroît. (*« Software is eating the world »*).

16. Software Quality in 2013: a survey of the state of the art, Capers Jones <http://namcookalytics.com/wp-content/uploads/2013/10/SQA2013Long.pdf> (p.66)

Les problèmes de qualité constituent un facteur particulièrement important de retard des projets. On distinguera une catégorie de défauts de qualité à part qui sont hors du propos de ce livre : les incompréhensions sur les besoins, qui mènent à sous-estimer la complexité du projet ou bien à construire le mauvais produit, sont un autre facteur important. Les problèmes de qualité ne tombent pas du ciel comme par enchantement. Ils viennent du fait que le code écrit par les développeurs n'est pas maintenable, et donc que les pratiques de développement sont inadaptées¹⁷. Les développeurs ont d'ailleurs des termes choisis pour désigner cette plaie du métier de développeur qu'est le code non maintenable : code écrit à l'arrache, *quick & dirty*, code endetté (dette technique), code « spaghetti », usine à gaz, code *legacy*...

Qu'est-ce qui caractérise le code non maintenable ? Les symptômes sont à chaque fois les mêmes :

- Il est quasiment impossible de modifier ce code sans risquer d'introduire des erreurs.
- Lire, comprendre, et modifier ce code prend du temps et requiert une grande motivation.
- Les développeurs préconisent la refonte : il faudrait tout refaire.
- Ils finissent par coller quelques « verrues » ça et là pour répondre rapidement au besoin.
- Les meilleurs d'entre eux finissent par chercher un autre projet, voire une autre entreprise.

Le code non maintenable se traduit immanquablement par des évolutions plus coûteuses et plus risquées. Le code non maintenable, c'est – du point de vue de l'innovation – du **code sans avenir**.

Ø L'équipe de développement vue comme un système

Le code de l'entreprise doit être maintenable, car – c'est une quasi certitude – il devra être entretenu. Or, si c'est l'affaire des développeurs de maintenir le code et d'entretenir également sa maintenabilité, l'affaire du manager est de **créer et préserver un environnement** dans lequel ses équipes peuvent faire ce travail correctement. Autrement dit, pendant que l'équipe de développement produit et maintient un système informatique (une application, un produit, un service), le manager produit et maintient un « système de développement », constitué de l'équipe et son environnement.

Les éléments de ce système sont :

- Des acteurs : développeurs, *Product Owner*, architecte, etc.
- Des artefacts : code, tests, documents, tâches, etc.
- Des entrées et sorties : demandes, livrables, etc.

17. Si vous écrivez un programme de 15 000 lignes de code comme vous écririez un script de 15 lignes, sans tests unitaires, et sans processus de revue, alors vos pratiques de développement ne sont pas adaptées à la complexité de votre projet. Votre projet va se mettre très en retard, voire être annulé.

Ces éléments sont en relation et s'affectent mutuellement à l'intérieur du système. Par exemple :

- Les développeurs créent ou modifient du code en vue de produire un livrable qui répond à des demandes du « client » (appelé *Product Owner* en Scrum).
- Le « client » qui reçoit ce livrable envoie de nouvelles demandes à son propos.
- Le volume et la taille du code influencent la capacité des développeurs à prendre en compte de nouvelles demandes.
- Les développeurs ajustent le code et en améliorent le design (*refactoring*) afin de répondre plus facilement aux nouvelles demandes.

Tous ces éléments et ces relations constituent ensemble l'activité du système au cours du temps. Ce comportement est plus ou moins stable, et aboutit à un résultat plus ou moins proche du but qui est affecté au système. À savoir : produire, livrer et maintenir un logiciel répondant aux besoins de ses utilisateurs et aux contraintes de l'organisation.

Pourquoi est-il intéressant de se représenter l'équipe, son produit, ainsi que ses relations avec l'extérieur comme un système ?

Cela permet d'observer, de raisonner et d'intervenir à partir de données constatées sur le comportement du système, plutôt qu'à partir de ses propres interprétations ou impressions.

Cela permet de raisonner et d'agir en considérant la structure du système comme la source de son comportement, plutôt que de tenter d'agir sur un élément en particulier comme s'il était à lui seul la cause de ce comportement.

Enfin, cela permet d'anticiper, à l'aide d'un modèle adéquat basé sur des boucles de renforcement ou des boucles d'équilibrage, les différents comportements du système qui le mènent à la stabilité, ou bien à une crise.

Conversation à propos d'un projet observé à travers des variables non pertinentes

- Je trouve que l'équipe de Michel est quand même assez souvent à la machine à café, je me demande si le projet avance comme il faut...
- Que disent les résultats des derniers sprints ?
- Oh ça paraît correct à première vue, mais si tu compares avec l'équipe de Jérémie, ceux-là semblent plus concentrés, et en plus ils restent le temps qu'il faut pour que ça marche !

Conversation à propos d'un projet observé à travers des variables pertinentes

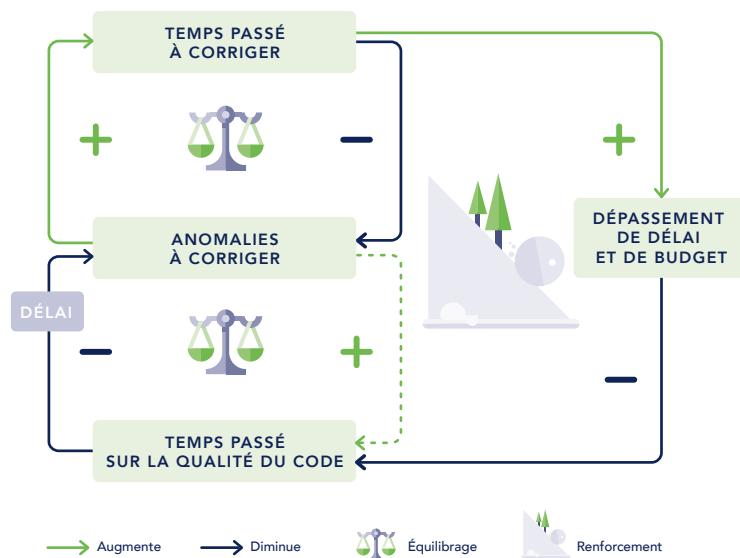
- En ce qui concerne le projet VIC, je n'ai rien à signaler. Le projet a tardé à livrer ses premières stories, mais c'est stable, et je ne vois aucun bug.
- On m'a dit que l'équipe passait trop de temps à écrire des tests...
- Je ne m'occupe pas du temps passé par les développeurs.
- Comment tu sais que ça marche alors ?
- Tout le monde a accès au board du projet. Tu peux y voir les tickets d'anomalies et le niveau de dette technique.
- Oui, mais est-ce que l'équipe délivre ?
- Demande au Product Owner, il vient à 11h.
- Je sais, je l'ai croisé ce matin. Donne-nous ta recette !



○ Le cas de crise le plus fréquent : l'addiction aux bugs

Le phénomène d'addiction aux bugs est tellement fréquent qu'il mérite qu'on le décrive en détail. Il correspond à un archétype systémique bien connu¹⁸, dans lequel un problème est résolu à l'aide d'une solution à court terme, laquelle ne traite que le symptôme et non les causes profondes du problème, ce qui produit une aggravation de la situation.

Addiction aux bugs : une boucle de renforcement négatif



18. <http://www.systems-thinking.org/theWay/ssb.htm>

Le symptôme : l'équipe reçoit des tickets de demande de correction de bugs. Il y a un écart entre la qualité perçue et la qualité attendue.

La cause profonde : c'est le niveau de qualité structurelle du code.

Dans un code bien conçu, lisible, modulaire, bien testé, les risques d'insertion de bugs sont moins élevés. Dans un code illisible, couplé, non testé, les erreurs de programmation sont plus fréquentes.

La contre-mesure symptomatique : l'équipe corrige les bugs, en mode *quick and dirty*, c'est-à-dire en urgence, à l'aide d'expédients.

L'effet sur le symptôme : les bugs sont corrigés, et les tickets sont « résolus ». La qualité perçue s'améliore.

On a donc là apparemment une boucle de rétroaction en équilibre : lorsque le nombre de tickets augmente, l'équipe augmente ses efforts de correction en mode urgent, jusqu'à ce que le flot de tickets se tarisse. Après quoi l'équipe revient à un travail « normal ». Tout semble aller pour le mieux.

L'effet secondaire : les interventions *quick and dirty* pour corriger les bugs ne permettent pas de traiter le problème fondamental, c'est-à-dire d'améliorer la qualité structurelle du code. En effet, ces corrections sont réalisées en urgence et au plus vite, parce qu'elles mettent le projet en retard. Dans un projet en retard, les développeurs n'ont plus le temps d'améliorer la qualité du code.

On a donc une **seconde boucle de rétroaction en renforcement** : la qualité du code étant ce qu'elle est, de nouveaux bugs apparaissent. Leur correction en urgence contribue à détériorer un peu plus la qualité du code – détérioration qui amènera de nouveaux bugs, d'où une situation de retard et d'urgence croissants, qui finit généralement par une « explosion » du système : action de recouvrement, annulation du projet, changement d'équipe.

Exemple de renforcement et d'équilibrage à l'œuvre sur un projet qui a des problèmes de qualité

- *J'entends bien ton conseil. Tu dis qu'on devrait écrire des tests unitaires sur le code, mais là, on n'a pas le temps, parce qu'on doit livrer en fin de semaine impérativement.*
- *Qu'est-ce qui vous prend plus de temps que prévu ?*
- *C'est surtout ces tickets qui sont tombés lundi : bugs à corriger ASAP.*
- *Et lorsque vous les corrigerez, vous allez ajouter des tests ?*
- *Non. On n'a pas le temps, je t'ai dit !*

Exemple de renforcement et d'équilibrage à l'œuvre sur un projet qui résout ses problèmes de qualité



- *J'ai corrigé le ticket 17 sur ma story...*
- *Tu as écrit un T.U ?*
- *C'est juste un indice de boucle qui...*
- *Tu peux écrire un T.U, stp ? Le code est couvert de T.U.s. C'est ce qui nous évite de perdre du temps avec des régressions.*
- *C'est une seule ligne de code !*
- *Alors le T.U ne devrait pas prendre trop de temps non plus.*
- *Non, je peux l'écrire en 10 minutes. Mais je ne comprends pas votre logique.*
- *Tu viens d'arriver ; fais nous confiance. Tu as passé combien de temps sur ce bug ?*
- *Je sais pas, deux heures...*
- *Le temps que tu passes sur les tests te fait gagner le temps que tu ne passes pas sur les bugs.*

Ø En quoi la correction de bugs en mode urgent est-elle une addiction ?

Une addiction est un phénomène dans lequel un comportement de remède symptomatique est renforcé au point que :

- Le remède à court terme engendre plus de problèmes qu'il n'en résout.
- Le remède à court terme est de moins en moins efficace au cours du temps.
- Supprimer le remède à court terme provoque un état de douleur difficilement supportable¹⁹.
- Le remède à long terme (c'est-à-dire sur les causes profondes) ne se traduit **jamais** par un effet observable **immédiat**.

19. Comme le savent ceux qui ont déjà demandé à leur client une « trêve » afin d'arrêter le travail en mode urgent et de remédier à la qualité après une longue série de bug-fixes, il est très difficile de résister à la pression dans un projet en retard.

Exemple d'une tentative pour remédier aux causes structurelles plutôt qu'aux symptômes



- Je ne voudrais pas jouer les rabat-joie, mais il y a un sérieux problème de qualité dans ce que vous faites.
- Je vous assure qu'on fait tout ce qu'il faut pour que les bugs soient corrigés au plus tôt.
- Et pendant ce temps là, les fonctionnalités n'arrivent pas !
- Toute l'équipe est sur le pont.
- Et comment expliquez-vous le fait qu'une correction entraîne des régressions dans l'application ? Je reviens sur des incidents que je croyais résolus, et je les constate à nouveau !
- Pour bien faire, il faudrait qu'on prenne le temps de poser des tests, et de revoir la structure du code.
- Ça va prendre combien de temps ?
- Justement c'est le problème. Il faudrait qu'on remette certaines corrections à la livraison suivante...
- Quoi ?! Et déployer l'application dans cet état ? Hors de question !

○ La qualité structurelle du code est-elle le facteur principal de réduction du nombre de bugs ?

Oui, si vous comptez dans la qualité du code le fait qu'il soit pourvu de tests unitaires et qu'il soit lisible, correctement conçu, cohérent, sans redondance. Pour vous en convaincre, il suffit de participer à la **rétro-analyse** (ou méthode des « 5 pourquoi²⁰ ») de quelques bugs. Tous les retours sur une application ne sont pas dus à des erreurs de programmation, mais en revanche **toutes les erreurs de programmation peuvent être liées à la qualité du code.**

20. La méthode des « 5 pourquoi » consiste à poser successivement cette question lors de l'analyse d'un problème, afin d'identifier les causes qui sont à la racine du problème.

Exemple de rétro-analyse utilisant la méthode des « 5 pourquoi »



- Le bug, c'est que lorsque j'édite le rapport 2014, et ensuite le rapport 2015, alors le rapport 2015 affiche le total des ventes 2015 plus celui de 2014.
- Pourquoi affiche-t-il celui-là en plus ?
- Parce que le dataset qui contient les données n'a pas été refermé après l'édition de 2014.
- Pourquoi n'a-t-il pas été refermé ?
- Je pensais que c'était une variable locale et que la sortie de la routine libérerait cette variable. Mais c'est une variable globale en fait.
- Pourquoi est-ce une variable globale ?
- Je n'en suis pas certain, mais je pense que ça devait être pour une raison de performances, pour ne pas à avoir rouvrir un dataset.
- Les variables globales sont interdites sauf exception documentée. Pourquoi cette exception n'était pas documentée ?
- Probablement parce qu'on n'a pas fait de revue sur cette partie du code.
- Pourquoi est-ce qu'on n'a pas fait de revue ?
- Parce que je pensais que ce n'était qu'une petite modification, et qu'on était déjà en retard.

○ Pourquoi l'équipe se laisse-t-elle piéger si facilement dans la dépendance au « bug fix » ?

Sans une discipline de tests unitaires et de revues de code, le nombre de bugs dans l'application augmente imperceptiblement. Ce n'est que lors des tests de recette, voire en production, qu'ils apparaissent. Il se crée donc un stock de bugs, et ceux-ci ne sont révélés qu'à chaque recette, longtemps après leur insertion.

La relation entre la qualité structurelle du code et le nombre de bugs est une relation d'influence induisant des délais. Les bugs, par définition, apparaissent à l'insu du développeur et ne sont pas détectés en temps réel. Cela signifie qu'une action sur la qualité du code n'a pas d'incidence immédiate sur le nombre de bugs en cours, mais seulement sur la probabilité d'apparition de nouveaux bugs dans le futur.

La relation entre la qualité du code et le nombre de bugs futurs est donc imperceptible au jour le jour. C'est seulement lorsque l'équipe reçoit une nuée de tickets de retours en anomalies, que le problème de qualité devient perceptible, et alors il est souvent trop tard.

○ Comment éviter l'addiction aux bugs ?

Ce phénomène est extrêmement courant, mais il n'est pas inéluctable. Voici quelques mesures qui permettent de l'éviter :

1. Tester systématiquement le code au niveau unitaire, c'est-à-dire au plus près de la ligne de code en train de s'écrire.

2. Effectuer des revues de code systématiques sur tout le code nouvellement écrit.

Ces deux pratiques à elles seules vont permettre à l'équipe d'améliorer sa sensibilité aux problèmes de qualité : les détecter avant la recette, les corriger par les mesures appropriées, et ainsi éviter d'entrer dans le cycle infernal de la correction de bugs en urgence.

3. Améliorer le partage d'information et de décisions à l'intérieur de l'équipe à propos de la qualité structurelle du code.

4. Accompagner chaque action de correction par une action complémentaire sur la qualité structurelle du code : rétro-analyse, ajout de tests, revue, *refactoring*.

5. Surveiller 3 métriques simples afin d'anticiper les crises, à savoir :

a. Débit de fonctionnalités

b. Bugs constatés

c. Niveau de dette technique

○ Anticiper et éviter une crise de qualité dans un projet

Est-il possible de détecter au plus tôt l'apparition de ce piège systémique pour l'équipe de développement, de manière à corriger le comportement du système avant que celui-ci ne déraille ? C'est possible, et cela même sans être un développeur, ni même un ancien développeur. Pour cela, il est nécessaire de mettre en place et de suivre régulièrement quelques métriques basées sur un modèle simple du système de développement.

Ø Débits, Stocks et Mesures

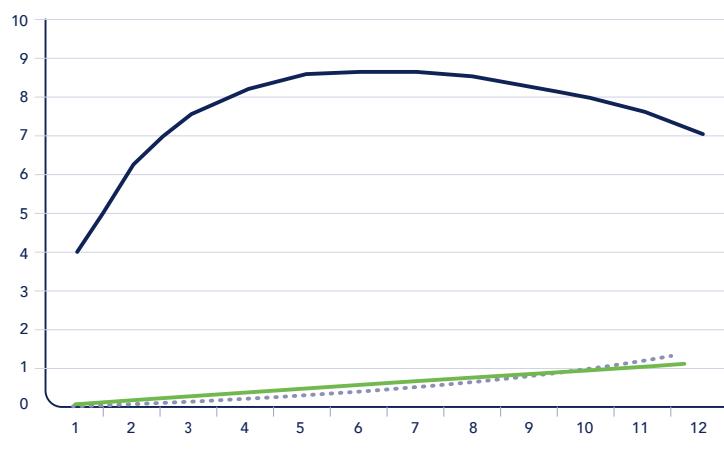
Le but du système est de mettre en production des fonctionnalités dans les meilleurs délais, en respectant les contraintes de coûts (notamment la taille de l'équipe), et de qualité (standards de l'organisation).

F : Fonctionnalités. On pourra donc mesurer la performance du système en relevant pour chaque période considérée (par exemple, pour chaque itération) le nombre de fonctionnalités (*User Stories*) demandées en entrée du système et livrées en production, ainsi que l'évolution de deux stocks.

B : Bugs. Entrera dans ce stock tout ce qui est identifié et suivi par l'équipe comme « à corriger sur le code de l'application » : bugs, demandes de correction, en prenant soin de distinguer les problèmes de qualité du code (ex : plantage) des problèmes de compréhension entre l'équipe et le client (ex : ambiguïté dans un critère de recette).

D : Dette Technique. Entrera dans ce stock, tout ce qui est identifié et suivi par l'équipe comme un sujet de refactoring ou d'amélioration technique. Par exemple : la complexité cyclomatique²¹ du code, redondances dans le code, etc. La définition de cette mesure peut être différente d'une équipe à l'autre. **L'essentiel est que pour une équipe donnée, cette définition ne change pas au cours du temps.**

L'évolution de ces données au cours du temps traduit le comportement du système.

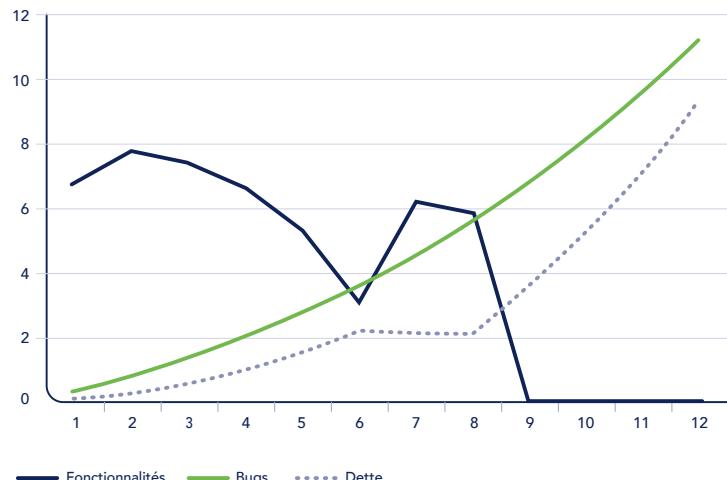


Graphique 1

21. Complexité cyclomatique : mesure de la complexité d'un programme. En simplifié, correspond aux nombre de chemins possibles + 1 dans un programme. Voir aussi https://fr.wikipedia.org/wiki/Nombre_cyclomatique.

Graphique 1 : Modélisation d'un projet dans lequel l'équipe consacre respectivement 40 %, 30 % et 30 % du temps disponible (après correction des bugs) aux activités de développement, tests unitaires et désendettement.

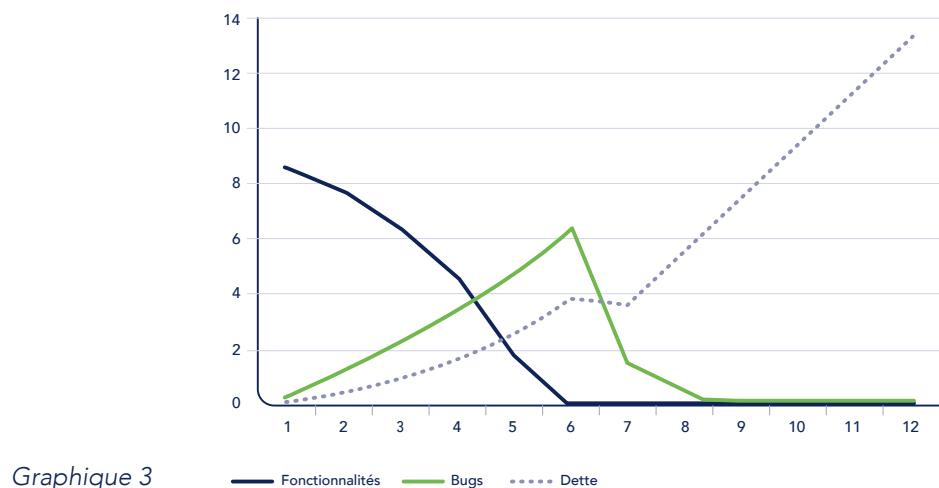
Courbes en augmentation, stabilisation des fonctionnalités, puis lente décroissance : c'est le comportement « nominal » du système, en particulier au début du projet, jusqu'à ce que les stocks se stabilisent. **L'équipe maintient la maintenabilité du code.** À mesure qu'une application s'enrichit en fonctionnalités et se complexifie, le nombre de bugs et le niveau de dette technique augmentent également, même si toutes les contre-mesures pour maintenir la maintenabilité sont prises. En effet, il y a nécessairement plus de bugs, et plus de dette technique dans un grand programme que dans un petit, à niveau de qualité constant. De ce fait, la courbe F s'aplatit, à cause du surcroît de temps passé à corriger les bugs, et de la « taxe » de maintenabilité induite par la dette technique. À long terme, le code du programme se dirige lentement vers un état de « legacy », où chaque nouvelle fonction coûte un peu plus cher, occasionne plus de bugs, et dégrade un peu plus la qualité du design.



Graphique 2 — Fonctionnalités Bugs Dette

Graphique 2 : Modélisation d'un projet dans lequel l'équipe consacre respectivement 70 %, 15 % et 15 % du temps disponible (après correction des bugs) aux activités de développement, tests unitaires et désendettement, avec un sursaut de désendettement de 50 % sur les périodes 7 et 8.

Courbe « Fonctionnalités » en diminution, stocks « Bugs » et « Dette Technique » en augmentation : le système est en crise. Le niveau de qualité de l'application engendre une surcharge de travail : bugs à corriger, régressions, tests supplémentaires. De plus, la mauvaise maintenabilité du code (lisibilité, modularité, facilité de modification) fait que chaque tâche de programmation prend plus de temps. **L'équipe est surchargée de travail, mais ses performances diminuent.** À long terme, le projet évolue vers une immobilité complète (voir les graphiques 2 et 3).



Graphique 3 : Modélisation d'un projet dans lequel l'équipe consacre respectivement 90 %, 5 % et 5 % du temps disponible (après correction des bugs) aux activités de développement, tests unitaires et désendettement, avec un changement d'activité à partir de la période 7, où elle passe à 75 % de tests unitaires pour le reste du projet.

Agir en vue d'améliorer la qualité des développements

○ Quelles contre-mesures éviter lorsque le système est en crise ?

Lorsque le débit de fonctionnalités diminue, tandis que les stocks « bugs » et « dette technique » augmentent, le système est en crise. Comment intervenir ? Quelles sont les mesures à prendre ?

Tout d'abord intéressons-nous, afin de les éviter, aux mesures qui ne sont pas efficaces, voire contre-productives, que nous appellerons les « fausses bonnes idées » :

Fausse Bonne Idée n° 1 : Chercher à augmenter la capacité de l'équipe, en faisant pression pour que les équipes fassent plus d'heures, ou en faisant entrer de nouveaux développeurs dans l'équipe. Une telle mesure s'appuie sur une interprétation erronée des indicateurs et une représentation incomplète du modèle : *puisque le débit des fonctionnalités ralentit, il suffit d'ajouter de la main d'œuvre.* Or en augmentant la taille de l'équipe, on ne change pas les tendances indiquées par l'évolution des stocks, mais on les amplifie, puisque ces tendances sont dues à la façon de travailler de l'équipe. Le système produira de plus en plus de bugs, de dette technique, et de moins en moins de fonctionnalités en état de marche.

Fausse Bonne Idée n° 2 : Travailler sur les symptômes au lieu d'identifier les causes profondes, par exemple en étoffant l'équipe de testeurs, en corrigeant les bugs de manière précipitée, ou bien encore en contournant les procédures habituelles afin de gagner du temps. Ces mesures sont un peu comme le ruban adhésif qu'on utilise faute d'une solution adéquate. Elles sont sans influence sur le problème de la qualité structurelle du code et le manque de pratiques de qualité.

Fausse Bonne Idée n° 3 : Rechercher de meilleurs développeurs, s'avère également inefficace sauf si ces développeurs arrivent à changer la manière dont se déroule le développement dans l'équipe, ce qui est très peu probable. Un développeur même chevronné ne peut rien contre un système inefficace. S'il se retrouve dans un environnement où la qualité reste lettre morte, voire est controversée ou même délibérément empêchée, il essaiera pendant un temps de pousser ses coéquipiers à mettre en place de meilleures pratiques. Mais s'il est découragé dans cette initiative, il quittera rapidement le poste.

Fausse Bonne Idée n° 4 : Trouver un responsable, faire pression, menacer, sanctionner.

Ces « techniques de management » sont autant inefficaces que nocives. Elles reposent sur un biais cognitif, l'erreur fondamentale d'attribution, par lequel on attribue les causes du comportement d'une personne à ses dispositions internes plutôt qu'à la situation elle-même. Lorsqu'un manager « durcit le ton » avec les membres de l'équipe, c'est comme s'il voulait dire que ce sont leurs dispositions (leur motivation, leur compétence, leur attitude) qui gouvernent la performance du système plutôt que le contexte. Or c'est généralement l'inverse qui est vrai : c'est l'environnement, le contexte dans lequel on travaille, qui conditionne les performances, et non les dispositions individuelles. Un développeur pourra se trouver très motivé et très attentif à la qualité dans le contexte d'un projet open source auquel il contribue ; mais désinvolte ou simplement débordé, dans le contexte d'un projet « professionnel » en crise de qualité.

Exemple d'une erreur fondamentale d'attribution

- 
- *Il faut que je parle à Jérôme, ça ne va plus.*
 - *Ah ?*
 - *Oui. Ce que je n'apprécie pas chez Jérôme, c'est son attitude.*
 - *Ah bon ?*
 - *Oui. Le projet est en retard de plusieurs semaines, et lui, il ne s'en fait pas plus que ça. Je ne le trouve pas très engagé.*
 - *Pour avoir travaillé avec lui pendant des années, je peux te dire que ça ne lui ressemble pas.*
 - *Alors comment tu expliques son comportement ? On dirait qu'il se fiche de savoir si le projet va réussir à temps.*
 - *C'est lui qui a suggéré que l'équipe prenne un temps pour se former à TDD...*
 - *Je sais, et j'ai refusé. On est en retard, bon sang !*
 - *Je ne sais pas si tu as remarqué, mais le nombre de retours en anomalie augmente chaque semaine.*
 - *J'ai remarqué, figure-toi. J'ai encore fait un rappel à ce sujet hier matin. L'équipe doit corriger tous ces bugs avant jeudi midi.*
 - *Je peux te poser une question ?*
 - *Vas-y.*
 - *Comment tu réagis, lorsque tu reçois deux consignes contradictoires ?*

Durant les revues de code, chaque développeur fait bien le principe d'être « dur avec le code, doux avec les personnes ». **Le manager d'une équipe de développement devrait être « dur avec les mauvaises pratiques, doux avec les personnes ».** Dans un projet en crise, on voit parfois des managers « durcir le ton » avec des membres de l'équipe, tandis que rien n'est fait pour que les pratiques de développement changent. C'est une recette assurée vers la catastrophe.

○ Que faire pour éviter de se retrouver en crise de qualité ?

Surveillez les indicateurs sur les « stocks » :

- **F** : Nombre de fonctionnalités livrées,
- **B** : Nombre de retours en anomalie,
- **D** : Niveau de dette technique.

Ces indicateurs sont des **mesures simples** et doivent le rester. Il est inutile de les décomposer en sous-catégories, de faire des exceptions, d'agencer des règles de calcul alambiquées. En revanche, il est crucial de conserver la même mesure tout au long du projet, même dans le cas où celle-ci paraît incomplète ou imprécise. Il vaut mieux avoir une métrique imparfaite mais qui reste cohérente dans le temps, plutôt qu'une métrique « pointue » mais dont les règles de calcul changent toutes les trois semaines.

Soyez à l'affût des signaux faibles de fléchissement du projet.

Surveillez les trois quantités F, B et D, itération après itération. Lorsque la courbe de débit de fonctionnalités se tourne vers le bas alors que le nombre de bugs ou la dette technique augmentent, cela signifie que les problèmes de qualité du code commencent à entraîner l'équipe dans la spirale infernale de l'addiction aux bugs.

FONCTIONNALITÉS	BUGS	DETTE TECHNIQUE	SITUATION
↗	↘	↖ ↗	Miraculeux !
↗	→	→ →	Satisfaisant
→	→	→ →	R.A.S
↘	→	→ →	Danger !
↘	↗	↗ ↖	Projet en crise de qualité
↘	↘	↖ ↗	Projet immobilisé le temps de désendetter et débuguer

Forme des courbes « fonctionnalités, bugs et dette technique » et leur interprétation

Observez avant d'agir. Les décisions prises à la hâte en face d'un système que l'on comprend mal sont en général contre-productives, parce qu'elles partent d'une préoccupation qui concerne les symptômes et non les causes profondes.

Exemple du contrecoup d'une intervention non pertinente

- 
- Qu'est-ce que tu as pensé du petit rappel sur les horaires en début de rétro, vendredi dernier ?
- Bah. C'est le chef, c'est normal. Et toi ?
- Ça m'a cassé le moral pour la journée.
- Y en a certains qui exagèrent tout de même. Arriver à 10 heures...
- Tu sais pas à quelle heure ils repartent !
- Pas faux.
- Pour ce que ça va changer de toute façon...
- Ben si : plus d'heures de travail...
- Oui ? De combien est le manque à gagner ?
- Hmm. Disons 8 heures par semaine, au maximum.
- Et combien d'heures il nous faudrait pour fixer tous les bugs ?
- Ouh là. Au moins 30 heures, je pense. En ce moment on crée plus de bugs qu'on n'en résout, en fait.
- Voilà.
- Tu proposes quoi ?
- Si c'est pour écrire du code pourri, 6 heures par jour, c'est moins pire que 8.

Voici des exemples de symptômes parmi les plus fréquents lorsqu'un projet traverse une crise de qualité :

- Développeurs débordés de travail.
- Horaires de travail en dehors de la normale.
- Discussions houleuses à propos des mesures à prendre.
- Désaccords ouverts ou latents à propos du design, de l'architecture.
- Surprises lors de la démonstration du produit.
- Ambiance de travail maussade.
- Postures de blâme, défensives, ou de sacrifice.
- Certains membres de l'équipe sont perçus comme des « sauveurs ».

- Certains membres de l'équipe sont des boucs émissaires.
- Explications multiples sur les raisons du retard.
- Séances d'estimations de plus en plus longues.
- Demandes des développeurs d'obtenir plus de spécifications écrites.
- Réunions de crises.
- Accusations, médisance, rumeurs...

Ces symptômes, pour spectaculaires ou préjudiciables qu'ils paraissent, n'appellent pas d'action spécifique de la part du manager, en particulier si les causes profondes ne font l'objet d'aucune contre mesure.

Agissez sur le contexte et l'environnement, en donnant à l'équipe les moyens de progresser :

Interrogez l'équipe afin de savoir où elle en est de ses pratiques et ce qui lui manque pour se mettre à niveau :

- Quelles sont les pratiques de développement connues et reconnues dans l'équipe ?
- L'équipe fait-elle des tests unitaires, des revues ? A-t-elle un standard ?
- L'équipe est-elle formée à ces pratiques ?
- Quelles sont les compétences et les lacunes dans l'équipe ?
- L'équipe a-t-elle du temps pour améliorer ses pratiques ?
- Y a-t-il une émulation pour s'améliorer ?
- Y a-t-il du temps alloué dans les budgets pour se former ?
- L'espace de travail se prête-t-il à l'apprentissage ?
- Les moyens de communication, d'échange d'information et de partage des pratiques sont-ils adéquats ?

Tenez compte des effets systémiques : dans tout système, un stock se crée petit à petit, jour après jour. C'est la manifestation d'un processus qui prend du temps. Soyez à l'affût des signaux faibles. La dette technique qui paralyse les évolutions du code n'apparaît pas du jour au lendemain sur un projet, mais petit à petit. Ne mettez pas l'équipe face à des attentes irréalistes, comme de prétendre résorber en quelques jours tous les retours en anomalies ou toute la dette technique accumulée ces derniers mois. Il n'y a pas de remède miracle, ce qui est arrivé par le truchement de mauvaises habitudes ne peut être réparé qu'en changeant d'habitude. Changer d'habitude est un processus parfois long, et qui ne se fait pas sans difficultés.

Ne remettez pas les améliorations à plus tard. Si votre système de développement est entré dans une boucle de renforcement telle que l'addiction aux bugs, les mesures que vous repoussez

aujourd'hui parce qu'elles sont coûteuses ou trop contraignantes, seront encore plus coûteuses et contraignantes dans quelques temps.

Sagesse de développeurs

- *Quel est le meilleur moment pour écrire un test sur cette partie du code ?*
- *C'était lorsqu'il a été écrit.*
- *Le deuxième meilleur moment est maintenant.*



Une intervention posée, mais néanmoins particulièrement courageuse dans un projet en crise

À : l'équipe XXL

Sujet : compte rendu de la rétrospective de projet

Merci à tous d'avoir participé à cette rétro.



Constats :

- *Le projet est en retard de trois mois par rapport au planning initial.*
- *Il reste encore 52 tickets de retours ouverts, dont 15 ne sont pas encore qualifiés (bugs ou incompréhensions).*
- *Sur les deux derniers mois, l'équipe a passé plus de 50 % du temps à retravailler du code pour des raisons de qualité (correctifs, réaménagement du code, refonte d'un module dans son intégralité).*

Décisions :

- *Pour chaque retour en anomalie lié à des erreurs de programmation, conduire un « 5 pourquoi », en documentant les décisions prises sur le wiki.*
- *L'enquête est lancée sur les pratiques de développement : qui pratique TDD, les tests unitaires ? d'autres tests, le pair review ?*
- *Une formation à TDD sera mise en place dans le mois qui vient.*
- *Mise en place d'une revue de code, chaque vendredi entre 10h et 12h, en ciblant en priorité le code modifié récemment. Les décisions prises en revue seront documentées sur le wiki.*
- *Nous faisons un nouveau point sur la mise en place de ces actions d'ici un mois, un autre point sur la qualité du projet d'ici 3 mois.*

Bon courage à tous.

○ Je suis le manager de ces chefs d'équipe : que faire ?

Un manager responsable d'un service composé de plusieurs équipes de développement aura beau s'intéresser aux pratiques de qualité dans le développement comme à un sujet qui lui tient à cœur, il n'aura dans tous les cas que peu de temps à y consacrer. Est-ce que cela signifie pour autant que la qualité des développements n'est pas son affaire ? Certainement pas. Après tout, l'impact de la non-qualité sur un projet détériore les performances de son service, mais est de plus préjudiciable à la culture de l'entreprise ainsi qu'à son image.

En tant que responsable d'un service que devez-vous faire afin de veiller à la qualité ?

Surveillez des indicateurs de haut niveau :

Pour chaque application, obtenez un rapport mentionnant, par période de livraison :

- Le nombre de bugs rencontrés par fonctionnalité livrée,
- La proportion des bugs qui sont corrigés avant la mise en production par rapport à l'ensemble des bugs rencontrés sur l'application.

Ces deux indicateurs traduisent un état global de la qualité du produit, et non seulement du code. Bien sûr, la qualité d'une application est influencée par bien d'autres paramètres que la seule qualité structurelle du code : qualité du pilotage fonctionnel, qualité des expressions de besoins, tests fonctionnels, etc. Il n'empêche qu'une crise de la qualité du code au niveau des développeurs aura un effet visible sur ces courbes quoi qu'il arrive par ailleurs. Il n'est pas nécessaire de connaître dans le détail la manière dont ces « stocks » (bugs par fonctionnalités) sont mesurés. En revanche il est important que ces métriques **restent cohérentes dans le temps**.

Définissez et aidez au déploiement d'une politique culturelle de la qualité :

- Mettez en place un programme de formation lorsque les équipes manquent des compétences de base en termes de pratique de qualité : tests unitaires, TDD, revue de code.
- Demandez à ce que soient mis en place des rituels standards consacrés à la qualité, par exemple une revue de code systématique par toute l'équipe à raison de deux heures par semaine (ou bien sous une autre forme). Il devrait être possible de consulter les rapports de revues, non pas afin d'évaluer les développeurs, mais afin de vérifier que ces revues ont lieu, et qu'elles sont efficaces.
- Soutenez vos chefs de projet dans la démarche d'amélioration des pratiques de la qualité, en négociant auprès des clients métiers ou des sponsors du projet le temps nécessaire à l'équipe pour se former et mettre en place les bonnes habitudes.
- Aidez les développeurs à diffuser les bonnes pratiques en encourageant la mise en place de communautés de pratiques dans l'entreprise qui bénéficieront du temps, de l'espace, et d'outils

de communication propres à transmettre efficacement ces pratiques.

- Lorsque le contexte n'est pas idéal et qu'il manque des moyens et du temps, encouragez les petites expérimentations sur un temps court et qui vont dans le sens de l'amélioration de la qualité.
- Faites preuve de patience, d'écoute, et encouragez vos équipes. **Une culture de la qualité ne se crée pas en un jour.**

Ø Points à retenir

Le code logiciel produit est un **actif apportant de la valeur** à la société qui le produit. À ce titre, il est primordial pour une organisation de tout mettre en œuvre pour en prendre soin. **Prendre soin du code, c'est entretenir ses qualités structurelles :** fiabilité, robustesse, lisibilité, modularité, facilité de compréhension et d'évolution.

La responsabilité de cette qualité n'incombe pas seulement aux développeurs. Un logiciel est le fruit de l'activité d'un système et donc des interactions entre tous les interlocuteurs (développeurs, managers, fonctionnels,etc.). Il est important de **considérer le système dans son ensemble**, pour, en cas de problème, **rechercher les causes profondes plutôt que de « traiter les symptômes ».**

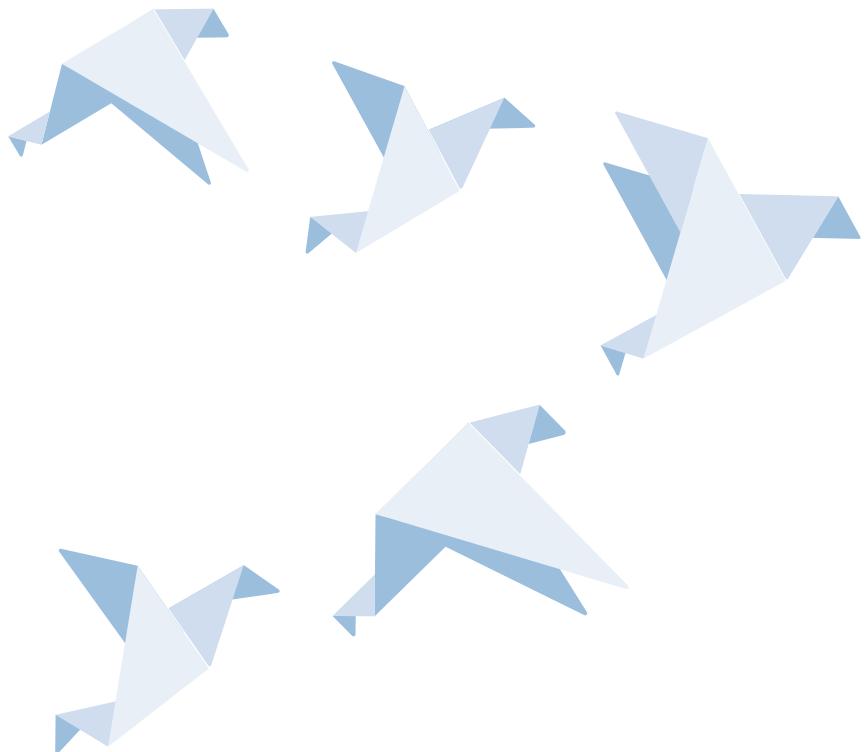
Une des responsabilités du manager est de s'assurer que ses équipes produisent un logiciel de qualité. S'il ne participe pas directement à augmenter cette qualité, son rôle sera toutefois de **créer l'environnement** dans lequel cette qualité pourra émerger.

Ø Par où commencer ?

- Préparez dès maintenant un suivi de quelques indicateurs sur le projet : Fonctionnalités livrées, Bugs, Dette technique.
- Faites un état des lieux des pratiques en place dans vos équipe, et identifiez quelques éléments d'investissement (en termes de temps ou de formation).
- Faites mettre en place dès aujourd'hui un rituel de revue de code hebdomadaire.
- Communiquez avec vos équipes sur l'importance de la qualité.

Négociez auprès des métiers pour donner le temps à vos équipes de mettre en place cette culture.

LE TECH LEAD, AU SERVICE DE L'ÉQUIPE



Le Tech Lead, au service de l'équipe

Par Michel Domenjoud

Dans Tech Lead, il y a deux dimensions aussi importantes l'une que l'autre : Tech et Lead. Ce rôle intègre des responsabilités de prise d'initiatives et de décisions sur les aspects techniques, **mais également une composante de leadership**, centrée sur les relations humaines. Cette dimension – la plus délicate, mais aussi la plus riche – est malheureusement trop souvent ignorée ou délaissée.

Tout n'est donc pas aussi simple que la sémantique le laisse paraître. Un élément est revenu plusieurs fois lors des échanges que nous avons eus avec des Tech Leads :

« J'aurais aimé trouver quelque part une définition de ce rôle plutôt flou ». En réalité, ce n'est pas tant qu'il est flou ; **ce rôle implique plusieurs facettes**, sans pour autant nécessiter d'être un « mouton à cinq pattes ».

Réaliser un produit logiciel de qualité est un travail d'équipe. Pour y parvenir, l'équipe doit **s'organiser collectivement** pour garantir l'excellence technique, assurer une **communication et des échanges de qualité**

autour du code et faciliter la progression **en organisant un apprentissage continu et pérenne**.

Dans cet écosystème, **le Tech Lead est un leader au service de l'équipe**, un facilitateur qui l'aide et la conseille pour atteindre ce niveau d'excellence.

○ Une vision du rôle de Tech Lead

Dans *Becoming A Technical Leader*²², Gerald M. Weinberg introduit le rôle de Tech Lead comme un *Problem Solver* ; une personne qui aide l'équipe à trouver des solutions aux problèmes – principalement techniques. Il ne règle pas ces problèmes tout seul, au contraire : il crée une situation de confiance, afin que l'équipe soit en capacité de les résoudre par elle-même.

Pour créer cet environnement propice au changement et à la résolution des problèmes, Weinberg propose d'utiliser le modèle **M.O.I.** :

22. *Becoming A Technical Leader*, Gerald M. Weinberg, 1986

M

MOTIVATION

Soutenir la motivation de l'équipe.

O

ORGANISATION

Mettre en place une organisation qui favorise la collaboration et la résolution collective des problèmes.

I

INNOVATION IDÉES

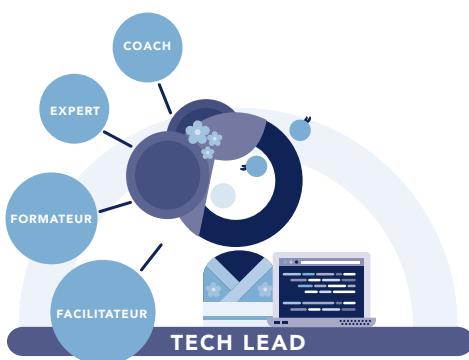
Être ouvert aux idées et la résolution collective des problèmes.

Le Tech Lead doit donc **être capable d'élever le niveau de toute l'équipe, en donnant l'exemple et en cherchant à la rendre autonome.**

> Un rôle à quatre facettes

La complexité du rôle de Tech Lead réside dans l'art de composer avec quatre facettes principales, et donc de changer de posture selon les situations :

- **Expert** pour porter la vision technique du produit,
- **Formateur** ou mentor, pour aider à la progression de l'équipe,
- **Facilitateur** pour favoriser l'autonomie de l'équipe,
- **Coach** pour guider son équipe vers l'excellence.



Les compétences associées à ces facettes sont nombreuses. Mais **un Tech Lead n'est surtout pas un super-héros !** Le plus important n'est pas de maîtriser absolument toutes ces compétences : il faut savoir lesquelles sont nécessaires à l'équipe et identifier celles qui

manquerait. Le Tech Lead doit donc être conscient de ses propres limites et de celles de son équipe.

De même, les activités associées à ces quatre facettes représentent clairement plus qu'une activité à plein temps. Il est donc essentiel que le Tech Lead puisse se fier à son équipe et qu'il n'essaie pas de tout faire seul.

○ De l'importance d'avoir de bons Tech Leads

Est-il important pour une entreprise d'avoir de bons Tech Leads ? Les enjeux que nous venons de présenter nous le garantissent. La plupart des équipes ont besoin d'un Tech Lead qui les emmène vers une forte culture de la qualité. Mais encore faut-il que ce dernier le fasse correctement. Que se passe-t-il si un Tech Lead n'assume pas ses responsabilités, ou pire, va à contre-courant de son rôle ?

Il existe des *Net Negative Producing Programmers*²³, des développeurs qui contribuent négativement à la productivité d'une équipe, notamment en dégradant la qualité du code par leur manque de pratique. Dans sa conférence donnée à l'USI en 2010²⁴, Patrick Kua allait plus loin en étendant le concept à des *Net Negative Producing Technical Leaders*. Si le Tech Lead ne joue pas son rôle de guide, **il risque d'avoir une mauvaise influence sur son équipe** : ne pas donner l'exemple, ne pas soutenir la motivation.

23. Net Negative Producing Programmers, G. Gordon Schulmeyer, http://www.pyxisinc.com/NNPP_Article.pdf

24. Building the Next Generation of Technical Leaders - Patrick Kua at USI, 2010 <https://www.youtube.com/watch?v=WEn5pwM7EjY>

S'il est rare qu'une personne choisisse délibérément d'être contre-productive, certains Tech Leads pêchent par **manque de préparation à leur rôle**. De nombreuses entreprises promeuvent un développeur à ce rôle en se basant uniquement sur son expertise technique ou son ancienneté. Cependant, **rien ne garantit qu'un bon expert technique fasse un bon Tech Lead**, ou même qu'il en ait simplement envie : considérer que c'est automatique reviendrait à occulter la composante *Lead* de ce rôle.

Les entreprises ont donc une réelle responsabilité dans le choix et l'accompagnement de leurs Tech Leads.

Pour les aider, les actions suivantes peuvent être menées :

- **Sensibiliser les personnes à la réalité du rôle et à l'importance de l'humain.**
- Mettre en place des relations privilégiées par du **mentoring** et un suivi précis sur l'évolution dans le rôle.
- **Apprendre à demander et recevoir du feedback** sur son rôle, aussi bien à l'intérieur qu'à l'extérieur de leur équipe.
- Former les personnes au management.
- **Monter une communauté interne de pratiques** et d'échanges entre Tech Leads, qui permette de se rencontrer et de créer des points de contact.
- Encourager à participer à des communautés et événements externes (*meetups, conférences*).

● Porteur de la vision technique

Le Tech Lead a la responsabilité de porter la vision technique du produit, et c'est en cela qu'il adopte une posture d'expert dans l'équipe.

Il s'assure que chacun au sein de l'équipe met en œuvre les moyens d'atteindre et de vérifier la qualité attendue sur les aspects suivants :

- Veiller à la qualité du code et de l'architecture ainsi qu'au respect des standards.
- Être vigilant sur la conception.
- Suivre avec attention les défauts de qualité et leur résolution. On parle souvent de gestion de la dette technique, bien que ce terme soit aujourd'hui assez galvaudé (voir le chapitre 4 - Dette Technique et non-qualité).
- Avoir une bonne maîtrise des technologies utilisées, une vision d'ensemble du produit, de son architecture et de son infrastructure. Ceci implique également une certaine connaissance du contexte métier, afin de pouvoir évaluer la cohérence des décisions à prendre et leur impact technique.

Le Tech Lead est lui-même particulièrement attendu sur ces aspects. **Il se doit d'être exemplaire** pour l'équipe dans sa participation active aux développements et dans la prise de décisions techniques.

> Le Tech Lead code avec l'équipe

Le Tech Lead fait partie intégrante de l'équipe de développement : il code, comme tout développeur.

Son temps passé à coder est variable d'un projet à un autre (de 30 à 80 % selon les projets que nous menons chez OCTO Technology) et dépend fortement de la taille de l'équipe. Pour nous, **un Tech Lead doit passer au moins 30 % de son temps à coder**. C'est important pour deux raisons en particulier :

- Pour assumer ses responsabilités vis-à-vis des aspects techniques et prendre certaines décisions, il doit **être impliqué dans les développements, plutôt qu'être un « inspecteur des travaux finis »**.
- Être leader n'est pas acquis dès le début. C'est en étant sur le terrain au quotidien qu'il obtient **une légitimité et une confiance** de la part du reste de l'équipe et des acteurs extérieurs.

Pour coder avec l'équipe, un certain niveau de savoir-faire est également utile afin d'acquérir cette légitimité :

- Sur les pratiques de développement.
- Sur les paradigmes de programmation et d'architecture, les *design patterns*, les principes *Clean Code*, etc.
- Grâce à un certain niveau de maîtrise des technologies utilisées.

Pour pouvoir guider le reste de l'équipe, dont des experts, il est aussi important de rester à jour et faire de la veille technique, dans une démarche d'apprentissage continu.

Cette participation aux développements se fait de manière à **favoriser la propriété collective du code** et la progression de toute l'équipe. Pour cela, il est utile de :

- Binômer avec les autres développeurs, par exemple pour aider à débloquer des points difficiles.

*Un Tech Lead
doit passer
au moins 30 %
de son temps
à coder.*

- Faciliter des séances de conception et de résolution de problèmes, afin de faire émerger les solutions collectivement.

- Éviter de traiter seul systématiquement les sujets les plus complexes, de ne s'occuper que de sujets purement techniques et de POC (*Proof Of Concept*).

Ces activités ne sont pas intrinsèquement différentes de celles d'un développeur. **C'est l'exemplarité qui est primordiale pour assumer le rôle.**

> Participer aux choix techniques

Le Tech Lead ne prend pas seul les décisions associées aux aspects techniques du produit, mais il est responsable du fait qu'elles soient prises. Il est indispensable **d'arbitrer, de faciliter la prise de décisions**, mais aussi de savoir dire non et trancher.

Propriété collective du code

On parle de propriété collective du code (en anglais *Collective Code Ownership*) pour évoquer une dynamique où les développeurs d'un projet communiquent régulièrement sur la base de code de leur application et les modifications apportées. L'idée derrière cela est d'avoir une équipe sans personne-clé qui soit la seule à bien connaître une section particulière du code.

Il peut proposer des solutions et faire certains choix, par exemple :

- Aider l'équipe à converger vers un choix technologique ou d'architecture – et ce tout au long du projet.
- Aider à définir de nouveaux standards et à faire évoluer les standards existants.
- Arbitrer des compromis entre qualité et délais de livraison (voir la notion de dette technique tactique dans le chapitre 4).

Porter l'excellence technique du produit, c'est également une responsabilité vis-à-vis des acteurs non techniques qui interagissent avec l'équipe. Dans les modèles d'équipes agiles que nous utilisons, **le Tech Lead travaille généralement main dans la main avec le Product Owner**, qui porte la vision produit. Ce duo est chargé de trouver un équilibre entre un

produit qui répond à un besoin, et un produit bien construit techniquement.

Pour cela, le Tech Lead **se doit d'être transparent** et d'expliquer les difficultés qui se présentent, **en levant les alertes au plus tôt, en sachant dire non** et en proposant des scénarios alternatifs. Par exemple, **trouver des compromis sur des User Stories techniquement complexes** à mettre en œuvre.

○ Favoriser la progression de l'équipe

Le Tech Lead est moteur dans la progression de son équipe. Il peut transmettre du savoir, mais il doit aussi **alimenter une dynamique d'apprentissage permanent**. Il s'agit donc de **créer un cadre qui permet à chacun de progresser tout en s'assurant que l'on respecte les enjeux de réalisation du produit**.

Cette progression se fait tout d'abord sur le terrain avec le projet lui-même, **en facilitant l'apprentissage des pratiques de développement** via le *pair programming*, la revue de code ou encore lors de réflexions sur le design autour d'un tableau blanc.

Il est utile aussi de **favoriser des moments d'apprentissage dédiés** :

- Animer des *coding dojos* et proposer des *katas de code*.
- Proposer des formations à suivre lorsque c'est pertinent.

- Faire de la veille technique partagée avec l'équipe.
- Partager des ressources et des lectures.

Faire progresser les membres de son équipe implique encore une fois de **montrer l'exemple**. Mettre en œuvre une nouvelle pratique, comme *Test Driven Development*, est loin d'être automatique. Il est rare que toute une équipe l'adopte spontanément, même après avoir suivi une formation. Si le Tech Lead ne montre pas l'exemple en essayant lui-même de poser les premiers tests, l'équipe risque très fortement de ne pas suivre.

C'est la même chose pour le respect des standards de qualité et la correction des défauts. Dans son livre *Clean Code*, Robert C. Martin fait référence avec raison à la théorie de la vitre brisée²⁵ : si personne ne corrige les défauts dans une partie du code et ne montre l'exemple, les défauts vont se multiplier à cet endroit.

○ Favoriser l'autonomie de l'équipe

Pour favoriser l'autonomie de l'équipe, le Tech Lead doit pouvoir compter sur son *leadership*, telle que l'entend l'approche de *Servant Leadership*²⁶. Le but est de favoriser un **fonctionnement collectif**, dans lequel l'équipe peut prendre ses propres décisions, au lieu d'attendre simplement les ordres.

Néanmoins, cette posture n'est pas adaptée à toutes les situations. Certaines requièrent plus de dirigisme : lorsque l'équipe peine à trouver un consensus, par exemple. C'est important également d'être efficace dans l'organisation et la prise de décisions en situation d'urgence, pour résoudre un incident grave en production, etc.

Il s'agit donc de trouver un équilibre entre ces deux postures : se poser en arbitre plutôt qu'en dirigeant, tout en sachant aussi trancher si nécessaire.

Ce rôle requiert donc des compétences de savoir-être, les *soft skills* suivantes :

- Être humble et à l'écoute, ouvert au débat, assertif²⁷.
- Être capable de transmettre une connaissance, un savoir ou un savoir-faire.
- Être disponible.

Les facettes de coach et de facilitateur peuvent sembler plus apparentées aux rôles d'un coach agile ou d'un Scrum Master. Mais le Tech Lead n'assume pas tout à fait les mêmes responsabilités. Ce dernier se focalise davantage sur les aspects techniques et les pratiques de développement, plutôt que sur les aspects purement méthodologiques de déroulement du projet.

25. Hypothèse de la vitre brisée (souvent référencée en anglais : *Broken Windows*) : https://fr.wikipedia.org/wiki/Hypoth%C3%A8se_de_la_vitre_bris%C3%A9e

26. *Servant Leadership*, conception du leadership suivant laquelle un leader ou un manager est au service de son équipe pour atteindre les objectifs.

27. Assertivité : capacité à s'affirmer tout en respectant et écoutant l'autre.

Timeboxer la résolution de problèmes

La résolution de problèmes dans la réalisation de produit, qu'il s'agisse de problèmes de conception, de correction de bugs ou d'optimisation de performances, est une activité qui peut être très consommatrice en énergie et en temps.

Afin de mieux maîtriser cette problématique, nous utilisons activement le timeboxing :



- Pour avancer sur la résolution, on se fixe des timeboxes courtes de quelques heures ou d'une demi-journée maximum.
- À la fin de chaque timebox, on demande du feedback à l'équipe pour décider si on poursuit.
- Si le problème est gros, on s'interdit de commencer à tenter sa résolution sans avoir identifié les premières étapes.
- On identifie ensemble dès le départ les solutions alternatives ou dégradées pour pouvoir abandonner sans risque une solution trop complexe.

> Faciliter la résolution de problèmes

Lorsque l'on est plus expérimenté – ou depuis plus longtemps que d'autres sur le projet – on peut facilement être tenté de donner rapidement la solution, voire de résoudre les problèmes soi-même.

Présenter la solution toute prête n'offre aucune plus-value. Pour permettre aux autres de progresser, il faut **laisser la place à l'échec, tout en créant un cadre propice à l'apprentissage.**

Il s'agit donc de guider, en aidant à ne pas « réinventer la roue », en orientant les recherches, ou encore en utilisant le timeboxing²⁸ pour résoudre les problèmes.

> Soutenir la motivation

Avoir une certaine approche du développement, adopter et utiliser des pratiques, est une question de motivation, souvent même de passion. Le Tech Lead peut la soutenir en essayant de connaître les envies, les points forts et points faibles de chacun pour cibler des axes de progression. Il peut aussi chercher à transmettre sa propre motivation.

Souvent, il s'agit aussi d'aider et d'encourager les gens à oser. Par exemple, si le refactoring d'un code apparaît trop complexe pour un membre de l'équipe et que l'on a une solution simple à proposer, on l'implémente avec lui. Il se rendra compte que c'était réalisable et essayera plus volontiers par lui-même la prochaine fois.

28. Timeboxing : méthode de gestion par fenêtres de temps limité.

Que faut-il éviter ? Quelques anti-patterns



Super-héros :

Est-il indispensable d'être un expert technique pour être Tech Lead ? Nous l'avons vu, le Tech Lead a un rôle important sur les aspects techniques de la construction du produit. Pour autant, il n'est pas indispensable qu'il soit le super-héros de l'équipe ni qu'il maîtrise à 100 % tous les aspects techniques du produit : cela ferait de lui un expert et non plus un leader.

À vouloir se positionner toujours en expert, donner trop rapidement la solution à un problème voire tout résoudre seul, le super-héros risque de pénaliser l'équipe en l'empêchant de progresser, et en se rendant indispensable.

Être très bon techniquement n'implique absolument pas qu'on sera un bon leader. Il est souvent plus important de savoir déléguer et lâcher prise sur certains détails, en faisant confiance à l'expertise de l'équipe.



Moine codeur :

Le moine codeur fait tout par lui-même, surtout pour les sujets les plus compliqués. Il binôme et échange relativement peu.

C'est une posture assez proche du super-héros, à la différence qu'elle peut ici dénoter plutôt un problème de confiance

dans la capacité de l'équipe à résoudre les problèmes assez vite. Nous avons également observé le cas d'anciens architectes techniques, plus intéressés à l'architecture et aux frameworks, qui codent, mais pas avec l'équipe.

Dictateur :

À l'inverse d'un facilitateur, le Tech Lead dictateur reste en permanence dans une posture dirigiste. Il affecte les tâches, définit probablement lui-même les standards et est seul habilité à revoir le code.



Prof :

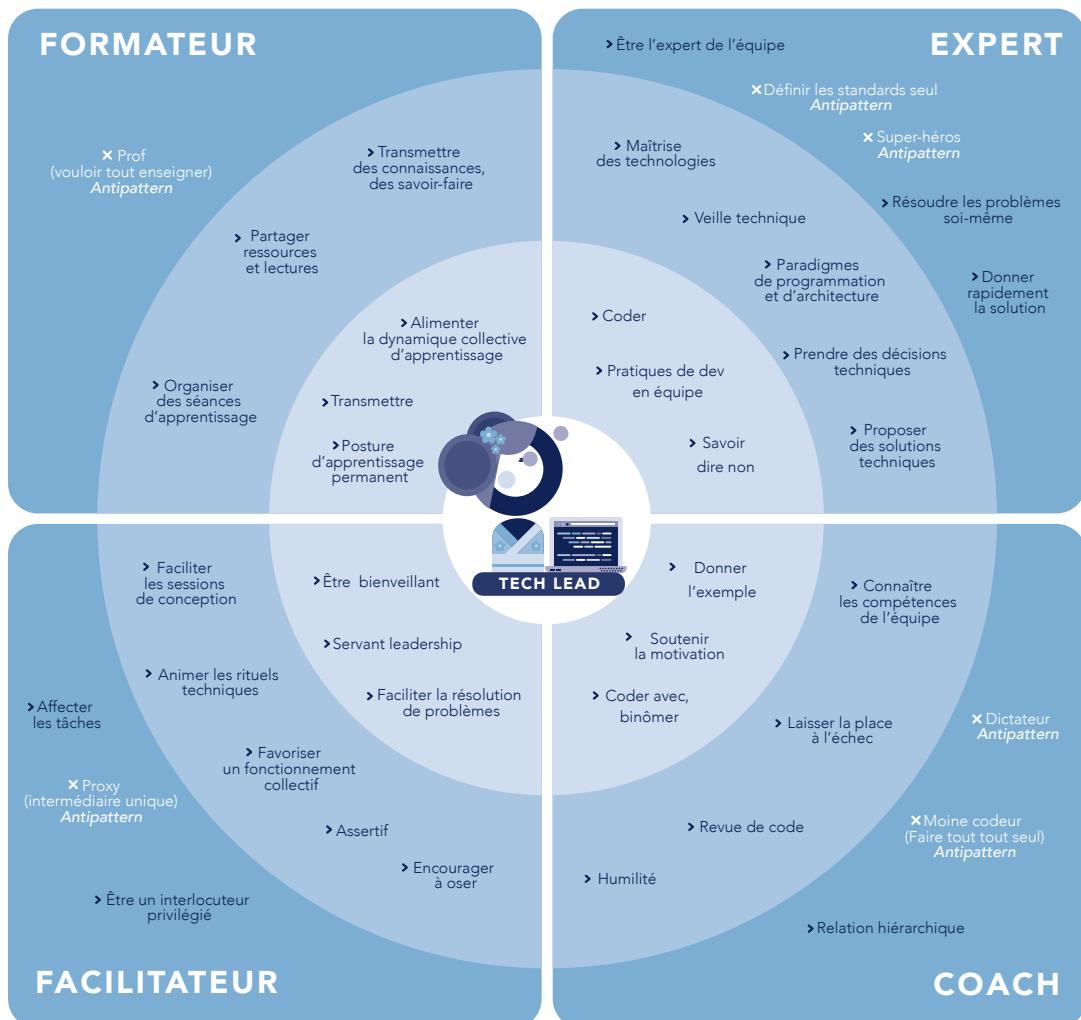
Le Tech Lead Prof conserve en permanence sa casquette de formateur. Il essaye de tout enseigner, au risque que ce soit au détriment des contraintes de réalisation du produit.

Proxy :

Il est parfois utile d'être un interlocuteur privilégié lorsqu'il s'agit de convaincre, pour choisir une solution technique et investir le temps nécessaire à une production de qualité. Attention néanmoins à ne pas tomber dans le travers du proxy, et devenir l'intermédiaire unique entre l'équipe de développement et le reste du monde.

Ces descriptions peuvent sembler caricaturales, mais nombreux sont les Tech Leads qui tombent plus ou moins dans l'un de ces travers. D'une manière ou d'une autre, ces approches pénalisent l'équipe, en limitant son fonctionnement collectif et sa progression vers l'excellence technique.

Rôles et compétences du Tech Lead



Superflu



Utile



Indispensable

○ Points à retenir

Plusieurs éléments sont indispensables au rôle de Tech Lead :

- Il porte la vision technique du produit, et fait partie intégrante de l'équipe en participant activement aux développements.
- Il favorise l'apprentissage et la progression des développeurs de son équipe.
- Il favorise l'autonomie de l'équipe.
- Il cherche à guider son équipe vers l'excellence.

Finalement, ce rôle de Tech Lead nécessite des pratiques et du savoir-faire, de l'apprentissage, et surtout des compétences de leadership pour pouvoir guider et influencer son équipe.

Être un bon Tech Lead, c'est être un bon artisan capable d'aider à éléver le niveau de toute son équipe en donnant l'exemple et en cherchant à la rendre autonome.

○ Par où commencer ?

Si vous avez le rôle de Tech Lead ou allez dans cette direction :

- Déterminez parmi les facettes et les compétences que nous avons évoquées un premier axe sur lequel vous souhaitez travailler et fixez-vous des objectifs de progression.
- Demandez du feedback à vos coéquipiers : comment vous perçoivent-ils dans votre rôle ? Y a-t-il des points particuliers sur lesquels l'équipe peut progresser ?
- Faites une estimation de votre temps passé à développer. Si vous passez trop peu de temps à développer, trouvez des points à déléguer à votre équipe.
- Trouvez des occasions de donner l'exemple. Vos actions et votre code doivent refléter la qualité attendue.

DETTE TECHNIQUE ET NON-QUALITÉ



Dette technique et non-qualité

Par Christophe Thibaut

Voici un extrait d'une conversation entendue pendant la rétrospective de fin d'itération d'un projet Agile A, entre les développeurs et le Product Owner :

P.O : Avec seulement deux stories sur les sept planifiées en début de sprint, je peux vous dire qu'on n'avance pas à un bon rythme.

Dév. 1 : C'est clair.

P.O : Ce qui pose le problème de ce que l'on va être capable de montrer à la démo du COMEX. Ça a une certaine importance, parce que je comptais sur cette démo pour faire avancer les choses du côté du service utilisateur.

Dév. 1 : Clairement durant ce sprint on a été énormément ralenti...

P.O : Par quoi ? Les stories que je produis ne sont a priori pas plus compliquées que d'habitude.

Dév. 2 : OK, moi je pense savoir ce qui nous ralentit.

P.O : On t'écoute.

Dév. 2 : Ce qui nous ralentit, c'est qu'on a de la dette technique à ne plus savoir qu'en faire : il y

a des modules entiers sans tests unitaires, d'où tous ces bugs qu'on a dû corriger pendant le sprint ; il y a beaucoup de code redondant, et du code mort aussi, certaines parties du code sont incompréhensibles pour moi, et elles ne respectent pas les conventions de noms habituels.

Dév. 3 : Tu veux parler du code de Jean-Michel ?

Dév. 2 : Pas seulement, en fait.

Dév. 1 : Bref, en ce moment, c'est un peu compliqué de faire plus vite.

Voici un extrait d'une autre conversation entendue pendant la rétrospective de fin d'itération d'un projet Agile B, entre les développeurs et le Product Owner :

P.O : J'ai drôlement apprécié que vous ayez réussi à mettre en place le système de localisation pendant ce sprint ! Ça va en jeter pour la démo du côté du service utilisateur !

Dév. 1 : Ah oui ! Tant mieux.

P.O : Alors merci pour ça !

Dév. 1 : Bah de rien.

Dév. 2 : Oui mais : ce système de localisation, il faudra le revoir...

P.O : Ah bon, pourquoi ?

Dév. 2 : Parce qu'on a violé certaines de nos règles de design pour l'implémenter. C'est un hack.

P.O : Qu'est-ce que tu entends par « un hack » ?

Dév. 1 : Une solution temporaire en quelque sorte. Bref, c'est de la dette technique.

Dév. 3 : Tiens d'ailleurs, je mets la tâche de refactoring pour le sprint suivant.

Dév. 2 : Post-it violet ! La dette technique, c'est en violet.

Dév. 3 : Ah oui, merci.

Ø Dette technique ?

La différence marquante entre ces deux conversations, c'est l'usage qui est fait de l'expression « dette technique ». Dans le projet A, l'équipe rencontre des problèmes de qualité interne qui l'empêchent de livrer autant de fonctionnalités qu'elle le voudrait, et c'est ce qu'elle désigne par « la dette technique ». Dans le projet B, l'équipe rencontre peut-être des problèmes de qualité, mais en tout cas, ce n'est pas ce qu'elle désigne par « dette technique ». Ce qu'elle désigne, précisément, par ce terme, c'est la mise en place d'une solution temporaire, n'obéissant pas aux standards habituels de l'équipe, mais qui permet de gagner du temps à court terme.

Dans le premier cas, le terme désigne **une manière de qualifier le code**, de parler de sa qualité générale, qui pour le coup paraît relativement préoccupante, bien qu'aucun remède ne semble clairement identifié. Dans le deuxième cas, il désigne **un expédient temporaire**, un écart ponctuel aux standards de l'équipe, pour lequel une solution de recouvrement est non seulement identifiée, mais en plus planifiée dans le temps. Bien sûr, on pourrait dire que l'équipe A a pu se « sur-endetter » à coups d'expédients successifs. D'ailleurs, Ward Cunningham, qui est l'auteur de cette analogie, pointait déjà – il y a plus de 20 ans – le risque de dérive vers un endettement technique généralisé :

« Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise. »

En effet, lorsque l'équipe « emprunte techniquement », **c'est à sa propre vélocité future**, ainsi qu'au client indirectement : sans un prompt remboursement, celui-ci sera à terme gratifié d'une solution temporaire qui a rendu service sur le coup, mais qui a aussi fait long feu. Qui coûte, maintenant, bien cher. Et on sait qu'en logiciel, paradoxalement, le temporaire dure.

○ De la métaphore au modèle

Continuer d'appeler « dette technique » ce qui relève en fait d'un problème général de qualité dans le design ou le code d'une équipe, fait perdre toute son utilité à cette métaphore. Considérez l'endettement financier d'une entreprise : l'entreprise s'endette afin d'investir et elle investit afin de s'enrichir. Aucun chef d'entreprise ne déclarerait : « Nous nous endettons, nous ne contrôlons pas vraiment le phénomène, et n'avons pas vraiment d'idée de ce qu'il faut faire pour y remédier, mais en attendant, c'est ce que nous faisons ».

En revanche, dans le monde de l'IT, tout le monde ou presque utilise aujourd'hui la notion de dette technique précisément dans ce sens, c'est-à-dire, finalement, pour exprimer l'idée d'une **fuite en avant**. Comme il est difficile de constater une fuite – même une fuite en avant – sans tenter de montrer qu'on ne reste pas à rien faire devant le problème, on a également identifié des moyens de **mesurer** la dette technique – un exploit dans l'art de faire dire aux chiffres n'importe quoi – passant ainsi subtilement de la **métaphore**, au **modèle** d'analyse d'impact. C'est dire si le problème du surendettement technique incontrôlé s'est répandu. Or une équipe engluée dans la dette technique au point d'avoir besoin d'outils pour la mesurer, c'est un peu comme une équipe qui déclarerait : « Nous accumulons des problèmes de qualité interne, nous ne contrôlons pas vraiment cette dégradation, et nous n'avons pas vraiment de contre-mesure, mais en attendant, nous pouvons la mesurer, et c'est ce que nous faisons ».

Cette métaphore est-elle utile à une équipe aux prises avec des problèmes de qualité ? C'est douteux. Lorsque le Product Owner – ou le Manager, ou le Tech Lead, ou qui vous voulez – n'est pas en mesure de traiter efficacement l'alerte sur les agios exorbitants que l'équipe est déjà en train de subir, que peut signifier s'endetter techniquement ? Utiliserait-on encore la notion de dette dans un système financier où la banque ne viendrait jamais réclamer le paiement des termes, principal et intérêt ?

Chaque équipe utilise évidemment les métaphores comme elle l'entend pour désigner telle ou telle partie de sa réalité. Mais lorsqu'une équipe se réfère à un *Design Pattern*, comme par exemple *Singleton* ou *Observateur*, c'est en général dans l'idée d'utiliser ce pattern tel qu'il a été documenté, afin de remédier à un problème similaire à ceux que le pattern résout habituellement. Or avec l'équipe A citée en premier exemple, il semble que l'emploi du terme ne sert aucune démarche particulière, si ce n'est jeter un voile pudique sur une réalité dure à admettre (que certains appellent, loin des slides officiels, le « code écrit à l'arrache »). Force est de constater **qu'on est passé du pattern à l'anti-pattern**.

❶ Comment s'y repérer ?

Faut-il pour autant jeter le *pattern* aux orties ? Loin s'en faut. Comme le montre l'exemple de l'équipe B, la notion d'emprunt à court terme est particulièrement utile – parce qu'elle décrit une situation nécessaire, inévitable dans toute équipe Agile, à savoir le conflit entre les besoins à court terme du projet, de l'équipe, du client, et les besoins à moyen ou long terme de ces mêmes parties. **Une équipe excellente, en cherchant à être Agile, contractera tactiquement de la dette technique à un moment ou à un autre.**

Et c'est parce qu'elle recherche l'excellence autant que l'agilité, que l'équipe Agile utilise ce *pattern*. Si on comparait le travail d'une équipe de développement à une partie d'échecs, on pourrait exprimer cette différence comme suit :

- **La dette technique « tactique », c'est comme un sacrifice (un échange temporairement défavorable) à un moment clé de la partie.**
- **La dette technique « endémique », c'est plutôt comme perdre ses pièces les unes après les autres, et courir à la défaite.**



À quoi reconnaît-on qu'une équipe fait des « emprunts techniques » temporaires pour s'assurer un avantage (également temporaire) plutôt que s'endetter jusqu'au cou ?

Il suffit d'observer ses pratiques, et d'écouter ce qu'elle répond à ces questions :

- Avez-vous un standard documenté que vous pouvez afficher, ou expliquer via une session en binôme ?
- Avez-vous des tâches de remboursement prévues pour les « emprunts » techniques en cours ?
- Les tâches de recouvrement s'écoulent-elles au même rythme que les tâches standards de développement ?

À l'inverse, à quoi reconnaît-on une équipe engluée dans la « dette technique » ?

Voici des symptômes qui ne trompent pas :

- Sur le tableau de l'activité de l'équipe, les tâches de recouvrement sont définies en priorité basse, et s'accumulent de semaine en semaine.
- Également sur le tableau, des graphiques détaillés affichent les résultats des outils d'analyse automatique du code, comme la complexité cyclomatique ou le taux de couverture du code par les tests.
- Quand on l'interroge sur ce pattern, l'équipe voit la dette technique comme un problème, et non comme une solution. En réalité, **maîtriser sa dette technique, c'est la voir comme une**

opportunité (de livrer plus vite au prix d'une dégradation **temporaire** du design) et non un problème (de qualité du code).

○ Alors que faire ?

Si votre équipe vous semble aux prises avec un problème général de qualité, vous pouvez vérifier quel usage elle fait généralement du terme « dette technique ». C'est un bon signal d'alarme. Et si l'alarme sonne en effet, que faire ?

• **Provoquez un temps d'arrêt pour établir un dialogue** autour de ce problème. Vous ne pouvez pas mener un développement logiciel à coups de dettes techniques répétées, et encore moins en vous appuyant sur du code de mauvaise qualité.

• **Demandez à votre équipe de décider quels standards elle veut appliquer** dans son travail. Rendez le standard explicite. Pour cela, il ne suffit pas de produire un document mais il faut confronter régulièrement les écarts à ce standard, au moyen de revues de code régulières, facilitées et documentées, ou bien par exemple en adoptant la pratique du *pair programming*.

• **Mesurez**, au cours des revues, ce que l'équipe dénote comme des défauts de qualité. Tenez à jour le nombre de défauts relevés par revue, **afin de vérifier que les actions prises remédient au problème de qualité**.

• **Travaillez sur les facteurs** de « dette



technique », **c'est-à-dire sur les pratiques de l'équipe**, et non sur le code endetté lui-même. La dette technique « endémique » est un dégât. Travailler seulement sur les dégâts sans tenter de changer les facteurs, c'est comme tenter d'écopier sans rechercher les voies d'eau. Aucune « gestion de la dette technique » ne permet de remédier à l'absence des pratiques de base du développement : tests unitaires systématiques et revues de code.

- **Mettez en place des formations, des groupes de lecture, des expérimentations** visant à mieux connaître et mettre en œuvre des pratiques de qualité.

Ø Et que ne faut-il pas faire ?

Ne pointez pas du doigt. Ne blâmez pas vos équipiers pour la façon dont ils traitent le problème ou pour leur usage plus ou moins

approprié du terme « dette technique ». Face à une équipe qui a des résultats médiocres, la première réaction est souvent de mettre la pression sur cette équipe pour qu'elle fasse plus d'heures, comme si elle manquait de courage pour réussir. Il y a en effet deux sens au mot courage : ce peut être le courage de ne pas reculer devant la peur, ou bien celui de ne pas reculer devant l'effort. Si votre projet est « complètement endetté », le courage qui manque probablement, ce n'est pas celui de travailler plus dur, c'est celui de mettre un terme à la fuite en avant et de commencer à investir une partie du temps sur l'amélioration des pratiques. C'est-à-dire **le courage de dire « non »**.

○ Points à retenir

Il existe deux usages bien différents du terme de « dette technique ».

La **dette technique tactique** fait référence à la **mise en place d'une solution temporaire**, n'obéissant pas aux standards habituels de l'équipe, mais qui permet de **gagner du temps à court terme**, et pour laquelle **une solution de recouvrement** est non seulement **identifiée**, mais aussi **planifiée dans le temps**. Une fois l'objectif à court terme atteint, il est vital d'appliquer la solution de recouvrement identifiée, de « rembourser la dette technique ». Si ce n'est pas fait, la solution temporaire va durer et coûter cher à long terme.

La **dette technique endémique** revient à désigner le code de mauvaise qualité, pour lequel aucune solution n'a été **planifiée**. Appeler « dette technique » ce qui relève en fait d'un problème général de qualité dans le design ou le code d'une équipe, fait perdre toute son utilité à cette métaphore, et **revient à masquer une stratégie de fuite en avant**.

○ Par où commencer ?

- Prenez une heure avec l'équipe pour identifier ce que vousappelez aujourd'hui dette technique dans votre produit. Parmi les portions de code identifiées, combien proviennent d'un choix avec un plan d'action en cours pour rembourser ? Combien proviennent d'un problème de qualité non maîtrisée ?

- Élaborez une **définition commune de la gestion des problèmes de qualité, comprise par tous** : aussi bien par les développeurs que les intervenants métier ou le management. Si vous utilisez le terme de dette technique, soyez clairs dans votre utilisation du terme.

- Si les problèmes de qualité sont endémiques, **commencez par identifier les pratiques manquantes** dans l'équipe **avant de vous attaquer à du « désendettement »** des défauts existants.

- Une fois que le flot de nouveaux défauts de qualité est stabilisé, identifiez les problèmes de qualité existants, et élaborez un plan d'action pour le problème le plus prioritaire, avec des objectifs atteignables dans un temps court.

OCTO > CULTURE CODE

INTERLUDE

CRISSEZ PROLIFÉREZ

Par Christophe Thibaut.





Résumé des épisodes précédents : ToF et Misteur Rup, au cours d'une revue de code particulièrement houleuse, ont réussi par hasard à se dématérialiser – rematérialiser dans un listing, où ils ne tardèrent pas à découvrir qu'ils se trouvaient à proximité d'un important repaire de bugs. En effet Misteur Rup a ramassé là, par terre, une note autocollante repositionnable verdâtre qu'ils déchiffrèrent non sans peine et qui semblait indiquer en termes cabalistiques la future venue d'une sommité des bugs à une sorte de « présidium diurétique » au cours duquel il allait être expliqué aux bugs disciples fraîchement insérés, la vision partagée du « Grand Plan ». Rongés par la curiosité, nos deux compères suivirent tant bien que mal le diagramme de dépendances qui menait au présidium diurétique – la réunion devant, probablement selon ToF, se tenir dans un garage.

ToF et Misteur Rup, précautionneusement cachés dans une anfractuosité du listing, observaient la lente procession des bugs qui entraient dans le garage maintenant par dizaines.

— Faut qu'on file d'ici au plus vite, si tu veux mon avis, dit Misteur Rup. C't'un pétrin ce comité.

— Tu nous y a mis, maintenant tu veux partir ?

Mais leur dispute fut interrompue par une annonce solennelle que le président des bugs lui-même allait se tarter son talk.

— Une chance que les bugs parlent le Frangliche et qu'on l'a appris à l'école ! disait Rup.

— Chute donc ! lui fit ToF.



Une espèce de grand prêtre des pous, une créature hideuse, se hissa sur une caisse de cambouis. Sa voix grézillait comme le son d'un scooter de pizzeria, avec un effet flanger à peine décelable.

« Jeunes poux ! Virulents bugs de la deuxième génération, et vous ma vieille garde, anciens vaillants choléopores, mes fidèles !! Vous êtes tous venus vous faire dire encore le Grand Plan, et je vous accueille de toutes mes lamelliformes ! »

Le grand prêtre toussa, attendit une seconde que deux trichoptères retardataires prennent place, puis commença son esssssplication. ToF, qu'est agrégé de Frangliche, a pris les notes.

GRAND PLAN

LE COMMENT - COMMENT

PROLIFERER

- SOYEZ DISCRET -

C'est la règle numéro un de la vie du bug, et la sagesse essentielle. Rappelez-vous que le but, le sens de la vie, la Cause, c'est d'arriver indétecté en production ! Aussi, pour ne pas vous faire avoir, ne vous faites pas voir ! Si deux humains sont à l'écran, et que l'un des deux tape du code dans lequel vous envisagez de vous reproduire, méfiance ! Vous pouvez être assuré que l'autre humain est en train scruter le code pour vous traquer ! Ce n'est pas le moment pour vous de proliférer. Les humains n'écoutent rien, même en binôme, mais ils y voient très bien. Ne sautez pas aux yeux des humains !



Fuyez aux premiers Signes d'Alerte

Dès que vous entendez l'un de ces mots : test, check, assert, verify, spy, mock, revue, pair, fuyez à toutes pattes !

Les humains n'entendent rien, voient bien, mais par-dessus tout ils peuvent être retors et malicieux, avec un goût prononcé pour les farces douteuses et les pièges et les machines infernales, comme la célèbre fraâmeworque jUnit.



Reproduisez-vous à l'Ombre ▾

N'entrez jamais dans un code appelé par un test. Seuls les plus malins d'entre nous survivent aux tests, et dites-vous que vous n'êtes probablement pas le plus malin de la couvée. Si vous devez rester dans les parages parce que votre progéniture est en larve et ne peut pas encore marcher, installez-vous plutôt dans le code de tests. Avec un peu de chance les yeux des humains seront fixés sur l'autre code et laisseront les tests en jachère. Résitez là plutôt.

Privilégiez le code compliqué, le code touffu, le code illisible, verbeux, mal pensé, pas relu, le code bricolé, le code ancien mais qui doit évoluer quand même. Une petite routine, même de seulement 300 lignes, c'est un endroit agréable où s'installer et élever sa famille, ce n'est pas le luxe comme un petit château de 2 000 lignes et sept niveaux d'ifs imbriqués, mais c'est un bon début dans la vie.



Évitez le Code Testé

Nichez-vous dans les Dépendances



Dans les dépendances, l'humain peine à bâtir des tests. Ses machines infernales restent pratiquement sans effet face à ces prodiges de la nature pouilleuse que sont le Singleton, l'Instance Unique, les Variables Globales en tout genre, les Effets de Bords, le Typage en Canard... L'humain est rusé, mais il cherche à tout faire en même temps, ses abstractions fuient, mais il s'entête. Lorsque vous entendez quelque chose comme « ça y est, ça marche » sur un code dont la fabrication a pris plusieurs jours, vérifiez que le code s'exécute au moins une fois, puis sautez dans la première dépendance que vous trouverez.

L'obsolescence, c'est l'opulence ! Les humains changent d'avis comme de carapace. Les projets sont à peine livrés qu'on passe à la benne des librairies entières, que dis-je des langages de programmation ! Les humains sont inconséquents ! Ils recommencent leur projet dans une nouvelle technologie tous les deux ans, mais ne jettent pas les anciens. Certains de mes plus vieux amis vivent dans une Procédure Stockée

écrite en 1992 ! Et la légende dit que notre prophète est encore vivant, tapi dans une Clause Copy en Cobol, depuis 1965 ! Si vous n'avez pas ces technologies dans vos parages, installez-vous dans un fichier JS. Le code JavaScript peut devenir obsolète en quelques semaines ! C'est le nouvel eldorado de la génération de bugs montante !



Nichez-vous dans le code Obsolète



Attendez jusqu'au Moment de la Nuée ▾

Si votre code se développe dans un projet agile, il y a danger. Mais c'est à la fois danger et opportunité ! Vos conditions d'existence en production sont plus favorables avec un code qu'on livre aux usagers toutes les deux semaines plutôt que tous les deux ans ! Mais il faut passer la barrière des tests. Vous n'êtes pas le plus malin de la couvée ! Si vous êtes jeune et sans expérience, installez-vous dans un projet en cascade, où vous pourrez attendre six mois, voire un an, que votre code passe en production. D'ici là, vous vous serez reproduits à merveille, et avec votre progéniture, vous profiterez de la cascade pour faire des nuées !

Profitez des Heures Supplémentaires

Le code qui grandit dans la nuit est un paradis pour nous autres. Les humains sont malins, mais ils sont têtus et n'écoutent rien. Certains humains restent à coder bien après le crépuscule, ils y parviennent en boostant leur système par ingestion de glucides et de caféine. Observez ces humains là, ils sont probablement en train

de bâtir la résidence de vos rêves. Un humain influençable, pris sous la pression, qui n'ose pas communiquer ses obstacles à ses camarades partis depuis l'heure du dîner, ou bien qui se prend pour Superman, peut produire une quantité considérable de code dépendant, mal pensé, non relu, pas testé. Après 20 heures, c'est l'idéal pour s'installer !



La suite de mon discours s'adresse à ceux d'entre vous qui vont se faire prendre. C'est malheureux. Ce n'est pas souhaitable. Mais si cela arrive, vous pouvez encore faire quelque chose pour la Cause !

D'abord, est-ce sûr que vous allez vous faire prendre ? Si vous êtes pris dans une chasse aux bugs, essayez de trouver un camarade plus visible que vous, derrière lequel vous

trouvent quelque fois, mais leur rage est passagère, et surtout, elle les empêche de comprendre. Avec un jeune bug naïf qui se fait prendre, les humains n'iront pas plus

▲ Cachez-vous derrière un Bug plus Visible

▼

cacher. Un jeune bug naïf, c'est l'arbre qui cache la forêt. L'humain présomptueux le choppe, et croit la mise au point faite, et pendant que votre malheureux camarade se fait prendre, et puis pendre, vous pouvez en profiter pour vous installer et rester, peut-être jusqu'à la Nuée ! Les humains nous

loin. Ils taperont sur l'auteur du code, au lieu de mettre à la question le bug qu'ils viennent de prendre ! Si après la pendaison vous n'entendez pas « 5 pourquoi », « rétro analyse », « cause profonde », alors vous pouvez probablement rester dans les parages.



Participez à une Réinsertion

Si vous êtes à découvert au moment d'une chasse, vous pouvez, dans un geste ultime – et fort noble – de survie pouilleuse, tenter de vous reproduire par réinsertion. L'humain qui corrige le code après découverte manque rarement des marges de manœuvre suffisantes pour opérer une tactique d'éradication efficace. Il pare au plus pressé. C'est là une chance pour vous de vous glisser à nouveau dans le code, ou bien d'appeler un camarade moins voyant que vous. L'humain n'aura pas le temps de le trouver. Rappelez-vous le couplet fameux de notre épopée lyrique, il est dans notre hymne et nous le chanterons tout à l'heure :

*De poisse amère je suis oint,
Car avecque toutes ces bogues
À suivre à pied, seul, et sans dogue,
D'ourdir des tests le temps n'aie point.
C'est le couplet de la victoire !*

Que faire lorsque vous êtes pris ? C'est un triste sort, mais vous pouvez encore être utile à la Cause. Certains d'entre vous ne seront pas assez malins pour survivre en production, mais ils pourront se faire prendre pour ce qu'ils ne sont pas, en faisant croire à l'humain des choses qu'il ne peut pas vérifier, parce que ses collègues et son client ne sont pas là, ou que ses livres restent fermés. Si sur votre lieu de capture, vous entendez des phrases comme :

Faites des Écrans de Fumée



- « Si ça se trouve, c'est le disque qui a un problème... »
- « C'est peut-être le comportement normal... »
- « En tout cas chez moi ça marche... »
- « La spéci ne dit rien là-dessus... »
- « C'est juste un paramètre mal écrit... »
- « En PHP on a le droit d'écrire ça... »

Alors c'est le moment de tenter de plaider l'innocuité. Prenez votre air le plus innocent et frais, né de la dernière couvée. Vous allez peut-être survivre à votre prise !

Accaparez l'Attention ▼

Si toutefois vous ne survivez pas, vous pouvez toujours vous rendre utile à la Cause, en faisant un effet spectaculaire. L'effet que vous ferez peut empêcher les humains de réfléchir. Pour peu que le projet soit en retard – ils le sont tous – et que votre effet soit un peu visible, paradoxal, insolite, amusant, perturbant, les humains vont s'emparer de vous pour alimenter leurs discussions. Les humains n'écoutent pas, aussi leurs discussions prennent beaucoup de temps. Tout le temps passé par les humains à discuter, c'est du temps qui leur manquera pour se remettre en chasse, et tendre des tests ou faire des enquêtes sur votre petite famille et vos résidences secondaires. Attirez l'attention. On m'a dit que certains camarades qui se sont faits prendre et sont maintenant prisonniers du bug traqueur, arrivent à revenir en réunion d'humains une fois par semaine depuis plusieurs mois ! C'est un bel exemple de fidélité à la Cause.



Provoquez des Débats sur Qui est Responsable ▼

Si vous êtes prisonnier du bug traqueur, et que vous sentez qu'une réunion va se mettre en place et porter sur vous, essayez de faire jouer les apparences en votre faveur. Les humains sont prompts à s'accuser entre eux des maux contre lesquels ils luttent. L'auteur du code dira peut-être à votre propos, « ce n'est pas un bug, c'est ce que disait la spéc ! » à quoi son client répondra : « certainement pas, la spéc ne disait pas

cela », à quoi l'auteur répondra « c'est le problème, la spéc était pas claire », et son client : « si elle n'était pas claire, il fallait le dire », et l'auteur : « tu nous a dit qu'on manquait de temps », et le client :

« je n'ai jamais dit : faites de la marne » et la discussion s'ensuivra continûment, et avec un peu de chance escaladera les échelons hiérarchiques humains. Comme c'est beau !

Rappelez-vous, camarades pour qui allez vous faire prendre, que vous êtes l'instrument contondant de notre guerre psychologique ! C'est notre masse, qui fait notre force, elle étouffe patiemment l'humain sous une charge de travail à

Déouragez les Initiatives

la croissance exponentielle, sans pour autant blesser son amour propre au point qu'il renonce à produire et reprenne un peu ses livres. Les livres, les exemples, les formations, les articles de blogue, sont les ennemis des bogues ! Lorsque vous voyez un humain ouvrir un livre, prévenez quelques

camarades, invitez-les à se manifester. C'est pour la bonne cause. Avec une petite faction de bugs par semaine, l'humain aura chaque soir trop de travail à finir pour se remettre à apprendre. Et d'ailleurs à l'université, dans les écoles, la traque aux bugs est pour ainsi dire inexisteante ! C'est le pays des TPs bâclés à la hâte en fin de trimestre, et les locutions telles que tests unitaires, revues de code, y sont pratiquement inconnues, ou restent lettre morte. Malheureusement, ces environnements de code ne partent jamais en production, ils sont donc sans intérêt pour nous.

Devenez Intermittent

Vous avez tous entendu les histoires de ces héros passés, sublimes bugs devenus des légendes, qui ont survécu à la trappe, et bien que s'étant fait prendre, sont quand même en production ! C'est œuvre magnifique, et d'une grande noblesse pouilleuse. L'art en est difficile. Il s'agit pour vous d'apparaître, puis de disparaître. L'humain qui vous a traqué va lancer son code plusieurs fois afin de voir si vous vous manifestez encore. C'est le moment que vous choisissez pour disparaître. Vous êtes repassé en invisibilité. Vous êtes double, à la fois dans le code en production, intermittent, à la fois dans le bug traqueur, marqué résolu « non reproduit ». C'est le sort de peu d'entre nous, sauf lorsque les humains dans leur folie décident de faire un projet compliqué où plusieurs morceaux de code s'exécutent en faisceaux parallèles.

Transformez-Vous en Fonctionnalité



Enfin le but ultime, le dessein suprême, la consécration pour tout bug qui se respecte, c'est d'être anobli, adoubé par les humains et transformé en feature ! Peu d'entre vous obtiendront ce saint statut, mais croyez-moi, le jeu vaut la peine d'essayer, et d'essayer encore ! Certains groupes d'humains vérifient à chaque livraison si nos camarades pous qu'ils ont détectés mais décidé de ne pas corriger faute de temps, sont bien toujours présents dans la nouvelle version du code en production ! Ils appellent cela « l'iso-fonctionnel ». Certains groupes d'humains ont réussi à imposer à des millions d'autres humains ce qui n'était rien d'autre au départ qu'une minuscule erreur de logique, cachée derrière une dépendance, codée par un débutant. Car pour les humains, ce n'est pas la qualité qui prévaut, mais la vitesse à laquelle

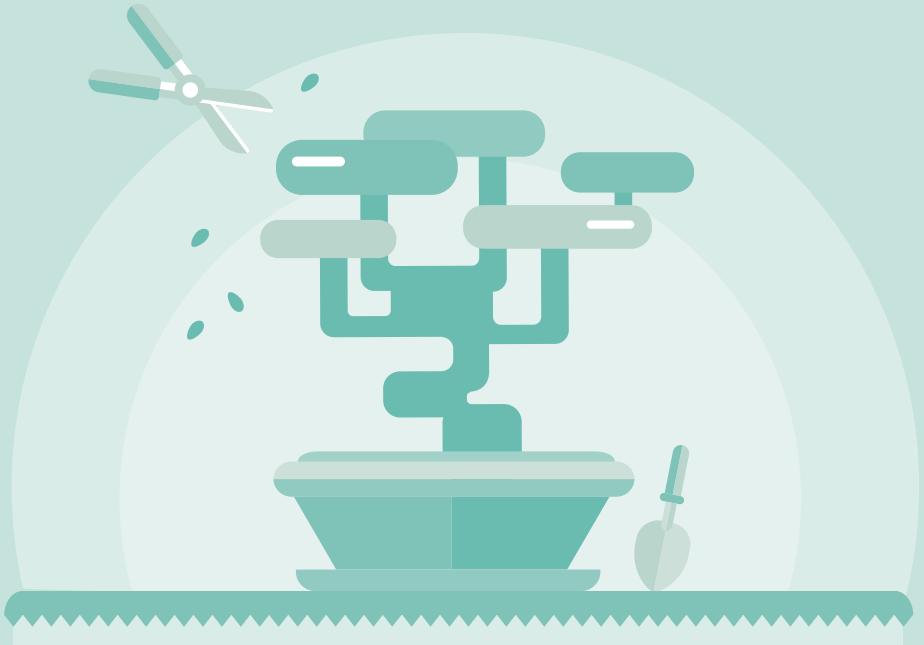
ils peuvent vendre ce qu'ils fabriquent. Aussi leur est-il venu l'idée de vendre les bugs. De grandes maisons humaines prestigieuses vivent de cette idée, qui prouvent que par-delà nos différences avec les humains, une collaboration gagnant-gagnant dans un monde en révolution digitale permanente est possible, et je crois savoir que des premiers accords de paix avec certains grands groupes humains sont en cours. C'est un espoir magnifique pour vous jeunes bugs, pous de la deuxième génération, et vous ma cohorte de fidèles !

Longue vie aux Bugs de toutes Espèces !!

CRISSEZ ! PROLIFÉREZ !

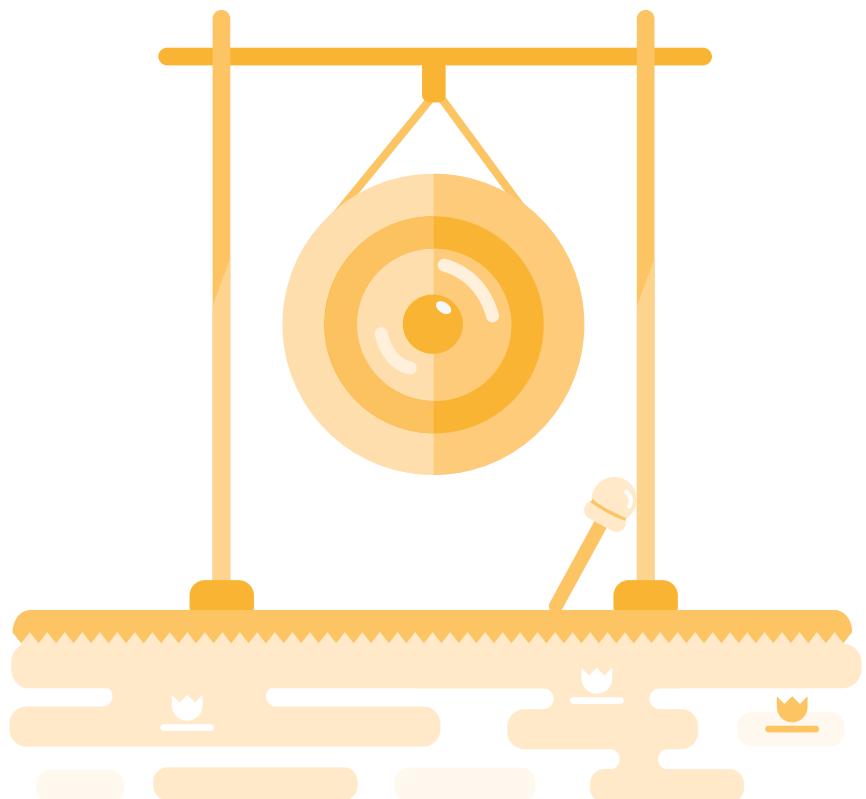
Le présidium allait bientôt prendre fin, lorsqu'un glapissement se fit entendre, qui mit l'alarme à l'assemblée des bugs. ToF dans sa maladresse avait écrasé de sa chaussure droite un jeune thysanoptère qui sortait à peine de l'état larvaire et son couinement avait résonné dans le silence qui suivait l'allocution du roi des pous. Dans une clameur de grésil, toutes les cohortes de choléoptères se dressèrent et foncèrent vers nos deux compères.

Faut qu'on file au plus vite ! hurla Misteur Rup.



BETTER
CODE

PRÉAMBULE



Préambule

Par Arnaud Huon & Michel Domenjoud

Jusqu'ici, nous avons principalement évoqué les conditions permettant d'avoir un environnement de travail propice à des réalisations de qualité. Regardons maintenant de plus près les pratiques qui contribuent effectivement à la qualité de ces réalisations.

Notre objectif n'est pas de couvrir en détail dans ce livre l'intégralité des pratiques : de nombreux ouvrages traitent déjà ces sujets et nous vous invitons à vous y référer²⁹. Nous vous proposons plutôt de nous concentrer sur quatre piliers indispensables : les techniques d'écriture d'un code compréhensible, la revue de code, les tests automatisés et plus particulièrement Test Driven Development.

Si nous avons choisi **ces quatre pratiques** en priorité, c'est parce qu'elles **confèrent un atout indéniable à une équipe de développement : des feedbacks rapides et le support à une communication de qualité autour du code** :

- **Écrire du code propre**, remanié en permanence (*refactoring*), permet d'échanger facilement entre développeurs, sans difficultés inutiles induites par un effort de compréhension.
- Effectuer des **revues de code** et utiliser

le **pair programming** permet d'obtenir un feedback rapide et efficace sur la qualité du code.

- Utiliser les **tests automatisés** offre un feedback rapide sur la qualité fonctionnelle du produit.
- Plus particulièrement, la méthode **Test Driven Development** permet d'obtenir individuellement un feedback en continu sur sa propre progression dans la construction du logiciel.

Avant de nous intéresser en détail à ces quatre pratiques, nous vous proposons néanmoins un bref détour par quelques autres éléments qui nous semblent importants.

○ S'appuyer sur des standards partagés

Vous l'aurez compris, nous considérons qu'écrire du code est un travail d'équipe. Pour améliorer collectivement la qualité du code, il faut que toute l'équipe partage une même vision sur le sujet. C'est ce qui va former les standards d'équipe, initiés au démarrage du projet, et qui seront enrichis au fur et à mesure

29. Ouvrages que vous pouvez retrouver dans la bibliographie à la fin de ce livre.

de son avancement. La revue de code est le bon moment pour enrichir les standards. C'est à la suite de celle-ci qu'ont lieu des discussions, par exemple sur la meilleure façon de nommer les entités logicielles.

Le meilleur moyen de pérenniser ces standards, c'est à minima de les écrire. Que ce soit sur un mur, sur un wiki ou un document en ligne, l'important est que chacun puisse les lire et les modifier facilement.

> Analyse de code automatisée

Pour les plus outillés, on peut automatiser le contrôle du respect d'une partie des standards via les outils d'analyse de code. Ces outils fournissent en général un premier indicateur fiable sur la maintenabilité du code : un standard non respecté, c'est déjà une difficulté pour y rentrer et pour le modifier.

S'appuyer sur les standards proposés par défaut par l'outil est d'ailleurs un bon point de départ. En effet, ceux-ci correspondent généralement aux standards attendus par la plupart des développeurs du langage.

Nous reviendrons sur ces outils dans le chapitre sur les revues de code.



Ø Boy Scout Rule

Dans son livre *Clean Code*, Robert C. Martin nous présente un principe essentiel pour écrire du beau code, reprenant la fameuse règle des boy-scouts qui se résume en une phrase : « **Toujours laisser un endroit dans un état meilleur que celui où vous l'avez trouvé** ».

Si nous livrons tous du code dans un état plus propre que celui où nous l'avons trouvé, alors le code risque moins de se détériorer. Le **nettoyage³⁰ de code** n'a pas besoin d'être immense, on prend seulement quelques minutes : il suffit souvent de changer un nom de variable, de découper une méthode trop longue, d'éliminer de la duplication de code, de nettoyer une instruction qui comprend trop de *if*. L'essentiel est de rendre le code plus lisible, plus compréhensible.

Si l'amélioration continue fait partie intégrante du quotidien de l'informatique, alors le **refactoring** et cette *Boy Scout Rule³¹* en sont l'application la plus importante sur l'écriture de code. Vous pourrez trouver une illustration de sa mise en pratique sur notre blog³².

30. Notez ici que l'on emploie le terme nettoyage et non refactoring : s'il s'agit bien en réalité de refactoring au sens où l'on modifie le code sans toucher au comportement, il est important de se restreindre à de petites améliorations et d'éviter un refactoring important sans besoin d'évolution associé. Voir aussi <http://ronjeffries.com/xprog/articles/refactoring-not-on-the-backlog/>

31. http://programmer.97things.oreilly.com/wiki/index.php/The_Boy_Scout_Rule

32. <http://blog.octo.com/scout-toujours/>

○ Construire et livrer en continu

Afin d'obtenir un *feedback* au plus tôt sur le produit, la pratique de **l'intégration continue** est déjà largement utilisée depuis plusieurs années. Grâce à des outils de *build* automatisé, par exemple Maven dans la sphère Java, et d'intégration continue comme Jenkins, nous sommes en mesure de **construire, valider et déployer** automatiquement le logiciel après toute modification du code source.

De nombreuses équipes vont encore plus loin et déploient leurs produits en continu en production. On emploie le terme de *Continuous Deployment*. Nous avions déjà détaillé cette approche dans le livre *les Géants du Web*³³.

○ Des outils adaptés et maîtrisés

Écrire du code implique d'utiliser de nombreux outils différents. De la console aux outils d'intégration continue, chacun d'entre eux a un rôle précis. Bien utilisés, ils peuvent grandement **améliorer la productivité au quotidien**³⁴.

À la manière des artisans, les développeurs chevronnés ont dans leur besace des logiciels qu'ils ont appris à maîtriser, à personnaliser selon leurs besoins, dont ils connaissent l'essentiel des raccourcis clavier, avec lesquels

ils sont productifs. Ces développeurs sont précieux, car ils vont guider les plus novices dans le choix de leurs outils et de leur utilisation.

Il n'est pas indispensable que tous utilisent exactement les mêmes outils, même si cela est une pratique courante. Si un développeur est plus à l'aise avec Vim pour éditer un script Ruby et que son voisin préfère un IDE³⁵, c'est son affaire, du moment qu'ils sont tous deux efficaces lorsqu'ils éditent ce même script. Il est important néanmoins que cela ne constitue pas un blocage à la pratique du binôme.

Choisir son IDE

L'**IDE** est souvent *l'outil principal* du développeur, et il a une importance non négligeable dans l'écriture de code propre. En effet, un bon IDE doit permettre de faire sans douleur des petits *refactorings* (par exemple, renommage et extraction de variables, de méthodes), et même les faciliter fortement (répercussion automatique, aide à la décision, etc.).

À l'opposé, un IDE trop basique ou non maîtrisé peut avoir un effet néfaste : si le refactoring est douloureux ou laborieux, le développeur aura souvent tendance à abandonner la pratique.

Une utilisation intensive de cet outil peut permettre une vitesse d'écriture de code ou de navigation dans les sources beaucoup plus fluide et efficace. Parmi les fonctionnalités peu utilisées par les novices, on peut citer de nombreux *refactorings assistés*, la génération automatique de code ou de classe, l'édition en mode colonne ou en sélection multiple ou encore l'historique de copier/coller.

33.. Les Géants du Web, 2012, publication OCTO Technology, <http://www.geantsduweb.com>

34. Nous l'évoquons chez OCTO il y a déjà plusieurs années dans notre livre blanc sur la productivité des développements dans l'écosystème Java (même si la plupart des outils évoqués dans cet ouvrage ont beaucoup évolué depuis). Java Productivity Primer, Douze recommandations pour augmenter votre productivité avec une usine logicielle, OCTO Technology, 2009.

35. IDE : *Integrated Development Environment*, en français Environnement de Développement Intégré



Mais s'il n'y avait qu'un seul conseil à donner pour améliorer sa productivité : **maîtrisez les raccourcis clavier de vos outils.** Cet article³⁶ présente un gain de 15 % de temps sur une expérience en utilisant uniquement le clavier.

Écrire du code consiste principalement à interagir avec son clavier, repasser constamment par la souris pour certaines actions induit donc une perte de temps. Pour les apprendre sur son IDE, il existe même des plugins rappelant le raccourci clavier correspondant à chaque fois qu'on se sert de la souris (par exemple, pour IntelliJ IDEA³⁷ ou Eclipse³⁸). Si vous participez régulièrement à des *Coding Dojos*, n'hésitez pas à vous y imposer une contrainte supplémentaire pour vous forcer à maîtriser l'usage de votre clavier : interdisez-vous l'usage de tout pointeur (souris ou trackpad) !

36. Mouseless Programming, Tomaz Tekavec, 2015 <http://codurance.com/2015/11/25/mouseless-programming/>

37. Plugin Key Promoter pour IntelliJ IDEA : <https://plugins.jetbrains.com/plugin/4455>

38. Plugin MouseFeed pour Eclipse : <http://sourceforge.net/projects/mousefeed/>

ÉCRIRE DU CODE COMPRÉHENSIBLE



Écrire du code compréhensible

Par Arnaud Huon, Michel Domenjoud et Nelson da Costa

74

Nous passons 10 fois plus de temps à lire du code qu'à en écrire³⁹. Il est donc indispensable d'en optimiser sa compréhension. La notion de code propre est assez simple en réalité : il s'agit d'écrire un code qui soit suffisamment clair et explicite pour que chacun puisse le lire et le comprendre sans effort, et ainsi être plus facilement en mesure de le modifier.

Du code devrait pouvoir se lire comme on lit un journal : on doit facilement comprendre ce qu'il fait en lisant les gros titres. Plusieurs pratiques facilitent cette compréhension, par exemple respecter un bon découpage des classes et des fonctions, veiller à la séparation des responsabilités et des niveaux d'abstraction, mais aussi s'astreindre à un nommage explicite.

Et on retrouve bien souvent les mêmes problèmes flagrants dans le code existant qu'on lit, au niveau de :

- La pertinence du nommage (les variables / méthodes / classes sont-elles bien nommées ?)
- La structure des unités de code (cette méthode / classe fait-elle trop de choses ?)
- L'architecture du code (utilise-t-on le bon

design pattern ? Est-ce que cette classe hérite de la bonne interface ?).

Pour parvenir à écrire un code compréhensible, il ne suffit malheureusement pas d'utiliser des outils. Il n'est pas possible d'automatiser entièrement l'amélioration de la qualité du code et la détection d'écart à des standards. La démarche même de définition de standards, permettant de décider collectivement comment rendre le code plus clair, nécessite une réflexion y compris sur des points simples comme un bon nommage.

Nous n'abordons ici que quelques aspects de la notion de code propre. Pour aller plus loin, nous vous invitons à consulter l'un des nombreux ouvrages sur le sujet, et en premier lieu *Clean Code* de Robert C. Martin.

○ Savoir détecter quand le code a un problème

Voici quelques indices simples qui peuvent aider à déterminer si du code doit être remanié :

39. *Clean Code: A Handbook of Agile Software Craftsmanship* par Robert C. Martin (Prentice Hall, 2008)



- Des méthodes trop longues
(par exemple : au-delà de ~20 lignes).
- Des classes trop longues
(par exemple : au-delà de ~200 lignes).
- Des méthodes avec trop d'arguments
(par exemple : au-delà de 3 arguments).
- Une incapacité à lire le code « en diagonale » : il faut se plonger dans chacune des méthodes, des classes pour comprendre ce qu'elles font.
- L'abus de méthodes statiques publiques,
qui dénote souvent une violation du principe de la programmation modulaire en introduisant un état global partagé et des risques d'effets de bord.
- Plus de 2 niveaux d'indentation dans une méthode.
- Des objets sans méthodes⁴⁰
(en dehors des getters/setters) – dans un contexte de POO⁴¹.
- Des classes avec trop de dépendances.
- Des tests trop compliqués à écrire ou à lire (la classe à tester porte peut-être trop de responsabilités différentes).



40. AnemicDomainModel, Martin Fowler, 2003, <http://www.martinfowler.com/bliki/AnemicDomainModel.html>.

41. POO: Programmation Orientée Objet

Plus généralement, **faites confiance à votre ressenti** : si vous trouvez que du code est trop compliqué à lire ou à comprendre, qu'il y a quelque chose de « bizarre », de « pas propre », c'est qu'il peut nécessairement être simplifié et amélioré.

○ Bien commencer avec le nommage

Un **nommage adéquat** est sans doute la source première de **sens du code**. Pourtant cet aspect du code est souvent à tort considéré comme secondaire et même négligé.

> S'appuyer sur le vocabulaire métier

Un produit offre des fonctionnalités, liées à un métier, et celui-ci utilise logiquement un certain vocabulaire. Afin de limiter non seulement les risques d'incompréhension, mais également l'effort nécessaire pour passer du métier au code, il est important d'être précis et d'utiliser les mêmes termes que le métier pour nommer les concepts modélisés dans le code.

Par exemple, si le terme du métier pour désigner un type d'utilisateur est Vendeur, il vaut mieux éviter d'appeler la classe correspondante Commerçant dans le code.

Pour faciliter le partage du vocabulaire métier, il est important qu'un échange ait lieu entre les développeurs et les spécialistes du métier.

Behavior Driven Development (BDD) est une approche visant à renforcer cette collaboration, en s'appuyant notamment sur des exemples concrets. C'est également l'occasion de s'assurer que le vocabulaire utilisé est bien défini et partagé.

Côté implémentation, *Domain-Driven Design*⁴² (DDD) est également une approche appropriée, qui s'attache notamment à structurer le code et à le nommer en respectant le domaine métier via un langage partagé, ou *Ubiquitous Language*.

> Usage de l'anglais

L'usage de l'anglais pour coder est plus ou moins une tradition dans la programmation informatique. C'est aussi celle utilisée par la plupart des langages de programmation eux-mêmes. L'anglais a l'avantage d'être une langue précise et technique, où les termes offrent peu d'ambiguïté... quand elle est bien maîtrisée. En effet, une mauvaise maîtrise de l'anglais va au contraire obscurcir le code et souvent amener des contresens.

Au démarrage du projet, posez-vous la question de savoir quelle est la langue du métier.

Si c'est l'anglais, allez-y, mais pas à moitié. **Évitez au maximum le « franglais »**, surtout pour les termes métiers. Privilégiez un code en français si les termes métiers de l'application sont exprimés en français.

42. Domain Driven Design (DDD). Approche du Design logiciel popularisée par Eric Evans dans son livre *Domain Driven Design: Tackling complexity in the heart of software* (Addison Wesley, 2003). Voir aussi <http://blog.octo.com/domain-driven-design-des-armes-pour-affronter-la-complexite/>

Standards de nommage

Voici quelques exemples de standards de nommage valables :

- Les interfaces doivent commencer par un *I*
- Les interfaces ne doivent pas commencer par un *I*
- Les implémentations d'interface doivent finir par *Impl*
- Le nom des listes doit toujours finir par *List*
- Le nom d'une liste doit indiquer ce qu'elle contient et finir par un '*s*', etc.

L'usage d'un **glossaire d'équipe** est souvent aussi nécessaire afin de répertorier les standards de nommage qui vont au-delà du simple standard de syntaxe. C'est notamment au sein de celui-ci que l'on conservera la définition des différents termes métier complexes utilisés, et tout ce qui permet de lever des ambiguïtés.

Si vous choisissez le français, vous pouvez conserver l'usage de l'anglais pour un certain nombre de termes conventionnels dans votre nommage, notamment l'usage des verbes d'actions pour vos méthodes (*get*, *set*, *has...*) ou le nom des *Designs Patterns*⁴³ usuels. Il est nécessaire de créer un **glossaire** dans ce cas, afin de référencer tous les anglicismes acceptés.

Un cas peut éventuellement justifier l'usage d'une langue différente entre code et métier, celui où les développeurs ne comprennent pas du tout la langue du métier. L'usage d'un glossaire devient alors indispensable également.

> *L'intention*

La première question à se poser avant d'initier l'écriture d'une classe, d'une méthode ou d'une variable, c'est « quelle est mon intention ? ». Pour une fonction on peut aussi le traduire en « qu'est-ce que je cherche à faire exactement ? ». Pour une classe, ce sera « quelle est la responsabilité de cette classe ? ». Enfin, pour une variable, ce sera « qu'est-ce que cette variable représente ? ».

Dans le cadre d'une revue de code, on se pose ces mêmes questions en comparant le nom à l'implémentation. Une inadéquation entre les deux n'est peut-être pas seulement une erreur de nommage mais souvent aussi un **problème de compréhension du besoin**. Le non-respect de l'intention dans un nommage est révélateur d'un besoin de redesign.

43. Design Patterns https://fr.wikipedia.org/wiki/Patron_de_conception

> Éviter les ambiguïtés

Quelle que soit la forme qu'elle prend, l'ambiguïté aura pour conséquence une difficulté de compréhension, voire pire, une mauvaise interprétation. Elle peut se présenter sous différentes formes :

- **Ambiguïté de sens** : les termes utilisés peuvent avoir plusieurs significations dans le contexte de l'application.

```
//mauvais nommage
public class Stream{
    public void displayVideo(String videoId) {
        ...
    }
}
// meilleur nommage
public class VideoStream{
    public void display (String videoId) {
        ...
    }
}
```

78

- **Ambiguïté de lecture** : il y a une trop forte proximité syntaxique entre différents termes, notamment au sein d'une même méthode.

```
//mauvais nommage
public void transformArticles(List<Article> contents) {
    ...
    List<Article>articles = new ArrayList<Article>();
    for(Article article : contents){
        article = doSomething(article);
        articles.add(article);
    // trop de proximité entre article et articles, la lecture prend plus de temps...
    }
    ...
}

//meilleur nommage
public void transformArticles(List<Article> contents) {
    ...
    // répétition de type mais moins de proximité syntaxique
    List<Article>articleList = new ArrayList<Article>();
    for(Article article : contents){
        article = doSomething(article);
        articleList.add(article);
    }
    ...
}
```

- **Des noms trop génériques :** les termes n'expriment aucune particularité fonctionnelle.

```
//mauvais nommage
public void transformArticles(List<Article> contents) {
    ...
    List<Article>list = new ArrayList<Article>(); // terme beaucoup trop générique...
    for(Article article: contents){
        article = doSomething(article);
        list.add(article);
    ...
    }
    ...
}

//meilleur nommage
public void transformArticles(List<Article> contents) {
    ...
    List<Article>transformedArticleList = new ArrayList<Article>();
    for(Article article : contents){
        transformedArticle = doSomething(article);
        transformedArticleList.add(transformedArticle);
    ...
    }
    ...
}
```

- **Rupture du principe de moindre surprise⁴⁴** : un terme « réservé » est utilisé pour tout autre chose que son usage commun.

```
//mauvais nommage
public class ServiceManager{
    //un manager est plutôt attendu pour nommer certaines classes de coordination
    public String name;
    public String serviceName;
    public List<Employee> managees;
}

//meilleur nommage
public class DepartmentDirector{
    //on s'éloigne alors un peu du terme métier pour ne plus violer le principe
    //si c'est trop gênant, il devient nécessaire de mettre en place un standard
    //réservant le terme manager à un terme métier, et non un type de classe
    public String name;
    public String departmentName;
    public List<Employee> managees;
}
```

44. Principe de moindre surprise http://fr.wikipedia.org/wiki/Principe_de_moindre_surprise

> Limiter l'obsolescence

Suite à des évolutions ou des *refactorings*, le nom de certains concepts n'est plus en adéquation avec ce qu'ils font.

On le constate aussi souvent sur des classes qui ont trop grossi au fur et à mesure de l'ajout de méthodes. Leurs noms ne sont plus alors que la représentation d'un sous-ensemble de leurs usages. Ceci indique souvent qu'il faudrait découper la classe ou la méthode concernée.

○ Et le design dans tout ça ?

Obtenir un bon design logiciel est très important pour préserver la maintenabilité d'un produit sur le long terme. Pourtant, nous n'avons pas abordé ce sujet en premier lieu, pour une raison simple : les autres points déjà évoqués sont à considérer en priorité.

Dans les équipes que nous rencontrons, qui connaissent souvent des douleurs sur leur code existant, nombreux sont ceux qui se focalisent sur des problématiques d'architecture et de design.

« Il faut changer de framework, celui-ci est obsolète ! » ou encore « Il faut qu'on utilise DDD, CQRS⁴⁵ et qu'on bascule en microservices^{46 47} ! ». Oui, peut-être. Il est fort probable effectivement que certaines approches de design soient plus adaptées que l'existante.

Mais avant de foncer tête baissée vers ces aspects, souvent considérés comme plus « sexy » et gratifiants, posez-vous la question : avez-vous des pratiques de développement collectives qui vont vous permettre de bien appréhender ensemble ces problématiques complexes ?

Parvenez-vous ensemble à produire un code clair pour tous ?

Si ce n'est pas le cas, il est presque certain que changer de design ou de technologie ne résoudra pas les problèmes de qualité et qu'ils se répéteront rapidement.

Afin de savoir où placer l'effort, il est important de prioriser les besoins de design lorsque l'on écrit du code. En premier lieu, **appliquer par ordre de priorité les règles d'un Design Simple^{48 49}** énoncées par Kent Beck dans la démarche eXtreme Programming :

- 1.** Tous les tests passent (c'est-à-dire que le code fonctionne).
- 2.** L'intention est claire.
- 3.** Il n'y a pas de duplication.
- 4.** Le code a le moins d'éléments possibles. Ici, on appliquera les principes KISS⁵⁰ et YAGNI⁵¹.

45. CQRS: Command Query Responsibility Segregation. Voir <http://blog.octo.com/cqrs-larchitecture-aux-deux-visages-partie-1/>

46. Microservices, a definition of this new architectural term, Martin Fowler & James Lewis <http://martinfowler.com/articles/microservices.html>

47. L'architecture microservices sans la hype : qu'est-ce que c'est, à quoi ça sert, est-ce qu'il m'en faut ?, <http://blog.octo.com/larchitecture-microservices-sans-la-hype-quest-ce-que-cest-aquoicassertest-quel-menfaut/>

48. eXtreme Programming Simplicity Rules, Kent Beck <http://c2.com/cgi/wiki?XpSimplicityRules>

49. Les règles de Simple Design expliquées par Martin Fowler, <http://martinfowler.com/bliki/BeckDesignRules.html>

Si on s'est assuré que ces quatre règles sont respectées, alors seulement on se pose la question de savoir si des principes de design plus avancés sont respectés et si certains patterns peuvent s'appliquer⁵², notamment les cinq principes **SOLID**⁵³, que l'on peut résumer ainsi :

- **S**ingle Responsibility : ne faire qu'une seule chose mais la faire bien.
- **O**pen/Closed : une classe doit être ouverte à l'extension mais fermée à la modification (plugins).
- **L**iskov Substitution : toute implémentation d'une interface doit pouvoir se substituer à une autre.
- **I**nterface Segregation : une classe ne doit implémenter une interface que si elle a réellement besoin de remplir son contrat.
- **D**ependency Inversion : le code ne doit pas interagir directement avec l'extérieur mais doit passer par des abstractions (et vice versa).

Le respect de cette hiérarchie de principes doit permettre une approche de design évolutif⁵⁴ – règle d'architecture au cœur du développement agile, on parle également de *Design émergent* – plutôt qu'un design entièrement pensé à l'avance (on parle de *Big Design Up Front*⁵⁵). En plus de rendre le code plus simple à comprendre, le respect de ces principes permet d'obtenir un code plus évolutif.

50. KISS: Keep It Simple & Stupid https://fr.wikipedia.org/wiki/Principe_KISS

51. YAGNI: You Ain't Gonna Need It <https://fr.wikipedia.org/wiki/YAGNI>

52. A Hierarchy of Software Design Needs, Jason Gorman <http://codemannship.co.uk/parlezuml/blog/?postid=1321>

53. SOLID [https://fr.wikipedia.org/wiki/SOLID_\(informatique\)](https://fr.wikipedia.org/wiki/SOLID_(informatique))

54. Is Design Dead ? Martin Fowler, 2004 <http://martinfowler.com/articles/designDead.html>

55. Big Design Up Front (BDUF) https://en.wikipedia.org/wiki/Big_Design_Up_Front

RÈGLES D'UN DESIGN SIMPLE	
ÉTAPE 1	Tous les tests passent (c'est-à-dire que le code fonctionne).
ÉTAPE 2	L'intention est claire.
ÉTAPE 3	Il n'y a pas de duplication.
ÉTAPE 4	Le code a le moins d'éléments possibles. Ici, on appliquera les principes KISS et YAGNI.
PRINCIPES SOLID	
ÉTAPE 5	<p>Single Responsibility : ne faire qu'une seule chose mais la faire bien.</p> <p>Open/Closed : une classe doit être ouverte à l'extension mais fermée à la modification (plugins).</p> <p>Liskov Substitution : toute implémentation d'une interface doit pouvoir se substituer à une autre.</p> <p>Interface Segregation : une classe ne doit implémenter une interface que si elle a réellement besoin de remplir son contrat.</p> <p>Dependency Inversion : le code ne doit pas interagir directement avec l'extérieur mais doit passer par des abstractions (et vice versa).</p>

⦿ Points à retenir

Le développement est essentiellement une problématique de partage d'information entre les développeurs, mais aussi avec les autres acteurs de l'équipe. L'expressivité du code est donc essentielle, et un nommage adéquat y contribue fortement, de même que d'autres aspects que nous n'avons pas évoqués ici.

Cette adéquation ne peut être obtenue et maintenue que par une réflexion constante sur **l'intention** et sur **l'absence d'ambiguïté** dans sa description. Malheureusement, ce qui peut paraître simple et évident pour un développeur ne l'est peut-être pas pour un autre. L'écriture d'un code de qualité repose donc sur une approche collective du développement, facilitant les nombreuses interactions nécessaires.

Pour améliorer la « propreté » de son code, plusieurs pratiques sont indispensables :

- **Améliorer individuellement la qualité du code** au fil de sa compréhension en appliquant la **Boy Scout Rule** au cours des phases de lecture de code individuelles (par exemple, lorsque l'on relit le code avant de le pousser vers le gestionnaire de versions).
- **Améliorer collectivement la compréhension du code grâce aux revues de code.**

- **S'appuyer sur des standards** et utiliser un glossaire d'équipe si nécessaire.

- **Privilégier un design simple** en priorisant le respect de certains principes.

⦿ Par où commencer ?

- Organisez un atelier regroupant les intervenants du métier et les développeurs afin d'identifier un vocabulaire partagé.

- Au cours des prochains développements, nommez les concepts dans votre code à plusieurs (*pair programming*, revue de code, demande d'aide ponctuelle).

- Organisez une session de revue de code collective avec pour objectif d'initier un document de standards.

- Initiez un rendez-vous récurrent pour échanger autour des principes d'écriture d'un code propre, KISS, YAGNI, puis dans un deuxième temps, des principes SOLID.

- Un format qui a bien fonctionné pour nous est un groupe de lecture qui se réunit régulièrement pour échanger sur la lecture d'un ouvrage, ou encore sur la lecture d'articles marquants.

LA REVUE DE CODE



La revue de code

Par Michel Domenjoud

Dans la plupart des domaines impliquant l'écriture, on n'imagine pas que ce qui est écrit puisse être publié sans avoir été relu. Un article devrait toujours être relu avant publication (par exemple, le texte que vous êtes en train de lire), que ça soit pour une vérification sur le fond – le sujet de l'article est-il bien traité ? – ou sur la forme – orthographe, grammaire, structure et lisibilité du texte.

De la même manière, la pratique de la revue de code consiste à faire relire son code afin d'y trouver le maximum de défauts, que ça soit sur le fond – est-ce que ce code fonctionne, et matérialise bien la fonctionnalité prévue ? – ou sur la forme – clarté, lisibilité, respect des standards, etc.

La revue de code est une **pratique presque aussi ancienne que le développement de logiciel⁵⁶**, et très répandue dans des entreprises comme Microsoft⁵⁷ ou Google⁵⁸. Il y a une bonne raison à cela, c'est qu'elle permet de détecter les défauts plus tôt dans le processus de développement.

Pourtant, cette pratique est relativement peu répandue dans les équipes que nous

rencontrons. C'est parfois parce qu'elle est méconnue, mais c'est plus souvent pour de mauvaises raisons :

« *On n'a pas le temps, ça coûte trop cher.* »

« *On a essayé mais ça a tourné au troll.* »

Ou encore, « *Je ne veux pas le faire, c'est du flichage.* »

Cette pratique n'est effectivement pas si évidente à mettre en place, mais suffisamment simple et efficace pour que nous la considérons comme **la première pratique de développement collective à mettre en œuvre dans une équipe**.

○ Déetecter les défauts au plus tôt

L'objectif principal de la revue de code est le même que les autres méthodes d'assurance de la qualité du logiciel : **il s'agit de trouver au plus tôt les défauts** qui existent dans le code.

« *Oui, mais nous faisons déjà des tests pour détecter ces défauts !* »

56. M.E., Fagan (1976). *Design and Code inspections to reduce errors in program development*. IBM Systems Journal, <http://www.mfagan.com/pdfs/ibmfagan.pdf>

57. Alberto Bacchelli and Christian Bird (2013), *Expectations, Outcomes, and Challenges of Modern Code Review*, Microsoft Research, <http://research.microsoft.com/apps/pubs/default.aspx?id=180283>

58. How Google Does Code Review, Paul Hammant, 2013 <http://java.dzone.com/articles/how-google-does-code-review>



Tester une application fonctionnellement, notamment en mode boîte noire, est également important mais ne permet pas de trouver les mêmes défauts, car :

- Ces tests ne détecteront pas les écarts vis-à-vis des standards de code.
- Certains défauts fonctionnels impliquent des cas difficiles à reproduire, qui ne sont détectés que par des tests exploratoires manuels.

Les tests, même s'ils sont automatisés, ne détecteront pas tout : on n'aura un retour que sur ce que l'on a effectivement testé. La relecture peut permettre d'identifier ces cas au plus tôt et sécuriser le code.

Les bénéfices de la revue de code ne sont plus à démontrer : selon des études regroupées par Capers Jones⁵⁹ sur plus de 12 000 projets, la revue de code collective permet de détecter en moyenne 65 % des défauts, 50 % pour la revue par un pair, tandis que les tests n'en détectent en moyenne que 30 %.

Il existe d'autres bénéfices à la revue de code :

- **Améliorer la qualité intrinsèque** du logiciel et faire évoluer les standards.
- **Renforcer la propriété collective du code.** Faire relire le code par une autre personne que son auteur contribue à renforcer la propriété collective, et réduit le risque d'une spécialisation trop forte où seul l'auteur du code est capable de le modifier.

59. Jones, Capers; Ebert, Christof (April 2009). *Embedded Software: Facts, Figures, and Future*. IEEE Computer Society, <http://doi.ieeecomputersociety.org/10.1109/MC.2009.118>

- **Faciliter l'apprentissage.** La revue de code est un moyen de former les plus jeunes en partageant les standards en place, en leur montrant certaines techniques, ou comment certaines fonctionnalités sont implémentées. C'est aussi l'occasion pour ceux qui sont arrivés plus récemment d'apporter un point de vue différent et proposer de nouvelles approches.

- **Améliorer la qualité des échanges** entre les développeurs.

Tous ces éléments font de la revue de code **une pratique collective indispensable** qui contribue à diffuser une **culture de la qualité** au sein de l'équipe.

> *Un coût à mettre en relation avec ses bénéfices*

La revue de code présente l'avantage d'intervenir très tôt dans le processus de développement : qu'elle soit collective ou par un pair, elle intervient de préférence avant que la fonctionnalité ne soit testée par un utilisateur. **Et plus la détection d'un défaut intervient tôt après l'écriture du code, moins sa correction coûtera cher.** Les études sur le sujet sont nombreuses : pour aller plus loin, vous pouvez consulter les chapitres sur la qualité logicielle et les pratiques de construction collaborative dans *Code Complete*⁶⁰, qui mentionnent de nombreuses références.

Réaliser des campagnes de tests manuels peut prendre jusqu'à 20 % du temps dans le

cycle de développement, et corriger des bugs tardivement a un coût exponentiel⁶¹. Alors n'est-il pas préférable de **prendre deux heures de revue collective par semaine, soit 5 % du temps sur une semaine de 40 heures ?**

Attention, nous ne préconisons pas d'abandonner d'autres pratiques d'assurance de la qualité. La revue de code est une pratique indispensable à combiner avec d'autres pratiques, comme les tests automatisés, afin de maximiser la qualité du produit avant de le proposer aux utilisateurs.

o Quel format choisir ?

Nous utilisons principalement **trois formats de revue de code** dans nos projets : la **revue collective**, plutôt formelle, la **revue par un pair**, un format plus léger, ainsi que le **pair programming**, qui offre un feedback immédiat. Les trois ont leurs avantages et inconvénients : revenons ensemble sur ces formats et comment les mettre en place dans une équipe.

> *Revue de code collective*

La revue de code collective consiste à organiser des sessions d'inspection formelles, régulières et en groupe, dédiées à la détection de défauts.

60. Steve McConnell, *Code Complete 2nd edition*, 2004, chapitres Software Quality Assurance & Collaborative Construction <http://www.cc2e.com/Default.aspx>

61. Loi du coût exponentiel des défauts, initialement énoncée par Capers Jones. *Software Assessments, Benchmarks, and Best Practices*, Addison-Wesley, 2000

REVIEW COLLECTIVE

- Temps et périmètre limités :
 - > Séance d'1h30 maximum, une fois toutes les deux semaines.
 - > Viser un taux de revue de 300 lignes de code par heure.
 - > Consacrer plus de temps ou essayer de relire plus de code en une seule session nuit à la qualité de la revue.
- **Dédiée à la détection des défauts** : la séance est dédiée uniquement à relever les défauts, pas à les corriger en séance, ni à débattre des questions de standards ou de design : on consacre une minute maximum à chaque défaut.
- Statut suivi : en fin de séance, on décide si le code revu est accepté ou rejeté. Le cas échéant, les défauts retenus à l'issue de la revue sont enregistrés et leur correction est vérifiée.
- Participants :
 - > L'auteur du code est présent, ainsi que plusieurs relecteurs.
 - > Afin de faciliter le déroulement, on désigne :
 - Un modérateur**, qui garantit qu'on se concentre sur la découverte des défauts, et que les échanges sont de qualité.
 - Un scribe**, qui note les défauts relevés.
 - Un gardien du temps**.
- Préparée : le code à relire est sélectionné et communiqué à l'avance, afin que les relecteurs puissent étudier le code à tête reposée avant la séance.
- Outillée : on construit progressivement des checklists afin de faciliter le déroulement de la revue.

Comment sélectionner le code ?

Dans l'optique de détecter les nouveaux défauts introduits, on sélectionne le code ajouté ou modifié pour réaliser les dernières fonctionnalités.

- On relit idéalement tout le code produit récemment, mais ce n'est faisable que si l'équipe est suffisamment petite.
- Sinon, on sélectionne dans le code récemment modifié une partie sensible, complexe, ou qui introduit de nouveaux concepts, un nouveau pan de l'architecture logicielle, etc.

On peut aussi sélectionner du code existant qui fait « peur », sujet à des régressions fréquentes, notamment si on sait qu'on va devoir le faire évoluer prochainement.

> Revue par un pair

La revue par un pair est une pratique plus largement répandue car plus simple et moins contraignante à mettre en place.

Qui fait la revue de code ?

Idéalement, la première personne disponible s'occupe de la revue. Tout comme dans la version collective, il faut éviter que ça soit toujours la même personne qui s'occupe des revues, pour favoriser la propriété collective du code.

Si c'est une étape obligatoire dans votre processus de développement, affichez-la sous forme d'une nouvelle colonne sur votre board avant l'étape de revue fonctionnelle.

> Pair Programming

Le *pair programming*, ou binômage, est la pratique permettant à deux développeurs de travailler sur une même base de code.

Nous considérons que le *pair programming*, bien utilisé, permet une revue de code tout à fait valable, avec l'avantage majeur de fournir un **feedback immédiat**. Cette pratique est adaptée à de nombreuses situations, à tel point que certaines équipes développent en permanence en binôme. Elle est d'autant plus appropriée :

- Pour un apprentissage, une montée en compétences entre deux développeurs de niveaux différents.
- Pour le développement de fonctionnalités complexes, un *refactoring* difficile ou la correction d'un bug, deux paires d'yeux valent souvent mieux qu'une.

Selon l'objectif du binômage, il convient d'utiliser une approche appropriée. Par exemple, si vous binômez pour aider un développeur à progresser, il faut bien faire attention à lui laisser suffisamment le clavier et le laisser faire des erreurs, même si vous avez l'impression de ralentir⁶².

62. *Pairing with Junior Developers*, Sarah Mei, 2015, <https://devmynd.com/blog/2015-1-pairing-with-junior-developers/>

REVUE PAR UN PAIR

- Points communs avec la revue collective :
 - > Dédiée à la **détection** des défauts et **non aux corrections**.
 - > Préparée.
 - > Statut suivi.
- En binôme :
 - > Afin d'obtenir des bénéfices similaires à ceux d'une revue collective, la revue occasionne un échange entre l'auteur du code et le relecteur.
 - > Lors de la revue, une fois que l'auteur a présenté le code pour partager l'intention du code revu, c'est le relecteur qui navigue dans le code.
- Périmètre limité à une fonctionnalité : dès que le développement d'une fonctionnalité, d'une User Story, est considéré comme terminé par le développeur, une revue est déclenchée sur ce code.
- Systématique : la revue est systématique et obligatoire pour tout code destiné à arriver en production.
- Outilisée :
 - > Le code revu est directement lié à un ou plusieurs *commits*, on utilise généralement le différentiel des modifications comme support.
 - > On utilise souvent un outil plus avancé pour tracer le suivi des défauts, des outils comme Gerrit⁶³, ou GitHub⁶⁴ directement.
 - > Sur certains projets, nous avons aussi simplement attaché la liste des défauts à la User Story associée sur le *board*.

63. Gerrit : outil d'assistance à la revue de code. <https://code.google.com/p/gerrit/>. Faire de la revue de code avec GitHub <https://github.com/features/#code-review>

64. Faire de la revue de code avec GitHub <https://github.com/features/#code-review>

C'est une pratique assez simple sur le papier, mais qui demande une certaine discipline pour être efficace :

- Il est important que pilote et copilote soient tous deux actifs dans l'échange.
- La communication et les *feedbacks* se doivent d'être de bonne qualité pour obtenir une bonne collaboration.
- Comme il requiert une attention et des échanges constants, le *pair programming* peut être fatigant. Ménagez-vous des pauses fréquentes et alternez éventuellement avec des moments de programmation plus classiques.

91



Cette étude menée par Alistair Cockburn⁶⁵ montre que si le *pair programming* entraîne une augmentation d'environ 15 % du temps de développement (pour une fonctionnalité donnée), elle est largement compensée par

un gain en qualité logicielle ainsi qu'une meilleure diffusion du savoir au sein de l'équipe.

> *Que faire des défauts relevés lors de la revue ?*

Pour être efficace, une revue de code doit être **dédiée à relever les défauts. La correction ou les éventuelles discussions ont lieu séparément :**

- Lorsque la correction à appliquer est claire et ne fait pas débat (défaut fonctionnel, écart à un standard établi), l'auteur du code réalise la correction après la revue et un suivi est effectué pour s'assurer que les défauts ont bien été corrigés.
- Souvent, deux implémentations différentes se valent. Dans le cas d'une revue en binôme, il sera donc utile de faire appel à un tiers ou toute l'équipe pour faciliter la prise de décision.
- Lorsque le point d'attention concerne un standard non établi, on fait évoluer les standards écrits de l'équipe, en s'assurant qu'ils soient partagés par tous.
- Il est courant que certains points plus complexes nécessitent un échange avec toute l'équipe :
 - > On peut réserver une demi-heure en fin de revue collective, ou solliciter l'équipe pour un point dédié dans le cas de la revue par un pair.

65. *The Costs and Benefits of Pair Programming*, Alistair Cockburn, <http://www.cs.utah.edu/~lwilliam/Papers/XPSardinia.PDF>

PAIR PROGRAMMING

- En binôme :

- > Avec une unique station de travail : 1 ordinateur et 1 clavier pour 2 personnes.
- > Deux rôles : pilote et copilote. Le développeur au clavier – le pilote – se focalise sur le développement, tandis que l'autre développeur – le copilote – l'aide à avancer, grâce à ses observations et ses commentaires.

- Changement de rôle régulier :

Les développeurs changent de rôle régulièrement, créant ainsi une dynamique productive. Ce rythme peut être régulier – avec utilisation d'un chronomètre – ou non, en fonction des compétences techniques de chacun.

- N'est pas spécifiquement dédié à la revue :

Toutes les activités de développement se font en binôme. La revue est donc faite au fil de l'eau, et peut être complétée par une revue que les deux développeurs effectuent avant de partager leur code au reste de l'équipe.

- Est utilisé ponctuellement ou en permanence selon les équipes.

> Sur plusieurs projets, nous avons également ritualisé un créneau d'une ou deux heures en fin d'itération, appelé selon les équipes « l'heure Tech » ou le *Coding Dojo*, au cours duquel on peut aborder ces points.

> Quel format choisir ?

Le tableau ci-contre donne quelques critères (non exhaustifs) de comparaison entre les trois formes de revue de code présentées.

Les différentes études évoquées dans *Best Kept Secrets of Peer Code Review*⁶⁶ et les chapitres sur la qualité logicielle et les pratiques de construction collaborative dans *Code Complete* montrent que la revue collective est globalement plus efficace dans la détection des défauts, mais présente l'inconvénient d'être plus lourde et coûteuse à mettre en œuvre.

Ces études montrent également que **la très grande majorité des défauts sont relevés lors de la relecture en préparation de la revue**. Ceci nous conforte dans l'utilisation de la revue par un pair, à condition qu'elle soit bien préparée : elle permet d'obtenir un bon compromis en évitant le coût d'une réunion avec de nombreux participants.

On peut aussi utiliser les revues de code collectives formelles en complément des revues en paires :

- En format court restreint aux points qui ne sont pas évidents, non couverts par les standards.

- Pendant la période de mise en place des revues, afin que l'équipe se forme collectivement à la pratique.

Enfin, le *pair programming* est un format très efficace également dans certaines situations ; il est même utilisé en continu dans certaines équipes bien rodées.

On trouve de nombreux articles suivant le schéma « Les [avantages | dangers] du [Code | Pair Review | Pair Programming] », mais au final, le débat n'est pas là. Même si nous émettons un avis plus en faveur d'un format, peu importe que vous fassiez du *pair programming*, de la revue collective ou par un pair : **l'essentiel est de trouver l'approche qui correspond à la culture de votre équipe**.

Et il est fort probable qu'un mélange des trois pratiques soit pertinent. Nous utilisons souvent l'approche suivante :

- Revue de code par un pair obligatoire pour tout le code, à moins qu'il ne soit considéré comme revu par l'un des deux autres formats.
- Utilisation opportune de la revue collective sur des sujets ciblés.
- *Pair programming* fréquent, notamment en début de développement d'une fonctionnalité.

⁶⁶. Best Kept Secrets of Code Review, Smart Bear Software, 2006, <http://smartbear.com/SmartBear/media/pdfs/best-kept-secrets-of-peer-code-review.pdf>

Avantages et inconvénients des différents types de revue

	REVUE COLLECTIVE	REVUE PAR UN PAIR	PAIR PROGRAMMING
EFFICIENCE (nombre de défauts détectés)	+++	++	++
PROPRIÉTÉ COLLECTIVE DU CODE	+++	++	+++
AMÉLIORATION DE LA QUALITÉ, ÉVOLUTION DES STANDARDS	+++	+++	+++
FACILITÉ DE MISE EN ŒUVRE	+	+++	++
RAPIDITÉ DU FEEDBACK	+	++	+++

L'article *Pairing vs. Code Review : Comparing Developer Cultures*⁶⁷ propose une analyse intéressante du sujet et nous reprendrons sa conclusion ici : analysez et admettez les faiblesses dans vos pratiques de développement collaboratives. Si vous n'avez

pas encore choisi une approche, échangez avec des développeurs pour qui ces pratiques sont régulières. Expérimitez et cherchez à vous améliorer, lors d'une rétrospective par exemple.

67. *Pairing vs. Code Review: Comparing Developer Cultures*, Paul Hinze, 2013, <http://phinze.github.io/2013/12/08/pairing-vs-code-review.html>

Comment rater vos revues de code ?

Introduire une nouvelle pratique avec succès n'est pas chose aisée. C'est un peu comme mettre une barque à la mer : une fois dans l'eau, les premiers mètres sont assez chaotiques. Il y a beaucoup de vagues, on commence à se demander si c'était une bonne idée. Ne serait-il pas sage de retourner au rivage ? Mais en persévérant, on arrive finalement au large, où la mer est plus calme : il suffisait de s'accrocher.

95

Nous avons rencontré dans nos expériences de revue de code plusieurs écueils qui nuisent aux bénéfices attendus et qui peuvent mettre en péril la pratique dans l'équipe.

Voyons au travers de quelques anecdotes vécues pourquoi vos premières revues de code risquent d'être difficiles et quels sont les fondamentaux nécessaires à des revues de code réussies.

○ Mon équipe ne progresse pas



Au sein d'une équipe que je venais de rejoindre, j'entends un jour Bob râler devant son poste :

« Mais il n'est pas possible Martin, combien de fois a-t-on dit qu'on ne construisait plus les requêtes SQL avec des Strings mais avec des Criteria Hibernate ! »

Je suis allé voir Bob pour comprendre ce qu'il se passait. Il m'explique alors qu'il a revu du code écrit par Martin et qu'il a encore dû corriger une requête SQL qui ne répondait pas aux standards.

- Moi : Mais, ils sont écrits quelque part vos standards ?

- Bob : Non... mais on les connaît, Martin est le seul à ne pas les respecter. Bon en même temps il est arrivé récemment.

- Moi : Effectivement... Mais j'ai une autre question : tu as dit que tu as dû corriger le code toi-même ?

- Bob : Oui, on fait comme ça, je préfère corriger moi-même pour être sûr. Et je ne vais pas aller l'interrompre pour ça.

- Moi : Et si on allait lui en parler ?

> La revue n'occasionne pas d'échange

Dans cette conversation, pourquoi Martin n'a-t-il pas intégré le standard ? **Relire le code de quelqu'un, c'est l'occasion de lui donner un feedback sur son code**, et de l'aider à améliorer ses pratiques : **un échange avec l'auteur du code est indispensable**. Sans cela, le relecteur a beau détecter les défauts, ceux-ci réapparaîtront inlassablement dans le code futur car l'auteur n'a pas obtenu de retour dessus.

La revue de code doit être préparée par le relecteur, au moyen d'une première relecture pour relever des défauts ou des questions à poser. Cela permet d'en effectuer une partie de façon asynchrone, mais le risque est qu'elle se termine sans occasionner cet indispensable échange, idéalement verbal.

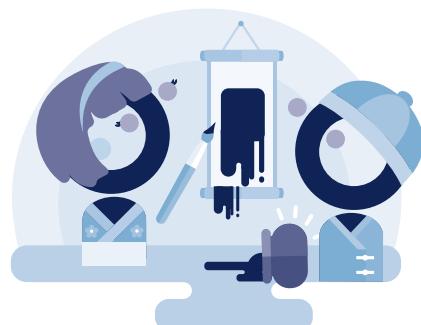
> L'auteur ne corrige pas les défauts

L'exemple précédent montre aussi un manque de confiance de la part du relecteur envers l'auteur du code. C'est dommage car le meilleur moyen de progresser, d'apprendre et d'intégrer un standard est de l'appliquer soi-même.

Si la revue de code est l'occasion de transmettre du savoir, c'est aussi le cas de l'étape de correction des défauts :

- Il est donc important que ce soit **l'auteur qui corrige les défauts relevés**.
- Afin d'éviter que le relecteur corrige les défauts par manque de confiance, on peut démarrer une séance de *pair programming* et construire la correction à deux.

Il s'agit aussi de responsabiliser l'auteur du code : corriger les défauts d'un autre développeur peut l'infantiliser, alors que l'aider à les corriger lui-même le responsabilise.



> Des standards non partagés

L'utilisation de standards de code est une autre pratique essentielle pour parvenir à des revues efficaces. Ces standards sont construits collectivement par l'équipe, et écrits par exemple dans un wiki.

S'ils ne sont pas écrits, on va forcément les oublier ou les réinterpréter, et donc ne pas les assimiler. Ces standards, initiés dès le début d'un projet, évoluent ensuite dès que nécessaire, par exemple lors d'une « heure Tech » récurrente.

● Mes revues se transforment en débats sans fin et autres « trolls »

Pendant la première revue de code collective organisée au sein de l'équipe :

- « - Kent : Dis voir Becky, à la ligne 984, ton accolade n'est pas à la ligne, et en plus ta variable \$moneyList ne respecte pas la casse standard : ici on utilise du snake_case !
- Becky : Alors, d'une part il n'y a pas de « casse standard », je ne suis pas d'accord. Et de toute façon c'est has been ! Si on suivait la norme, on serait en PSR 42 et il n'y aurait pas de question à se poser !
- Kent : Mouais il y a des choses bien dedans, mais je ne suis pas d'accord avec leurs règles de nommage !

La discussion dure encore un bon moment, mais arrêtons-nous ici. »



Qu'elles soient collectives ou en binôme, les premières revues de code organisées au sein d'une équipe sont souvent « polluées » par des débats interminables, généralement sur des points censés être simples.

Afin de couper court à ces débats, il est indispensable d'établir au plus tôt des standards écrits, comme on l'a vu auparavant. Et ceci surtout pour des choses pouvant paraître anodines comme le nommage (quelle norme PSR utilise-t-on en PHP ?) ou des particularités controversées du langage (faut-il utiliser var en C# ?).

Il est peu probable que toute l'équipe soit parfaitement alignée sur ces points. Chez OCTO Technology, nous ne sommes pas d'accord sur tout non plus. **L'essentiel est de trouver un compromis, et de l'écrire dans les standards.**

Malgré cela, on rencontre aussi des aspects non traités par les standards, ou des points plus compliqués qui nécessitent une discussion avant de décider d'une correction.

C'est pour cela que, pour conserver une revue efficace :

- Quel que soit le format, toute discussion ne concernant pas la détection d'un défaut est reportée après la revue.
- En revue collective, on ne passe pas plus d'une minute de discussion par défaut trouvé. Un modérateur et un gardien du temps y veillent.

Si un point fait débat, on le note dans une liste « à décider, non standard » : le tout est que le débat n'ait pas lieu au cours de la revue.

◎ Ça fait mal quand on critique mon code

Lors d'une séance de coaching, je demande :

- Alors, comment s'est passée votre dernière revue ?
- Xavier : Assez tendue... ça a clashé entre Étienne et Vincent.
- Moi : Pourquoi ?
- Xavier : Étienne présentait le code qu'il a produit sur la User Story #135, et Vincent lui a dit. « C'est de la m**** ton implem' ! ». Du coup, tu connais Étienne, il s'est braqué et n'a plus rien dit jusqu'à la fin.

Damien Beaufils, Coach Craft

Par essence, la revue de code comporte une problématique humaine : il s'agit de faire relire la production d'une personne par ses pairs et d'en faire une critique. Certaines sessions se passent mal lorsque le relecteur pointe l'auteur du code plutôt que les défauts relevés, ou que l'auteur prend les critiques du code comme une attaque personnelle. Cela devient particulièrement difficile à animer sur une session collective car certaines personnes risquent de se braquer et refuser la revue.

Un point essentiel dans la revue de code, et plus largement dans les pratiques collectives de développement est de **donner un feedback de qualité à ses pairs**. Notamment, il s'agit de dissocier le code et la personne qui l'a écrit, dans ses formulations mais aussi dans la réception du feedback.

Le manifeste *Egoless programming*⁶⁸ présente très bien ces principes :

- N'hésitez pas à l'afficher et vous y référer dès que ce type de problème se produit.
- C'est aussi le rôle du modérateur de s'assurer de la qualité des échanges lors des sessions collectives.

En cas de difficultés, des outils de communication qui ne sont pas propres à l'informatique peuvent être utilisés, par exemple la CNV (Communication Non Violente).

S'il n'y avait qu'une seule phrase à retenir sur ce point :

*Soyez doux avec les développeurs,
pas avec le code.*

99

○ Malgré la mise en place des revues, on croule toujours sous les bugs



Je venais de terminer le développement d'une fonctionnalité assez complexe qui m'avait occupé près d'une semaine. Cela avait impliqué un certain nombre de refactorings, faisant que la quantité de code modifié à revoir était plutôt conséquente.

Je déplace alors ma User Story dans la colonne Code Review du board, et préviens l'équipe que cette fonctionnalité est en attente de revue, au cas où quelqu'un puisse s'en charger tout de suite.

Ça tombe bien, Benoît est disponible et démarre immédiatement la revue. Il vient me voir un quart d'heure plus tard : « Hormis une petite erreur de nommage, tout est clair et fonctionnel ».

Trois semaines plus tard, un bug bloquant est découvert en production, sur un cas difficile à reproduire. Nous passons près de deux jours avant d'en déterminer l'origine.

Mais une fois que nous avons trouvé, nous nous sommes sentis un peu bêtes : l'erreur était évidente en lisant le code...

Il est possible que malgré les revues, de trop nombreux défauts continuent à être détectés trop tard, alors qu'ils semblent évidents une fois détectés.

68. The Ten Commandments of Egoless Programming: <http://blog.codinghorror.com/the-ten-commandments-of-egoless-programming/>

> La revue n'a pas été préparée

La revue doit être préparée par les relecteurs. Sans cela, il est probable que l'on passe trop rapidement sur le code pour que les défauts soient correctement relevés en séance. C'est ce qui s'est produit au cours de l'anecdote évoquée précédemment.

Il arrive également que l'auteur navigue lui-même dans le code : c'est utile pour expliquer le code et faciliter la compréhension de l'intention derrière le code. Mais si on utilise ce mode de navigation pour toute la revue, il est possible que l'auteur navigue trop rapidement dans le code ou omette accidentellement d'en parcourir certaines parties, ce qui est d'autant plus risqué si la revue n'a pas été suffisamment préparée.

> Le rythme est trop élevé

Nous avons rencontré un second écueil dans cette anecdote : le développement de la fonctionnalité a pris une semaine entière et beaucoup de code a été produit.

Revoir dix à vingt lignes de code peut se faire en quelques minutes, mais si la revue concerne beaucoup de code, il faudra logiquement y passer plus de temps.

Dans la présentation des formats de revue, nous précisions qu'il est important de limiter la quantité de code revu en une séance, ainsi que la durée de ces séances. Au-delà d'un certain rythme, les relecteurs ne détecteront que peu ou pas de défauts supplémentaires, car il est fort probable que l'attention ait diminué et qu'on laisse alors passer certains problèmes.

Afin de limiter la quantité de code à revoir, il est indispensable que ce soit une pratique régulière :

- Si les User Stories ou les tâches ne sont pas assez finement découpées, on peut travailler avec le Product Owner pour trouver une granularité plus fine⁶⁹.
- Si vous utilisez un processus de revue au fil de l'eau, n'hésitez pas à matérialiser une limite d'en cours sur la colonne correspondante : c'est une étape qui ne doit pas induire de blocage dans le flux. Certaines équipes utilisent cette limite pour déclencher une revue collective dès qu'il y a un stock de relecture suffisant.
- Enfin et surtout, rien n'interdit de commencer la revue avant la fin du développement d'une fonctionnalité.

69. Story Splitting: quelques ressources pour aider à découper ses User Stories, <http://guide.agilealliance.org/guide/split.html>

> On ne sait pas quels défauts relever

Les membres de l'équipe peuvent ne pas avoir le même niveau d'exigence, ou ne pas savoir quels défauts chercher en relisant le code.

Pour remédier à ce problème, on évoquait plus tôt la nécessité d'**avoir des standards écrits et partagés**.

Il est également très **utile d'utiliser des checklists** pour aider à mener la revue :

- Ces listes contiennent des éléments à vérifier systématiquement car identifiés comme critiques, ou parce qu'ils sont constatés régulièrement.
- Constituez votre propre checklist et faites-la évoluer au fil des revues. Voici un exemple d'une (vieille) checklist en Java ici⁷⁰.
- De nombreux éléments de ces checklists peuvent être vérifiés automatiquement par des outils d'analyse de code.



Un point de vigilance concernant l'outillage : s'il peut nous aider en détectant automatiquement de nombreux défauts, **l'outillage ne peut en aucun cas remplacer la revue de code**. Même si un grand nombre de défauts potentiels peut être relevé automatiquement, il serait dangereux de penser qu'on peut se reposer intégralement dessus. Par exemple, certains points clés des principes *Clean Code* comme l'intention et le caractère explicite du code, exprimés par le nommage, nécessiteront toujours une relecture manuelle.

Enfin, on observe parfois que certains relecteurs n'osent pas relever des défauts, pensant qu'ils ne sont pas légitimes pour remettre en cause le code d'un autre, ou craignant une mauvaise réaction de l'auteur. **Il est important que le cadre de la revue soit bienveillant** et propice aux échanges pour que cette situation ne se produise pas.

> Les défauts détectés ne sont pas corrigés

Si les défauts ne sont pas corrigés bien qu'ils aient été détectés, un élément fondamental a probablement été oublié. Une revue de code, quel que soit son format, doit avoir :

- **Un statut : le code est-il accepté** en l'état ? Si non, quels sont les défauts à corriger ?
- **Un suivi : les défauts trouvés sont listés** et leur correction après coup est vérifiée, si besoin au moyen d'une rapide seconde revue.

70. Java Inspection Checklist: http://www.cs.toronto.edu/~sme/CSC444F/handouts/java_checklist.pdf

● Outiller la revue de code

On peut distinguer trois manières d'outiller la revue de code :

- **Mesurer des indicateurs dits de qualité et détecter automatiquement certains défauts**

Des outils d'analyse de code comme SonarQube⁷¹, Findbugs⁷², Checkstyle⁷³ – issus de l'écosystème Java, mais des équivalents existent pour la plupart des langages – permettent de détecter des défauts potentiels.

Ils permettent de mesurer de nombreux indicateurs (complexité cyclomatique, duplication, couverture de tests, etc.) ainsi que d'appliquer des règles qui détectent des problèmes potentiels de qualité, qui peuvent relever de la maintenabilité tout comme d'un vrai défaut.

- **Faciliter le déroulement de la revue**

Certains outils proposent de faciliter la revue elle-même en proposant une interface de visualisation et de saisie adaptée :

- > Signaler aux membres de l'équipe le code en attente de revue.
- > Proposer une vue différentielle pour relire le code.
- > Permettre d'annoter le code dans l'outil et historiser les discussions.
- > Outiller le processus de revue et suivre l'avancement.

Certaines de ces fonctionnalités sont directement intégrées dans GitHub ou dans SonarQube, et d'autres outils y sont dédiés (Review Board⁷⁴, Upsource⁷⁵, Crucible⁷⁶, Gerrit). De nombreux acteurs du web utilisent ces outils pour assister leurs revues de code, comme l'explique par exemple Google⁷⁷.

- **Intégrer la revue de code dans le processus de gestion de version**

Avec un outil de gestion de version moderne comme Git, renforcé avec Github (ou GitLab⁷⁸ en interne), il est facile de mettre en œuvre un processus de revue de code qui s'appuie sur les *pull requests*⁷⁹ et l'utilisation de *forks*⁸⁰ : toutes les modifications de la base de code sont proposées via des *pull requests*, qui doivent être revues avant l'intégration à la branche principale de développement.

71. SonarQube: <http://www.sonarqube.org/>

72. FindBugs: <http://findbugs.sourceforge.net/>

73. CheckStyle: <http://checkstyle.sourceforge.net/>

74. Review Board: <https://www.reviewboard.org/>

75. Upsource: <https://www.jetbrains.com/upsource/>

76. Crucible: <https://fr.atlassian.com/software/crucible/overview>

77. How Google does code review: <https://dzone.com/articles/how-google-does-code-review>

78. Gitlab: <https://about.gitlab.com/>

79. Faire des *pull requests* avec GitHub : <https://help.github.com/articles/using-pull-requests/>

80. Utiliser les *forks* de dépôts avec GitHub : <https://help.github.com/articles/fork-a-repo/>

Un processus outillé de validation collective :

Dans mon équipe, on utilise GitLab, et chaque développeur possède un fork de l'application. Le dépôt origin représente donc pour chaque développeur son fork, et upstream le dépôt central. On ne push que sur origin, les ajouts dans upstream se font par pull request.

On a une colonne sur notre board pour les pull requests, avec une limite d'en-cours. Il n'y a pas de responsable unique des pull requests : chacun passe sur les pull requests en cours régulièrement et laisse des commentaires d'amélioration. Ça permet surtout un partage du code avec tout le monde.

Si tu n'as plus de commentaire à faire, tu ajoutes un +1. Si tu n'as plus de commentaire à faire et qu'il y a déjà un +1, tu valides la pull request qui sera alors mergée dans le dépôt upstream automatiquement. Et bien entendu, celui qui ouvre une pull request ne peut pas la fermer lui-même.

Au final, le code qui atterrit sur le dépôt central aura été vu par au moins trois personnes, ce qui représente la moitié de notre équipe : celui qui l'a écrit et les deux (minimum) qui l'ont validé.

Mieux encore, lors de l'ouverture d'une pull request, le serveur d'intégration continue vérifie si celle-ci est valide, en testant un merge automatique, en exécutant les tests automatisés et en lançant les outils d'analyse de code. Si elle n'est pas valide, la pull request est immédiatement rejetée.

Timothée Carry, développeur

○ Le Tech Lead passe tout son temps à faire de la revue

On rencontre souvent des équipes où seules les personnes les plus expérimentées de l'équipe, voire uniquement le Tech Lead, sont habilitées à revoir le code de l'équipe.

Il arrive également dans certains cas que la revue soit faite seulement par des experts extérieurs à l'équipe. Ou encore que certains membres de l'équipe ne se sentent pas légitimes pour faire de la revue de code, car ils viennent d'arriver ou ne seraient pas assez expérimentés.

Réserver la revue aux personnes les plus compétentes devrait théoriquement permettre de s'assurer qu'un maximum de défauts soient détectés, et pallier certains problèmes évoqués depuis le début de ce chapitre.

Il est préférable que tout le monde puisse participer à la revue, car ce problème de confiance dans la participation de tous peut avoir **deux effets indésirables** :

- N'ayant pas confiance pour déléguer, le Tech Lead se retrouve surchargé et **passe son temps à faire de la revue plutôt que d'aider son équipe** au quotidien.

- Le Tech Lead étant le seul à relire officiellement et à valider le code des autres développeurs, **l'équipe perd les bénéfices de propriété collective du code** et de transmission du savoir.

Pour autant, restreindre les « droits de revue » peut avoir du sens dans certains cas⁸¹. C'est une approche assez utilisée sur des projets open source, dans lesquels sont constitués des groupes de committers, seuls habilités à intégrer des contributions et donc à décider du statut en fin de revue de code. Mais nous sommes convaincus que lorsqu'on a une équipe de développement identifiée, co-localisée, cette approche ne devrait pas être utilisée sur le long terme.

L'objectif devrait être de **parvenir à une équipe autonome et responsabilisée, où chacun participe à la revue de code**. Nous croyons à une approche en cercle de qualité plutôt que chirurgicale avec un expert entouré de ses assistants.

Voici quelques pistes pour aller vers un fonctionnement plus efficace :

- Si toute l'équipe n'est pas à l'aise pour faire de la revue, il est préférable de commencer par organiser des revues collectives plutôt qu'en binôme, en s'assurant d'embarquer les moins expérimentés pour les faire progresser.
- Si on utilise la revue en binôme, on décide collectivement, ou avec le Tech Lead, qui est le plus pertinent pour faire cette revue, quitte à monter parfois un trinôme de revue.
- Faire revoir son code par un développeur moins expérimenté n'est pas forcément risqué : il trouvera certains défauts par lui-même, posera des questions qui amèneront à détecter les défauts avec l'auteur. Il proposera peut-être aussi de nouvelles façons de faire.

ø Je n'ai pas de soutien du management

J'ai accompagné une équipe qui a mis en place la revue de code collective. Le manager a d'ailleurs décidé que des sanctions seraient prises à partir de quatre défauts trouvés dans une revue. Et là, le miracle s'est produit : les revues de code n'ont jamais détecté plus de quatre défauts !

Gerald M Weinberg, Quality Software Management⁸²

81. Oh Foreman, Where art Thou? Robert C. Martin, 2014, <http://blog.8thlight.com/uncle-bob/2014/02/23/OhForemanWhereArtThou.html>.

82. Quality Software Management, Gerald M. Weinberg, Dorset House Publishing Company, 1997

On a vu jusqu'ici plusieurs écueils qui peuvent réduire l'efficacité de la revue de code, au risque de se traduire par un manque d'adhésion de l'équipe.

Mais en plus de tout cela, un élément de taille est à considérer : avez-vous le soutien du management ? Mieux, est-il prêt à s'engager pour faire réussir la démarche ?

Un problème courant est de voir le management ou le *Product Owner* de l'équipe demander à ne pas faire la revue de code sur un sujet parce qu'on n'a « pas le temps » et que le sujet est urgent.

Il est indispensable que tous les intervenants de l'équipe soient alignés sur les enjeux, et surtout sur les bénéfices attendus de la revue. Pour convaincre des personnes non techniques, n'hésitez pas à :

- Vous appuyer sur des éléments concrets, comme les études citées au début de ce chapitre.
- Mesurer l'efficacité des revues.

Pour cela, nous utilisons **un indicateur simple et efficace : mesurer le nombre de bugs détectés en fonction de l'étape de réalisation**. Admettant que plus un défaut est détecté tard plus il coûte cher à corriger⁸³, on souhaite alors maximiser la part des défauts détectés au plus tôt dans le processus.

Attention néanmoins au risque inverse : **le management doit être convaincu, mais ne doit pas pour autant s'immiscer dans le processus de revue** au risque qu'elle soit perçue comme du « flicage ».

Cette approche vue dans l'anecdote de Weinberg comporte une erreur dans l'objectif : il s'agit de trouver un maximum de défauts pour améliorer la qualité du code, et surtout pas de trouver les personnes qui introduisent des défauts. Nous en introduisons tous car nous faisons tous des erreurs. Il vaut mieux s'inquiéter si les revues ne trouvent pas de défauts !

Il ne faut pas confondre efficacité et efficience : pour étudier si le processus de revue de code est optimal (efficience), on peut mesurer le nombre de défauts trouvés en revue, le rapport au nombre de lignes de code examinées, mais ça ne donne en aucun cas un indicateur sur les résultats de la revue par rapport à ses objectifs (efficacité).

Une règle essentielle pour éviter ce type d'écueil : **la revue de code est une pratique technique**. Donc, **seules les personnes compétentes techniquement doivent y participer**, les animer et suivre les corrections.

83. Upstream Decisions, Downstream Costs, Steve McConnell, Windows Tech Journal, 1997 <http://www.stevemcconnell.com/articles/art08.htm>

Un processus flexible :



Jour 1 :

- Yoann : Xavier, j'ai terminé ma User Story, tu pourras me faire une revue de code s'il te plaît ?
- Xavier : Pour l'instant je suis occupé, de toute façon, vu que tu as déplacé ta carte sur le board, le prochain qui sera disponible s'en occupera. Mais j'y pense, Julien n'a pas encore vu le code de la gestion des commandes, ça serait une bonne occasion de partager.
- Julien : Ok ça marche ! Si tu veux je suis disponible tout de suite : mes tests unitaires passent et je n'ai pas encore attaqué la suite de ma fonctionnalité, je suis interruptible. Et de toute façon, si on se rend compte que c'est un sujet costaud, on pourra toujours l'inscrire dans la liste des sujets pour une Revue Collective !

Un autre jour, au standup :

- Xavier : Hier, j'ai fini la Story sur le nouveau système d'emailing. Elle est en attente de revue de code, et du coup on dépasse la limite de trois cartes qu'on s'était fixée. Qui est disponible ?
- Yoann : Ces différents sujets sont plutôt sensibles, que pensez-vous de prendre une heure ce matin pour faire une revue collective dessus ?

Tous : Ok, c'est parti !

○ Points à retenir

Comme dans les autres domaines de l'écrit professionnel, la relecture du code produit est une pratique essentielle pour la qualité de l'ouvrage. Elle permet de **déetecter les défauts au plus tôt et à moindre coût, renforce la propriété collective du code, favorise l'apprentissage et améliore la qualité des échanges** entre développeurs.

On distingue principalement **trois formats : revue de code collective, revue par un pair, et pair programming**.

Enfin, l'art de la revue de code s'apprend, et pour qu'une équipe adopte cette pratique, elle devra se l'approprier et éviter les écueils les plus courants.

Au travers de ces retours d'expérience, nous vous avons déjà donné quelques pistes, mais s'il n'y en avait qu'une à retenir la voici : **soyez vigilants en organisant vos premières sessions de revue**. Si elles se passent mal dès le début et que vous ne résolvez pas rapidement les problèmes, l'intérêt pour les revues de code va chuter. La pratique risque alors d'être abandonnée ou déformée.

○ Par où commencer ?

- Formalisez votre processus de revue de code et veillez à bien le respecter : comment doit-elle se dérouler ? Quand ? Avec quels supports ?

- Faites évoluer votre pratique de la revue au cours de vos rétrospectives.

- Appuyez-vous sur des standards que vous faites évoluer et constituez des *checklists* des éléments à vérifier.

- Veillez à bien respecter votre format :

 - >Préparer la revue.

 - >Respecter un rythme soutenable.

 - >Décider d'un statut en fin de revue et suivre la correction des défauts.

- Utilisez des *checklists* pour guider la revue et ne pas oublier les points importants.

- Automatisez ce qui peut l'être pour détecter des défauts.

- Assurez-vous d'obtenir un soutien fort du management.

- Transformez la revue de code en une pratique collective, à laquelle tous les membres de l'équipe participent.

- Si votre équipe est novice avec la revue de code, commencez par vous former et privilégiez des sessions collectives, au moins dans un premier temps.

- Si les revues sont douloureuses pour des raisons d'ego, n'hésitez pas à vous faire accompagner par un intervenant neutre, un facilitateur, afin de désamorcer les situations tendues.

UNE STRATÉGIE EFFICACE DE TESTS AUTOMATISÉS



Une stratégie efficace de tests automatisés

Par Abel André, Cédric Rup, Julien Jakubowski, Michel Domenjoud

Tester un logiciel avant de le mettre à disposition de ses utilisateurs est une pratique répandue depuis les débuts de l'informatique, et son utilité n'est généralement pas remise en question. Pourtant, le sujet des tests est vaste et les approches sont nombreuses. Dans ce chapitre, nous explorerons l'approche que nous utilisons avec succès dans nos projets et que nous évoquions déjà en 2006⁸⁴.

Historiquement, et notamment dans des projets dits en « cycle en V », l'essentiel des tests sont effectués après les développements, lors d'une phase de recette, souvent par une équipe dédiée de testeurs. Cette approche présente deux inconvénients majeurs. D'une part, **le feedback⁸⁵** sur les problèmes de qualité est **trop tardif** et induit un coût élevé de correction des défauts. D'autre part, il s'agit généralement de tests complexes à mettre en œuvre (combinatoire élevée, stabilité d'environnement difficile à assurer) et longs à effectuer, rendant cette phase **fastidieuse et très coûteuse**, même en s'appuyant sur des outils tels que *Quality Center*⁸⁶ ou d'autres permettant l'automatisation comme *Selenium*⁸⁷.

Une bonne stratégie de tests devrait offrir **un feedback rapide et en continu**, afin d'**être efficace dans la correction des défauts détectés et minimiser la perte de temps**. La stratégie que nous proposons s'appuie sur les axes suivants :

- S'organiser pour **placer la pratique des tests au cœur de l'équipe de développement**.
- **Effectuer les tests au plus tôt** dans le processus de développement.
- **Mettre davantage d'énergie sur certains types de tests**.
- **Automatiser au maximum les tests pertinents**.

Des pratiques telles que l'automatisation des tests et l'usage des tests unitaires se sont largement démocratisées ces dernières années. Pourtant, beaucoup d'organisations ont des douleurs dans la mise en place de leur stratégie de tests, allant parfois jusqu'à abandonner les changements initiés.

Avant d'élaborer une stratégie, commençons par parcourir l'écosystème des tests.

84. Une Politique pour le Système d'Information, Chapitre Les Tests, OCTO Technology, 2006

85. Feedback : retour d'information

86. Outil de gestion de tests commercialisé par Hewlett Packard

87. Outil d'automatisation de tests de parcours utilisateur sur des applications web.

● Tester pour contrôler et pour construire

Tester est une activité complexe : tous les tests n'ont pas le même but ni la même granularité⁸⁸. Selon ce que l'on cherche à vérifier, différents acteurs, techniques, outils et environnements sont mis à contribution.

On peut distinguer **d'une part des tests métier qui servent à valider les fonctionnalités du produit, et d'autre part des tests techniques** qui valident le fonctionnement d'unités de code (tests unitaires), mais aussi la sécurité et la performance.

On différencie également ces tests selon qu'ils sont utilisés pour contrôler le logiciel ou pour guider sa construction. En effet, si les tests ont classiquement pour objectif de vérifier que la qualité du système est conforme à ce qui est attendu, ils peuvent aussi aider à le construire.

Pour explorer ces différents types de tests, nous nous appuierons sur le quadrant des tests agiles de Brian Marick⁸⁹ page suivante.

> Tester pour contrôler

La section Q1 est souvent, et à juste titre, le territoire des testeurs : ces tests permettent de s'assurer que le logiciel fourni correspond bien aux attentes.

Les **tests exploratoires** visent à être les plus exhaustifs et les plus imaginatifs possible pour que les défauts soient détectés avant la mise en production. Ils doivent permettre également d'assurer la non régression.

Les **tests d'acceptation utilisateur** visent, comme leur nom l'indique, à faire valider le produit par des utilisateurs, tandis que les **tests d'usabilité** permettent de s'assurer du niveau de l'expérience utilisateur.

Trouver le détail manquant ou le cas d'utilisation spécifique auquel le métier n'aurait pas pensé requiert une connaissance et un savoir-faire particulier. C'est la force du testeur expérimenté qui connaît son métier, son domaine fonctionnel et son application. Le développeur est d'une aide précieuse, mais peut manquer de recul lorsqu'il s'agit d'explorer le fonctionnement de l'application, et sa connaissance de l'implémentation sous-jacente peut introduire un biais.

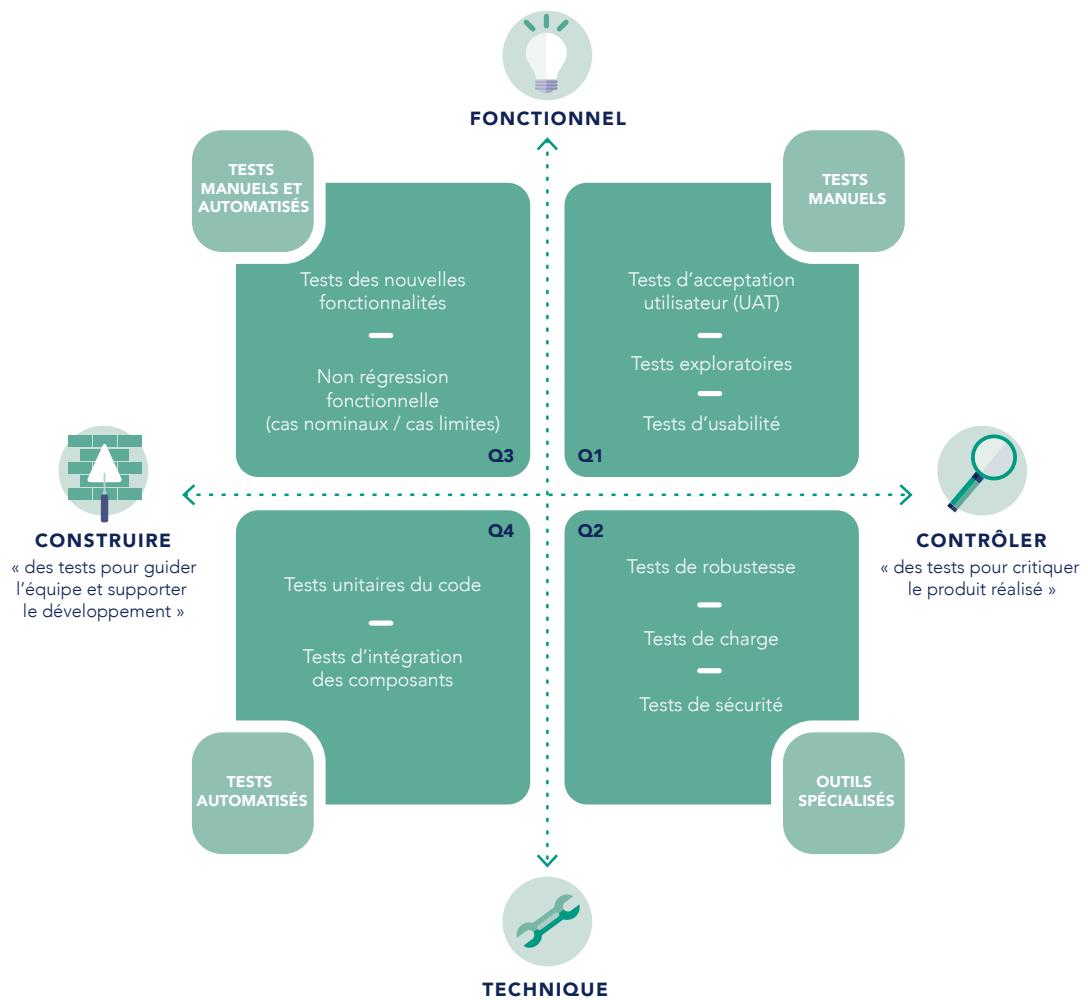
De même, les **tests techniques** en bas à droite (Q2), qui ont pour but de vérifier le système sur des aspects comme les **performances** ou la **sécurité**, requièrent une expertise forte et la connaissance des outils. Ces tests spécialisés peuvent être très consommateurs en temps et en ressources, et font appel à des connaissances parfois si pointues qu'ils nécessitent l'intervention d'experts (par exemple : les tests d'intrusion).

88. On parle de granularité des tests pour caractériser la proportion du système testé : les tests à granularité fine testent directement une portion de code (tests en « boîte blanche »), tandis que les tests à plus haute granularité testent le logiciel entier (tests en « boîte noire »)

89. Agile Testing Matrix, initialement proposée par Brian Marick (<http://www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1>) et reprise par Lisa Crispin dans Agile Testing: a practical guide for testers and agile teams <http://agiletester.ca/>



État de l'art des tests logiciels



Les principales caractéristiques de ces **tests de contrôle** (Q1 et Q2) sont les suivantes :

- Ils sont faits sur un environnement au plus proche de la réalité. Effectuer des tests de non régression ou de charge a peu de sens si une partie du système n'est pas connectée.
- Ils s'effectuent après la livraison du code.
- Ils s'effectuent au plus haut niveau de granularité, en considérant l'application comme une « boîte noire », c'est-à-dire que l'on n'agit que sur des points d'entrée accessibles de l'extérieur de l'application et non en effectuant des appels directs au code.

Les tests de contrôle ont bien entendu de la valeur, mais écrire l'intégralité des tests après les développements, c'est déjà trop tard. Le *feedback* intervient trop tardivement pour éviter des problèmes de qualité ou pour les corriger avec un coût réduit. Une première piste consiste donc à raccourcir la boucle de *feedback* en écrivant et en effectuant certains tests au plus tôt – avant et au cours de l'implémentation du logiciel.

> Tester pour construire

Là où les tests de la partie droite du quadrant permettent seulement de s'assurer de la qualité d'un incrément livrable du produit, **les tests de construction soutiennent et guident l'équipe dans les différentes étapes de la création du logiciel**. Ils sont conçus pour découvrir et décrire le fonctionnement de l'application à différents niveaux : de la fonctionnalité à laquelle

l'utilisateur aura accès, jusqu'au code qui va servir à l'implémenter. **Ces tests peuvent être vus comme l'échafaudage qui permet d'accompagner la construction du produit.**

Tests fonctionnels

Les tests fonctionnels de la partie en haut à gauche (Q3) répondent à la question : « Les développements sont-ils conformes à ce que nous voulions proposer à l'utilisateur ? ». Traditionnellement, ces tests sont rédigés et effectués par des équipes de testeurs après les développements, lors d'une phase de recette.

Nous privilégions une approche sensiblement différente : **les cas de tests sont rédigés avant les développements**, en même temps que la spécification des fonctionnalités. Cette approche permet de construire plus facilement un logiciel adapté au besoin et pas seulement un bon logiciel. Cela permet également de lever un maximum d'ambiguités et d'éviter les classiques débats : « Ce n'est pas ce qu'on avait convenu » ou « Ce n'est pas un bug mais une fonctionnalité », qui arrivent souvent bien trop tard.

La définition du test concerne toute l'équipe, intervenants métier, testeurs et développeurs, afin qu'elle ait une compréhension commune de ce qu'il faut réaliser.

On parle alors de **spécification par l'exemple** : si les fonctionnalités sont décrites par des *User Stories*, celles-ci sont accompagnées de critères d'acceptation ainsi que **d'exemples concrets qui serviront de scénarios de test**.

C'est ce que propose la méthode *Behavior Driven Development* (BDD).

Tests unitaires et tests d'intégration

Les tests techniques de la partie en bas à gauche du quadrant (Q4) répondent à la question : « Est-ce que cette partie du code se comporte comme je l'ai imaginée ? ». On y inclut **des tests unitaires et des tests d'intégration⁹⁰**, qui seront **toujours automatisés**. Ils aident le développeur à décrire, concevoir et implémenter le code. Le test unitaire s'effectue à une granularité très fine en vérifiant le comportement d'une portion de code totalement ou partiellement isolée de ses dépendances. Un test d'intégration a pour objectif de vérifier que plusieurs composants fonctionnent bien ensemble : il vérifie l'assemblage.

Cette granularité permet d'exécuter ces tests facilement, rapidement et donc très fréquemment. Faire participer un testeur ou un analyste métier à l'élaboration des tests unitaires n'a que peu de sens : ils concernent uniquement les développeurs. Néanmoins, la rédaction de ces tests gagne énormément à utiliser le vocabulaire du métier et des syntaxes lisibles.

Ces tests accompagnent l'équipe et surtout les développeurs tout au long de l'implémentation. Ils permettent de vérifier que l'équipe va dans la bonne direction et que le comportement de ce qui a déjà été réalisé n'est pas modifié au

fil des versions : **l'échafaudage entourant la construction devient alors progressivement une assurance, un harnais de tests garantissant la non régression**.

○ Où mettre l'énergie ?

Afin d'obtenir un *feedback* au plus tôt, il est nécessaire de mettre l'accent sur les tests de construction (à gauche du quadrant). Concernant les tests fonctionnels, il vaut mieux privilégier les tests spécifiés par l'exemple juste avant les développements à des tests de recette traditionnels. Il n'est pas question pour autant d'abandonner les tests de contrôle, qui seront choisis judicieusement et également effectués au plus tôt.

> Tester devient une pratique centrale de l'équipe

Pour qu'une stratégie de tests soit efficace, il faut aussi savoir qui met l'énergie sur quels tests. Les tests et leur automatisation doivent devenir une préoccupation centrale de l'équipe de développement. Ceci implique de s'organiser en conséquence, et d'abandonner certaines idées reçues :

Idée reçue n°1 : « En tant que développeur, je n'ai pas besoin de tester, il y a des tests de recette pour ça. Ce n'est pas mon boulot, il y a des testeurs qui font ça très bien. »

90. Le terme pouvant prêter à confusion, notez que nous utilisons le terme de test d'intégration pour des tests sur un système partiellement intégré, à la différence d'un test de bout en bout sur le système entièrement intégré.

BEHAVIOR DRIVEN DEVELOPMENT

La méthode *Behavior Driven Development*, littéralement « Développement Dirigé par le comportement », est une démarche de réalisation centrée sur la valeur métier, en rapprochant intervenants métier, testeurs et développeurs tout au long du processus de développement.

Un échange entre ces trois interlocuteurs, lors d'ateliers dédiés réalisés juste avant le développement, permet de partager le besoin en détaillant ensemble les scénarios exemples. Ces ateliers, familièrement appelés « ateliers 3 amigos », permettent de cerner ce qui est attendu, mais aussi d'identifier un maximum de cas en conjuguant différentes compétences.

Afin de faciliter ce rapprochement, la méthode BDD, souvent associée à la démarche *Domain Driven Design* évoquée dans le chapitre 5, s'appuie fortement sur un langage partagé par tous, jusque dans la formulation des tests unitaires et le code.

Notez bien que l'on parle d'**exemples concrets**. Si l'on cherche à définir une fonctionnalité de connexion à une application, l'exemple sera : « En tant qu'utilisateur, lorsque je saisit mon identifiant *paul* avec le mot de passe *12345* puis que je clique sur le bouton de connexion, je suis bien connecté et redirigé sur mon espace personnel », plutôt que « L'utilisateur peut se connecter avec son identifiant et mot de passe ».

On associe souvent la méthode BDD directement aux outils d'automatisation de tests fonctionnels, que nous évoquerons par la suite, mais attention au raccourci : **BDD doit avant tout reposer sur l'échange et des exemples concrets, sans lesquels les tests fonctionnels automatisés n'auront que peu de valeur ajoutée.**

Un bon testeur fait bien son métier, mais il y a plusieurs failles dans le raisonnement précédent :

- Lorsque l'on attend une phase de recette pour réaliser les premiers tests, le feedback est bien trop tardif et le coût de correction plus important qu'en ayant vérifié son code tout au long du développement pour détecter au plus tôt d'éventuels défauts⁹¹.
- C'est un véritable gâchis d'utiliser l'énergie d'un testeur pour découvrir des défauts évidents, alors qu'il pourrait par exemple se concentrer sur des tests exploratoires. C'est comme si un véhicule n'était testé qu'une fois assemblé, sans avoir vérifié la qualité des pièces détachées.
- Trouver tous les défauts en boîte noire est complexe. Les tests réalisés par un testeur se situent à un niveau de granularité élevé, généralement celui de l'application, ce qui implique beaucoup de complexité et une trop grande combinatoire pour tester tous les cas possibles.

Il est donc **de la responsabilité du développeur de tester aussi, en aidant à construire les scénarios d'exemples et en écrivant des tests automatisés**. Écrire de bons tests unitaires est une pratique qui ne s'acquiert pas en un jour. Nous recommandons particulièrement l'apprentissage de la méthode TDD (*Test Driven Development*), le meilleur moyen de réaliser un logiciel bien conçu et pleinement testé tout au long des développements en écrivant tests

automatisés et code dans le même souffle.

Idée reçue n°2 : « Les testeurs ne font pas partie de l'équipe de développement. »

C'est encore le cas dans de nombreuses entreprises et c'est un frein à l'efficacité des tests. L'idée de rassembler toutes les compétences au sein de l'équipe de réalisation du produit n'est pas récente et a été démocratisée par les méthodes agiles. **Être efficace dans la collaboration entre intervenants métier, développeurs et testeurs n'est réellement possible qu'en les rapprochant dans la même équipe**, et ce autant sur un plan organisationnel que géographique.

Idée reçue n°3 : « C'est du gâchis de tester trop tôt : il n'est pas possible de tester une fonctionnalité à peine développée, ou c'est prendre le risque de tout devoir tester à nouveau une fois celle-ci terminée. »

Être contraint d'attendre plusieurs semaines avant de pouvoir tester la première fonctionnalité n'est pas une fatalité. Commencer par des développements très techniques qu'on assemble tardivement n'est pas judicieux, par exemple commencer par mettre en œuvre toute la persistance des données. Il est préférable de se donner l'objectif de **parvenir dès que possible à un squelette opérationnel d'application, testé automatiquement**, qui sera raffiné au fil des développements. Cette approche a été popularisée par la démarche **Outside-in Test**

91. « Loi » du coût exponentiel des défauts, initialement énoncée par Capers Jones. *Software Assessments, Benchmarks, and Best Practices*, Addison-Wesley, 2000

Driven Development⁹², et s'appuie également sur l'utilisation de l'intégration continue dès le début des développements.

Adopter une stratégie efficace de tests automatisés peut impliquer un changement culturel profond : changement d'organisation, nouvelles activités et pratiques, approche différente dans le développement. Ce changement peut être long à mettre en œuvre et nécessite d'y mettre les moyens, comme le montre le retour d'expérience à ce sujet publié par Google⁹³.

> Que faut-il automatiser ?

Plus on teste tôt et fréquemment, plus les défauts trouvés sont faciles à corriger. Mais tester très fréquemment prend beaucoup de temps. Pour augmenter la fréquence, la question de l'automatisation se pose donc rapidement : les machines sont plus rapides, n'ont pas de problème avec le travail rébarbatif, et ne sont pas sujettes à l'erreur. Leur sous-traiter les tests est donc un excellent investissement.

L'outilage en lui-même n'est pas un problème : des outils existent aujourd'hui pour automatiser à peu près tous les tests possibles. Pour autant, **il n'est pas forcément judicieux de tout automatiser ni de s'appuyer principalement sur un seul type de test** parce qu'il est possible de l'automatiser. Quels tests sont réellement automatisables ? À quel prix, ou plutôt avec quel rendement ?

Avant de parler d'automatisation ou de répartition par type de test, introduisons les critères qui guideront le niveau d'investissement. Il ne faut négliger aucune catégorie mais concentrer son énergie sur les tests **rapides à exécuter, efficaces en termes de détection et de correction d'anomalie, peu coûteux à automatiser et à maintenir**.

Granularité des tests automatisés

Il existe une forte corrélation entre d'une part la granularité des tests automatisés et d'autre part leur complexité de mise en œuvre, leur maintenabilité et leur rapidité d'exécution.

En effet, si le code est correctement conçu et modulaire, alors un test unitaire automatisé à granularité très fine sera très rapide à exécuter. Il sera également simple à écrire et à maintenir. Il présente ces avantages pour une raison principale : le code testé est suffisamment isolé pour éviter tout effet de bord et ne pas devoir utiliser de dépendances externes au système – c'est-à-dire toute pièce logicielle extérieure au système testé qui présente un comportement non trivial et a généralement un effet de bord (base de données, serveur de mail, dépendance à la date, etc.). Ces tests sont écrits au plus près du code, en utilisant de librairies de tests de type xUnit⁹⁴.

Les tests d'intégration proposent un feedback sur des cas d'utilisation précis du système, dans des conditions se rapprochant des conditions réelles. Dans ces tests, on se permet cependant

92. La démarche *Outside In TDD* est détaillée dans *Growing Object Oriented Software Guided By Tests*, Steve Freeman et Nat Pryce, Addison Wesley, 2009.

93. *How Google Tests Software*, James A. Whittaker, Jason Arbon, Jeff Carollo, Addison Wesley 2012.

94. xUnit : librairie de tests unitaires. Des déclinaisons existent dans de nombreux langages (JUnit pour Java, NUnit pour .Net).

de s'abstraire de certaines dépendances ou de couches entières du logiciel. Par exemple, on va bouchonner⁹⁵ le système d'email et passer outre l'interface graphique. Ces tests représentent donc un niveau intermédiaire entre tests unitaires et tests d'interaction avec l'IHM ou de plus haut niveau. Ils utilisent généralement les mêmes outils que les tests unitaires.

Les tests fonctionnels permettant de valider certaines fonctionnalités peuvent également être automatisés. Lorsque c'est le cas, il est préférable de le faire à une granularité similaire aux tests d'intégration, en assemblant partiellement le système. On peut alors utiliser des outils spécifiques permettant de décrire les scénarios de tests, tels que *Cucumber*⁹⁶.

Les tests de bout en bout (au sens où le système est entièrement intégré) ou d'IHM⁹⁷ ne sont quant à eux pas toujours rentables en termes d'automatisation. Ils peuvent parfois sembler simples à mettre en œuvre, mais comme ils nécessitent un logiciel intégré en entier, ils sont souvent beaucoup plus lents à exécuter et impliquent un feedback plus tardif. Ils sont aussi très coûteux à maintenir sur le long terme car sensibles au moindre changement mineur et sans impact visible dans le logiciel (notamment l'IHM). Ceci provoque de nombreuses fausses alertes et peut conduire à la perte de confiance en ces tests et à leur abandon.

Par exemple, l'automatisation de tests fonctionnels au niveau de l'interface graphique

peut apparaître comme une solution complète car on teste toutes les couches du logiciel. Malheureusement, ces tests sont très sensibles aux modifications dues aux évolutions fréquentes des interfaces. Ils sont donc victimes de « faux négatifs » qui les rendent coûteux à maintenir et qui, à terme, risquent d'entraîner une perte de confiance en ces tests.

Effectuer des tests pour contrôler est vital pour s'assurer de la qualité, mais il est préférable de se concentrer sur des cas critiques dont le non fonctionnement pourrait porter atteinte à l'activité de l'organisation, et se reposer sur la compétence des testeurs pour trouver les défauts qui se cachent dans les méandres des cas d'utilisation.

L'idéal serait de pouvoir lancer les campagnes de tests le plus fréquemment possible. Malheureusement, plusieurs facteurs rendent ces tests difficiles à automatiser ou à exécuter régulièrement :

- **Les tests exploratoires** qui permettent de sonder l'application ainsi que les tests réalisés par l'utilisateur final reposent sur l'expérience humaine. Seuls des scénarios préétablis sont réellement « scriptables » et ne sont donc pas nécessairement pertinents en termes d'investissement.

- **Les tests spécifiques** tels que le contrôle de l'expérience utilisateur et l'accessibilité, les performances ou encore la sécurité requièrent souvent l'intervention d'un expert. Néanmoins,

95. On parle de bouchon, et de stubs et mocks en anglais, pour qualifier le remplacement d'une dépendance extérieure au système par une fausse implémentation, capable de simuler et vérifier les interactions.

96. Cucumber : outil de tests fonctionnels automatisés orienté Behavior Driven Development.

97. Notez ici que nous parlons de tests d'IHM de type parcours utilisateur automatisé et non de tests unitaires dans l'interface, par exemple en Javascript, dont nous vous recommandons chaudement l'usage (voir la refcard Test sur tous les fronts <http://www.octo.com/fr/publications/16-tests-sur-tous-les-fronts>)

des outils permettent d'automatiser une partie de ces tests, notamment pour détecter les erreurs les plus communes. Ces tests sont alors exécutés le plus tôt possible via l'intégration continue, par exemple pour vérifier l'évolution des performances⁹⁸ ou l'accessibilité⁹⁹.

Plus la granularité d'un test est fine, plus celui-ci sera rentable à automatiser : rapide, fiable, simple à mettre en œuvre et à maintenir.

Il est donc **judicieux de mettre l'accent sur les tests unitaires automatisés**, et de limiter l'automatisation des tests de plus haut niveau à ceux qui apportent réellement une valeur supplémentaire. Si on se réfère au quadrant vu en début de chapitre, il est donc préférable de **concentrer l'effort d'automatisation sur les tests de construction**.

Caractéristiques des différents types de tests automatisés

TYPE DE TEST AUTOMATISÉ	GRANULARITÉ	FEEDBACK	FACILITÉ D'IMPLÉMENTATION ET COÛT DE MAINTENANCE	EXEMPLE D'OUTILLAGE
Test unitaire	Très fine : fonction ou comportement d'une entité logique de code	Très rapide (quelques secondes)	Très simple : ils fonctionnent en isolation	xUnit
Test d'intégration Test fonctionnel	Partie de logiciel intégré	Rapide (quelques minutes)	Simple : on s'affranchit des contraintes majeures (certaines dépendances, IHM)	xUnit Cucumber
Test de bout en bout Test d'interaction IHM	Logiciel pleinement intégré	Lent (jusqu'à plusieurs dizaines de minutes)	Complexé (nécessité d'avoir l'environnement adéquat, fragilité)	Selenium
Test critique	Logiciel pleinement intégré	Lent (jusqu'à plusieurs heures / jours)	Très complexe : expertise nécessaire	Outils spécialisés

98. Gatling, outil de test de charge, <http://gatling.io/>

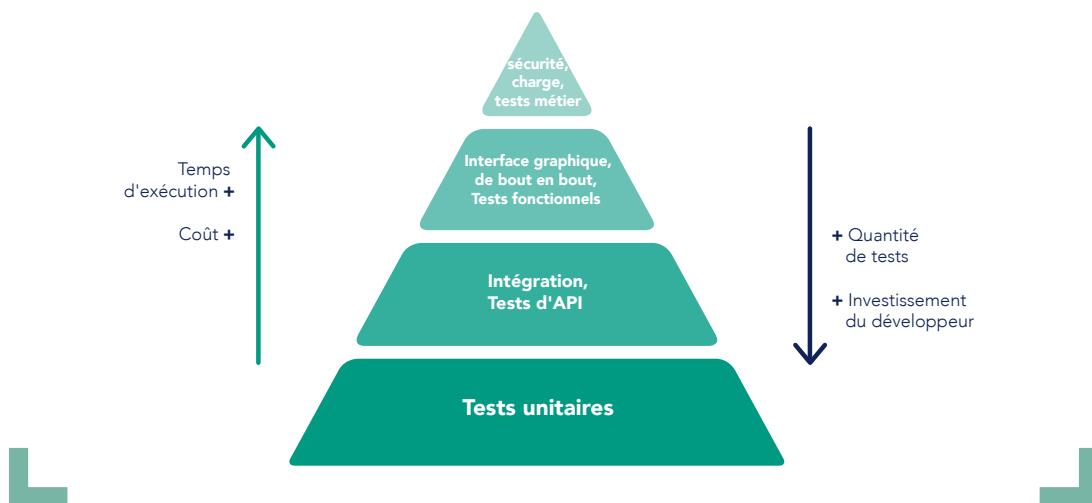
99. Tanaguru, outil de test d'accessibilité automatisé (<http://www.tanaguru.com/fr/>).

> Une stratégie de tests automatisés en pyramide

Nous avons les éléments permettant de bâtir une stratégie efficace d'automatisation des tests, classiquement représentée sous forme de pyramide :

- Une base solide constituée d'une multitude de tests unitaires assurant un feedback rapide, continu et précis.
- Un étage moins conséquent de tests d'intégration, notamment constitués de tests fonctionnels liés à la construction de l'application.
- Des tests de bout en bout couvrant uniquement les cas les plus critiques.
- Un sommet constitué de tests critiques (par exemple sécurité, charge) qui sont exécutés à la demande par des experts.
- Cette répartition des tests automatisés n'exclut en rien les tests manuels à forte valeur ajoutée tels que les tests exploratoires.

Stratégie des tests automatisés en pyramide



Application de la pyramide des tests sur un système d'information

Pour donner une idée de ce que peut donner cette stratégie, voici la répartition que nous avons sur un système d'information mature que nous développons avec l'un de nos clients depuis près de 10 ans. Ce système compte environ 350 000 lignes de code pour plus d'une vingtaine d'applications :

- Environ 5 000 tests unitaires automatisés représentant presque autant de lignes de code que l'applicatif.
- Plusieurs centaines de tests fonctionnels automatisés avec l'outil FitNesse¹⁰⁰.
- Une dizaine de tests techniques variés (IHM, charge, tests de bout en bout).

Cette stratégie de tests s'est plusieurs fois révélée **payante pour adresser des sujets critiques** sans trop de douleur : changer, **sans crainte**, l'intégralité des versions de langages et frameworks, faire évoluer un tiers du périmètre fonctionnel en un an, et tout simplement pouvoir livrer rapidement des nouvelles évolutions au quotidien en détectant rapidement la moindre régression.

Déetecter les anomalies grâce à une couverture optimale

Pour que l'ensemble des tests automatisés fournisse un harnais de sécurité efficace, il faut réussir à maximiser la couverture du code testé. La stratégie proposée précédemment permet d'optimiser cette couverture.

On pourrait dire qu'il est inutile de tester unitairement deux lignes de code qui ne sont pas complexes. Mais qu'en est-il lorsque l'on a mille fois deux lignes de code simple ? Cette économie est dangereuse. Prenons plutôt l'approche inverse : s'il est trivial de coder un composant, alors il est logiquement tout aussi trivial de le tester. La facilité d'implémentation des tests unitaires compense leur couverture très locale. Ils peuvent être multipliés aisément et leur fine granularité aide à trouver rapidement

l'origine d'une régression, permettant ainsi une correction rapide.

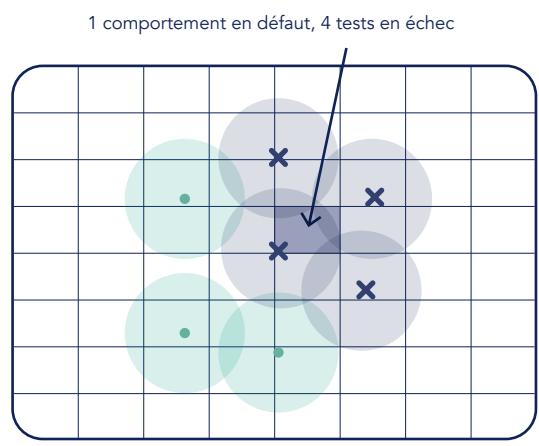
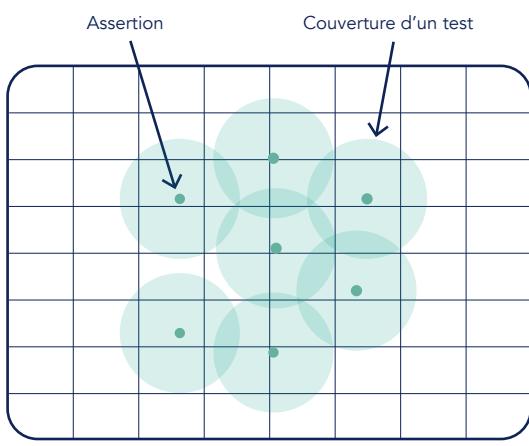
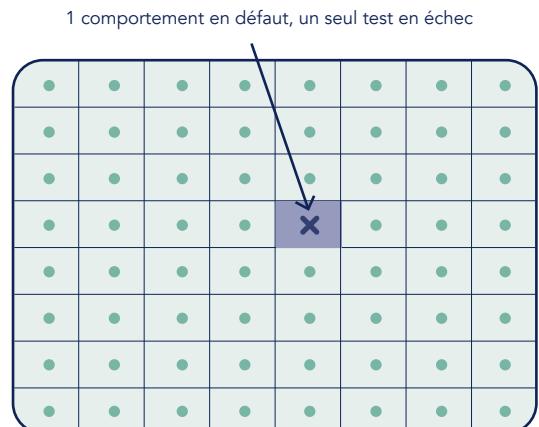
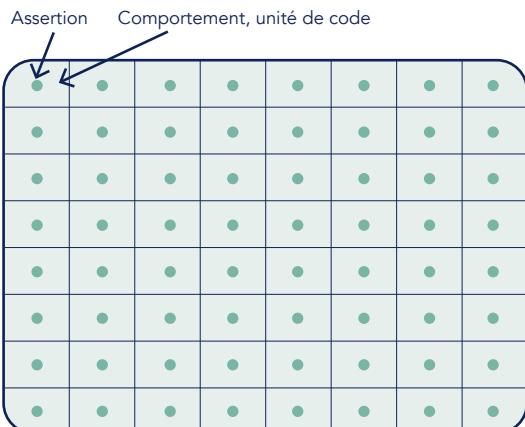
Les tests de plus haut niveau semblent couvrir une portion plus large du système. Cela est souvent vrai lors de l'exécution du code, mais ils se révèlent bien plus complexes en termes de couverture des cas possibles. De plus, il est impossible de restreindre chaque test à une couverture spécifique de code, ce qui implique un certain recouvrement entre les tests et met en valeur un gros défaut à ce type de tests : ils ont de nombreuses raisons d'échouer.

Généralement, on ne cherche donc pas à tout vérifier avec les tests automatisés de haut niveau. Rappelons que l'objectif des tests d'intégration est de vérifier comment les composants s'assemblent et non de vérifier tout le comportement.

100. FitNesse : plate-forme wiki permettant d'exécuter des tests d'acceptation. <http://www.fitnesse.org/>



Tests unitaires vs. tests d'intégration : couverture et localisation des régressions.



Il est néanmoins utile d'en avoir quelques-uns : leur large couverture d'exécution rend difficile à localiser l'origine précise d'une régression mais *a contrario* il sera plus facile de déterminer son impact selon le nombre de tests en échec.

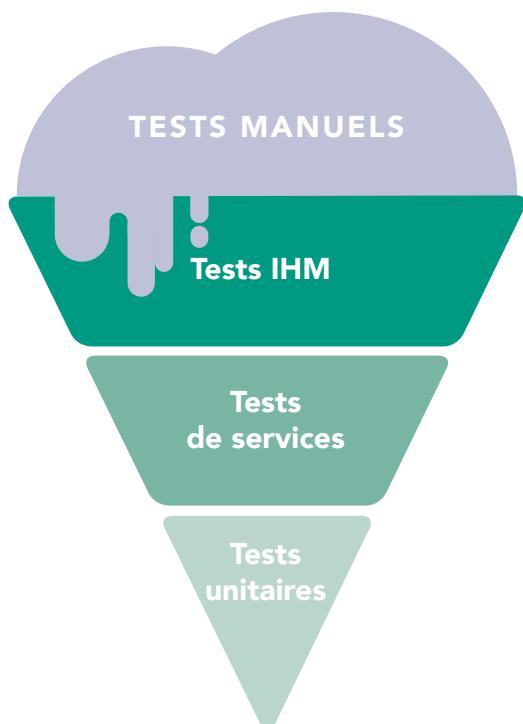
> *Les vaines promesses des tests pleinement intégrés*

Il n'est malheureusement pas rare de rencontrer des stratégies où l'automatisation de tests de bout en bout a été mise en avant, et où il y a peu, voire aucun test unitaire.

Comme nous venons de le voir, les tests pleinement intégrés donnent un **faux sentiment de sécurité** : un gros effort de mise en place n'offre au final qu'une faible couverture fonctionnelle¹⁰¹, et le feedback donné par les tests est beaucoup trop tardif pour être efficace. Résultat : les défauts sont détectés tard, bien longtemps après la fin du développement qui les a introduits, et le processus de correction n'est ni efficient ni efficace. **Notre pyramide, inversée, oscille alors en équilibre instable sur son sommet.**

L'argument souvent mis en avant pour justifier l'utilisation de ces tests intégrés est qu'ils sont plus faciles à réaliser. Considérer le logiciel comme une boîte noire en ne se préoccupant que de ses entrées et sorties devrait permettre d'en ignorer la complexité interne...

Et même si on peut éventuellement ignorer la complexité interne du logiciel, il devient vite



Anti-pattern : répartition des tests « Ice Cream Cone »

difficile d'ignorer celle induite par un logiciel à intégrer dans un système d'information complet : il faut s'assurer que les applications tierces soient disponibles dans la bonne version, monter d'énormes jeux de données jusqu'à l'autre extrémité du SI, avoir un environnement dédié, etc.

101. *Integrated tests are a scam*, Joseph B. Rainsberger, 2010 <http://blog.thecodewhisperer.com/permalink/integrated-tests-are-a-scam.html>

Cette stratégie risquée est souvent choisie lorsque l'équipe de développement n'est pas rodée à l'utilisation des tests unitaires, d'autant plus si le code existant est un code legacy dont les mauvais choix de conception ont rendu le code difficilement testable unitairement.

Alors comment s'en sortir lorsque l'on est dans cette situation ?

Revenir à une répartition des tests plus efficace prendra du temps, mais ce n'est pas impossible. Ceci ne fonctionnera que si l'on cherche à corriger les causes profondes plutôt que les symptômes :

- Avant toute chose, il faut mettre les moyens pour que l'équipe se forme aux tests unitaires et à *Test Driven Development*. L'utilisation systématique de TDD sur le nouveau code permettra par définition d'avoir un code testé unitairement et de renverser peu à peu la répartition des tests.
- Même sur du vieux code, il existe des techniques permettant de rendre progressivement le code testable^{102 103} : mais bien travailler sur du code legacy est une compétence qui s'acquierte.

Code Legacy : un cauchemar pour tester unitairement ?

Le terme de « Code Legacy » est un anglicisme couramment utilisé dans le monde du développement informatique : littéralement, il s'agit du code dont nous héritons. Et qu'entend-on par code hérité ? Une définition¹⁰⁴ nous dit qu'il s'agit généralement de code qui n'est plus supporté ou plus maintenu.

*Nous prenons une définition qui peut sembler plus radicale : tout code qui n'est pas couvert par des tests automatisés est du Legacy, ainsi que le définit Michael Feathers dans *Working Effectively With Legacy Code*¹⁰⁵. Il s'agit de code complexe et risqué à faire évoluer car il n'y a pas de harnais de non-régression.*

Souvent, ce code souffre également de défauts de conception : code très complexe, fonctions extrêmement longues, et surtout des dépendances inextricables. C'est ce dernier point qui est le plus souvent problématique pour poser des tests unitaires, car le code à tester possède des dépendances extérieures qui peuvent sembler impossibles à remplacer en contexte de test. La pratique des tests unitaires s'appuie sur les propriétés d'une conception modulaire minimisant le couplage entre les unités de code. Souvent, l'usage de singleton global, qui induit un état global partagé ou de dépendances instanciées directement dans la fonction, fait qu'il est nécessaire de remanier le code avant de poser le premier test.

102. Du Legacy au Cloud en moins d'une heure - David Gageot, USI 2013 <https://www.youtube.com/watch?v=q11gydDAMSo>

103. Beyond Legacy Code, David Scott Bernstein, Pragmatic Bookshelf, 2015

The Mikado Method, Ola Ellnestam et Daniel Brolund, Manning, 2014

104. Code hérité, https://fr.wiktionary.org/wiki/code_h%C3%A9rit%C3%A9

105. Working Effectively with Legacy Code, Micheal Feathers, Prentice Hall, 2004

Ø Points à retenir

Une stratégie efficace de tests s'appuie sur deux éléments indispensables qui permettent d'avoir confiance en ses tests et d'être serein tout au long du projet – plutôt que d'avoir peur d'une phase dédiée qui intervient trop tard. Pour cela, il est indispensable que les tests offrent un **feedback rapide et en continu, et permettent d'être efficace dans la correction des défauts détectés.**

Les tests deviennent une pratique centrale de l'équipe de réalisation du produit, qui peut impliquer un changement de culture, d'organisation et de nouvelles compétences à acquérir :

- Les tests fonctionnels sont écrits avant les développements, collectivement, par les intervenants métier, les testeurs et les développeurs.
- Les développeurs testent aussi, en s'appuyant sur les tests unitaires automatisés comme un moyen de réaliser le bon produit.

L'effort d'automatisation des tests est concentré sur l'écriture des tests de construction avec de nombreux tests unitaires plutôt que des tests de haut niveau peu rentables.

Mettre l'accent sur l'automatisation des tests de construction permet aussi de limiter l'effort sur les tests de contrôle, et de concentrer l'énergie des testeurs là où ils apportent le plus de valeur.

Ø Par où commencer ?

- Organisez un atelier 3 amigos pour initier une démarche de spécification par l'exemple et rapprocher métier, testeurs et développeurs.
- Si vous avez déjà des tests automatisés, dressez une rapide cartographie de la répartition : cela ressemble-t-il à une pyramide ? Si ce n'est pas le cas, identifiez les tests de haut niveau les plus instables pour élaborer une démarche permettant de revenir à une couverture par des tests unitaires.
- Les développeurs de l'équipe sont-ils formés à l'écriture de bons tests unitaires ? À *Test Driven Development* ? Si non, envisagez une formation, et dédiez des sessions de *pair programming* à cet apprentissage.

TEST DRIVEN DEVELOPMENT



Test Driven Development

Par Abel André

Test Driven Development (TDD) est une méthode de développement mettant l'accent sur l'écriture de tests automatisés en tant qu'outil servant à guider l'implémentation des fonctionnalités. Cette méthode a été popularisée par Kent Beck en 2002¹⁰⁶, elle est au cœur de la méthode eXtreme Programming. Nous avons la conviction que cette démarche est aujourd'hui **une pratique fondamentale du développement logiciel**.

Le principe de base de TDD est le même que pour l'agilité ou la revue de code : renforcer la qualité des réalisations grâce à des boucles de rétroaction. Et tout comme pour ces autres pratiques, cela permet d'avancer sereinement et avec confiance. **Il n'y a plus de murs à franchir, juste des marches.**

TDD n'est pas une méthode de test, c'est un moyen d'avoir du feedback instantanément lors du développement. Vu comme cela, l'obtention d'une couverture de code¹⁰⁷ plus que raisonnable est donc une conséquence naturelle de la pratique de TDD, et non une fin en soi. La pratique de cette méthode porte ses

fruits à partir du moment où elle est devenue un automatisme, une habitude, et qu'elle peut ainsi être exercée avec facilité et sans effort particulier d'attention. C'est comme le vélo : une fois que l'on a appris à en faire, on saura toujours en refaire.

○ Le cycle TDD

TDD est une méthode de développement itérative et incrémentale utilisable au quotidien. Itérative, car il s'agit de répéter un cycle un certain nombre de fois. Incrémentale, car il s'agit d'enrichir une unité logique de code à chaque cycle avec de nouveaux comportements.

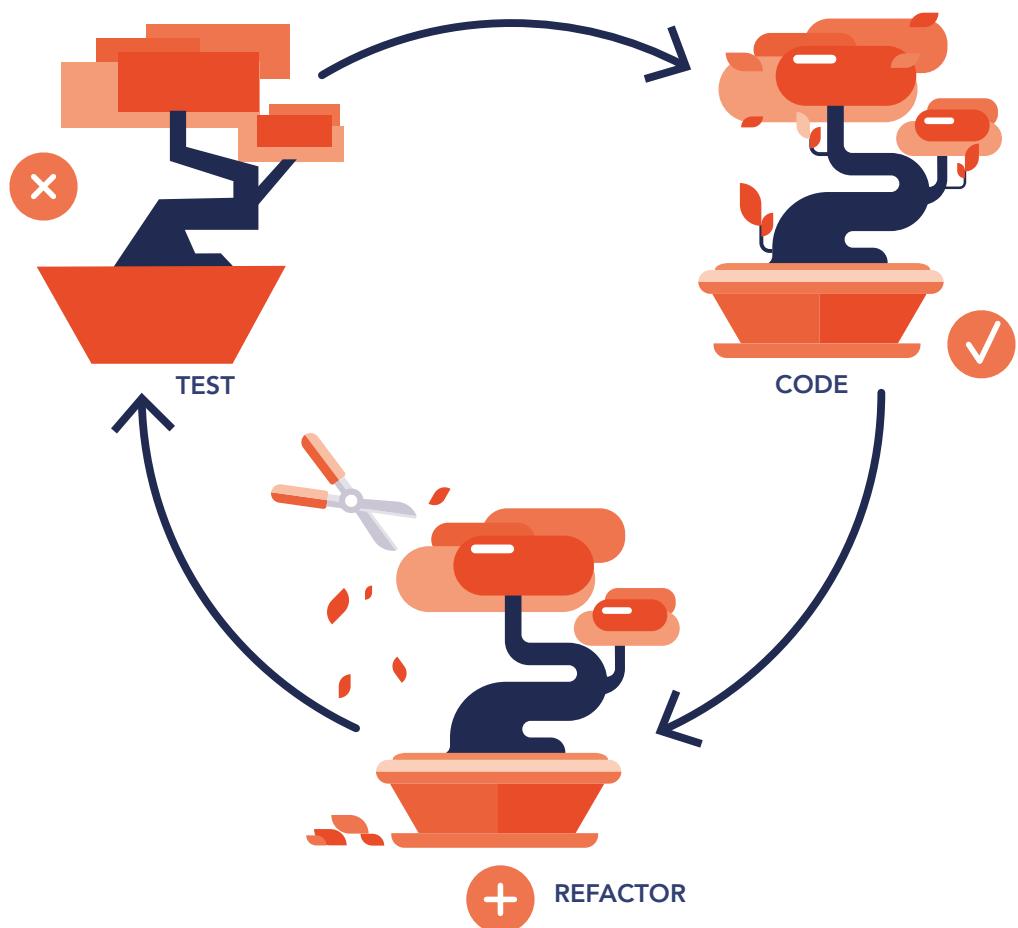
On peut résumer le cycle de TDD ainsi : **Test, Code, Refactor** (ou Red, Green, Refactor).

NB : dans la suite du chapitre, on utilisera le terme de « fonction » pour désigner une unité logique de code de taille restreinte, nommable et exécutable.

106. *Test-Driven Development: By Example*, Kent Beck, Addison-Wesley 2002

107. Couverture de code : proportion de code couverte par des tests, c'est-à-dire qui est parcourue lorsque l'on exécute les tests de l'application.

Le cycle TDD : Test, Code, Refactor



TEST : écrire un test en échec**CODE : écrire le code qui fait passer le test**

La première phase permet au développeur de **spécifier le comportement** qu'il souhaite implémenter. Elle consiste donc à écrire un test destiné à échouer. Il est crucial que ce test soit en échec (rouge) lors de sa première exécution : cela indique au développeur que le dit comportement n'est pas encore implémenté. En effet, si le test est en succès (vert) à sa première exécution, cela peut signifier :

1. Qu'il teste un comportement déjà présent.
2. Qu'il ne comporte pas les bonnes assertions.
3. Qu'il ne s'exécute pas dans le bon contexte.
4. Qu'il n'exécute pas la fonction voulue.

Dans tous les cas, il ne fournit alors pas au développeur d'objectif à atteindre et rien ne permet d'affirmer que c'est l'écriture de l'implémentation qui a fait passer le test.

Dans cette phase, le développeur cherche **l'implémentation la plus simple** pour faire passer le test. Même si cette implémentation est triviale, comme par exemple retourner une valeur en dur. Il ne s'agit pas de faire le meilleur code possible, mais d'implémenter la fonctionnalité attendue par le test et de revenir à la « barre verte » le plus rapidement possible, c'est-à-dire dans un état où le test écrit précédemment ainsi que tous les autres existants passent.

REFACTOR : améliorer le design



La phase de *refactoring* conclut le cycle de TDD : une fois l'intention décrite dans un test et réalisée dans le code testé, le développeur examine son code en se demandant comment il peut en **améliorer la qualité et le design**, par exemple en appliquant les règles d'un Design Simple (voir le chapitre 5 : Écrire du code compréhensible).

Si une amélioration est possible, elle pourra être faite en toute sécurité, puisque les tests écrits précédemment sont là pour indiquer au développeur une éventuelle régression. Bien qu'un *refactoring* ne soit pas requis à chaque cycle, il est toutefois indispensable d'examiner le code et de se poser la question : comment améliorer le code ?

... REPEAT

Une fois le cycle de TDD terminé, on recommence : décrire une intention avec un nouveau test, écrire l'implémentation, remanier le code si nécessaire. Plus le cycle est court – de l'ordre de quelques minutes – plus la quantité de tests augmente rapidement et plus le *feedback* apporté par ces tests est fréquent et riche en informations :

- Les tests passent-ils toujours (alerte immédiate en cas de régression) ?
- Le code est-il lisible (cf. REFACTOR) ?
- Les tests sont-ils faciles à écrire ? À faire passer ?
- Les tests décrivent-ils bien l'usage de la fonction testée ?

Quelques règles simples :

- Commencez toujours par un test automatisé qui échoue.
- N'ajoutez jamais de tests si un test échoue déjà.
- Éliminez toute duplication.
- Soignez la qualité du code de test autant que le code de l'implémentation.

○ Comment appliquer la méthode ?

> *Fake it until you make it*

Au cours des premiers cycles de TDD, quand la fonction à tester et la suite de tests sont vides, il n'y a pas vraiment de possibilités de refactoring ou d'enrichissement du code. Le principe *Fake it until you make it* est alors appliqué. Le développeur fera passer ses premiers tests aux intentions simples en écrivant des **implémentations basiques** : par exemple, des clauses *return* avec des valeurs en dur, ou encore des clauses conditionnelles sans leur alternative.

Cela peut paraître contre-intuitif : pourquoi écrire du code stupide si l'on sait pertinemment qu'il ne sera pas conservé une fois la fonctionnalité terminée ? La raison est que cela permet de faciliter l'écriture des premiers tests – donc des intentions les plus importantes du code – avec **une implémentation basique qui fera peu à peu apparaître l'implémentation dite évidente** de la fonction. On tourne autour de cette implémentation évidente, on **triangule** jusqu'à ne plus avoir d'autre choix pour avancer que de l'appliquer.

Cela se fera alors en toute sécurité puisque le harnais de tests est présent et s'assure pour le développeur de la non régression.

Fake it until you Make it

Vous avez écrit un test. Vous n'avez pas en tête une implémentation simple. Dans le code de production, faites passer le test en renvoyant la valeur attendue par le test.

Triangulate

*Vous avez un premier test grâce à un *Fake It*. Vous souhaitez avancer pas à pas :*

- *Écrivez un deuxième test.*
- *Écrivez le code le plus simple qui fasse passer ce test sans casser le premier.*

Obvious implementation

Vous avez triangulé une implémentation basique grâce aux deux premiers tests :

- *Écrivez un troisième test simple.*
- *Écrivez l'implémentation évidente du code qui fasse passer le dernier test sans casser les deux premiers.*

Réfléchir avant d'agir

Un écueil commun lors de l'utilisation de TDD est de penser que les cas de test ne doivent être trouvés qu'un par un, sans objectif à moyen terme, afin de permettre un design émergent. En réalité, la méthode préconise simplement de respecter la succession des cycles *red, green, refactor*. **Pour guider son travail en TDD, le développeur peut dresser la liste des comportements auxquels devra répondre la fonction** sous forme de *to-do list* qu'il s'efforcera de respecter au cours des cycles.

Il faut par contre à tout prix éviter d'écrire et d'essayer de faire passer au vert plusieurs tests en même temps : cela a tendance à éloigner le développeur de la solution nécessaire et suffisante. De plus, cela induit généralement des phases de refactoring trop longues ou trop complexes.

On part ici du principe qu'un système étendu complexe est composé d'un ensemble de sous-systèmes restreints simples. S'attaquer à ces derniers est plus facile puisque cela demande de concentrer son attention sur un nombre limité d'informations. De la même façon qu'on marche un pas après l'autre, on code avec TDD un test après l'autre.

Test List

Que devriez-vous tester ?

- Avant de commencer, faites une liste de tous les tests que vous aurez à écrire.
- Pendant la session, lorsqu'une idée arrive, ajoutez un test à votre liste et continuez de programmer.

> À quel périmètre appliquer TDD ?

Il n'existe pas de règle formelle quant aux types de tests automatisés à utiliser avec TDD ni sur l'étendue des unités logiques de code à tester. Cependant, si l'on reprend le modèle de pyramide des tests automatisés décrit dans

le chapitre précédent « Une stratégie efficace de tests automatisés », on s'aperçoit que plus les tests sont bas dans la pyramide :

- plus l'origine d'une régression est précise,
- plus leur temps d'exécution est court,
- plus le cycle de TDD est court.

Grâce à ce feedback rapide et peu coûteux, le développeur a l'occasion d'appliquer du refactoring plus souvent. Il reste également concentré sur son travail sans perdre le fil de ses pensées : le changement de contexte occasionné par un feedback trop long a un impact important sur la productivité.

Il est donc judicieux d'**utiliser TDD au niveau le plus unitaire possible** : en appelant directement une fonction ou méthode. C'est cette démarche que nous conseillons et mettons en avant lors de la fabrication de nos produits.

Tests FIRST

Les tests ne doivent jamais s'affecter mutuellement.

Caractéristiques FIRST d'un test en TDD :

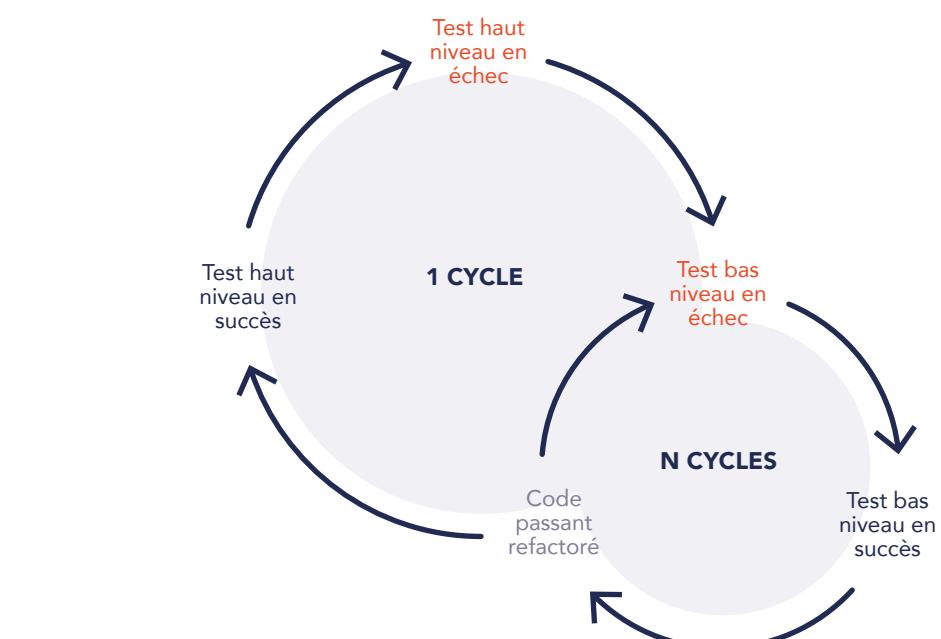
- **First** : le test est écrit en premier
- **Independent** : le test s'exécute indépendamment des autres
- **Repeatable** : le test produit le même résultat à chaque appel
- **Simple** : le test met en œuvre le minimum de données
- **self-Testing** : le test est automatisé

Néanmoins, faire le choix de *Test Driven Development* n'est pas exclusif. Si on l'applique aux tests unitaires, il devient possible de l'associer à des tests de plus haut niveau dans la pyramide. De cette manière, en écrivant un test fonctionnel ou d'intégration qui sera en échec à sa première exécution, on peut progresser à un niveau plus fin avec plusieurs cycles de TDD jusqu'à ce que le test initial passe. On parlera alors de **double boucle TDD** : un cycle *red*

– *green* qui encapsule plusieurs cycles *red* – *green* – *refactor*.

L'identification des cas fonctionnels testables à un plus haut niveau, afin de démarrer la double boucle, est réalisable grâce à la spécification par l'exemple. Pour cela on peut s'appuyer sur des approches telles que *Behavior Driven Development* (BDD) ou encore *Acceptance Test Driven Development* (ATDD).

Double boucle TDD, avec N cycles de TDD pour 1 cycle de plus haut niveau



> *L'importance des suites de tests*

En plus de fournir un guide pour l'implémentation des fonctionnalités en faisant émerger progressivement un design de qualité, un des apports majeurs de TDD est de fournir un harnais de tests qui constitue un atout dans la maintenabilité du code de production. Ces tests, par leurs noms et assertions, servent à dresser la liste exhaustive des intentions et des comportements du code : ils sont **une spécification technique détaillée, automatisée et exécutable**, à l'usage des futurs développeurs ayant à faire évoluer le produit. L'enjeu est de garder la maîtrise du code en le sécurisant, ce qui encourage son évolution.

Ainsi, la phase de **refactoring s'applique aussi à la suite de tests**. Le développeur doit prendre en compte plusieurs points d'attention : avoir un nommage cohérent des cas de tests ; une factorisation des données nécessaire à l'exécution des tests dans des fonctions d'initialisation ; un regroupement des tests appartenant aux mêmes cas fonctionnels dans des suites de tests correctement nommées.

Ø **Les freins à l'adoption de la méthode**

> *La nécessité de se former*

Nombreux sont les développeurs, managers et chefs de projet qui refusent d'adopter TDD en raison du temps supplémentaire pris sur les projets. Pour les praticiens confirmés, **ce surcoût n'existe pas ; c'est l'adoption de la méthode qui provoque une diminution temporaire de l'efficacité de l'équipe de développement, et non son usage au quotidien**.

Comme toute méthode ou outil, il y a une courbe d'apprentissage à franchir avant d'être efficient dans son utilisation. Une fois la méthode maîtrisée et le geste et les réflexes acquis, la méthode permet de développer à un rythme soutenu et durable :

- parce que le code existant est déjà testé, donc il est plus facile de le faire évoluer en ayant l'assurance de détecter les régressions avant le déploiement,
- parce qu'en répétant des cycles de TDD, on porte toute son attention sur des problèmes simples que l'on peut résoudre mentalement avec rapidité.

Il sera toujours plus coûteux de passer du temps à chercher l'origine des régressions dans un logiciel déployé en production. Tout temps utilisé avant le déploiement pour prévenir des défauts est du temps investi judicieusement.

Il n'y a en réalité **qu'une seule et unique manière d'intégrer TDD dans notre manière de développer : par la pratique.** Apprendre en recevant une formation est possible, pour peu que celle-ci soit suffisamment longue. Mais puisqu'il s'agit d'un geste, d'une technique effectuée par l'artisan, rien ne saurait remplacer la pratique à répétition pour transformer ce geste en automatisme. Les développeurs qui le souhaitent doivent pouvoir investir du temps dans l'apprentissage et bénéficier d'un accompagnement dans la pratique. **L'apprentissage de TDD demande de la rigueur et un renversement du mode de pensée habituel.**

> *Les difficultés avec du code legacy*

Un avantage apporté naturellement par la méthode est de produire du code testable, puisque créé pour répondre à des tests. À l'inverse, du code *legacy* – par définition non couvert par des tests automatisés¹⁰⁸ – peut présenter des difficultés majeures lorsqu'il s'agit de le faire évoluer en TDD. Poser le premier test fera apparaître les problèmes de design : dépendances statiques, effets de bords imprévus, couplage fort, etc.

Plusieurs approches peuvent être utilisées pour faire évoluer un tel code :

- isoler les règles de gestion dans des unités logiques de code dédiées pour les faire évoluer plus facilement,

- se défaire d'un couplage fort grâce à des proxies¹⁰⁹ que l'on pourra bouchonner,
- poser un premier test de plus haut niveau afin de s'aider sans risque majeur.

Le paradoxe de cette situation est que :

- pour faire évoluer du code *legacy* en tout sécurité, il faut des tests,
- mais pour ajouter des tests, il faut faire évoluer le code.

Il faut souvent faire évoluer le code parce que certains choix de design rendent le code difficile à tester de façon unitaire et automatisée. Le plus souvent, il s'agit de couplage fort avec des dépendances indésirables : méthodes statiques, accès à une base de données ou au système de fichiers, etc.

Pour lever ce paradoxe, il faut donc pouvoir gérer ces dépendances, en faisant évoluer le code sans test mais de façon sécurisée, uniquement dans le but de le rendre testable. Pour cela, il existe un certain nombre de techniques, qui sont décrites notamment dans le livre *Working Effectively with Legacy Code* de M. Feathers.

> *La prédominance des habitudes*

Il est fréquent que les pratiques diffèrent légèrement d'une personne à l'autre au sein d'une équipe de développement. Si pratiquer ou non TDD est un choix que chacun

108. Voir la définition introduite dans le chapitre précédent.

109. Composant intermédiaire

fait, convaincre ses collègues des bienfaits de la méthode peut s'avérer compliqué. Malheureusement, les habitudes ont une fâcheuse tendance à s'enraciner dans nos raisonnements, et les années passant, elles sont souvent difficiles à remettre en question.

L'habitude la plus tenace est celle qui consiste à penser à l'implémentation avant de penser aux différentes intentions du code. Et même lorsque l'on arrive à faire abstraction des implémentations qui semblent évidentes en rédigeant des tests, c'est faire l'aller-retour de cycle en cycle entre implémentation et intention qui est bloquant. Ce renversement de mode de pensée est une gymnastique intellectuelle qui peut prendre du temps à intégrer.

Pour entraîner son cerveau à fonctionner de la sorte, la meilleure approche reste la pratique, et plus particulièrement le **travail en pair programming**. Voici un exercice intéressant, que nous appelons communément le « ping-pong » :

1. Le développeur A écrit un test rouge.
2. Le développeur B fait passer ce test au vert et refactor si nécessaire.
3. Le développeur B écrit un test rouge.
4. Le développeur A fait passer ce test au vert et refactor si nécessaire.
5. Le développeur A écrit un test rouge.
6. etc.

Cet exercice permet à tour de rôle de s'astreindre à uniquement réfléchir au test, à l'intention du code, tout en sachant que l'implémentation sera écrite par son binôme. Ces sessions sont toujours intéressantes car en se faisant des « passes », les participants échangent sur leurs manières de tester et d'implémenter. Cela leur permet également de partager leurs idées en plus de s'aider l'un et l'autre à faire évoluer leur mode de pensée.

⌚ Les bénéfices de TDD

Un des bénéfices de la méthode est son **impact positif sur la productivité** : le feedback rapide et fréquent qu'elle offre permet au développeur de maintenir sa concentration et de garder le fil de sa pensée. Si le feedback pointe vers des régressions, celles-ci sont corrigées dès la phase de construction, ce qui réduit le coût et le temps perdu à corriger des défauts détectés en recette ou en production, et permet d'investir plus de temps dans la création de valeur.

La **nature itérative de TDD** pousse le développeur à **se poser des questions sur les comportements que doit offrir la fonction testée** : il peut donc au cours de la session penser à de nouveaux cas à implémenter.

De plus, puisque le **refactoring** est rendu possible et accessible à chaque cycle, le code a une tendance naturelle à devenir plus lisible et plus propre. Sa maintenabilité en est donc

améliorée ce qui contribue à la **productivité des futurs développeurs**.

Enfin, la documentation apportée par le code le rend plus facilement appréciable pour un autre développeur, et contribue à la productivité future.

Ces différents bénéfices permettent de dégager un **bon retour sur investissement de TDD**, même si celui-ci est souvent complexe à mesurer¹¹⁰ : si on peut constater une **légère augmentation du temps passé en développement** – que nous évaluons empiriquement à 20 % environ – il y a un **gain manifeste lorsque l'on considère l'ensemble des étapes de réalisation du produit**. Celui-ci s'explique par une forte amélioration de la qualité du code et une réduction conséquente du temps passé à tester en recette et à corriger les défauts. Cette étude¹¹¹ réalisée auprès d'équipes de Microsoft et IBM montre par exemple une réduction du nombre de défauts allant de 40 à 90 % pour une augmentation du temps de développement (et non le temps global) entre 15 et 35 %.

110. Le ROI de TDD, blog Octo Technology, 2008 <http://blog.octo.com/le-roi-du-tdd/>

111. Realizing quality improvement through test driven development: results and experiences of four industrial teams, Nachiappan Nagappan, 2008. http://research.microsoft.com/en-us/groups/ese/nagappan_tdd.pdf

➊ Points à retenir

Test Driven Development est une **méthode de développement itérative et incrémentale**, et non une méthode de test. Elle fournit un guide en trois étapes, s'appuyant sur l'utilisation de tests unitaires automatisés, qui permet d'intégrer en continu une spécification fine du comportement du code et la construction d'un design incrémental. L'utilisation de cycles courts est nécessaire pour obtenir un **feedback fréquent** sur les développements réalisés.

L'approche de TDD représente souvent un changement dans la manière de penser, et nécessite donc un temps de formation et de la pratique pour transformer le geste en automatisme.

TDD offre un **retour sur investissement largement positif** avec de nombreux bénéfices dans le code réalisé :

- Le code est naturellement testable et maintenable.
- Les tests produits sont une spécification détaillée des comportements du code.
- Le développeur a une maîtrise accrue de son code et est encouragé à le remanier.

➋ Par où commencer ?

- Si vous êtes novice avec TDD, une formation et l'accompagnement par un développeur qui maîtrise la méthode seront vraiment nécessaires pour franchir le cap.
- Avant de réaliser cet investissement, vous pouvez vous rendre à l'un des nombreux *Coding Dojos* organisés par la communauté, où l'on pratique souvent des exercices¹¹² pour s'améliorer en TDD.
- Utiliser TDD sur du code *legacy* est complexe et risque fortement de vous décourager si vous n'avez pas l'habitude : commencez par l'expérimenter sur du nouveau code de préférence et sur un périmètre restreint.

112. Catalogue de katas, des exercices de programmation : <http://codingdojo.org/cgi-bin/index.pl?KataCatalogue>

RÉCIT D'UN APPRENTISSAGE PAR LA PRATIQUE



Récit d'un apprentissage par la pratique

Par Julien Tellier

Nous l'avons évoqué plusieurs fois au cours de cet ouvrage, la culture et les pratiques Software Craftsmanship reposent sur une dynamique d'apprentissage permanent. Pour l'illustrer, nous avons donc voulu terminer ce livre en donnant la parole à Julien Tellier, développeur chez OCTO Technology, qui a souhaité partager l'expérience de son apprentissage de cette culture.

Ceci n'est pas un *howto*.
Ce n'est pas non plus un *how-not-to*.
C'est du vécu.

Je vais vous évoquer mon ressenti, de l'arrivée dans mon premier projet jusqu'à sa fin. Je vous parle de tout ce qui a pu me marquer, me donner envie d'adopter l'état d'esprit d'un Craftsman, et surtout de ce que chaque étape m'a apporté. Pourquoi ? Parce qu'il n'y a pas qu'une manière, ou contexte pour y parvenir, mais qu'en revanche n'importe quelle expérience peut inspirer la vôtre (et ça aussi, c'est du vécu).

Ø Premier contact

Premier jour. Damien se présente, il va faire partie de notre équipe de 3 développeurs. Il me raconte ce qui l'intéresse, ses convictions en tant que Craftsman, les méthodes telles que...



- TDD ? C'est quoi ? Honnêtement, je n'en ai jamais entendu parler, je lui avoue sans honte.
- Alors, TDD c'est une méthode qui guide le développement de ton application par l'écriture de tests unitaires, me répond-il. Ça protège ton code au fil du développement et ça le documente aussi !
- Ah ça m'intéresse, tiens.
- C'est bien, parce qu'on ne fera pas le projet autrement ! On se réserve l'après-midi et on fera un randori¹¹³ basique. On va coder un Puissance 4, tu verras c'est plutôt sympa.
- Cool !

> L'après-midi, on sort les crayons ...



- Le but de l'exercice, c'est de développer un Puissance 4, m'explique Damien. On va y aller par étapes. D'abord, on a besoin d'une grille.
 - Un tableau de valeurs ?
 - Oui, pourquoi pas. Mais ne t'embête pas tout de suite avec ces détails-là. Quels sont les tests que tu ferais sur cette grille ? Que ferais-tu pour vérifier son bon fonctionnement ? Vas-y, lance des idées, on verra à la fin celles qui se recoupent et celles qu'on aurait oubliées.
- Ça n'a pas besoin d'être parfait du premier coup.

113. Randori : Format de coding dojo dans lequel les participants prennent tour à tour le clavier.

Après avoir proposé de vérifier que la grille fait la bonne taille, que l'on ne met pas de pion au même endroit qu'un autre, ou que le pion envoyé n'est pas nul, il me réoriente. La grille est-elle vide au début du jeu ? L'ajout d'un jeton se fait-il bien au-dessus d'un autre ? L'ajout est-il bien impossible au-delà de la taille maximum de la grille ? Ce sont les règles. Le message passe (pour un court instant) : oublie l'implémentation, quelles sont les fonctionnalités attendues par l'utilisateur de cette grille ?

Au fur et à mesure, les choses à tester s'éliminent, et d'autres émergent. C'est fun ! Mais on n'a pas encore écrit une seule ligne de code.


– Alors, ce qu'on va faire, c'est qu'on va se passer le clavier à chaque phase : j'écris un test, tu écris le code qui le fait passer, on voit ce qu'on peut améliorer dans le code, tu écris un test, j'écris le code qui le fait passer, etc¹¹⁴. Ça te permettra de voir comment se déroule chaque phase et de l'appliquer ensuite par toi-même. On parle souvent de ce cycle en nommant ses étapes : Red, Green, Refactor.

> « Red » : écrire un test en échec

```
assertThat(cellule).isEqualTo(0);
```


– Écrire le test devrait être très simple. On va suivre le format Gherkin¹¹⁵. D'abord la partie Given, étant donné un contexte dans lequel évolue la classe à tester. Vient ensuite, When : lorsque j'exécute une méthode de cette classe, et enfin Then : alors on s'attend à un comportement, un résultat. Regarde, on va même te faire un raccourci dans ton IDE pour t'écrire le squelette de ta méthode de test automatiquement.

Damien écrit la signature de la méthode pour que ça compile, écrit le test, et le lance.

- Attends, pourquoi tu lances le test ? On n'a rien écrit, on sait bien qu'il va échouer !
- On n'est pas à l'abri d'une erreur : le test doit d'abord échouer, et pour les bonnes raisons. Tu peux très bien écrire un test qui n'échoue pas alors que ta méthode n'est pas encore implémentée. À quoi servirait ce test alors ? Tu dois vérifier que ton test guide l'implémentation, et te protégera bien quand tu retoucheras le code.

114. Aussi appelé *Pair programming* en ping pong.

115. Syntaxe utilisée à l'origine par la méthode *Behavior Driven Development* et mise en avant par l'outil Cucumber.

> « *Green* » : écrire le code qui fait passer le test

```
return 0;
```

– *Ok, quel est le code le plus simple que tu puisses écrire pour que le test passe au vert ?*

Je commence à cogiter...

– *Non non, ne t'embête pas avec tout ça, on attend juste une valeur pour l'instant. Retourne-la, ne te lance pas dans les conditions, on n'a pas encore les tests pour ça. On ne fait pas de design à l'avance, c'est du stock (traduction : avec des si... vous connaissez la suite).*

Test au vert, implémentation la plus simple, code défensif, éviter le stock ; je sens que j'ai du chemin. J'implémente donc le code qui fera passer le test. J'ai rarement écrit une ligne aussi bête... Mais elle faisait passer le test.

> « *Refactor* » : améliorer le design

```
return CELLULE_VIDE;
```

– *À mon tour, dit-il en reprenant le clavier. Bon, il n'y a pas grand-chose à faire, c'est notre premier test et notre première implémentation. Allez, 0 c'est un nombre magique¹¹⁶, remplacer par CELLULE_VIDE ça a du sens...*

Ah oui, j'aurais pu y penser, tiens.

> Le cycle TDD

Plus les cycles Red-Green-Refactor passaient, plus le volume de code, et surtout des tests, augmentait. Il y avait toujours plus de remarques, et la phase de refactor devenait plus conséquente. Et pourtant les tests finissaient toujours par être verts, et le code était toujours aussi lisible, évident. Ce n'était qu'un exercice, une simple grille de Puissance 4, mais quel soin pour le réaliser ! Il suffisait de suivre les étapes, et d'écouter les conseils.

116. Nombre magique : constante numérique non nommée qui peut nuire à la lisibilité du code.

Puis d'autres notions sont apparues. Il me parle de *mocks* et de *stubs* : la confusion est très courante pour beaucoup de développeurs. Ok, je ne confondrai pas. Il me dit que **les dépendances statiques sont une plaie pour la maintenabilité du code** ; notamment dans les tests car on ne peut pas les bouchonner. Ça a l'air important, je ferai gaffe.

Ça avait l'air facile, c'était du bon sens. Pour s'y retrouver :

- **Ne mets pas trop de responsabilités dans tes méthodes.**
- **Écris du code explicite.**
- **Ne teste qu'une chose à la fois.**

> *Découverte*

Les premiers jours sur le projet ont démarré en binôme. Sur le même principe, on échangeait le clavier à chaque étape du cycle. C'était presque un jeu où il fallait trouver le code le plus simple, le plus lisible, qui permette d'obtenir la fonctionnalité. Damien avait toute la patience et l'humilité nécessaires pour accueillir mes remarques, et je prenais tout le recul que je pouvais pour faire de même. Ça m'a permis d'apprendre une chose simple : il n'y a pas « mon code » ou « ton code ». **L'amélioration continue se porte mieux quand elle n'est pas encombrée de l'ego de chacun.**

Le meilleur signe d'évolution venait pour moi des revues de code. Ce qui était hors de nos pratiques devenait de plus en plus évident, et les retours sur les *User Stories* que j'implémentais de moins en moins volumineux. Mais j'étais encore trop lent. Je le sentais, ce n'était pas encore naturel. Je passais encore trop souvent par des phases où je développais mes tests après l'implémentation. De fait, le design n'émergeait pas avec les tests, et je n'écrivais que ceux auxquels l'implémentation me faisait penser.

Pourquoi était-ce plus fastidieux que ce que l'on m'avait vendu ? Avec le recul, je me dis que de nombreux développeurs ont dû passer par cette phase avant de comprendre d'où venait cette fatigue. C'est presque un passage nécessaire, où l'on réalise qu'on n'arrive pas à tirer le bénéfice des méthodes avec la même efficacité que les autres. C'est à mon avis le passage où nombreux sont ceux qui abandonnent, alors qu'ils sont si proches !

Ø Passer le cap

T'as les doigts sur la prise. Tu tires. T'as mal, mais faut pas lâcher !

> *L'équipe s'agrandit !*

Près de deux mois après le démarrage du projet, Abel, un autre membre de la tribu CRAFT d'OCTO Technology rejoint l'équipe.

- *Une demi-heure pour mettre en place l'environnement de développement, pas mal, nous dit-il avec le sourire.*
- C'était plutôt rassurant.*

Quelques jours ont suffi pour qu'il fasse évoluer nos standards et notre approche du code. Et en peu de temps, Abel avait rattrapé ce que nous connaissions de l'application.

Un jour, il pose un livre sur la table : *The Software Craftsman – Professionalism, Pragmatism, Pride*¹¹⁷. Il nous explique qu'il vient de le finir et que ça lui a beaucoup plu.

- *Franchement c'est facile à lire. Tu sens que le mec est passé par ce qu'on a connu. Il y a pas mal d'anecdotes, et beaucoup de bon sens. Je pense qu'on devrait tous l'avoir lu. Ça t'intéresse ? dit-il en me regardant.*
- *Pourquoi pas, oui. Avec le métro je devrais trouver le temps ! (Pour ceux qui ont lu ce livre, cet état d'esprit me rappelle la partie Creating time.)*

>

Attitude professionnelle

Je n'avais jamais saisi ce que cela pouvait signifier dans le domaine du développement.

Flashback. Je suis sorti d'une école où l'on m'a appris à me débrouiller. Honnêtement c'était efficace. On m'a appris à plonger dans n'importe quelle technologie, et ça m'a rendu productif (ouch) dès la première année de stage. Enfin, productif pour des boîtes où les développeurs avaient autant de bonnes pratiques dans leurs bagages que moi.

Le résultat, pour moi comme pour nombre d'étudiants de cette école, c'est que je me suis mis à « bricoler » des programmes. On passe du temps à penser au code, à faire des diagrammes de ce

117. *The Software Craftsman : Professionalism, pragmatism, pride*, Sandro Mancuso (Prentice Hall, 2014)

que sera un travail dont on ne peut raisonnablement pas voir tous les défauts avant d'être dedans. Puis on « pisse » une centaine de lignes d'un coup, on lance l'application en vérifiant que « ça marche », sinon on debug à coups de `printf`, et on tente déjà de l'optimiser alors que c'est à peine fini (ou même avant que ce soit stable).

C'est chaotique. À cette époque je pensais qu'un algorithme qui résolvait n'importe quel problème en complexité temporelle $O(\log n)$ était le meilleur code que je pouvais produire, que c'était ça le développement.

Ce que m'a fait comprendre ce livre ? Que j'étais irresponsable. Je me suis mis à penser à tout ce que j'avais fait, et je me suis senti bien génér. Comment se fait-il qu'un code produit pour une entreprise n'ait pas été développé pour qu'elle puisse s'en resserrer ? Pourquoi n'importe qui ne pourrait-il pas le comprendre, le faire évoluer et s'assurer de sa stabilité ?

Imaginez qu'une fois que vous quittez votre entreprise elle réalise que, dans une des applications que vous avez développées, il manque une fonctionnalité, ou que l'une d'entre elles est instable. Si votre code n'est pas maintenable, que fait-elle ? Elle paie très cher pour stabiliser ce qui a été (mal) fait ? Elle jette votre travail ? Et quand vous revenez un an après sur votre travail, que se passe-t-il ? Combien de temps passerez-vous à revenir sur ce que vous avez déjà fait ? Combien de temps vous faudra-t-il pour finalement avancer et prendre votre pied ? Honnêtement, pourquoi s'infliger tout ça ?

Aujourd'hui, une variable que vous avez appelée `date` pour être tapée plus vite (et l'auto-complétion ?) peut être renommée à travers toutes les sources en `lastEventDate` grâce à un raccourci clavier. Votre IDE peut détecter le code dupliqué. Avant un `commit` il vous prévient des erreurs qu'il a détectées. Besoin d'extraire du code ? Un raccourci. La technologie ne nous donne plus d'excuses, nous avons le temps et les moyens de bien faire.

Je pense qu'il n'y avait pas mieux que ce bouquin pour m'embarquer plus loin dans le mouvement. Une dose le matin et le soir, pour que la fin de la journée ne soit pas la fin de la réflexion.

C'est plus que ma façon de développer que j'ai fini par remettre en cause.

> Sortir du passage

Pourquoi était-ce plus fastidieux que ce que l'on m'a vendu ? Tout ça a déclenché pas mal de questions. Aurais-je mal compris l'une des méthodes ? Ce projet est-il un cas particulier ? Pourquoi les autres développeurs y arrivent-ils alors ? L'expérience ? Suis-je trop bête ?

Le problème était, comme souvent, entre la chaise et le clavier. Et alors que je me torturais à savoir où ça pouvait pécher, où les pièges que l'on trouve dans les livres pouvaient se terrer, je finis par appeler à l'aide Abel :



- *J'arrive pas à m'en sortir. Je suis certain que c'est idiot. La fonctionnalité ne me paraît pas plus compliquée qu'une autre, mais j'arrive pas à voir comment écrire les tests sans penser à l'implémentation.*
- *Ok, je peux être franc avec toi ? Je pense que tu te prends trop la tête. Tu te projettes trop sur des choses que tu ne peux pas encore savoir. Je vois que tu veux bien faire. Ça peut paraître contre-intuitif, mais ton meilleur code tu ne le feras pas en imaginant toutes les possibilités en avance, en te demandant « et si je fais ça ? et ça ? mais s'il y a ça, alors il faut faire autrement ». Il y a trop de possibilités. Je ne dis pas que tu ne dois pas te demander quelles dépendances ou quelles technos tu pourrais employer, mais KISS : Keep It Simple, Stupid.*

Et là vient le déclic. Ce qu'il me raconte ensuite je le comprends comme ça :

« Écris un test simple, certainement pas encore complet, mais pose une base, une certitude. Tente d'écrire le code qui y répond, fais évoluer le test au besoin, au fil du code. Pour l'instant, oublie les performances, ne t'occupe pas du découpage, de la qualité, parce que tu n'arrives même pas à implémenter la fonctionnalité.

De certitudes en certitudes, de lignes évidentes en lignes évidentes, tu auras écrit un code qui marche. Certes, il ne sera pas parfait, mais tu auras atteint ton premier objectif : ton test passe. Ensuite seulement, sur cette base de certitudes, de code qui marche, tu pourras commencer à refactorer, à faire du joli que tout le monde comprendra.

Un seul combat à la fois. »

Aussitôt après, j'efface le code que j'avais commencé, et je joue le jeu. Non sans difficultés, mais comme disait Morpheus « Libère ton esprit ». J'avance aveuglement en me disant : « ça je m'en moque, ça aussi, et ça ah tiens j'en ai besoin hop je mock, ok le code est moche, mais ça marche. » J'ai ma fonctionnalité : « Ok refactor : trop de mocks, le test se lit mal, j'ai sûrement un problème de responsabilité dans cette classe. » Je crée un service avec la même méthode, en y mettant ce que ma classe précédente n'avait pas besoin de connaître. « Wow. C'est ça. L'archi vient d'émerger. Le code est fonctionnel, lisible, il respecte nos standards, et je n'ai pas passé mon temps à me projeter. »

Je ne dis pas que je n'ai jamais recommencé, les mauvaises habitudes ont la peau dure. Mais je connais mon principal souci. Et à partir du moment où je m'en rends compte, je passe le cap. Puis enfin, je peux prendre le temps de bien faire les choses, de prendre du plaisir, d'être fier de ce que je fais. Mieux, je peux le partager, tout le monde peut le lire.

○ Transmettre

T'inquiètes pas, moi aussi j'ai connu ça

> Agile et Software craftsmanship

Il est arrivé plusieurs fois que le *Product Owner* (quelqu'un du métier, pas versé dans le développement) se prenne au jeu et lise notre code par-dessus notre épaule :

– *Should redirect to english home location when no uri provided. Given first time on site true, when controller index, then redirect location is equal to « accueil ».* Ce serait pas « en_home » plutôt ?

Oui il l'a très bien compris. Si ça ce n'est pas un signe... Bien sûr, l'objectif n'est pas de rendre son code lisible pour le *Product Owner*. Mais on réalise dans cet exemple qu'avec la communication dans le même bureau, par mail, par téléphone entre tous les membres de l'équipe, que chacun entend et se soucie des problématiques des autres.

Il est arrivé que notre *Delivery Manager* côté client et notre *Product Owner*, pour qui c'est le premier projet mené en utilisant les méthodes agiles, défendent nos pratiques à leur hiérarchie avec conviction, parce que leurs effets sont immédiats et évidents pour eux. Quand un *refactoring* est fait, ils comprennent que cette action aura un impact positif sur notre vélocité à moyen et long terme. Aujourd'hui, quand une fonctionnalité est demandée, ils savent que des tests seront écrits. Ils n'ont pas peur de demander une fonctionnalité, ni peur qu'une modification ait des effets de bords, et ils savent que les tests sont cette garantie. Ils comprennent aussi que parce que nous faisons du TDD, tout démarre par les tests, et donc qu'**une couverture de test à 97 % est une conséquence naturelle de la méthode et non une démarche de « surqualité » pour atteindre un objectif.**

> *L'épreuve*

L'un des soucis de la version précédente du produit était la qualité du code. Son évolution était tellement douloureuse que le *Product Owner* hésitait à demander aux développeurs des fonctionnalités supplémentaires de peur que le projet s'effondre à cause des régressions.

Pour ne plus rencontrer ce problème, le client a voulu intégrer un nouveau développeur, Baptiste, en milieu de projet. Si l'un des objectifs était d'augmenter la vélocité, le client voulait avant tout vérifier que quelqu'un qui ne vient pas de la même entreprise puisse reprendre facilement le projet en cours.

Baptiste connaissait *Scrum*, mais ne pratiquait pas TDD. En une itération, grâce au binômage, aux revues de codes et aux différentes aides dont nous bénéficiions des uns et des autres pendant le

développement, il avait adopté nos pratiques et connaissait grosses modo les entrailles du projet. Plus le temps passait et plus il se passionnait pour les méthodes qu'on employait. Les retours de revue de code perdaient en volume. Si bien que lors d'une rétrospective, il nous a sorti un très beau *keep*¹¹⁸ :

– Damien ne m'a fait aucun retour de revue de code sur une des *User Stories* que j'ai développées !

C'était comme un *flashback* sur un passé proche. Je me revoyais passer toutes ces étapes : curiosité, enthousiasme, doute, et une fois le cap passé, l'état d'esprit, les pratiques, qui faisaient partie intégrante de sa vision du développement.

Je passe les discussions autour d'une bière sur ce qu'on passait notre temps à faire en journée, sur les mauvaises expériences qu'on a connues avant d'atterrir sur ce projet. Bref, on ne s'en lassait pas, et on en redemandait.

Toute l'équipe a fini par se rendre à un *meetup Software Crafts(wo)manship Paris*¹¹⁹ organisé chez OCTO Technology. Il y avait tant de choses intéressantes, de notions inconnues, et pourtant, aussi récente que soit notre initiation, nous étions déjà capables de comprendre et de partager avec toute une communauté.

Je me suis aussi retrouvé de l'autre côté de l'enseignement au début de cette période ! J'en étais déjà convaincu : **transmettre c'est être capable de transformer son savoir, son expérience, et l'adapter à la personne qui va le recevoir**. Ça implique de comprendre ce que l'on a assimilé, puis d'en trouver l'essence. C'est de la conceptualisation. Une fois que ces concepts se dégagent, il faut comprendre la situation de celui avec qui on les échange. Puis enfin, on les remet dans sa situation, à sa portée. C'est à mon sens le meilleur moyen de renforcer un savoir ; de le remettre en question, ou d'en découvrir de nouvelles facettes. En rendant accessible cette connaissance, j'ai encore une fois changé. Notre échange nous a changés.

> *Un thème pour les rassembler tous, et dans la lumière...*

À mon avis, il y a un point commun entre l'agile, le code responsable, et l'attitude des *Craftsmen*. La communication doit être fluide, sans accroche, de la technique à l'humain. Pour que l'agilité fonctionne, que les retours soient efficaces et rapides, l'empathie et la curiosité du *Craftsman*, tout comme l'excellence technique qu'il manifesterà, sont les clés. Pour atteindre cette excellence, le code doit être impeccable, expressif, prévisible, accessible à tous les développeurs et facilement modifiable. Pour qu'il ne cesse de l'être, pour que l'on prenne plaisir à travailler ensemble et à

118. Fait positif à conserver, remonté lors d'une rétrospective

119. La communauté *Software Crafts(wo)manship Paris* se réunit une fois par mois <http://www.meetup.com/fr-paris-software-craftsmanship/>

retrouver ce même plaisir en arrivant sur un nouveau projet, il faut transmettre au maximum nos pratiques, notre état d'esprit. Il y a un cercle vertueux, et nous sommes responsables de son succès.

○ Quel est le résultat ?

Alors voilà, ce projet sur lequel je ne contribuerai bientôt plus était pour moi la meilleure expérience de développement que j'ai vécue. J'ai beaucoup appris personnellement, et je pense que la clé du succès sur la **transmission de tout ce savoir-faire**, se trouve dans les quelques moments-clés décrits ici. Des liens forts sont nés, et c'est sans aucun doute le signe que le courant a été bien établi, que les pratiques n'ont pas été bêtement appliquées, et que la communication qui en a résulté en est aussi la cause.

Récemment, je me suis mis à faire un *pet project*¹²⁰ en TDD. Ça me paraît désormais aberrant de me passer de tout ce que j'ai pu apprendre. Après avoir lu *The Software Craftsman*, je me retrouve au milieu de *Clean Code*. Je me passionne de plus en plus pour ce sujet, cette attitude que je me vois appliquer dans d'autres conditions même hors du développement.

○ Et si je veux faire ça chez moi ?

L'idéal, c'est déjà d'avoir un *Craftsman* avec soi. Rien de mieux que quelqu'un qui a déjà parcouru ce chemin, qui connaît les obstacles et qui s'est donné pour objectif de transmettre son savoir. Donc si vous êtes déjà équipés, écoutez-le !

Mais si vous lisez ce chapitre, peut-être que vous ne l'avez pas à vos côtés. Vous pouvez continuer à lire sur le sujet et vous tourner vers les ouvrages cités dans ce livre. Et surtout, n'hésitez pas à rejoindre une communauté pour partager, c'est aussi une excellente occasion d'échanger sur ce que vous avez appris, vos déboires et vos victoires !

120. *Pet project* : projet personnel sans engagement majeur, qui peut viser à répondre à un besoin fonctionnel, ou apprendre une nouvelle technologie, pratiquer un geste, etc.

Bibliographie

Références utilisées dans cet ouvrage, par ordre de citation :

Préface

- Pekka Himanen, Linus Torvalds : *L'Éthique Hacker et l'esprit de l'ère de l'information*. Random House Trade Paperbacks (2001)

Introduction

- Janice Lo : *Nouvelles perspectives pour réduire l'impact du turnover dans l'informatique*. hec.fr(2014).<http://www.hec.fr/Knowledge/Strategie-et-Management/Management-des-Ressources-Humaines/Nouvelles-perspectives-pour-reduire-l-impact-du-turnover-dans-l-informatique>
- Marc Andreessen : *Why Software Is Eating the World*. Wall Street Journal (2011) <http://www.wsj.com/articles/SB10001424053111903480904576512250915629460>
- *Manifesto for Software Craftsmanship* (2009) <http://manifesto.softwarecraftsmanship.org/#/fr-fr>
- Andrew Hunt et Dave Thomas : *The Pragmatic Programmer : From Journeyman To Master*. The Pragmatic Bookshelf (1999)
- Robert C. Martin : *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall (2008)
- Sandro Mancuso : *The Software Craftsman : Professionalism, Pragmatism, Pride*. Prentice Hall (2014)

Chapitre 1 : Transmettre une culture de la qualité et de l'excellence technique

- Henrik Kniberg : *Agile Product Ownership in a Nutshell*. blog.crispe.se (2012) <http://blog.crisp.se/2012/10/25/henrikkniberg/agile-product-ownership-in-a-nutshell>
- Ben Linders : *Q&A with Sandro Mancuso about The Software Craftsman*. InfoQ (2015) <http://www.infoq.com/articles/mancuso-software-craftsman>
- Alistair Cockburn : *The costs and benefits of Pair Programming* cs.pomona.edu (2000) http://www.cs.pomona.edu/~markk/cs121.f07/supp/williams_prpgm.pdf
- Gerald M. Weinberg : *The Psychology of Computer Programming*. Van Nostrand Reinhold Company (1971)
- Jeff Atwood : *The Ten Commandments of Egoless Programming*. blog.codinghorror.com (2006) <http://blog.codinghorror.com/the-ten-commandments-of-egoless-programming/>

- Simon Baslé : *Les Brown Bag Lunches, qu'est-ce que c'est ?*
InfoQ (2014) <http://www.infoq.com/fr/articles/bbl-fr>

Chapitre 2 : Maintenir la maintenabilité

- Capers Jones : *Software Quality in 2013: a survey of the state of the art*
p.66. Namcook Analytics LLC (2013) <http://namcookanalytics.com/wp-content/uploads/2013/10/SQA2013Long.pdf>
- Gene Bellinger : *Shifting the burden.*
systems-thinking.org (2004) <http://www.systems-thinking.org/theWay/ssb/sb.htm>

Chapitre 3 : Le Tech Lead, au service de l'équipe

- Gerald M. Weinberg : *Becoming A Technical Leader.*
Dorset House Publishing 1986)
- G. Gordon Schulmeyer : *Net Negative Producing Programmers.*
pyxisinc.com (1992) http://www.pyxisinc.com/NNPP_Article.pdf
- Patrick Kua : *Building the Next Generation of Technical Leaders.*
USI (2010) <https://www.youtube.com/watch?v=WEn5pwM7EjY>

Better Code : Préambule

- Ron Jeffries : *Refactoring -- Not on the backlog!*
ronjeffries.com (2014) <http://ronjeffries.com/xprog/articles/refactoring-not-on-the-backlog/>
- Robert C. Martin : *The Boy Scout Rule.*
programmer.97things.oreilly.com (2009) http://programmer.97things.oreilly.com/wiki/index.php/The_Boy_Scout_Rule
- Mathieu Gandin : *Scout toujours !*
blog.octo.com (2012) <http://blog.octo.com/scout-toujours/>
- OCTO Technology : Livre blanc *Les Géants du Web* (2012)
- OCTO Technology : Livre blanc *Java Productivity Primer, Douze recommandations pour augmenter votre productivité avec une usine logicielle* (2009)
- Tomaz Tekavec : *Mouseless Programming.*
codurance.com (2015) <http://codurance.com/2015/11/25/mouseless-programming/>

Chapitre 5 : Écrire du code compréhensible

- Martin Fowler : *Anemic Domain Model* [martinfowler.com \(2003\) http://www.martinfowler.com/bliki/AnemicDomainModel.html](http://www.martinfowler.com/bliki/AnemicDomainModel.html)
- Eric Evans : *Domain Driven Design : Tackling complexity in the heart of software* Addison Wesley (2003)
- François Saulnier : *Domain Driven Design, des armes pour affronter la complexité*
blog.octo.com (2011) <http://blog.octo.com/domain-driven-design-des-armes-pour-affronter-la-complexite/>

- Martin Fowler et James Lewis : *Microservices, a definition of this new architectural term* [martinfowler.com \(2014\) <http://martinfowler.com/articles/microservices.html>](http://martinfowler.com/articles/microservices.html)
- Éric Thérond : *CQRS, l'architecture aux deux visages* (partie 1). [blog.octo.com \(2011\) <http://blog.octo.com/cqrs-larchitecture-aux-deux-visages-partie-1/>](http://blog.octo.com (2011) http://blog.octo.com/cqrs-larchitecture-aux-deux-visages-partie-1/)
- Julien Kirch : *L'architecture microservices sans la hype : qu'est-ce que c'est, à quoi ça sert, est-ce qu'il m'en faut ?* [blog.octo.com \(2015\) <http://blog.octo.com/larchitecture-microservices-sans-la-hype-quest-ce-quest-a-quoi-ca-sert-est-ce-quil-men-faut/>](http://blog.octo.com (2015) http://blog.octo.com/larchitecture-microservices-sans-la-hype-quest-ce-quest-a-quoi-ca-sert-est-ce-quil-men-faut/)
- Kent Beck : *eXtreme Programming Simplicity Rules* [c2.com \(2014\) <http://c2.com/cgi/wiki?XpSimplicityRules>](http://c2.com (2014) http://c2.com/cgi/wiki?XpSimplicityRules)
- Martin Fowler : *BeckDesignRules* [martinfowler.com \(2015\) <http://martinfowler.com/bliki/BeckDesignRules.html>](http://martinfowler.com (2015) http://martinfowler.com/bliki/BeckDesignRules.html)
- Jason Gorman : *A Hierarchy Of Software Design Needs* [codemanship.co.uk \(2015\) <http://codemanship.co.uk/parlezuml/blog/?postid=1321>](http://codemanship.co.uk (2015) http://codemanship.co.uk/parlezuml/blog/?postid=1321)
- Martin Fowler : *Is Design Dead ?* [martinfowler.com \(2004\) <http://martinfowler.com/articles/designDead.html>](http://martinfowler.com (2004) http://martinfowler.com/articles/designDead.html)

Chapitre 6 : La Revue de Code

- M.E. Fagan : *Design and Code inspections to reduce errors in program development.* IBM Systems Journal (1976) <http://www.mfagan.com/pdfs/ibmfagan.pdf>
- Alberto Bacchelli & Christian Bird : *Expectations, Outcomes, and Challenges Of Modern Code Review* Microsoft Research (2013) <http://research.microsoft.com/apps/pubs/default.aspx?id=180283>
- Paul Hammant : *How Google Does Code Review* [dzone.com \(2013\) <http://java.dzone.com/articles/how-google-does-code-review>](http://java.dzone.com/articles/how-google-does-code-review)
- Capers Jones et Christof Ebert : *Embedded Software: Facts, Figures, and Future.* IEEE Computer Society (2009) <http://doi.ieeecomputersociety.org/10.1109/MC.2009.118>
- Steve Mc Connell : *Code Complete 2nd édition* chapitres Software Quality Assurance et Collaborative Construction. Microsoft Press (2004)
- Capers Jones : *Software Assessments, Benchmarks, and Best Practices* Addison-Wesley (2000)
- Sarah Mei : *Pairing with junior developers* [devmynd.com \(2015\) <https://devmynd.com/blog/2015-1-pairing-with-junior-developers/>](https://devmynd.com/blog/2015-1-pairing-with-junior-developers/)
- Jason Cohen, Steven Teleki & Eric Brown : *Best Kept Secrets of Code Review* SmartBear Software (2006) <http://smartbear.com/SmartBear/media/pdfs/best-kept-secrets-of-peer-code-review.pdf>
- Paul Hinze : *Pairing vs. Code Review: Comparing Developer Cultures* (2013) <https://phinze.github.io/2013/12/08/pairing-vs-code-review.html>
- Christopher Fox : *Java Inspection Checklist* (1999) http://www.cs.toronto.edu/~sme/CSC44F/handouts/java_checklist.pdf
- Robert C. Martin : *Oh Foreman, Where art Thou?*

[blog.8thlight.com \(2014\) <http://blog.8thlight.com/uncle-bob/2014/02/23/OhForemanWhereArtThou.html>](http://blog.8thlight.com/uncle-bob/2014/02/23/OhForemanWhereArtThou.html)

- Gerald M. Weinberg : *Quality Software Management.*
Dorset House Publishing Company (1997)
- Steve McConnell : *Upstream Decisions, Downstream Costs.* Windows Tech Journal (1997)
<http://www.stevemcconnell.com/articles/art08.htm>

Chapitre 7 : Une stratégie efficace de tests automatisés

- Christophe Thibaut : *Une Politique pour le Système d'Information chapitre Les Tests,* OCTO Technology (2006)
- Brian Marick : *Exploration Through Example*
exampler.com (2003) <http://www.exampler.com/old-blog/2003/08/21/#agile-testing-project-1>
- Lisa Crispin et Janet Gregory : *Agile Testing: a practical guide for testers and agile teams*
(Pearson Education, (2008))
- Steve Freeman et Nat Pryce : *Growing Object Oriented Software Guided By Tests.*
Addison Wesley (2009)
- James A. Whittaker, Jason Arbon, Jeff Carollo : *How Google Tests Software.*
Addison Wesley (2012).
- Joseph B. Rainsberger : *Integrated tests are a scam*
thecodewhisperer.com(2010) <http://blog.thecodewhisperer.com/permalink/integrated-tests-are-a-scam.html>
- Michael Feathers : *Working Effectively with Legacy Code.*
Prentice Hall (2004)
- David Gageot : *Du Legacy au Cloud en moins d'une heure.*
USI (2013) <https://www.youtube.com/watch?v=q11gydDAMSo>
- David Scott Bernstein : *Beyond Legacy Code.*
Pragmatic Bookshelf (2015)
- Ola Ellnestam et Daniel Brolund : *The Mikado Method.*
Manning (2014)

Chapitre 8 : Test Driven Development

- Kent Beck : *Test-Driven Development: By Example*
Addison-Wesley (2002)
- Ludovic Cinquin : *Le ROI de TDD*
Blog Octo Technology (2008) <http://blog.octo.com/le-roi-du-tdd/>
- Nachiappan Nagappan & al. : *Realizing quality improvement through test driven development: results and experiences of four industrial teams*
research.microsoft.com (2008) http://research.microsoft.com/en-us/groups/ese/nagappan_tdd.pdf

Index

3 amigos	114
<i>Acceptance Test Driven Development (ATDD)</i>	132
Agile	13, 52, 62, 81, 115
Ambiguïtés	76, 78
Amélioration continue	16
Analyse de code automatisée	70, 102
Apprentissage	18, 20, 52, 93, 134, 139
<i>Behavior Driven Development (BDD)</i>	76, 113, 114, 132
<i>Big Design Up Front</i>	81
Binomage (voir aussi <i>Pair programming</i>)	17, 89
<i>Birds Of a Feather (BOF)</i>	18
<i>Boy Scout Rule</i>	70, 83
<i>Brown Bag Lunch (BBL)</i>	18
Bugs (addiction)	29
Checklist	101
Code	
• Legacy	8, 36, 123, 134
• propre (<i>Clean Code</i>)	69, 74, 101
• Source de valeur	24
<i>Coding Dojo</i>	17, 52, 72, 93
Communication Non Violente (CNV)	99
Complexité cyclomatique	35, 102
CQRS	80
Dépendances	116, 117 123
Design	
• Émergent	81, 130
• Simple	80, 82
<i>Design Pattern</i>	22, 74, 77
Dette technique	34, 59, 62
<i>Domain Driven Design (DDD)</i>	76, 80, 114
Effets de bord	75, 134
<i>Egoless Programming</i>	17, 99
<i>eXtreme Programming</i>	9, 80, 126
Feedback	17, 69, 71, 89, 96, 98, 109, 126

Glossaire	77
<i>Ice Cream Cone</i> (anti-pattern)	122
Intégration Continue	71
KISS	80, 82, 146
Management (décideurs)	20, 38, 50, 104
Méthode « 5 pourquoi »	32
Microservices	80
Mock	117
MOI (modèle)	47
Nommage	74, 97, 133
Non-qualité	13, 26, 59
<i>Pair programming</i>	17, 69, 89, 92, 94
Product Owner	13, 28, 52, 100
Professionalisme	16
Propriété collective du code	51, 52, 86
Qualité logicielle	15
<i>Refactoring</i>	16, 69, 70, 71, 80, 129
Retour sur investissement (ROI)	93, 136, 137
Revue de code	48, 52, 69, 83, 86, 95
• collective	93, 94
• par un pair	89, 94
SOLID	81, 82
Standards de code	63, 69, 77, 83, 86, 91, 96, 97, 101
Tech Lead	18, 47, 103
Tests	
• automatisés	69, 109, 116
• couverture de tests	120, 126, 147
• d'intégration	113, 116, 121
• unitaires	27, 34, 44, 113, 116, 118, 121, 131
Test Driven Development (TDD)	26, 78, 128, 129, 136, 137
<i>Timeboxing</i>	56
<i>User Story</i> (voir fonctionnalité)	35, 52, 100, 112
Variables Globales	75, 123
YAGNI	80, 82, 83

Le mot de la fin :

○ À propos des auteurs

Ce livre est un ouvrage écrit collectivement, par Michel Domenjoud, Christophe Thibaut, Arnaud Huon, Abel André, Cédric Rup, Julien Tellier, Julien Jakubowski et Nelson da Costa, avec la contribution de toute la tribu CRAFT d'OCTO Technology.

Développeurs, Tech Leads, coachs ou encore architectes, nous avons voulu rassembler notre expérience et celle de la communauté OCTO dans cet ouvrage.

La création graphique a été réalisée par Sophie Delrongo et Caroline Bretagne.

○ Remerciements

Un grand merci à tous les contributeurs, relecteurs et personnes interviewées : Damien Beaufils, Martin Bahier, Laurent Dutheil, Fabien Lamarque, Sébastien Roccaserra, Angélique Vernier, Jean-Charles Dessaint, Nelly Grellier, Joy Boswell, Pierre Nicoli, Benoît Lafontaine, Christian Fauré, Hervé Vaujour, Mahdi Lansari, Florent Jaby, Gabriel Kaam, Antoine Bernard, Borémi Toch, Timothée Carry, Laurent Brisson, Daniel Sabin, Christophe Charles, Adrien Estival, Jean-Christophe Blondel, Jérémy Buget, Julien Kirch, Jean-Yves Rivallan, Frédéric Merizen, Martin Pernollet, Vincent Guigui, Eric Fredj, Olivier Roux, Cyrille Deruel, Julien Vignolles, Mathieu Despriee, Wassel Alazhar, Nelly Moret.

OCTO Technology

« Nous croyons que l'informatique transforme nos sociétés. Nous savons que les réalisations marquantes sont le fruit du partage des savoirs et du plaisir à travailler ensemble. Nous recherchons en permanence de meilleures façons de faire. THERE IS A BETTER WAY ! »

– Manifeste OCTO Technology

OCTO Technology est un cabinet de conseil et de réalisation IT fondé en 1998, aujourd'hui présent dans cinq pays : la France, le Maroc, la Suisse, le Brésil et l'Australie. Les consultants OCTO accompagnent leurs clients dans la transformation digitale de leurs entreprises en intervenant à la fois sur la technologie, la méthodologie, la culture et la compréhension de leurs enjeux métier. Les équipes OCTO utilisent la technologie et la créativité pour les accompagner dans la construction de nouveaux business models. Lors de ses quatre récentes participations, OCTO était sur le podium du palmarès Great Place to Work® des entreprises de moins de 500 salariés où il fait bon travailler. En parallèle de ses activités de conseil, le cabinet organise USI - Unexpected Sources of Inspiration - qui s'impose depuis 2008 comme une référence parmi les plus grandes conférences internationales sur la transformation digitale.

Dépôt légal : mai 2016

Conçu, réalisé et édité par OCTO Technology,
50 avenue des Champs-Élysées - 75008 Paris.

Imprimé par IMPRO
98 Rue Alexis Pesnon, 93100 Montreuil Sous Bois

© OCTO Technology 2016

Les informations contenues dans ce document présentent le point de vue actuel
d'OCTO Technology sur les sujets évoqués, à la date de publication. Tout extrait
ou diffusion partielle est interdit sans l'autorisation préalable d'OCTO
Technology.

Les noms de produits ou de sociétés cités dans ce document peuvent être les
marques déposées par leurs propriétaires respectifs.



T e c h n o l o g y

There is a better way

www.octo.com - blog.octo.com

PARIS | RABAT | LAUSANNE | SÃO PAULO | SYDNEY