VOL. 01

CULTURE TEST

APPRIVOISEZ LA COMPLEXITÉ

THERE IS A BETTER WAY

TESTS AUTOMATISÉS : DÉFINITIONS ET DUALISME

Par Christophe Breheret-Girardin, Stéphane Bedeau, Sylvie Ponthus-Violland

octo.com | blog.octo.com

Les auteurs.

Christophe Breheret-Girardin

Coach craft, formateur et conférencier, Christophe partage avec enthousiasme ses connaissances, afin d'inspirer et de guider chacun vers son plein potentiel.

Stéphane Bedeau

Développeur et formateur, Stéphane a un goût prononcé pour les approches empathiques et la réalisation de produits numériques responsables de qualité.

Sylvie Ponthus-Violland

Développeuse, chef de projet, coach agile, Sylvie se passionne pour le développement de logiciels utiles, fiables et agréables à utiliser.

⊙ Contributeurs :

Adrian Staron, Alexandre Siguier, Alexis Chatillon (CNES), Benjamin Amanrich, Charles Pichery, Floris Tisseyre, Ivan Gorobenko, Samuel Retiere, Stéphane Pailha, Christophe Thibaut.

Direction artistique : Camille Vermorel et Rachel Savoie.

⊙ Relectrices et relecteurs :

Alexandre Jeambrun, Christian Fauré, Christophe Durand, Christophe Thibaut, Gaël Laurent, Hervé Taboucou, Sébastien Roccaserra, Sonny Klotz, Suxue Li.

Relecture formelle : Charlotte Petitbon, Alexandra Caussard.

Préface CNES

Alexis Chatillon
Spécialiste Qualité Logiciel du CNES, au sein du service “ Qualité Numérique, Exploitation et Opérations ”, dirigé par Sylvain TEODOMANTE.

Puisqu'il s'agit d'un livre, laissez-moi vous raconter une petite histoire.

Il était une fois une agence spatiale nommée CNES qui étudiait et explorait... l'espace ! L'espace est beau, fascinant, mystérieux mais aussi passablement hostile, notamment pour toutes les machines que nous envoyons là-haut, et qui souffrent de mille maux pour nos besoins scientifiques. Et n'oubliez pas qu'à l'intérieur, ce sont aussi des logiciels qui sont maltraités. Malgré tout ce qu'ils endurent, nous nous attendons à ce qu'ils fonctionnent parfaitement, quel que soit leur calvaire... Une seule panne pourrait rendre toute la mission inutile, ce qui n'est bon pour personne, et surtout pas pour notre agence qui a longuement travaillé, avec ses partenaires, pour produire ce satellite.

Fort d'une expérience conséquente, le CNES peut se féliciter d'avoir mené à bien nombre de missions, et d'avoir ainsi engrangé un certain savoir-faire.

Un savoir qui permet à nos engins spatiaux de “ mieux ” vivre la dure loi du vide, et à nos logiciels de mieux se préparer aux nombreuses surprises qui les y attendent.

L'histoire pourrait s'arrêter là, avec des acquis qui resteraient gravés dans le marbre, éternels. Cependant, vous qui lisez ces livres, vous vous doutez bien qu'il y a un “ mais ”, et notre agence aussi. Se remettre en cause, c'est aussi s'autoriser à progresser.

Les choses changent ici comme ailleurs, et on parle notamment de transformation numérique. Moderniser son approche autour de l'informatique, c'est un vaste chantier dans lequel le CNES s'est engagé.

Il faut accepter de revenir à l'esprit pionnier initial, explorer l'hypothèse d'un modèle “ Agile ”, rapprocher des métiers souvent éloignés et envisager d'expérimenter des méthodologies dites “ DevOps ”...

Ainsi, vouloir progresser ressemble de plus en plus à l'exploration de l'espace, non ?

Nous voilà à la place de nos satellites, confrontés à de nombreuses questions, à la recherche de réponses.

Au milieu de tous ces chantiers, nous parvient l'écho d'une discussion entre collègues, autour d'un café :

— Au fait, comment ça se passe sur le projet XYZ ?

— Écoute, tout allait bien jusqu'à ce qu'on prenne du retard. Maintenant, les développeurs courent après le temps pour produire le logiciel et les tests demandent beaucoup d'efforts.

— Cette histoire s'annonce mal on dirait. Il faut dire que le problème revient souvent et les tests deviennent rapidement le bouc émissaire idéal.

— C'est clair... mais est-ce qu'on n'aurait pas moyen de sortir de ce schéma ? C'est usant à force !

Hmm... Les tests, nous voilà donc avec un sujet pour progresser, et éviter des écueils que l'on connaît bien. Pour avancer dans cette réflexion, le CNES sollicite ainsi ses partenaires d'OCTO Technology. Seulement, même après de multiples brainstormings, interviews, et autres tentatives pour converger... Les tests constituent un sujet si vaste qu'il est difficile de savoir par où commencer. Il y a autant d'équipes que de besoins, d'outils, et de pratiques différentes !

Une chose est sûre, dans le monde standardisé du spatial, nous avons besoin de manuels afin de fournir des directives et de bonnes pratiques aux équipes projet.

Mais un manuel de tests sur quoi exactement ? Sur les outils ? Non, internet en est rempli. Sur l'automatisation, l'intégration dans un pipeline d'Usine Logicielle ? Non plus. Au fond, le souci n'est-il pas dans la façon dont nous abordons les tests ?

On les voit comme une exigence Qualité à remplir, une métrique à atteindre, mais au final les équipes s'en accommodent parfois difficilement, tout cela pour un bénéfice discutable.

Non, il faudrait plutôt changer notre regard sur ces tests, qu'ils ne soient plus perçus comme une obligation, mais plutôt comme un outil pour construire un logiciel. C'est ainsi que nos partenaires d'OCTO ont produit un guide pour déconstruire des idées bien ancrées au CNES, jusqu'à envisager le " Test First " et d'autres perspectives.

Très bien, notre agence possède enfin son guide sur les tests, fin de l'histoire ! Une fois de plus, vous vous doutez bien que non, ce serait trop simple.

Les guides c'est bien, mais ça se range facilement dans un coin de son bureau sans être utilisé. Des conférences ont bien été organisées, mais avec des agendas très chargés, pas facile de mobiliser tout le monde. Comment alors s'assurer que la connaissance parvienne jusqu'aux équipes projets ? Qu'une personne ouvrant le guide ait envie de tourner la page, de découvrir la suite ?

Un livre trop technique peut assommer, alors qu'un livre divertissant, on est curieux de connaître la suite ! Hmm... N'a-t-on pas justement conçu des fables pour transmettre des leçons ? Ne pourrait-on pas adapter ce concept à notre domaine technique ?

Difficile à envisager quand on n'est pas écrivain, n'est-ce pas ? Un immense défi de plus à relever. Décidément, notre histoire de tests ressemble vraiment à s'y méprendre à l'exploration spatiale. Et l'histoire qui va suivre, fruit de ce long, périlleux mais formidable travail de réflexion, n'est que le début d'une aventure plus vaste encore. En espérant qu'elle fasse écho à votre propre parcours, et résonne clairement dans votre esprit, je vous laisse poursuivre ce voyage...

La genèse de cette trilogie

Ces livres sont nés de discussions que nous avons eues avec le CNES à Toulouse. Leur demande initiale était que nous leur fournissions un état de l'art sur les pratiques de tests. Nous avons fait une première étude qui s'est achevée par une conférence de 2 heures.

À la suite de cette conférence, le CNES a souhaité que nous approfondissions le sujet et développions plus les pratiques. Nous avons travaillé itérativement avec le CNES à travers des échanges réguliers.

Cet ouvrage est le fruit de ce travail collectif.

Il fait aussi suite au précédent livre blanc " Culture Code ", paru en 2016 et toujours d'actualité. Si ce premier fait un focus sur l'équipe de développement et ses pratiques, incluant un chapitre sur les tests, nous allons ici nous concentrer sur les pratiques de tests. La lecture de Culture Code[1] est un complément à cette trilogie.

① https://publication.octo.com/en/download-whitepaper-culture-code

Ces livres s'adressent à vous si...

Ces livres s'adressent à toute personne qui désire apporter ou faire apporter de la valeur technique et/ou fonctionnelle à un produit logiciel, dans un but lucratif ou non.

Oui, ces livres ne sont pas réservés aux développeurs ! Ils ont d'ailleurs été construits pour mettre en avant les complexités du monde de la création logicielle tout en s'affranchissant au maximum de code.

Si vous êtes une personne qui code (Devs, Tech Leads, Lead Devs, DevOps, Data People, Machine Learning engineer... oui, toutes ces personnes doivent coder)

Vous découvrirez dans ces livres la philosophie qui se cache derrière les tests. Ainsi, vous aborderez les notions minimales, extensibles par vos démarches personnelles, qui vous permettront de :

- Créer un environnement de tests de qualité, intelligent et robuste, indépendamment du langage de programmation que vous utilisez ;
- Livrer un produit qui déchire et que vos utilisateurs auront plaisir à découvrir et à (devoir) utiliser ;
- Livrer un produit qui a un impact fort, rapide et certain dans la transformation du quotidien.

Si vous êtes une personne chargée de la vision et/ou de la gestion des impacts d'un produit logiciel pour votre organisation (PO, PM, Tech Leads, Lead Dev, C-levels, AMOA, QA...)

Vous découvrirez dans ces livres l'importance des tests dans :

- La vie et la survie de votre organisation ;
- La création de garanties et de sécurités ;
- Son impact fonctionnel et financier à moyen-long terme.

Si vous êtes une personne chargée de la gestion de votre organisation (RH, C-levels...)

Vous découvrirez dans ces livres l'histoire ainsi que les tenants et les aboutissants d'une culture d'équipe et d'organisation qui est :

- Propice à la survie financière de votre organisation ;
- Génératrice de bienveillance et de sérénité au sein des équipes ;
- Génératrice de garanties ;
- Génératrice d'attractivité dans le recrutement.

Si vous ne vous retrouvez pas dans ces catégories

Vous découvrirez dans ces livres une philosophie que vous pourrez probablement transposer tout ou partie dans votre quotidien. Après tout, le monde de l'informatique et de la création de logiciels est une modélisation du monde qui nous entoure et qui a pour but de répondre à des problèmes complexes pour l'humain et automatiser les tâches courantes rébarbatives.

Les organisations qui conçoivent des systèmes [...] tendent inévitablement à produire des designs qui sont des copies de la structure de communication de leur organisation.

Melvin Conway

En gros...

Que vous soyez Tech Lead, Lead Dev, PO, PM, RH, CTO, CEO, CPO, CFO, C3PO, AMOA, QA, une personne dont le job porte un nom à rallonge avec " manager " dedans, une personne curieuse...

... Que vous codiez ou non, dès lors que vous êtes en lien avec la production ou l'achat d'un logiciel pour votre organisation, vous êtes concerné par cette trilogie de livres !

" Les organisations qui conçoivent des systèmes [...] tendent inévitablement à produire des designs qui sont des copies de la structure de communication de leur organisation. "

Telle est la loi de Melvin Conway qui s'applique autant aux organisations qu'aux tests.

Ce qui ne sera pas abordé dans cette trilogie

Nous nous focaliserons, dans cette trilogie de livres, sur les phases de construction (build), c'est-à-dire principalement les tests automatisés. Ne seront pas abordés : les liens avec le product management, avec les tests utilisateurs, avec les MVP, avec les observations, ainsi que les tests du système social autour du produit.

Les points sur la dynamique d'équipe sont abordés en filigrane dans Culture Code.

Guide de lecture

Ces livres forment une histoire, c'est un documentaire-fiction. Nous avons utilisé toute notre expérience pour raconter l'histoire de Gaël et de ses collègues qui découvrent les tests. Gaël est fort, Gaël est fier et pourtant, il n'écrit pas de tests. Tout au long de cette histoire, nous expliquons les comportements de Gaël et les tests.

Ainsi, nous alternons histoire et explications techniques. Vous pouvez tout lire, l'histoire seulement ou les explications seulement, la mise en page vous y aide. Nous avons voulu cette alternance pour montrer ce qui se passe dans la tête de Gaël, et montrer que les tests rendent Gaël indestructible.

Cette trilogie n'est pas un manuel de tests à appliquer pour un projet parfait : la recette de cuisine unique exhaustive relève du fantasme. D'ailleurs, les cuisinières et cuisiniers les plus avertis connaissent leurs recettes[2] et malgré tout, ils utilisent leurs sens et acceptent ainsi la perfection de l'imperfection[3].

Ces livres proposent une approche pragmatique, à l'image des découvertes de Gaël et de son équipe. En vrac, nous allons parler de tests automatisés, de BDD, d'égo, de la place des tests, et de bien d'autres choses. Comment savent-ils qu'ils ont réalisé assez de tests ? Ils se rappellent pourquoi ils testent : pour construire leur code, pour bâtir des défenses, pour oublier, ne plus avoir peur et vivre sereinement. Si leurs tests leur permettent ça, alors oui ils ont réalisé assez de tests.

② Samin Nosrat, Sel Gras Acide Chaleur - Maîtriser les 4 éléments de la réussite, Éditions du Chêne, Novembre 2019, EAN : 9782812303616 (ou la série)
③ https://fr.wikipedia.org/wiki/Wabi-sabi

Sommaire

C’est ça le Flow, le graal ultime du développeur. Il est calme dans la complexité, dans l’océan des pensées. La concentration.

Dans une galaxie lointaine ...

BOUM... BOUM... BOUM...

La musique et les pulsations de son cœur se synchronisent, ils ne font qu'un. Les idées flottent et se présentent au rythme de la musique. Ses doigts parcourent le clavier, sont comme le prolongement de ses idées, ils leur donnent vie, les complètent et les rendent concrètes. Le sujet est simple : comment converger plus vite ? Les solutions sont infinies et les algorithmes possibles se mélangent.

L'algorithme génétique s'impose à lui, il ne voit plus que lui, il guide ses doigts. Mais la sélection des gènes est lente, les populations trop petites pour pouvoir déterminer si le temps de calcul est raisonnable.

Son problème mérite mieux que ça.

Il mérite mieux que ça.

Une idée incongrue vient se glisser, petite étincelle de folie dans cette partition : et si les descendants d'une génération n'étaient plus le fruit du hasard ? Et si cela devenait une intelligence, une sorte de sélection améliorée ? On fabriquerait alors les enfants en fonction de leur chance de survie.

[Vibration]

Comment converger plus vite ? Les solutions sont infinies et les algorithmes possibles se mélangent.

Les doigts s'activent déjà sur le clavier pour isoler le bout de code concernant la création des enfants. L'idée est déjà en train de prendre vie. La révolution est en marche. Darwin n'a qu'à bien se tenir. Une variation de plus dans la musique et l'étincelle se développe : Donald Knuth, son inspiration et son modèle depuis toujours, l'a déjà fait avant lui quand il s'est servi des mathématiques pour prédire la solution au mastermind[4], et ce, en un nombre de coups record.

[Secousses]

Cela semble de la folie, comment pourrait-on combiner un algorithme génétique avec la stratégie de sélection par élimination proposée par Knuth ? Et pourtant, s'il y parvient, qui sait les résultats qu'il peut obtenir ? Il flotte dans un océan de complexité, mais la complexité n'est pas son ennemie. Tel un chef d'orchestre, il sait en maîtriser toutes les subtilités, en tirer la puissance et l'harmonie. Le code se modèle selon ses envies. Ses doigts sont à l'ouvrage et guident les pensées autant qu'ils sont guidés par les pensées.

[" Gaël ! "]

C'est ça le Flow, le graal ultime du développeur. Il est calme dans la complexité, dans l'océan des pensées. La concentration. Toute pensée parasite a disparu. Tout est fluide et rien ne le freine dans son élan.

[" Hey Gaël ! Réveille-toi ! Nous sommes arrivés !!! "]

Confusion.

Mal de tête.

Son environnement s'estompe et un autre prend place.
Un train, semble-t-il.
Et quelqu'un qui le secoue.
Le retour à la réalité est brutal, tout ceci semble n'avoir été qu'un rêve.

④ The computer as master mind - Donald E. Knuth, https://www.cs.uni.edu/~wallingf/teaching/cs3530/resources/knuth-mastermind.pdf

C'est vrai que les nuits sont difficiles ces derniers temps, impossible de vraiment oublier ces problèmes d'optimisation qui le taraudent jour et nuit. Que fait-il dans ce train, déjà ? Ah oui, il est en route pour Space Inc., c'est un grand jour et il faut qu'il soit au meilleur de son art.

Il est plus que temps de chasser toutes ces pensées absurdes de sa tête. Et pourtant, est-ce vraiment si absurde ? Ne serait-ce pas plutôt son esprit qui vient de mettre le doigt sur quelque chose de révolutionnaire, mixer un algorithme génétique et les travaux de Donald Knuth ? S'il y parvient, cela sera sans précédent. Son talent sera à coup sûr reconnu, son heure de gloire viendra.

Non, non, non ! Il est impératif qu'il chasse tout cela de sa tête et tout de suite : la journée est bien trop importante pour se permettre d'être distrait.

La bulle de concentration

Une bulle de concentration est un état où le cerveau est monopolisé sur un sujet, sans être perturbé par d'autres pensées ou par des éléments qui ont lieu autour.

Des spécialistes estiment qu'il faut plusieurs minutes pour se créer une bulle de concentration et que celle-ci peut être brisée en une seconde par un élément externe (téléphone, question d'un collègue, popup intrusive à la suite de la réception d'un mail, d'une messagerie instantanée ou autre).

La conséquence de cette interruption est qu'il faut alors à nouveau plusieurs minutes pour recréer cette bulle.

Il existe des moyens pour améliorer la création d'une bulle, ainsi que sa durée (car il n'est pas possible de rester concentré 8h d'affilés) : planifier et découper ses travaux, utiliser la technique **pomodoro**[5], faire des exercices de respiration, écouter de la musique, etc.

Il y a également des astuces pour éviter les interruptions extérieures : mettre son téléphone en mode "ne pas déranger", désactiver les pop-up de l'ordinateur, fermer la porte de son bureau, mettre un casque en open-space, faire du télétravail, etc.

La concentration est surtout la capacité à rester au présent, c'est-à-dire de ne pas se laisser envahir par les images et pensées du passé ou du futur. La concentration est difficile à manier. Les sportifs se créent des routines durant l'entraînement pour se créer une bulle de concentration.

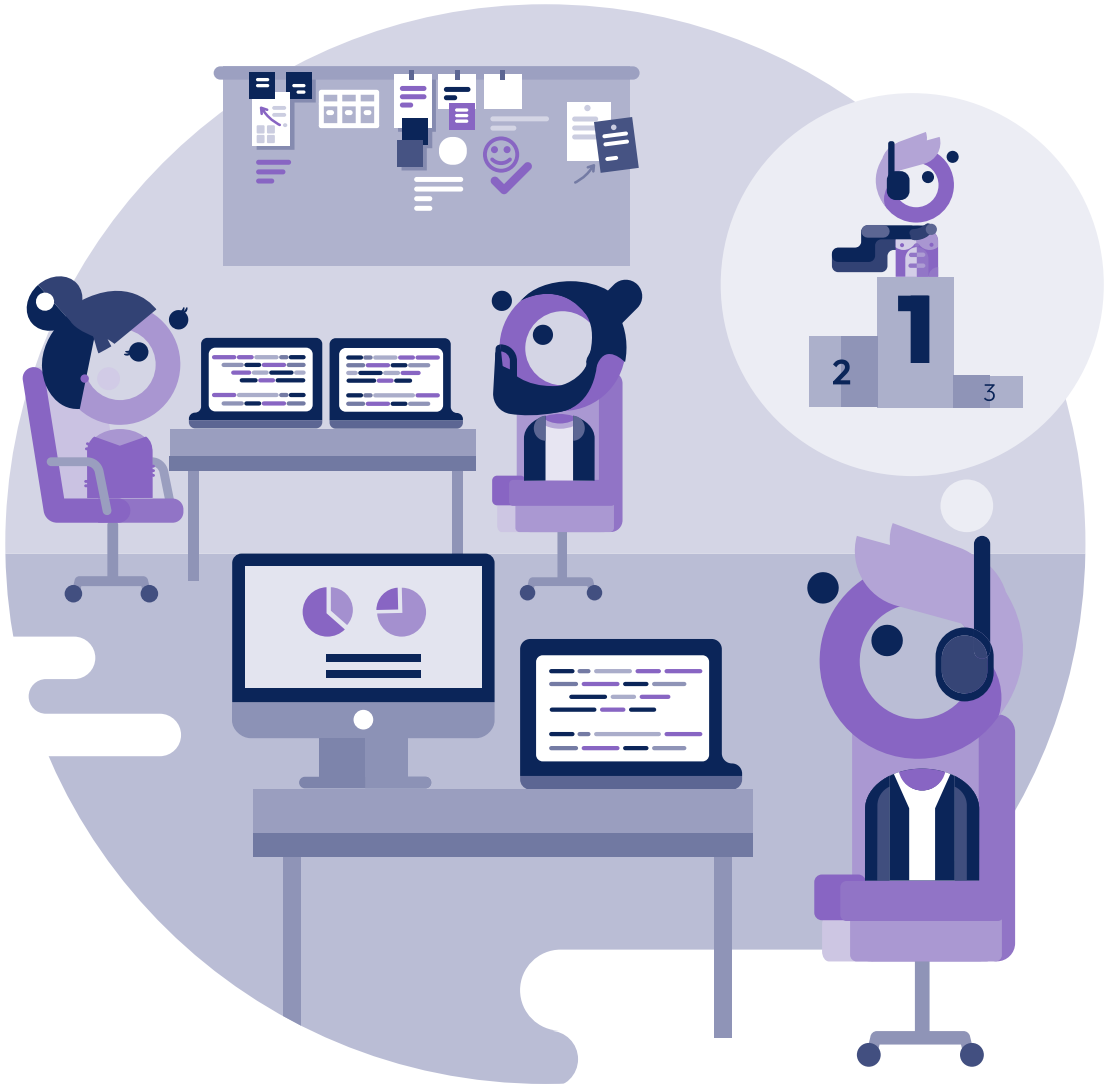
Il est possible de s'entraîner à se concentrer grâce à différent exercices comme la technique des 2 minutes qui consiste à fixer une horloge en ne faisant rien d'autre et sans penser à autre chose, les sudokus, les calculs mentaux, la lecture de longs articles ou livres, ou encore se concentrer sur nos actions en cours sans pensée extérieure.

⊙ Pour en savoir plus :

• https://medium.datadriveninvestor.com/deep-work-your-weapon-for-the-21st-century-3ca30c6600c1

• https://martinfowler.com/articles/developer-effectiveness.html

⑤ Technique qui consiste à découper le temps sur une tâche bien définie. Le découpage se fait en périodes de 25 minutes (pomodori) de concentration suivies de 5-10 minutes de distraction. On prend enfin une pause d'1 pomodori tous les 4 pomodori.



Ce n'est pas ça le travail d'équipe.

(In)justice League

La société Space Inc. a décidé de sélectionner l'entreprise qui travaillera avec elle sur son prochain sujet par l'intermédiaire de la résolution d'un problème.

Il s'agit de calculer le prix d'un panier rempli de livres Harry Potter.

Chaque exemplaire coûte 8 €, mais le libraire souhaite encourager l'achat des tomes différents de la série en appliquant les réductions suivantes :

- 5 % pour 2 livres différents ;
- 10 % pour 3 livres différents ;
- 20 % pour 4 livres différents ;
- 25 % pour 5 livres différents.

Par exemple, combien coûte ce panier de livres ?

- 2 exemplaires du premier livre ;
- 2 exemplaires du deuxième livre ;
- 1 exemplaire du troisième livre.

Réponse :

(3 * 8 €) - 10 % [premier livre, deuxième livre, troisième livre]

+ (2 * 8 €) - 5 % [premier livre, deuxième livre]

_____

= 36.80 €

Gaël jubile, c'est un problème d'optimisation, son terrain de jeu favori. Mais il n'a que deux heures et le problème est sûrement moins simple qu'il n'y paraît. Il s'agit de ne pas traîner et surtout de ne pas se laisser distraire par les tables voisines, sans parler des trois autres salariés de Blue Soft.

En effet, même si les trois autres se sont portés volontaires au même titre que lui, Gaël sait que tout repose sur lui et sur lui seul. Les autres ne sont pas au niveau et ne sont là que pour l'apparence.

Bien ! La première chose à faire est donc d'occuper ses acolytes pour qu'il puisse avoir le champ libre et se concentrer sur le problème :

— Écoutez, ce problème n'a pas été choisi par hasard, il comporte forcément un piège et j'ai besoin de vous pour l'identifier. Le problème consiste à faire des regroupements de livres et la clé réside dans la manière dont on les regroupe. Pouvez-vous me donner des heuristiques qui obtiennent les solutions optimales sur certains exemples, ainsi que me donner un contre-exemple qui montre que l'heuristique ne donne pas l'optimum dans tous les cas ?

— Bien sûr, tu peux compter sur nous, lui répond Paul, le manager de l'équipe.

Parfait, cela va les occuper une bonne partie des deux heures et il va pouvoir se concentrer sur la résolution. S'il arrive à retrouver le Flow de son rêve, la partie est gagnée.

L'exemple donné dans l'énoncé suggère qu'il faut faire les regroupements les plus grands possibles. Il y a une discontinuité dans les réductions : 5-10-20-25 n'est pas une suite linéaire. Soit le gap entre 20 et 25 est trop petit, soit le gap entre 10 et 20 est trop grand. Bien que Gaël n'ait aucune idée à ce stade de ce que cela va induire, il sent que le piège a toutes les chances d'être ici.

Alors qu'il est en pleine réflexion sur l'incidence de ces pourcentages sur la solution optimale, Farid s'exclame :

— Je sais !! La solution consiste à faire les regroupements les plus longs possibles, regardez ! Pour le panier de livres qui contient :

• 2 exemplaires du premier livre ;

• 1 exemplaire du deuxième livre ;

• 1 exemplaire du troisième livre.

La meilleure réponse est de faire un groupe avec les tomes 1, 2 et 3 et un autre avec seulement le tome 1.

— Euh merci, mais je parlais de trouver un piège… pas l'évidence à laquelle tout le monde pense en s'attaquant à ce problème, riposta Gaël, agacé d'avoir été déconcentré par une proposition aussi bête. Et maintenant si ça ne te dérange pas, j'aimerais me concentrer.

Gaël se plonge dans une concentration intense ; il lui faut trouver le lien entre les pourcentages de réduction et la détermination de l'optimum. Il y a clairement un effet de levier entre le pourcentage et le nombre de livres sur lequel la réduction s'applique. Sans qu'il ne puisse se l'expliquer, cela lui fait penser aux progressions géométriques. Oui, c'est ça, il doit y avoir un lien et il lui faut trouver l'équation qui régit la progression.

Mais pourquoi cela ne se résout-il pas ?

Pourquoi n'est-ce pas aussi fluide que dans son rêve d'il y a moins d'une heure ?

Et puis d'où provient tout ce bruit ?

Ne sont-ils pas censés être dans une salle d'examen ? Comment se fait-il que certaines tables se permettent de rire pendant une épreuve si déterminante ?

Il sent qu'il est sur le point de démêler tout ça lorsqu'il est de nouveau interrompu par Farid :

— Cette fois, c'est bon, je ne te dérange pas pour rien. Contrairement à ce que j'ai dit tout à l'heure, et à l'idée reçue qu'on pourrait avoir, la meilleure option n'est pas de faire les regroupements les plus longs, mais de les équilibrer. Regarde : si on prend 2 exemplaires du premier livre, 2 exemplaires du deuxième livre, 2 exemplaires…

— Excuse-moi de t'interrompre Farid, coupe Paul. Pour aider à mieux formuler et appréhender le problème dans notre esprit, je propose que nous adoptions une convention pour désigner le nombre d'exemplaires. La série (2, 3, 1, 4, 0) pourrait désigner 2 exemplaires du tome 1, 3 exemplaires du tome 2, 1 exemplaire du tome 3, 4 exemplaires du tome 3 et aucun exemplaire du tome 5. Qu'en pensez-vous ?

Tiens, pas bête ce Paul : utiliser un concept mathématique, les séries en l'occurrence, pour représenter un problème était forcément une bonne idée.

— Euh oui… si tu veux Paul, reprend Farid. Donc je disais que pour la série (2, 2, 2, 1, 1), la meilleure solution n'est pas de faire un paquet de 5 et un de 3 pour un total de 51,6 € comme on pourrait le croire de prime abord, mais plutôt deux paquets de 4 pour un total de 51,2 €.

Le piège est donc là : il faut systématiquement prendre la solution où la taille des paquets est la plus équilibrée possible, achève-t-il.

N'en pouvant plus, Gaël décide de lui mettre les points sur les i :

— Écoute Farid, merci pour la découverte mais ton premier exemple contredit la géniale conclusion que tu viens de nous livrer. En effet, si on suit ton raisonnement (2, 2, 1, 1), cela devrait donner 2 groupes de 2 ; or, la solution consiste à faire 1 groupe de 3 et 1 groupe de 1. Donc ça serait bien d'arrêter de m'interrompre toutes les deux minutes. Je dois me concentrer pour trouver la vraie solution.

Bien ! Ce coup-ci, les choses sont claires.

Où en est-il déjà ? Ah oui, l'équation permettant d'équilibrer les pourcentages et le nombre de livres en vue de trouver l'optimum.

Nouvelle interruption, de Paul ce coup-ci :

— Tu sais Gaël, cela ne fait pas quelques minutes mais 45 minutes que tu es perdu dans tes pensées, sans prononcer la moindre parole. Ce n'est pas ça le travail d'équipe et j'aimerais que tu témoignes d'un peu plus de respect à Farid qui fait de son mieux pour aider.

Allons bon, voilà qu'il va devoir faire dans le social et dans le tact en plein milieu d'une épreuve.

45 minutes quand même, presque la moitié de l'épreuve et il n'a pas encore écrit une seule ligne de code, ni même entr'aperçu la solution au problème. Il faut qu'il se penche sur cette non-linéarité du temps qui s'écoule. Mais ce n'est ni le lieu, ni le moment.

— Je pense avoir une solution à proposer, énonce alors Chloé.

Il faut croire que personne ne va laisser Gaël se concentrer. Pourtant, Chloé lui a semblé plus que compétente dans les quelques échanges qu'il a eu avec elle par le passé.

— J'espère que ça en vaut la peine, lui répond-il.

— Il me semble. Tout comme Farid, je suis tombée sur des exemples qui paraissent contradictoires. Je me suis dit que la solution n'est peut-être ni les regroupements les plus longs ni les plus "équilibrés", ou dit autrement, les plus courts. "Les plus longs" signifie qu'on

Ce n'est pas ça le travail d'équipe.

privilégie les groupes de 5, puis ceux de 4, 3 et enfin 2. “ Les plus courts ” signifie qu'on privilégie les groupes de 2, 3, 4 et enfin 5. Mais en réalité cela pourrait être n'importe quel ordre de préférence, explique Chloé.

Comment cela se fait que Gaël n'y ait pas pensé par lui-même ? Son raisonnement est brillant… En se limitant à quelques cas, elle simplifie la combinatoire et elle opère une réduction drastique de l'espace du problème !

Chloé remarque que Gaël est en train de se tortiller de frustration sur sa chaise, mais cela ne l'empêche pas de poursuivre :

— Si on y réfléchit bien, pour chaque livre, on le rajoute soit dans un nouveau groupe, soit dans un groupe 2, 3 ou 4 (impossible dans un groupe de 5 car il contiendrait déjà tous les tomes).

Il ne reste plus qu'à savoir dans quel ordre les rajouter si jamais on a le choix.

Pour cela, j'ai créé un panier avec un groupe de chaque taille possible et j'ai regardé l'incidence de l'ajout d'un livre tour à tour dans chaque groupe.

Il en résulte l'ordre préféré suivant : 3, 4, 2, 1

En triant les groupes suivant cet ordre et en choisissant le premier qui ne contient pas déjà ce tome, on obtient la meilleure combinaison possible.

Gaël est atterré : c'était à lui de mener ce genre de raisonnement. Comment a-t-il pu se faire voler la vedette par Chloé ? Il n'a même pas de quoi lui rétorquer que sa démonstration est fausse ou incomplète. En même temps, il ne doit pas être surpris : Chloé n'a pas caché sa volonté de faire carrière et cette soutenance est une occasion en or de se démarquer.

— Bravo Chloé, quelle belle analyse ! Je vais mettre tout cela au propre pour le présenter au jury. D'ailleurs, je propose aussi la notation `(g1, g2...)` pour désigner la taille de chaque groupe en sortie de l'algo. Je compte sur vous pour réaliser le code dans les 60 minutes qu'il nous reste. Dépêchez-vous, encourage Paul.

Aussitôt, Chloé et Farid se mettent en action, alors que Gaël reste pensif.

Egoless programming

De Bertrand Le Foulgoc[6]

D'après notre expérience, le milieu informatique peut être assez hostile, avec une raison majeure : l'égo du développeur.

Pourquoi tant d'ego ?

En informatique, nous nous posons beaucoup de questions, tout le temps. Pour être reconnu comme compétent dans un domaine, il faut souvent avoir raison. Certains poussent même le vice jusqu'à vouloir avoir raison tout le temps. De cette manière, aucun doute ne subsiste sur leurs compétences, techniques en tout cas. Le biais cognitif de gens incompétents se sentant supérieurs, appelé "effet Dunning-Kruger", n'arrange pas les choses. Quand le monde dans lequel nous baignons est dominé par des gens qui ont toujours (ou presque) raison, il devient difficile d'accepter d'avoir tort, car cela devient un aveu de faiblesse.

J'ai mal à l'égo

"Ego" vient du latin, et signifie littéralement "je". Voici une définition intéressante faite par la psychanalyse : le mot ego désigne en psychanalyse la part de la personnalité chargée d'équilibrer les différentes forces auxquelles est confronté le psychisme de l'individu. Ces forces incluent ses pulsions profondes, sa morale personnelle et la réalité du monde extérieur tel qu'il le perçoit.

Alors pourquoi tant de haine ?

Au regard de notre expérience, plusieurs raisons majeures nous viennent à l'esprit lorsque nous cherchons à justifier le manque de bienveillance dans un milieu :

• C'est un cercle vicieux ! ;

• C'est gratifiant. Quand nous marquons des points contre quelqu'un, nous montons d'autant plus haut que la personne est reconnue compétente ;

• C'est un excellent moyen de faire le "ménage", de forcer ceux qui n'ont rien à faire là, dehors ;

• C'est ancré dans la culture : nous jugeons beaucoup et nous aimons ça, nous avons la critique facile ;

• Le système éducatif note et classe dès le plus jeune âge, il y a donc une certaine continuité. Le travail personnel est aussi au centre des pratiques de l'école, il y a peu de cours sur le travail en équipe et la qualité des relations.

Heureusement, il est possible d'éradiquer ces mauvaises habitudes à l'échelle d'une entreprise, pour peu que la majorité soit motrice :

• Le cercle vicieux peut-être brisé assez rapidement si la communauté joue le jeu. Il faut d'abord que tout le monde prenne conscience du problème, et c'est souvent le plus dur ;

---

⑥ Egoless Programming - OCTO Talks ! https://blog.octo.com/egoless-programming/

• Cette forme de gratification n'est ni saine ni pérenne. Il suffit d'en être victime une fois pour redescendre en flèche. Et quand ça fait partie du jeu, ce n'est qu'une question de temps. Préférez la gratitude exprimée par les gens que vous aidez avec bienveillance, elle est nettement plus durable et constructive dans une équipe ;

• S'il y a besoin de "faire le ménage", c'est le signe d'un problème au niveau du recrutement. Intégrer une section culturelle au processus d'entretien permet d'éviter cette situation ;

• La culture d'une entreprise, bien qu'elle dépende évidemment du pays, est très malléable, c'est d'ailleurs une préoccupation importante, car si la mettre en place peut être difficile, il faut aussi veiller à la maintenir.

L'importance du vocabulaire

Un autre point qui a son importance est le vocabulaire. Un des principes de l'egoless programming est de ne pas trop s'attacher à son code ou à ses idées. C'est peut-être un détail, mais il peut néanmoins contribuer à l'ambiance tant que la culture n'est pas majoritairement bienveillante.

Le plus important est de faire preuve d'empathie. Et comme il est difficile d'enseigner ces valeurs, voici quelques exemples pour illustrer et vous mettre sur la bonne voie :

Ne dîtes pas
Mon code
Ma solution
Ton code est tout pourri
Tu comprends rien
Je te l'ai déjà dit, tu ne m'écoutes pas, je n'ai pas que ça à faire !
Ouais ça c'est facile je te le fais en deux heures

Dites plutôt
Le code
La solution que je propose
Tu pourrais améliorer ce code en...
Allez, on en discute au tableau
On a abordé un problème similaire hier, on peut développer un peu plus
Je connais bien, je pense pouvoir le faire assez vite

10 Commandements

Dans “ The Psychology of Computer Programming ”, Gerald M. Weinberg expose les dix commandements du code sans ego :

Comprenez et acceptez que vous allez faire des erreurs

Nul n’est infaillible et la meilleure façon d’apprendre est en faisant, et quand nous le faisons, nous faisons forcément des erreurs. Ce n’est pas un drame, c’est normal, et tout le monde passe par là.

Vous n’êtes pas le code

Nous passons tellement de temps à l’écrire, avec soin si possible, que le code devient une partie de nous-même. La preuve : supprimer d’énormes morceaux de notre code est douloureux au début. Quand quelqu’un critique notre code, il devient facile de le prendre personnellement : c’est une erreur, il faut s’en détacher. Il est vrai que la façon de délivrer le message peut incriminer directement (cf. l’importance du vocabulaire).

Peu importe ce que vous connaissez sur le “ karaté ”, quelqu’un en connait plus

Il y a toujours quelqu’un de meilleur que nous, c’est ce qui rend notre travail si intéressant d’ailleurs, inutile donc de prétendre être le meilleur, rien ne justifie d’être condescendant.

Ne réécrivez pas du code sans consultation

Réécrire du code sans en parler est dommageable, c’est une opportunité d’apprentissage perdue. C’est aussi très vexant de se rendre compte que le code qu’on a écrit a été réécrit par quelqu’un d’autre. L’ambiance dans l’équipe risque de se détériorer.

Traitez les personnes qui connaissent moins avec respect et patience

Encore une fois du bon sens, vous n’aimeriez sans doute pas que quelqu’un, qui en sait plus que vous, vous fasse des reproches et vous traite comme moins que lui. Sachez rester humble en toute situation, nous ne sommes tous que peu de chose.

La seule constante, c’est que le monde change

Une vieille citation d’Héraclite sur laquelle il convient de méditer. Dans notre contexte, il s'agit d’accepter que le code va changer, que nous allons changer aussi, les besoins, les collègues…

La vérité provient de la connaissance, pas de la position hiérarchique

L’autorité engendrée par la connaissance force le respect, plus que le grade. Cultiver ses connaissances est le meilleur moyen de gagner le respect dans un environnement collaboratif. N’hésitez donc pas à vous instruire (livre, tech blogs, podcasts, meetups, side projects…) et n’ayez pas peur du conflit avec vos supérieurs, c’est une bonne pratique.

› Battez-vous pour vos convictions, mais acceptez de perdre

Il est important de savoir défendre ses idées, surtout si elles nous paraissent être les plus bénéfiques à l'équipe, au produit. Accepter d'avoir tort et de le reconnaître est toutefois aussi important. Après un bon conflit, en cas de " défaite ", il convient d'accepter l'issue sans amertume, quitte à la documenter pour se donner bonne conscience.

› Ne soyez pas " l'autre là-bas dans la pièce "

Cette personne au fond de la pièce, toute seule dans le noir que personne ne connait et qui ne parle à personne. Le code appartenant à tout le monde, nous avons besoin de pouvoir collaborer.

› Soyez dur avec le code, doux avec les personnes

Ce point est intimement lié au point 2 " you are not your code ", il est important de ne pas pointer du doigt une personne, mais de proposer une amélioration du code. Un proverbe Sri-Lankais dit que lorsqu'on pointe quelqu'un du doigt, il y en a trois qui pointent vers nous (sur la même main, regardez vous-même).

⊙ À vous de jouer !

Prenez le temps de sonder votre égo : à quel niveau se trouve-t-il ? Avez-vous remarqué comme il pouvait changer en fonction des situations, donc des actions de chacun dans l'équipe ?

Un environnement bienveillant est plus attractif, plus productif et plus agréable à vivre. La puissance de la démarche réside dans le fait qu'elle part de la façon de coder et d'interagir avec ses collègues.

Les postures anti-pattern

Extrait de Culture Code

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 par l'architecture et par les 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, de choisir une solution technique et d'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 pénaliseront l'équipe en limitant son fonctionnement collectif et sa progression vers l'excellence technique.

Il tient son idée de génie : en appliquant de la mémoïsation [...]
il peut réduire drastiquement le nombre de calculs.

La gueguerre des étoiles

Au bout d'un certain temps, Gaël acquiert la conviction que quelque chose cloche dans le raisonnement de Chloé. Elle part du principe que le panier se remplit au fur et à mesure et fait le meilleur choix possible à chaque ajout. Mais Gaël connaît bien ce problème : cela revient à faire une optimisation de proche en proche. Or, si cela garantit de tomber sur un minimum local, dans bien des cas, cela n'amène pas forcément au minimum global.

Fort de son propre conseil, il préfère ne pas les déranger jusqu'à avoir développé quelque chose de valable. Quelques minutes plus tard, il trouve son contre-exemple :

- **En partant d'un panier (g2, g2)(un groupe de 2 et un autre groupe de 2)**
- **+1 — (g3, g2)**
- **+1 — (g4, g2)**
- **+1 — (g5, g2)**
- **+1 — (g5, g3)**

Alors que dans ce cas, l'optimum est **(g4, g4)**.

CQFD, un point pour Gaël. Il faut dire que ce second piège est loin d'être facile à trouver. Mais cela ne suffit pas pour gagner la partie contre Chloé : il reste à trouver une solution viable. Il les laisse donc à leur développement pendant qu'il met sur pied sa propre solution.

De toute évidence, l'ensemble des solutions possibles est une combinaison et donc un nombre de solutions à évaluer, évoluant suivant une factorielle, soit de l'ordre de 10^{12} pour seulement 15 livres. Bien trop pour obtenir des temps de calcul raisonnables. Mais n'y a-t-il pas des évaluations équivalentes ? Il se pourrait bien que Paul lui ait donné l'indice décisif avec ses notations sous forme de série. La plupart des évaluations ne diffèrent que par l'ordre, sans changer le résultat.

Il tient son idée de génie : en appliquant de la mémoïsation — c'est-à-dire le fait de garder en cache le résultat de toutes les évaluations précédentes — il peut réduire drastiquement le nombre de calculs. Pour chaque 5-tuple `(a,b,c,d,e)`, la fonction de prix est évaluée au maximum `(a+1).(b+1).(c+1).(d+1).(e+1)` fois.

Et cela, c'est parfaitement acceptable presque indépendamment de la taille de la liste. Il tient enfin son idée de génie ! Sauf qu'il ne lui reste que 40 min pour transformer cela en code.

Maintenant que Chloé, Paul et Farid sont occupés, il pourrait atteindre un Flow similaire à celui de son rêve, si seulement les autres tables ne faisaient pas autant de bruit. Quelle est cette manie de s'exclamer et de rire en travaillant ? Vu le temps restant, ils devraient aussi être en pleine phase de développement de leur solution, ce qui nécessite attention et concentration.

Gaël entend revenir des termes comme “red”, “green” et “refactor”. Ils prennent manifestement cette épreuve à la rigolade.

Dérangé une fois de plus, il lève la tête : pas moins de 12 tables comme la sienne occupent une grande salle de conférence. Ce qui veut dire 12 sociétés concurrentes pour décrocher ce fameux contrat avec Space Inc. La table voisine est de loin la plus bruyante : ils n'utilisent qu'un seul ordinateur, discutent constamment et s'exclament très régulièrement. Gaël entend revenir des termes comme “red”, “green” et “refactor”. Ils prennent manifestement cette épreuve à la rigolade et leur manque de professionnalisme est consternant. Ils ont l'air de trouver l'exercice facile. Pourtant, il est compliqué cet algorithme, et pas sûr que tout le monde soit aussi fort que Gaël en la matière. Si cela doit le retarder au point de ne pas finir, Gaël compte bien les dénoncer et porter réclamation auprès de la commission. Mais pour l'heure, il lui faut s'isoler dans sa bulle.

Quelques minutes avant le coup de sifflet mettant fin à l'épreuve, son code marche enfin. Gaël n'est pas peu fier de lui d'avoir eu cette idée et d'être parvenu à la coder en seulement 38 min. Heureusement qu'il avait déjà utilisé des librairies de mémoïsation par le passé.

— Écoutez, je dois vous présenter une faille dans le raisonnement de Chloé, annonce-t-il à son groupe.

En exposant son contre-exemple, ils sont bien obligés de reconnaître qu'il dit vrai et qu'ils sont tous les trois tombés les deux pieds dans le second piège. L'inquiétude est presque palpable et Farid finit par demander : " Et que va-t-on faire maintenant ? ".

Gaël leur demande de lui faire confiance pour la démonstration au moment même où la fin de l'épreuve est annoncée.

La sécurité psychologique

En 2012, Julia Rodosvsky lança au sein de Google le projet Aristote[7] qui avait pour but de découvrir les modèles et principes généraux qui se cachent derrière les équipes les plus efficaces. Après quelques essais infructueux, ils découvrirent que la capacité de s'exprimer et de prendre des risques sans crainte est une des clés. Pour pouvoir apprendre, nous devons pouvoir exposer nos faiblesses. Nous ne le ferons que si nous sommes dans un environnement qui nous permet de le faire.

Posons une définition : la sécurité psychologique, c'est la capacité à pouvoir s'exposer sans craindre personnellement des conséquences négatives, que ce soit en termes d'image personnelle, de statut ou de carrière.

Dans la notion de sécurité psychologique, on se réfère moins à la sécurité qu'à la confiance : **il s'agit de protéger suffisamment les personnes pour qu'elles se sentent dans un environnement de confiance**. En pratique, nous allons essayer de mettre les personnes dans les meilleures conditions possibles pour qu'elles puissent s'exprimer sans crainte.

Que se passe-t-il sans psychological safety ? C'est ce que nous retrouvons dans le modèle des 5 dysfonctions d'une équipe[8]. **L'absence de confiance et la peur du conflit empêchent l'équipe de grandir**. Cela se caractérise par la préservation d'une harmonie artificielle. Nous sommes loin du principe de congruence personnelle où nous pouvons être nous-même, quelles que soient les circonstances. Nous sommes davantage dans un jeu d'acteur et il devient difficile de discerner le vrai du faux. Le problème potentiel suivant, dans cette situation, c'est de prendre des actions et des décisions sur la base d'informations erronées à la base.

Comment favoriser ce sentiment de confiance ? Nous utilisons le mot favoriser car la confiance ne se met pas en place comme un outil. Une action qui favorise l'installation de la confiance dans une équipe, c'est de permettre et d'**encourager ses membres à exprimer leurs émotions**. Bien trop souvent, nous partons du postulat que le personnel et le professionnel doivent être séparés et que les émotions sont du ressort du personnel.

En réalité, y a-t-il beaucoup de situations plus angoissantes que l'appartenance à un groupe où nous sentons que nous ne pouvons pas exprimer ce que nous ressentons ? Faut-il toujours éviter de dire les choses telles que nous les vivons ? Tout rituel pensé pour favoriser la communication entre les membres d'une équipe est bénéfique. Progressivement, ce type d'action pragmatique construit une confiance solide.

⑦ https://rework.withgoogle.com/guides/understanding-team-effectiveness/steps/introduction/

⑧ Patrick Lencioni, Les cinq dysfonctionnements d'une équipe, Un monde différent, Janvier 2005

Le tout donne vraiment à la pièce un air de salle de classe qui,
il faut le dire, accentue la tension de Gaël.

Hello World ...

Le grand moment arrive enfin, cela fait presque 1 heure qu'ils s'impatientent dans cette grande salle avec interdiction de toucher à leur ordinateur ni même de discuter.

Une personne du staff vient les chercher pour les amener dans une petite pièce voisine où le jury est installé pour auditer chacun des groupes à tour de rôle. Les membres du jury ont aussi assisté à l'épreuve, mais Gaël était tellement absorbé par le problème qu'il n'a fait que les apercevoir lorsqu'ils ont quitté la salle et il peut seulement dire qu'ils étaient quatre.

On leur explique qu'ils doivent entrer sans faire de bruit et attendre que le groupe précédent ait terminé, de manière à enchaîner le plus vite possible pour faire perdre le moins de temps possible aux groupes suivants. Par gestes, Paul leur fait comprendre qu'il commencera la présentation, puis ce sera au tour de Gaël de jouer leur meilleure carte : la résolution du deuxième piège. L'heure d'attente a mis les nerfs de Gaël à rude épreuve. Il se sent bouillonnant, il veut rentrer dans l'arène. L'exercice algorithmique ne lui fait pas peur, il passe en revue les meilleures solutions possibles. Après tout, chaque entreprise a envoyé ses meilleurs développeurs. Et si les autres avaient trouvé une solution plus maligne à ce problème ? Aucune chance, Gaël est sûr de lui.

Et si les autres avaient trouvé une solution plus maligne à ce problème ?

En pénétrant dans la pièce, Gaël peut enfin détailler chaque membre du jury pour anticiper à qui il va avoir affaire. Tous les quatre sont

installés le long d'une grande table sur une petite estrade. Le reste de la pièce est occupé par des tables bien alignées face à l'estrade. Le tout donne vraiment à la pièce un air de salle de classe qui, il faut le dire, accentue la tension de Gaël.

Un petit homme aux lunettes rondes et une femme occupent la place centrale. Très rapprochés l'un de l'autre, ils s'échangent constamment des regards ; Gaël en déduit qu'ils travaillent ensemble au quotidien et, à la vue de la place centrale qu'ils occupent dans les échanges, cela veut forcément dire qu'ils seront impactés par le choix de l'entreprise. Peut-être l'un est-il responsable business et l'autre responsable informatique, les deux aspects que l'on retrouve dans la plupart des projets.

Sur leur droite, se tient une femme portant un tailleur marron et de petites lunettes carrées. Ses cheveux blonds sont noués en un chignon parfait. Elle se tient droite comme un "i" sur sa chaise et prend minutieusement des notes. Un très grand sérieux se dégage d'elle. Gaël l'imagine en qualiticienne. S'il la joue finement, la démarche scientifique qu'il compte présenter aura sûrement de quoi la séduire.

Complètement sur la gauche, est assis un homme assez imposant, cheveux châtains et sourcils épais. Il porte un costume bleu assorti d'une cravate et se tient en arrière sur sa chaise, les mains croisées sur le ventre. Sa posture le place clairement en retrait par rapport à ses trois acolytes, et il ne participe pas à leurs échanges. Et pourtant, Gaël en est sûr, il est de ces gens dont le regard vous transperce et qui ont le don d'appuyer là où ça fait mal. Gaël le sent, il devra se méfier de lui et guetter ses réactions.

Yes ! Le jury est conquis. Il ne reste plus qu'à transformer l'essai pour remporter la partie.

Il ne peut en rester qu'un

Quelle chance : le groupe en cours d'audition est le groupe ultra bruyant qui était installé à la table voisine pendant l'épreuve. Gaël espère vivement que le jury se montrera intransigeant avec eux pour leur manque de professionnalisme. D'ailleurs, leur résultat doit être à la hauteur de leur manque de concentration.

Malheureusement, Gaël déchante rapidement : non seulement, ils ont trouvé et résolu le piège de l'ordre préféré des groupes, mais en plus le jury ne semble pas leur tenir rigueur de leur comportement, bien au contraire. Gaël n'en revient pas, à aucun moment, le jury ne met en défaut ce groupe sur leur optimisation de proche en proche. Se pourrait-il que même le jury n'ait pas décelé cette faille dans le raisonnement ? Si c'est vraiment le cas, Gaël a une carte majeure à jouer qui leur assurera la victoire.

Contre toute attente, le duo central du jury conclut en les remerciant vivement pour cette présentation qui pour eux est de loin la meilleure depuis le début des auditions.

Ils ont trouvé et résolu le piège de l'ordre préféré des groupes, mais en plus le jury ne semble pas leur tenir rigueur de leur comportement, bien au contraire.

Si c'est vraiment le cas, celle de Gaël est encore mieux et aucun doute qu'ils vont être bluffés.

— Groupe suivant, c'est à vous, vous avez 5 min pour exposer votre solution et s'ensuivront 5 min de questions, annonce la femme au centre.

Paul explique de manière très claire et très concise les notations adoptées et la manière dont ils ont découvert et formulé la notion d'ordre préféré. Il conclut en 2 minutes et 13 secondes par :

— Mais nous n'étions pas au bout de nos surprises et Gaël va vous présenter la suite.

Gaël commence :

— Merci Paul. Bien que brillante, la notion d'ordre préféré s'appuie sur l'hypothèse que l'optimum puisse être trouvé en ajoutant les livres un par un au panier et en faisant le meilleur choix à chaque ajout ; ce qu'on pourrait qualifier d'optimisation de proche en proche.

Froncements de sourcils de la part du binôme central mais aussi de la présumée qualiticienne à droite. Cela semble confirmer que ce point n'a pas été anticipé, il a donc bien une carte maîtresse à jouer. Il explique alors son exemple en partant de `(g2, g2)` pour aboutir à `(g5, g3)` au lieu de `(g4, g4)`.

La femme au centre intervient :

— Je dois dire que je suis très surprise. C'est un exercice que nous avons joué à plusieurs reprises en interne et ni à ces occasions, ni les 6 groupes avant vous n'ont jamais évoqué ce point. Mais continuez, je vous prie, vous avez toute notre attention.

Yes ! Le jury est conquis. Il ne reste plus qu'à transformer l'essai pour remporter la partie. Il utilise alors toute sa capacité de vulgarisation pour expliquer en quoi la mémoïsation rend accessible dans un temps presque linéaire le parcours de la totalité des solutions.

Il conclut en affichant le code correspondant à cette solution, code qu'il s'est bien sûr attaché à rendre le plus concis possible pendant l'épreuve. Être pris par le temps est une chose, rendre du code non épuré en est une autre et Gaël ne fait pas partie de ces gens là.

Qu'est-ce qu'un bon code ?

Le demi-cercle, épisode 1[9]

Ça commence par la naïveté, par une sorte d'ignorance bénie. On crée du code sans être conscient des conséquences, et de la nécessité d'un retour d'information sur ce code. On bâtit naïvement une tour, avec ce qu'on trouve ici et là. Quand la tour frémit, on devient soudain extra-prudent, mais alors on prend un peu plus de temps pour chaque chose, et on subit la pression des clients et du management.

La tour s'écroule, on la remonte, à la hâte. On n'a pas le temps d'assurer la qualité parce qu'on a trop de problèmes imprévus à résoudre, parce qu'on a un faible niveau de qualité, parce qu'on n'a pas le temps d'assurer la qualité. On perd son calme, on agit de façon précipitée, désordonnée, irrationnelle.

On n'observe pas ce qu'il faudrait, on ne mesure pas les bons phénomènes et on en tire par conséquent des conclusions fausses, imaginaires. On recherche ce qui cloche en amont, dans les estimations, dans les prérequis, au lieu d'étudier ce qui nous ralentit le plus. On développe une sorte de mauvaise foi, qui sert à tout expliquer. On est sur la défensive.

Pendant ce temps, le code s'enfonce doucement, mais sûrement, dans l'entropie. Fuite en avant.

...

Les gestes de base, on les évoque : " il aurait fallu... ", on liste les " best practices ", on produit des présentations et on prescrit des " transformations ". Pourtant, on sait parfaitement — puisqu'on serait incapable, comme ça, à brûle-pourpoint, de restaurer un fauteuil Louis XVI, d'exécuter une sonate, ou de dispenser des soins après un pontage coronarien — on sait parfaitement

qu'il faut dix ans pour faire un expert

... c'est-à-dire que les gestes de base requièrent du temps pour devenir l'habitude, le savoir profondément ancré qui distingue le professionnel de l'amateur même passionné.

[9] https://blog.octo.com/le-demi-cercle-episode-1/

Qu'est-ce qu'un code propre ?

Les piliers[10]

Le " Clean Code " regroupe plusieurs règles et principes pour vous aider à construire, mais surtout refactorer votre code. En effet, comme le disait Michel[11] dans son article sur les artisans du code, le respect de ces différentes règles énoncées par Bob Martin a pour but d'offrir à votre code, entre autres, simplicité, lisibilité et structuration pour qu'il soit le plus évolutif et maintenable possible sur le long terme.

Voici des piliers qui vous soutiendront dans votre pratique de l'amélioration de la qualité de votre code.

Les piliers de la pratique du Clean Code

L'amélioration continue

Avant toute chose, pour avoir le code de la meilleure qualité possible, il est nécessaire que l'ensemble de l'équipe soit convaincu de la mise en place d'une pratique d'amélioration continue. Sans ça, tout bon Lead Dev/Tech Lead que l'on soit, la plupart des pratiques que l'on poussera seront malheureusement vite abandonnées…

Tests automatisés

Afin de s'assurer que toute action de refactoring n'a aucune incidence sur le fonctionnement de l'application, il est nécessaire de pouvoir s'appuyer sur un ensemble de tests automatisés suffisamment exhaustif pour remonter le moindre problème. Avant tout refactoring, il est donc nécessaire d'ajouter des tests en amont.

IDE " refactoring-friendly "

L'IDE[12] a une importance non négligeable dans la pratique du Clean Code. En effet, un bon IDE doit nous permettre de faire des micro-refactorings sans douleur (renommage ou extraction d'une variable ou d'une méthode, par exemple), voire même de les faciliter fortement (répercussion automatique, aide à la décision, etc.).

À l'opposé, un IDE trop basique va avoir tendance à provoquer l'inverse. En effet, si le refactoring est douloureux ou laborieux, le développeur aura souvent tendance à abandonner la pratique.

Boy Scout Rule

" À chaque fois que je parcours du code, je dois essayer de l'améliorer. " Cette amélioration peut être seulement une indentation, un saut de ligne ou le renommage d'une variable, du moment qu'il permet de rendre le code plus lisible, plus compréhensible.

Revue de code et/ou pair programming

En plus de la Boy Scout Rule, le meilleur moyen pour détecter un problème dans votre code,

⑩ https://blog.octo.com/ecrire-du-code-propre-les-piliers/
⑪ Les artisans codeurs chez Octo. Michel Domenjoud. https://blog.octo.com/les-artisans-codeurs-chez-octo/
⑫ Integrated Development Environment

c'est d'avoir l'avis d'un autre. Que ce soit via du pair programming où vous aurez du feedback en direct ou via de la revue de code, par paire ou en équipe, c'est vraiment la pratique qui fera la mieux apparaître les axes d'améliorations de votre code. En effet, pour soi, quand on vient de finir une tâche, il est souvent difficile de prendre suffisamment de recul pour critiquer son travail parce qu'il est encore frais dans notre tête. À ce moment-là de notre travail, notre code est souvent " simple " et " clair ".

› Analyse de code automatisée

La mise en place d'un outil d'analyse de code, type SonarQube, va vous permettre de remonter toutes les violations aux standards du langage/du framework que vous utilisez (amendé en général pour correspondre aux standards de l'équipe), ainsi que le taux de duplication de votre code. Ces indicateurs vont souvent ainsi être les pointeurs d'un endroit où vous pourrez appliquer les principes du Clean Code.

Ce genre d'outil fournit en général un premier bon indicateur sur la maintenabilité du code. Un standard non respecté, c'est déjà une difficulté pour y rentrer et pour le modifier, le corriger. D'ailleurs, je ne saurais que trop vous conseiller de commencer votre analyse en utilisant les standards définis par défaut. En effet, ceux-ci correspondent généralement aux standards attendus par la plupart des développeurs du langage/framework. Ça permet donc une meilleure entrée dans le code pour un nouvel arrivant.

Ces outils ne sont malheureusement pas suffisants. Les manques les plus flagrants concernent la pertinence des nommages (les variables/les méthodes/les classes sont-elles bien nommées ?), la structure des méthodes/des classes (cette méthode/cette classe fait-elle trop de choses ?) et l'architecture du code (est-ce que j'utilise le bon design pattern ? Est-ce que j'hérite bien de la bonne interface ?).

Pour combler ces manques, il est nécessaire de mettre en place des revues de code !

Quand doit-on refactorer ?
Savoir détecter que notre code a un problème

Les piliers[13]

En plus de l'utilisation d'outils d'analyse de code, il existe quelques indices simples pour nous aider à déterminer si notre code doit être refactoré :

• Des méthodes trop longues (au-delà de ~20 lignes) ;

• Des classes trop longues (au-delà de ~200 lignes) ;

• 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 (qui dénote souvent une violation du principe de la programmation orientée objet) ;

• Plus de 2 niveaux d'indentation dans une méthode ;

• Des objets sans méthodes (en-dehors des getters/setters dans un contexte de POO).

Plus généralement, faites confiance à votre intuition : 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, amélioré, refactoré !

[13] https://blog.octo.com/ecrire-du-code-propre-les-piliers/

Bien, puisque vous ne semblez pas comprendre, je vais aller droit au but :
vous ne nous avez rien dit sur les tests que vous avez utilisés
et sur la démarche que vous avez suivie.

Jusqu'ici tout va bien

Cinq minutes, tout pile, on ne peut même pas lui reprocher d'avoir dépassé le timing malgré une explication si poussée. Lorsqu'il observe son jury, il constate qu'ils sont tous stupéfaits. Seul l'homme à gauche affiche une expression neutre, peut-être un regard vaguement amusé. Et toujours dans cette posture en arrière avec les mains croisées sur son ventre. L'affaire est dans le sac et, il en est sûr, cela va faire du bruit tout autant au sein de son entreprise, BlueSoft, qu'au sein de Space Inc. D'ailleurs, le regard admirateur des trois membres de son propre groupe peut en témoigner.

Les membres du jury se regardent les uns les autres sans prononcer un mot.

Alors que le silence devient franchement gênant, l'hypothétique qualiticienne demande :

— Est-ce tout Monsieur ?

— C'est-à-dire ?, bafouille Gaël totalement pris au dépourvu par la question.

— N'avez-vous rien d'autre à nous présenter ? Êtes-vous sûr de ne rien avoir oublié ?, insiste-t-elle.

D'après les hochements de tête du binôme au centre, il sent qu'il est en train de rater quelque chose d'important. Et s'il était allé trop vite, avec ce satané timing ?

— Euh, non je ne pense pas, mais peut-être voulez-vous que je revienne plus en détail sur une partie de l'explication en particulier ?

Cette fois, elle paraît nettement plus énervée :

— Bien, puisque vous ne semblez pas comprendre, je vais aller droit au but : vous ne nous avez rien dit sur les tests que vous avez utilisés et sur la démarche que vous avez suivie.

Vous ne nous avez rien dit sur les tests que vous avez utilisés et sur la démarche que vous avez suivie.

— Comme énoncé par Paul, nous avons essayé diverses combinaisons et j'ai ensuite trouvé un exemple qui mettait en défaut l'optimisation de proche en proche, commence-t-il.

Au tour de l'homme du milieu d'intervenir :

— Bien, montrez-nous ces tests !

Où veulent-ils en venir ?

— Oui bien sûr. Paul, peux-tu projeter à nouveau les tableaux que tu as présentés en première partie ?

L'homme intervient de nouveau :

— Je pense que nous avons du mal à nous comprendre. Nous voulons voir le code des tests et le rapport d'exécution.

Si Gaël ne parvient pas à retomber sur ses pieds rapidement, l'affaire peut tourner au fiasco pour un simple malentendu.

— Euh oui oui, pardon pour la méprise. Voici le code d'exécution et voici le script permettant de lancer l'application. Comme vous le voyez, on peut saisir son exemple ici et voir dans les logs la solution choisie, son prix et le temps d'exécution, explique-t-il.

Un point de sémantique : le test et la vérification

L'image traditionnelle que nous avons des tests en développement logiciel est la suivante :

• Une fois la phase de programmation du logiciel considérée comme terminée, des testeurs, qui appartiennent généralement à un département nommé "Assurance Qualité", effectuent des séries d'essais du logiciel, guidés en cela par des documents dits "cas de tests", voire même "règles de gestion", lesquels ont été rédigés à partir des spécifications détaillées du logiciel ;

• Ces testeurs décrivent ensuite, via des rapports destinés aux développeurs et au management de projet, les défauts qu'ils ont trouvés.

Si cette image, qui fait référence aux tests communément appelés "tests manuels", a pu représenter la réalité des tests dans le passé, ce n'est plus le cas depuis le début des années 2010.

Les méthodes agiles ont transformé la façon de développer un logiciel, en mettant notamment l'accent sur :

• La **collaboration** ;

• L'importance du **feedback** et des délais courts ;

• L'enjeu crucial que représente une **chaîne d'intégration continue efficiente**.

Une étape essentielle de cette intégration continue consiste à exécuter plusieurs centaines, voire plusieurs milliers, de tests auto-vérifiants, de manière à **détecter rapidement toute régression** dans le comportement du système par rapport à la version précédemment déployée.

Est-ce à dire que tester le logiciel est devenu une activité entièrement automatique ? Loin s'en faut.

Lorsque nous parlons de tests automatisés, il serait en fait plus judicieux et pertinent de parler de **vérifications automatisées** (checks). La seule "assurance" produite par un test automatisé (et c'est déjà beaucoup), c'est que le comportement implémenté par certaines parties du code et les résultats de ce comportement sont conformes à ce qui est écrit dans le code de test, à savoir (grosso modo) **l'expression de l'intention des développeuses et des développeurs.**

Or tester un logiciel est une activité plus vaste que la simple vérification de conformité entre le code et l'intention exprimée à travers les tests. On doit donc désigner séparément deux activités dans la définition desquelles le mot "test" apparaît, mais qui pour autant n'ont pas le même but immédiat :

1. Tester un logiciel : essayer un logiciel en vue d'identifier des problèmes qui menacent sa valeur ;

2. Écrire des tests automatisés : construire un code qui vérifiera automatiquement certains aspects du comportement du système.

L'activité 1 est plutôt celle des testeurs et l'activité 2, celle des développeuses et des développeurs.

Que cherche-t-on quand on écrit un test automatisé ?

Le test automatisé est le reflet de plusieurs exigences :

- Ce que le code est censé faire ;
- Ce qu'il y a dans la spécification ;
- Ce que la personne qui développe a compris.

"It's not the domain expert's knowledge that goes into production, it's the developer's assumption of that knowledge." Alberto Brandolini

Bien entendu, les développeurs avertis le savent, et ne manquent pas de tester leur logiciel au sens 1, et parfois un test visuel est le seul moyen efficace et rapide pour le développeur d'assurer la non-régression. Évidemment, la plupart des testeurs utilisent et mettent en place des vérifications automatisées.

En agile, presque toutes les vérifications sont automatisées et presque toute l'activité de test est exploratoire.

Les progrès réalisés en matière d'outillage et de méthodologie nous incitent à conclure que des centaines de tests automatisés, réalisés par les développeurs, rendent caduque la nécessité de tester "manuellement" le logiciel.

It's not the domain expert's knowledge that goes into production, it's the developer's assumption of that knowledge.

Alberto Brandolini

Rien n'est plus faux : certes, les équipes de développement agiles assurent une construction mieux défendue contre les régressions possibles, mais elles le font sur la base de leurs propres interprétations issues de la collaboration avec les parties prenantes et non sur la base de spécifications détaillées.

Cette distinction entre vérification automatisée et activité de test est cruciale en termes de réussite de vos développements logiciels. De même qu'aucune équipe de "testeurs manuels" ne pourrait réaliser un travail de vérification avec la même efficience qu'une suite de tests automatisés, aucune réalisation, même "couverte" à 100 % par des tests automatisés, n'est à l'abri d'erreurs de conception, de conditions inattendues (i.e : qui ne figuraient pas dans le modèle mental de ses concepteurs), ou d'hypothèses irréalistes.

La place des tests dans le développement logiciel

Que les moments soient séquentiels ou parallèles, on peut en distinguer 4 majeurs dans la vie d'un logiciel. Si ces 4 moments peuvent se mêler, ils sont toujours bien présents. À chacun, on peut associer une typologie générale de tests :

- **La découverte du besoin et son exploration** : product management, tests utilisateurs des idées (MVP, UX…) ;
- **La construction du logiciel** (build) : conception, programmation, vérifications, tests automatisés ;
- **La recherche de problèmes** (test) : essais, tests exploratoires ;
- **Le logiciel en production** (run) : monitoring & supervision.

En parallèle de ces moments, il y a tout au long de la production des feedbacks sur le produit, des feedbacks sur la technique, un apprentissage sur le métier, sur l'organisation, du pair-programming, des mesures, des observations…

Comme annoncé dans l'introduction, dans ce livre, nous allons nous focaliser sur les tests pendant la phase de construction (build), c'est-à-dire principalement sur les tests automatisés.

— Monsieur, vous nous faites perdre notre temps..., s'agace l'homme aux lunettes rondes.
— Avant de les congédier, je souhaiterais vérifier quelque chose, le coupe sa voisine du centre.

Un plan sans accroc

Gaël est fier du code d'exécution qu'il présente au jury. Pourtant, la réponse est immédiate :

— Monsieur, vous nous faites perdre notre temps..., s'agace l'homme aux lunettes rondes.

— Avant de les congédier, je souhaiterais vérifier quelque chose, le coupe sa voisine du centre.

Est-ce qu'il venait vraiment de se faire disqualifier pour une mécompréhension et sans justification ? Alors qu'il venait de pointer du doigt une faille majeure dans toutes les solutions établies jusque-là ?

— Jeune homme au fond, pouvez-vous m'expliquer en détail le code de cet algorithme ? Vous êtes bien développeur n'est-ce pas ?, demande-t-elle en désignant Farid.

— Euh, oui... Je suis bien développeur. L'algorithme parcourt l'ensemble des solutions possibles tout en mémorisant en cache chaque évaluation de prix...

— Non non non, j'ai demandé en détail. Par exemple, expliquez-nous ce que fait la ligne 12, reprend-elle.

— Alors, elle semble filtrer certaines solutions. Peut-être que ce sont des solutions non valides, tente-t-il.

— C'est bien ce qu'il me semblait. En 2 heures, ce Monsieur a réussi l'exploit de produire un code inexplicable et inmaintenable. Ce sera tout pour moi aussi, ajoute-t-elle.

En 2 heures, ce Monsieur a réussi l'exploit de produire un code inexplicable et inmaintenable.

Non mais c'est quoi cette histoire à la fin !

En quoi le fait que Farid sache expliquer ou non la ligne 12 change quelque chose à la situation ? C'était à lui, Gaël, qu'il fallait demander. Lui sait parfaitement expliquer cette ligne.

L'homme au centre reprit :

— Bien, nous ne vous retenons pas plus longtemps…

Il suspend sa phrase lorsque l'homme à gauche sort de son immobilisme pour se pencher en avant. Les trois membres du jury le regardent et ne disent plus un mot. Il prononce quelques mots à leur intention, tellement doucement que Gaël ne peut en saisir aucun. Puis il reprend tranquillement sa posture en arrière.

Dans un mélange de honte et de colère contenue, l'homme du centre se reprend :

— Oui… euh… bon… comme on vient de nous le rappeler, vous avez accompli un acte remarquable en détectant une faille dans les solutions actuelles et cela mérite une sorte de récompense.

Il prend un long moment avant de prononcer la suite, comme si ça lui coûtait et qu'il se faisait violence pour prononcer ces mots :

— Nous avons donc décidé de vous laisser poursuivre l'aventure. Vous êtes sélectionnés pour la suite.

Avant qu'ils puissent se réjouir de cet heureux retournement de situation, la femme au centre déclare :

— Mais que cela soit bien clair, qu'importe ce que diront mes collègues (et elle lance un coup d'œil provocateur à l'homme mystérieux), si vous présentez un travail de ce genre la prochaine fois, vous serez disqualifiés. Je vous rappelle que nous voulons bâtir un partenariat pour les trois années à venir et nous ne cherchons pas des héros qui vont résoudre seuls les problèmes. Cela représente des missions pour des dizaines d'équipes. Le travail, les pratiques et les valeurs d'équipes en sont donc des éléments fondamentaux.

Nous ne cherchons pas des héros qui vont résoudre seuls les problèmes.

Prenez exemple, si vous le pouvez, sur le groupe qui vous a précédé [Non ? Pas encore eux ?]. Si vous n'êtes pas convaincants sur l'un de ces points, comptez sur moi pour vous opposer un refus formel. Est-ce suffisamment clair ?

Le groupe de Gaël ne peut que hocher la tête tant ils sont secoués.

Mais c'est sans compter sur la qualiticienne :

— Sachez que je partage totalement le sentiment de ma collègue. Et je rajouterais que je parlais bien évidemment tout à l'heure de tests automatisés. Vous avez donc plutôt intérêt à être également irréprochable sur ce sujet. Je ne sais pas dans quel monde vous vivez, mais sachez qu'ici à Space Inc., ne pas avoir une suite de tests qui couvre l'intégralité du logiciel est considéré comme un manque flagrant de professionnalisme. Si vous n'êtes pas en mesure de prouver le fonctionnement de votre logiciel à la soutenance finale, mieux vaut ne pas vous présenter.

Une source de vérité unique et partagée

Partons du principe que vos spécifications et votre code sont alignés et imaginons ce qu'il se passe dans la vision classique du développement. À la première correction dans le code, les spécifications ne sont plus alignées et la mise à jour nécessite un gros travail, qui n'est souvent pas fait tout de suite, puis au fil des mois, il devient impossible de rattraper cet écart. Finalement, on passe beaucoup de temps à rédiger des spécifications qui finissent par ne jamais être à jour.

Résultat : les développeurs ne regardent pas les spécifications, car elles ne disent pas comment fonctionne l'application et le métier, tout comme les testeurs ne peuvent pas se baser sur les spécifications, car ces dernières ne sont pas à jour.

Que se passe-t-il maintenant si l'on considère que les tests, définis en amont entre le métier et l'équipe de développement, sont les spécifications de notre logiciel ? Alors, un premier miracle se produit : les spécifications sont toujours à jour !

Peu après, on s'aperçoit d'un second miracle : alors même que l'on avait un combat entre le métier, pour qui la vérité était dans les spécifications, les testeurs, dans leur suite de tests, et les développeurs dans le code, **tout le monde adopte une seule et unique source de vérité : LES TESTS.**

Pour que cette source de vérité soit la mieux exploitée, les développeuses et développeurs doivent rendre les noms/descriptions des tests les plus lisibles et proches du langage métier possible. Dans tous les cas, c'est un exercice qui demande assiduité et évangélisation.

Est-ce qu'aller plus vite est plus économique ?

Parfois, le coût des tests fait débat : pour beaucoup de personnes et lorsque ce n'est pas naturel pour elles, les tests peuvent prendre un temps certain à écrire et le temps, c'est de l'argent ; et si écrire des tests coûtait trop cher ? **Les développeurs iraient plus vite s'ils n'écrivaient pas tous ces tests**.

Les arguments sont divers :

— Les développeurs ont l'air de se faire plaisir en écrivant tous ces tests ;

— Pourquoi ils scandent "red, green, refactor" tout le temps ? On dirait qu'ils s'amusent alors que la deadline approche et qu'on lutte pour obtenir des rallonges sur les échéances ! Ces tests ne servent à rien ! ;

— Les tests pourraient être réalisés par une équipe moins chère…

Si les tests sont moins nombreux, s'ils ne lèvent pas d'erreur ou s'ils sont réalisés par une autre équipe, alors ces tests ne permettent pas de construire le code. Ces tests là, indépendants de la phase de construction du code, seraient réalisés par principe. Ce serait comme passer la serpillière avant de passer le balai : on finit par déplacer de la saleté humide ; c'est toujours sale et il n'y a plus qu'à recommencer en passant d'abord le balai pour que ça soit réellement propre.

Mettons-nous d'accord : si les tests ne permettent pas d'améliorer la qualité du code, alors oui, ils ne servent pas.

Tout dépend de la manière dont sont écrits les tests. S'ils sont écrits avant chaque développement et donc s'ils sont tout le temps associés à l'activité de développement, alors ils permettent de :

• Construire le code ;

• Bâtir des défenses à court terme, voire à moyen-long terme ;

• Oublier : les libellés des tests écrits avec du vocabulaire métier sont une documentation compréhensible très utile pour le développeur, ils permettent d'oublier. Ces libellés de tests évoluent avec le code ; ils sont une documentation vivante.

Ces tests-là sont plus économiques, car ils permettent de solidifier le code, assurément.

Lien entre tests et qualité logicielle

Les équipes de testeurs et testeuses sont souvent appelées QA, Quality Assurance, ce qui montre bien que le lien est fait, au moins dans l'esprit de beaucoup, entre les tests et la qualité des logiciels délivrés. Il y aura qualité logicielle si le résultat de la phase de test est probant. Est-ce de bonne qualité s'il y a beaucoup de bugs **détectés ? Peu de bugs détectés ? L'objectif de la QA est de tester la qualité, alors que le test peut aussi servir à produire de la qualité.** Et la mesure n'est pas bien loin : nombre de bugs détectés, nombre de tests passés (sur un nombre défini a priori), etc.

Se pose déjà ici la question épistémologique de la mesure quantitative de la qualité : quantitatif vs. qualitatif. Sans entrer plus dans cette réflexion, nous vous demandons de la garder en tête pendant toute la lecture du présent ouvrage.

La qualité doit être "built-in", ce n'est pas une chose à ajouter, mais à embarquer au plus tôt.

Je profite de cette mise au point pour vous signaler qu'en plus,
votre solution ne fonctionne pas.

A touché le fond, mais creuse encore

Grâce à l'intervention de ses collègues, l'homme du centre reprend contenance et finit de les achever :

— Je profite de cette mise au point pour vous signaler qu'en plus, votre solution ne fonctionne pas.

Autant Gaël veut bien reconnaître l'avoir joué perso et ne pas avoir mis en place de tests automatisés, mais il ne peut pas se laisser attaquer sur le code sans broncher. Malgré les signes de Paul l'invitant à se taire, il rétorque :

— Je regrette, je suis certain qu'elle fonctionne.

Vous vous permettez d'introduire massivement de la complexité dans le code sans vous préoccuper des conséquences.

— Vraiment ? Étonnant cette certitude, alors que vous êtes incapable de prouver quoi que ce soit. Pouvez-vous me dire quelle est la mémoire disponible sur nos caisses enregistreuses ? À quel moment votre module de mémoïsation va-t-il la saturer ?, questionna le petit homme.

Non mais sérieusement, ne sont-ils pas capables de s'exprimer clairement sur au moins un sujet ?

— Caisses enregistreuses ? L'énoncé ne mentionne aucune caisse enregistreuse.

— Ah bon ? Pourtant, des experts étaient dans la salle pour répondre à toutes vos questions. Et cela démontre bien

mon point : nous ne sommes pas à l'école devant un énoncé. Vous vous permettez d'introduire massivement de la complexité dans le code sans vous préoccuper des conséquences. Notre rôle, à Space Inc., est de construire des systèmes complets. Et c'est la responsabilité de tous d'envisager la totalité des répercussions. Et c'est bien sûr ce que nous exigeons de nos partenaires.

Sur ces propos accablants, le jury les congédie pour passer au groupe suivant.

Parler avec des experts métier : utiliser le BDD

Décrire une règle de gestion ne suffit pas à générer une compréhension partagée[14].

L'idée que tout le monde se fait du BDD est que c'est un outil qui produit des tests automatisés. Or, comme toute bonne pratique craft qui se respecte, le BDD ne commence pas du tout par du code, cela commence par des conversations. Le BDD consiste à prendre en entrée des règles métier et à en parler en l'illustrant par des exemples. En l'illustrant d'une certaine manière, nous découvrons beaucoup de choses sur cette règle métier. À la base, c'est une conversation au minimum entre un développeur et une personne du métier et c'est encore mieux s'il y a un testeur (tres amigos). L'illustration avec des exemples permet de trouver différentes choses.

Première question : ce que l'on ne sait pas sur cette règle métier.

La conversation est sur des exemples :

— On a cette règle métier, peux-tu me donner un exemple ? C'est quoi cette date d'expiration ?

— C'est le 17/08/2017.

L'exemple est très précis, il y a des valeurs, des noms, des dates. On s'accroche à cette date et il y a des choses qui apparaissent :

— Que se passe-t-il le 16 ? Le 18 ? Le 17 ? Cela se passe où ?

Le fait de mettre des valeurs nous ouvre l'esprit sur les différentes possibilités qui sont cachées derrière cette règle.

Deuxième question : voit-on une situation où ça ne se passerait pas comme ça ?

C'est là que l'on va essayer de découvrir ce qu'il y a autour de cette règle de gestion. On peut découvrir, par exemple, que la date d'expiration n'est valable que sur un certain trimestre et qu'elle a un autre sens sur un autre trimestre, par exemple.

— Si la date est en août il fait plus chaud, le produit va périmer plus tôt.

— Si la date est en décembre, il fait plus froid et donc pas de problème de péremption.

Dans ce cas, on découvre une situation où l'exemple est faux, où l'on trouve ce qui ne fonctionne pas.

Les exemples sont des cas concrets qui permettent de découvrir ce qui se passe autour. Parfois, on découvre trop de choses et la conversation peut aboutir à une négociation sur l'ensemble des règles à prendre en compte.

On reconnaît un bon exemple BDD au fait que l'on peut voir comment l'application doit se comporter.

Un anti pattern est de parler interface utilisateur "quand je clique là que se passe-t-il ?". BDD parle de règle métier, pas d'interface utilisateur.

Puis viennent les outils pour automatiser les tests. Il y a tous les outils Gherkin : Cucumber, Behat, SpecFlow…

⑭ Behavior Design Development - Cédric Rup : https://blog.octo.com/le-bdd/

Tu sais, ton contre-exemple… J'ai bien peur qu'il ne fonctionne plus dès lors qu'on prend les livres dans l'ordre des tomes.

Ça va être tout noir !

Une fois dans le couloir Gaël laisse libre cours à sa colère :

— Qu'est-ce que c'est que ce cirque ? On résout le problème, mieux qu'ils ne l'espéraient, et c'est comme ça qu'ils nous remercient ? Et puis je ne suis pas un héros ! J'aime me concentrer sur mon travail, point !

— Écoute Gaël, ce n'est pas un jugement personnel envers toi. Space Inc. est une bien plus grosse entreprise que la nôtre et leurs pratiques sont différentes, voilà tout. Nous devons juste les comprendre et nous adapter et cela devrait bien se passer, tempère Chloé.

Gaël se rend compte que Chloé a d'autres choses à lui dire, mais n'ose pas devant son humeur hargneuse.

— Quoi ?!, l'invective-t-il.

Devant cette question directe, elle n'eut d'autre choix que de lui répondre :

— Tu sais, ton contre-exemple… J'ai bien peur qu'il ne fonctionne plus dès lors qu'on prend les livres dans l'ordre des tomes. Regarde :

Tome	G1	G2
un	1	1
deux	1	1
trois		
quatre		
cinq		

Tome	G1	G2
un	1	1
deux	1	1
trois	1	
quatre		
cinq		

Tome	G1	G2
un	1	1
deux	1	1
trois	1	1
quatre		
cinq		

Tome	G1	G2
un	1	1
deux	1	1
trois	1	1
quatre	1	
cinq		

Tome	G1	G2
un	1	1
deux	1	1
trois	1	1
quatre	1	
cinq		1

Gaël n'en revient pas, son raisonnement semblait parfaitement bon :

— Attends, tu veux dire que le seul truc qui vient de nous sauver la mise est une erreur de logique de ma part ? Que tout ce que je viens de démontrer est faux ? Mais… que va-t-il se passer quand ils vont s'en rendre compte ?

— C'est bien ce qui m'inquiète, répondit Chloé.

Tout ce que je viens de démontrer est faux ?

Biais d'over-engineering

Vous êtes-vous déjà rendu dans un parc pour enfants ? Le niveau d'agression sonore y est incroyable, avec des cris stridents entremêlés de pleurs, d'animateurs qui parlent au mégaphone et de musique souvent pas particulièrement plus agréable que le reste. Dans ce genre d'environnement, nous avons une tendance naturelle à nous mettre à parler beaucoup, beaucoup plus fort que d'habitude (ce qui n'arrange franchement rien).

Le parallèle peut être fait avec l'environnement dans lequel évoluent les développeurs : c'est un environnement très complexe, que ce soit au niveau du ou des langages qu'ils utilisent, qui sont très stricts et normés, des concepts qu'ils manipulent, des machines dont ils se servent, des environnements dans lesquels ils déploient, etc. À cause de ce même environnement, il est très compliqué pour les développeurs de réaliser du code simple. Un biais naturel apparaît et nous pousse à complexifier inutilement nos solutions et nos scénarios.

Tout l'enjeu de la pratique des tests (et notamment le test first[15]) est de nous pousser à simplifier notre manière de travailler afin de simplifier notre code.

⑮ Test first: pratique qui consiste à écrire un test avant d'écrire le code qui lui correspond

Le lendemain matin, Gaël arrive au bureau épuisé.
Il a enchaîné cauchemar sur cauchemar. Dans certains, son code de la veille s'en prend à lui et tente de le tuer de diverses manières

Votre mission, si vous l'acceptez

Le lendemain matin, Gaël arrive au bureau épuisé. Il a enchaîné cauchemar sur cauchemar. Dans certains, son code de la veille s'en prend à lui et tente de le tuer de diverses manières, sous le motif qu'il lui a donné vie sans raison d'être et qu'il erre, inutile, dans les limbes du réseau. Dans d'autres, Space Inc. le poursuit en justice pour escroquerie et abus de confiance et il se retrouve en prison au milieu de délinquants. Brrr... Rien que d'y penser, il en a encore des frissons.

L'ouverture de sa boite mail lui permet enfin de chasser tout cela de sa tête. C'est un mail de Space Inc. :

Bonjour,

Félicitations pour avoir réussi l'épreuve d'algorithmique hier. Nous avons sélectionné 5 entreprises, parmi les 12 en course, pour la seconde et dernière étape de ce concours.

Comme vous le savez désormais, Space Inc. s'est lancé un objectif très ambitieux de transformation, pour devenir leader sur le marché. Une brique essentielle de ce plan est de changer en profondeur la relation que nous avons avec nos prestataires. Nous ne voulons plus des exécutants, mais des personnes capables d'œuvrer avec nous, de nous proposer des idées innovantes et de nous secouer parfois.

Ainsi, bien au-delà de vos compétences dans le développement logiciel, nous serons

particulièrement attentifs à :

• La pérennité de vos solutions,

• Votre force de proposition,

• Vos pratiques d'équipes.

Nous avons conscience que cela demandera un effort majeur et qu'il ne peut s'envisager qu'en sécurité et dans la durée. C'est pourquoi le vainqueur de ce concours se verra attribuer l'exclusivité de tous nos marchés sur les trois ans à venir.

Cette dernière épreuve se déroulera sur 3 semaines et l'audition aura lieu le 7 mars dans nos locaux. Les heures de passage vous seront communiquées quelques jours avant.

Voici le descriptif du sujet de cette épreuve :

Dans le cadre du lancement de nos satellites d'observations nouvelle génération, nous souhaitons rendre disponible certaines informations recueillies à travers une application mobile.

Cette application pourra être utilisée par l'ensemble du personnel de Space Inc.

Dans un souci de partage, nous souhaitons également l'ouvrir au grand public et elle devra donc à terme être proposée sur le Play Store et l'Apple Store.

Chaque capteur du satellite remonte une information de mesure, mais également un ensemble de métadonnées, comme l'horodatage de la mesure, sa configuration, la position et l'orientation du satellite…

Par ailleurs, il faudra appliquer, par configuration, un ensemble de filtres sur les données recueillies. Pour n'en citer que quelques-uns :

• Moyenne glissante ;

- Moyenne glissante temporelle ;
- Régression linéaire ;
- Suppression des valeurs aberrantes ;
- Passe-bande ;
- Moyenne entre deux capteurs ;
- Etc.

Enfin, les écrans de restitution sur mobile devront être entièrement paramétrables :

- Choix du type d'affichage pour chaque élément graphique :
  - Vignette avec uniquement la donnée (collectée ou calculée) ;
  - Vignette avec la donnée et une ou deux métadonnées ;
  - Courbe temporelle ;
  - Vignette avec couleur conditionnelle (rouge, vert).
- Positionnement de chaque élément graphique dans les écrans.

Dans l'idéal, nous souhaiterions que cette configuration se fasse par l'utilisateur.

Nous vous souhaitons bon courage dans la réalisation de cette mission.

Cordialement,

Space Inc.

Eh bien, on peut dire qu'ils ont du pain sur la planche.

Garder le métier au centre de nos préoccupations

On pense souvent qu'un bon développeur est une personne qui est irréprochable techniquement, qui connaît par cœur les subtilités du langage qu'il utilise, en somme un geek qui saura tout implémenter, qu'importe la complexité qui en résulte.

Pour nous, la définition d'**un bon développeur est une personne qui sait tout mettre en oeuvre pour faire naître une application qui sert et représente au mieux un métier**. Ce travail est bien plus vaste que la maîtrise technique, plaçant le métier comme point de départ et au cœur de la création de logiciel.

À ce titre, si l'on arrive à faire en sorte que le métier et les développeurs raccourcissent au maximum les étapes intermédiaires de communication, on a tout gagné. Exprimé en langage métier, l'interface entre le métier et les développeurs devient le code. C'est pour cela qu'il est important, pour répondre au mieux aux besoins du métier, d'avoir des tests lisibles par les gens du métier.

Dans cette approche, les tests techniques disparaissent au profit des tests métier.

Voici encore un aspect qui vient légitimer

```
// nom de test technique, qui n'aide
pas à la compréhension
@Test
void testCandidatNull() {...}

// nom de test métier, qui aide à
sa compréhension
@Test
void devraitLeverUneAlerteSiAucun-
CandidatPresent() {...}
```

l'approche du test first, les tests étant l'interface de prédilection avec le métier qui concentre les discussions. Tout pousse à les créer avant de se lancer dans le code...

Nous pourrions essayer d'aller du plus abstrait au plus concret,
pour finir sur des choses sur lesquelles nous pourrons agir ?

Objectif Lune

Quand Gaël pénètre dans la salle de réunion, il est surpris d'y trouver Chloé, Farid et Paul déjà installés. Il regarde sa montre et a la confirmation qu'il cherche : 9h04, il n'est pas en retard. Comment se fait-il qu'ils aient tous trois l'air de l'attendre ? Et souriant de surcroît, alors que lui enchaîne les nuits pourries et n'a qu'une envie : retourner se coucher.

En plus, 3 gobelets traînent sur la table. Il espérait aller chercher un café avec les autres pour démarrer en douceur la journée, ça semble bien compromis.

Paul prend la parole :

— Bien, sans plus attendre, je vous propose de faire le tour de ce que nous avons à présenter. Qui commence ?

Comme à chaque fois ces derniers jours, Farid dégaine le premier :

— Pour ma part, j'ai fait un peu le tour des frameworks de tests unitaires, j'ai regardé les concepts d'intégration continue et fait quelques essais prometteurs.

— De mon côté, enchaîne Chloé, j'ai cherché à comprendre quels types de tests on peut utiliser et j'ai découvert une démarche intéressante intitulée la pyramide des tests et qui pourrait nous aider grandement. J'ai aussi lu un article très intéressant sur la tension entre confiance et flexibilité, mais je ne suis pas sûre de pouvoir vous transmettre davantage que l'idée générale.

J'ai découvert deux choses : le concept de zone d'impact et un indicateur : la couverture de code.

Les regards se tournent vers Gaël… qui de son côté n'est pas prêt à participer activement à cette réunion. Un silence pesant commence à s'installer. Il est brisé par Paul :

— Bien ! Et moi, j'ai découvert deux choses : le concept de zone d'impact et un indicateur : la couverture de code.

Gaël de son côté n'a, à vrai dire, rien à présenter. Il n'a fait aucune recherche. Toute cette histoire n'a ni queue ni tête. Il est développeur, et bon développeur de surcroît. Ces recherches sur les tests ne sont qu'une perte de temps. Et il souhaite juste que ce meeting se termine le plus vite possible pour retourner faire son vrai travail.

Il sent la colère monter à mesure que les regards de ses collègues se font de plus en plus insistants. Mais il éprouve aussi une grande honte depuis les événements survenus à Space Inc. Il concède :

— J'ai principalement avancé sur le développement des filtres. Ben quoi ? Ne me regardez pas comme ça ! Le code ne va pas se faire tout seul ! Il faut bien qu'au moins l'un d'entre nous fasse avancer les choses. Sinon, on aura l'air malin devant Space Inc. dans trois semaines… Enfin déjà plus que deux et demi.

J'ai quand même gardé tous les scripts et exemples que j'ai été amené à utiliser lors de la construction du code. Peut-être que nous pourrions mettre ça dans cette " intégration continue " dont Farid a parlé, précise-t-il sur la défensive.

— Bien, maintenant que nous avons fait le tour, il faut choisir dans quel ordre nous abordons ces sujets. Quelqu'un a-t-il une idée ?, demande Paul en bon manager.

— Même si je n'ai aucune idée de ce que ça va donner, je me dis que nous pourrions essayer d'aller du plus abstrait au plus concret, pour finir sur des choses sur lesquelles nous pourrons agir, proposa Chloé.

Comme personne ne réagit, elle ajoute :

— Mais si c'est débile… On peut faire autrement…

— Non, non, non ! Bien au contraire ! Ça semble plein de bon sens ! Je me demande juste d'où te viennent toutes ces bonnes idées, rectifie Paul.

La crispation se transforme en soulagement sur le visage de Chloé.

Pléthore de tests...

Toute une série de tests sont automatisables. Voici une liste de types de tests que vous pourrez retrouver sur vos projets :

Tests de non-régression : tests visant à vérifier que les nouvelles évolutions n'entraînent pas de régressions sur les évolutions précédemment développées. Ces tests peuvent être unitaires, d'intégration, de bout en bout...

Tests de performance : tests visant à vérifier les temps de réponse d'un système en fonction de sa sollicitation. Ce type de test peut bien entendu être automatisé, mais il peut prendre du temps, il n'est alors souvent pas intégré dans les tests joués lors de l'intégration. Il est toujours possible de l'intégrer dans des test joués régulièrement (par exemple : les nightly build).

Tests d'API : tests qui se concentrent sur l'API et qui impliquent des tests d'intégration. Le but de ces tests est de valider la connectivité de l'API et d'assurer la qualité de la donnée exposée.

Tests d'infrastructure : les pratiques actuelles amènent à automatiser le maximum, y compris la création de l'infrastructure (Infrastructure as Code), qui peut alors être testée, comme n'importe quel bout de code. Cette question est développée dans le volume 3 de notre série Culture DevOps, p. 18.

Tests manuels : sous cette dénomination, nous retrouvons souvent les tests exécutés par des humains, après les développements. Cette dénomination a un côté dévalorisant pour celles et ceux qui sont chargés des tests, car d'une part ces pratiques sont outillées, et d'autre part ce " manuel " s'oppose à " automatiser " qui serait plus efficace.

Tests exploratoires : il s'agit de tests à la marge organisés essentiellement par le métier. Il est difficile de simuler ces tests puisqu'ils sont étroitement liés à des comportements utilisateurs non-prédictibles ou des situations réelles non simulables. Prenons l'exemple d'une entreprise qui loue des voitures grâce à une application. L'application sert également de clef pour démarrer la voiture. Tout fonctionne par le réseau GSM ou Wifi. L'application est 100 % fonctionnelle, pas de bugs, beaucoup d'utilisateurs, tout va pour le mieux. Mais que se passe-t-il lorsqu'un utilisateur se gare dans une zone qui n'est pas couverte par le réseau, sous un pont, par exemple ? La voiture est bloquée. Difficile à prévoir dans le feu de l'action ! C'est un parfait exemple de test exploratoire.

Tests de charge : ces tests ont pour vocation de simuler des trafics utilisateurs pour mesurer la résilience des applications. Grâce à des librairies de tests comme artillery.io, gatling.io ou encore k6.io, on peut définir un ou plusieurs appels HTTP notamment pour définir des scénarios utilisateurs. On peut alors configurer l'amplitude, la fréquence et la périodicité de consommation API par une population d'utilisateurs fictive et ainsi évaluer les performances d'une API pour répondre à cette demande. Ces tests permettent également d'identifier d'éventuelles latences

au plus tôt. Il convient cependant de limiter leur utilisation, car la génération de population et les actions associées consomment beaucoup de ressources processeur.

Et il y en a d'autres !

Tests assistés, tests de sécurité, tests de haut niveau, Fuzz testing[16] (ou fuzzing), monitoring, tests property-based[17], tests d'interface utilisateur, tests de déploiement… La liste est longue… Vraiment longue ![18]

Pour s'y retrouver, l'important est de bien définir les intentions de tests et de répondre aux besoins métiers, utilisateurs et de développement[19] et seulement ensuite définir le type de test dont on a besoin et son implémentation.

Comme toujours, start with why![20]

⑯ https://en.wikipedia.org/wiki/Fuzzing
⑰ Property-Based Testing for everyone by Romeu Moura : https://www.youtube.com/watch?v=5pwv3cuo3Qk
⑱ https://en.wikipedia.org/wiki/Software_testing#Testing_types,_techniques_and_tactics
⑲ https://martinfowler.com/testing/
⑳ https://www.ted.com/talks/simon_sinek_how_great_leaders_inspire_action

Les 4 intentions de tests pour construire sa stratégie

Il y a pléthore de tests. Aussi, il est difficile de s'y retrouver et de décider où concentrer son énergie. Le schéma suivant, extrait de "Culture Code", permet de regrouper les tests selon 4 intentions pour y voir un peu plus clair :

Les tests dits "boîte noire" ou de contrôle, pour critiquer le produit réalisé au plus proche de la réalité, une fois qu'il est livré. Ces tests examinent uniquement les fonctionnalités d'une application :

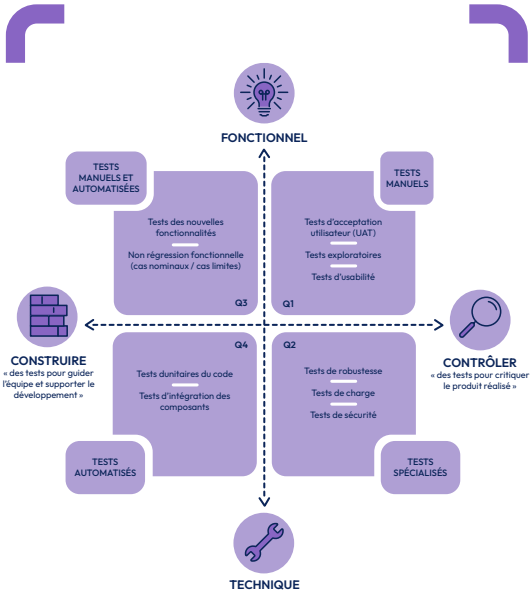
• **Q1 — Tests fonctionnels** : ce sont les tests qui permettent de vérifier ce que fait le système, de vérifier la conformité de l'application développée avec le besoin fonctionnel exprimé, ils sont basés sur la compréhension commune, entre les développeurs et le métier, des règles à implémenter.

• **Q2 — Tests techniques** : ils permettent d'évaluer comment le système fonctionne. Parmi les tests non-fonctionnels il y a les tests de performance, d'utilisabilité, de maintenabilité, de fiabilité, de portabilité et de sécurité.

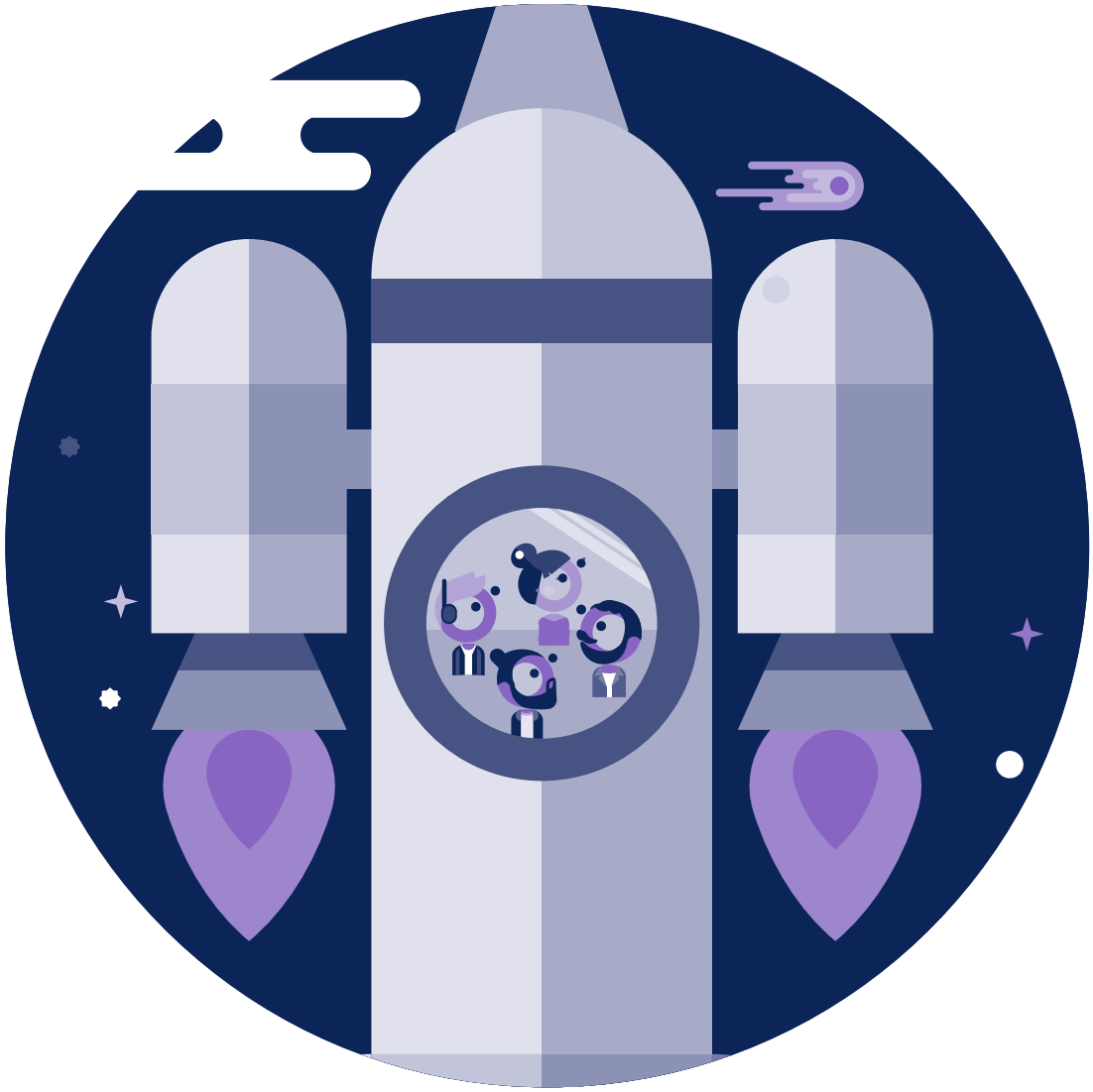
Les tests dits «boîte blanche" ou de construction, pour guider à la construction itérative d'un logiciel de qualité. Ces tests examinent le fonctionnement d'une application et sa structure interne :

• **Q3 — Tests liés au changement** : les tests de confirmation et les tests de non-régression. Les tests de confirmation sont ceux liés à un correctif faisant suite à un défaut. Ces tests exécutent des cas de tests qui ont été en échec la dernière fois qu'ils ont été exécutés. Leur objectif est de vérifier le succès des actions de correction. Les tests de non-régression vérifient que les modifications n'ont pas eu d'impacts négatifs sur les parties non modifiées du code.

• **Q4 — Tests structurels** : ce sont des tests qui sont basés sur une analyse de la structure interne du composant ou système comme le code, l'architecture et les flux de données.



Extrait du Culture Code



La science montre que c'est la confiance qui permet d'aller de l'avant.

Tirer n'est pas toucher

— Bien !, enchaîne Paul, (il faudrait qu'il arrête un peu de commencer toutes ses phrases par "bien"), cela nous donne l'ordre suivant :

- Chloé - Tension entre confiance et flexibilité
- Moi - Zone d'impact
- Chloé - Types de test
- Chloé - Pyramide des tests
- Moi - Couverture de code
- Farid - Intégration continue
- Farid - Framework de tests
- Gaël - Exemples et scripts de construction

Pas d'objection ? Bien ! (encore ? sérieux...), Chloé, c'est à toi de nous expliquer l'opposition entre confiance et flexibilité.

Chloé attaque :

— C'est un concept abstrait et je ne vais pas m'éterniser dessus. On peut considérer que les tests peuvent avoir deux fonctions :

- Augmenter la confiance ;
- Offrir de la flexibilité.

Les tests peuvent avoir deux fonctions :

• Augmenter la confiance ;

• Offrir de la flexibilité.

Si j'ai bien compris, à force de rajouter des tests pour rassurer tout le monde sur le bon fonctionnement du code, on finit par rendre plus difficile toute évolution ultérieure.

Paul ne peut s'empêcher d'intervenir :

— Mais n'importe quoi, bien au contraire ! La confiance génère dans le cerveau l'hormone nécessaire à la prise de…

Paul s'interrompt brutalement devant les regards sidérés de ses trois collègues.

— Euh… oui… je veux dire que euh… ce n'est pas ce qu'ont montré des expériences. Par exemple, un bébé rat dont la mère lui offre réconfort et sécurité sans limite ira, contre toute attente, beaucoup plus explorer et changer son environnement, qu'un bébé rat dont la mère le pousse à l'exploration.

Euh bref, la science montre que c'est la confiance qui permet d'aller de l'avant.

Paul est maintenant rouge de honte. Gaël est décontenancé. C'était quoi ça ?

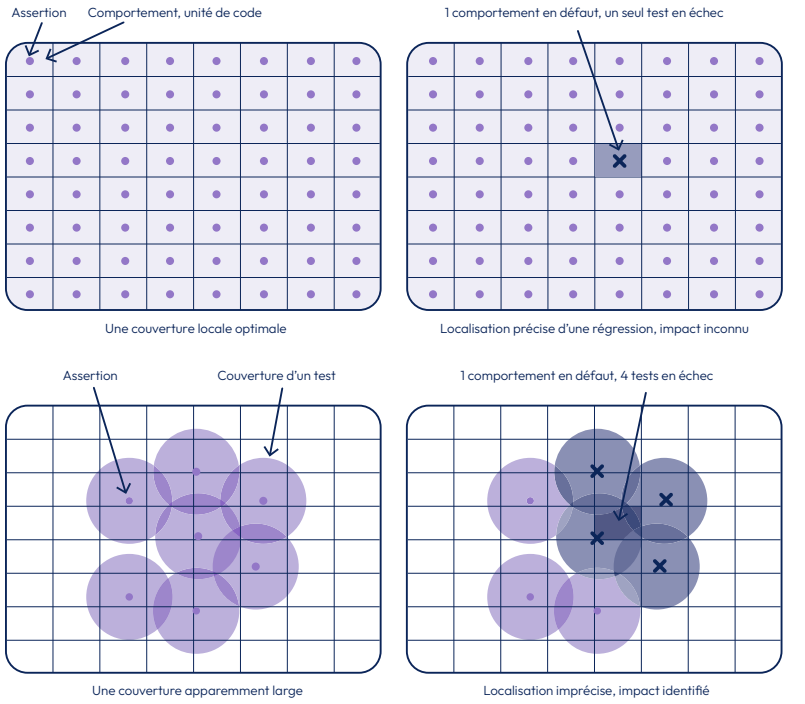
— En fait, je pense comme toi, répond Chloé, mais en repensant à notre épreuve chez Space Inc., je me suis dit que si nous avions eu des tonnes de tests, cela nous aurait rendu la tâche plus difficile.

— Waouh ! J'espère que tous les sujets ne vont pas être comme celui-là. Sinon il va falloir me récupérer à la petite cuillère d'ici midi. Vu notre niveau sur les tests, vous ne voulez pas qu'on passe à la suite ?, s'esclaffe Farid.

C'était apparemment ce que tous attendaient.

— Bien ! (ah non, ça suffit là !), pour la zone d'impact, ça me semble plus simple.

L'illustration ci-dessous en résume toute la teneur :



Extrait de Culture Code

Plus le fragment de code testé est petit, plus il est facile :

- De s'assurer que tout est testé ;
- De savoir ce qui doit être réparé quand un test passe au rouge.

— Ah ben ce coup-ci, c'est simple, il suffit de faire les tests les plus petits possibles, déclare assertivement Farid.

— Raté, car des tests plus larges correspondent bien mieux aux fonctionnalités métier du logiciel et donc en cas de problème, évolution ou bug, on peut immédiatement comprendre l'impact sur les fonctions rendues pour le business. D'autre part, on peut obtenir une couverture complète de son logiciel avec bien moins de code.

Ce coup-ci, Farid y va plus prudemment :

— Et donc c'est mieux des tests plus larges ?

— Et donc je n'en sais rien !, s'emporte Paul.

— J'imagine que c'est une question d'équilibre, comme tout à l'heure avec la confiance et la flexibilité. Ou alors qu'il faut les deux : des tests larges et aussi des tests fins.

Comme Farid ne va pas tarder à partir en dépression, Chloé intervient :

— Et si nous avancions vers des sujets plus concrets ?

Et c'est ainsi qu'ils passent au sujet suivant.

Plus le fragment de code testé est petit, plus il est facile :

- **De s'assurer que tout est testé ;**
- **De savoir ce qui doit être réparé quand un test passe au rouge.**

Un test peut en cacher un autre

Pour cet encart, nous vous recommandons la série d'articles OCTO : "Un test peut en cacher un autre" qui amène la réflexion un peu plus loin sur ce qu'il faut tester et comment.

• OCTO Talks ! Un test peut en cacher un autre – Un peu de théorie : https://blog.octo.com/un-test-peut-en-cacher-un-autre-un-peu-de-theorie/

• OCTO Talks ! Un test peut en cacher un autre - Tests unitaires : https://blog.octo.com/un-test-peut-en-cacher-un-autre-tests-unitaires-p1/

• OCTO Talks ! Un test peut en cacher un autre - Tests unitaires : https://blog.octo.com/un-test-peut-en-cacher-un-autre-tests-unitaires-p2/

• OCTO Talks ! Un test peut en cacher un autre - Tests d'intégration : https://blog.octo.com/un-test-peut-en-cacher-un-autre-tests-dintegration-p1/

• OCTO Talks ! Un test peut en cacher un autre - Tests d'intégration : https://blog.octo.com/un-test-peut-en-cacher-un-autre-tests-dintegration-p2/

Tu as parfaitement raison Gaël, tente de le calmer Chloé. Tu sais, l'agressivité n'est pas nécessaire, nous cherchons tous à faire de notre mieux...

On ne fait rien avec de mauvais outils

Chloé entre en scène :

— Ah, c'est au tour des types de tests ! Vous allez voir que c'est moins prise de tête. Sachez déjà que chacun y va de son terme pour qualifier un test et qu'on trouve des dizaines de dénominations. Je n'ai sélectionné ici que celles qui reviennent le plus fréquemment :

- **Unitaire** : porte sur une seule unité de code ou de comportement.
- **Intégration** : assure que les unités logicielles développées indépendamment fonctionnent correctement lorsqu'elles sont connectées les unes aux autres.
- **Fonctionnel** : vérifie l'adéquation avec un comportement métier attendu.
- **Non-régression** : vérifie la présence d'effets de bords indésirables.
- **API** : concerne les endpoints des services et leurs réponses.
- **IHM** : assure le bon fonctionnement de l'interface graphique.
- **Performance** : garantit la réponse aux exigences de rapidité, réactivité…
- **Installation** : vérifie que l'installation est conforme à l'attendu.

Gaël, silencieux jusque-là, ne peut se retenir davantage :

— Ne me dis pas qu'il va falloir écrire des tests pour chacune de ces catégories ? Je vous rappelle que nous n'avons que 3 semaines et que nous avons beaucoup de code à écrire pour satisfaire les attentes du client.

— Tu as parfaitement raison Gaël, confirme Chloé. Pour autant, l'agressivité n'est pas nécessaire, nous cherchons tous à faire de notre mieux…

Alors non seulement Chloé lui a volé en partie la vedette lors de l'épreuve, mais en plus maintenant elle le reprend sur la tournure de ses phrases ! Il a beau savoir qu'elle est carriériste, elle commence à pousser le bouchon un peu loin. De crainte qu'elle ne révèle à tous la faute de logique qu'il a commise chez Space Inc., il préfère s'abstenir de répondre.

Laissant Gaël à ses pensées noires, elle enchaîne :

— Je cherche à vous faire sentir qu'il y a un domaine, et un langage, autour du test. Ce n'est pas grave si nous n'arrivons pas à tout appréhender pour le moment. Et pour une mise en œuvre pratique, je vous propose de passer directement au sujet suivant : la pyramide des tests.

La pyramide des tests.

— Pyramide ?

— Oui. La base de la pyramide est grosse, puis plus on va vers le sommet plus la pyramide s'affine. Les tests unitaires sont la base de la pyramide, ils sont nombreux, car ils coûtent peu chers et ramènent des feedbacks rapides. Je crois que l'idée principale à retenir ici est que l'état de l'art est d'avoir beaucoup plus de tests unitaires que de tests d'intégration et enfin que de tests de bout en bout.

Ça me semble être une approche suffisamment pragmatique pour que nous puissions nous y mettre dès aujourd'hui.

Qu'en pensez-vous ?, annonce Chloé avec enthousiasme.

— Excellent !, approuve Paul avec entrain (Tiens ? Il connaît un autre mot que bien). Chloé, grâce à toi, nous tenons enfin un début de plan d'attaque. Si cela va à tout le monde, je propose de passer à mon sujet pour savoir à quel moment on peut considérer que nous avons assez de tests.

Facile, dès que le code fonctionne !, pense Gaël avec sarcasme.

Débat sur la définition des tests

Tests unitaires, tests d'intégration, tests de bout en bout : aujourd'hui, tout un vocabulaire et une taxonomie se sont développés autour des tests et font souvent l'objet d'âpres débats au sein des équipes de réalisation sur la nature de tel ou tel test.

Si un moyen tactique de se sortir du débat est de répondre que " mieux vaut une application testée (automatiquement) que non testée " et sortir ainsi du débat, cela reste léger. Il est nécessaire de se plonger dans le sujet et de bien comprendre ce que l'on fait quand on code.

Par exemple, si la définition d'origine des tests unitaires prenait comme unité : "la plus petite partie d'un code non divisible" ; elle ne fait aujourd'hui plus l'unanimité. En effet, ceux-ci sont très fragilisés lors de la restructuration du code, ce qui amène une mauvaise réputation aux tests car complexes à maintenir. Cela explique, en partie, l'abandon des tests par une partie des développeurs.

Aujourd'hui, on considère que c'est une unité de comportement. Quel comportement ? Le comportement observable d'un point de vue de l'utilisation du code à tester, c'est-à-dire **le dénouement produit par l'exécution du code (et non les interactions du code exécuté)**. Le test peut alors traverser plusieurs méthodes de plusieurs classes sans aucun problème. Le code est exécuté en mode boîte noire, c'est-à-dire sans aucune connaissance de la structure de ce dernier. Cela permet d'éviter cette fragilité en cas de restructuration, puisque tant que le comportement est le même, le test restera inchangé.

Mais faire évoluer une définition est complexe tant elle est solidement ancrée.

C'est pourquoi certains ont abandonné l'idée et s'accordent sur le terme existant : test d'acceptation (" acceptance test " en anglais).

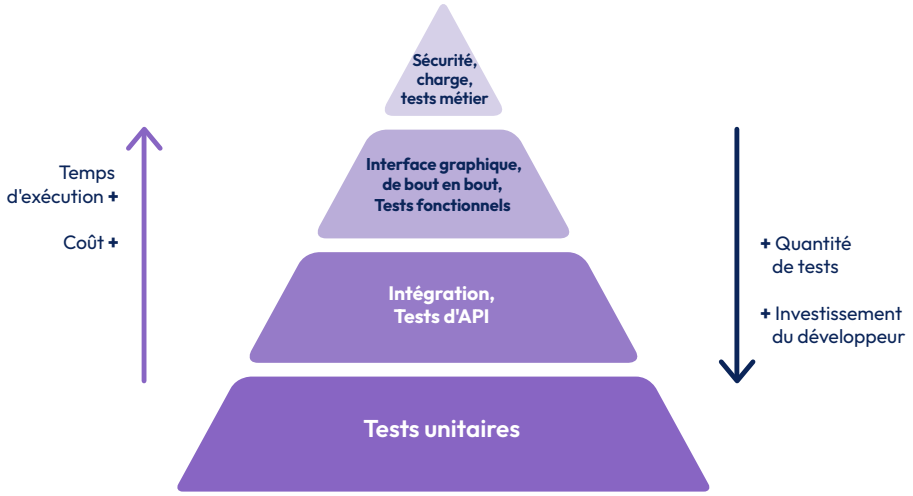
Mais si l'unité est le comportement et non plus une petite portion de code, qu'est-ce qu'un test d'intégration ? Il s'agit alors de l'intégration du système avec un élément externe, comme une base de données, par exemple.

La pyramide des tests

D'expérience, pour bien des équipes de réalisation, tant les managers que les développeurs, la définition pratique de ce qui se cache derrière chaque type de test reste floue.

Lorsque les comportements métier sont bien fournis, ces différents types de tests peuvent être représentés par la pyramide des tests, avec l'insistance sur l'importance d'avoir la pyramide à l'endroit[21].

Stratégie des tests automatisés en pyramide



Extrait du Culture Code

La pyramide est une métaphore nous invitant à suivre différents niveaux de tests. Elle nous indique aussi combien de tests doivent être réalisés à chaque niveau : un système stable sera construit avec des bases stables ; sinon, la pyramide ne tiendrait pas sur son socle.

[21] https://blog.octo.com/remettre-la-pyramide-des-tests-sur-sa-base/

Par ordre décroissant de quantité et d'investissement dans le développement, nous avons donc :

• Les tests unitaires ;

• Les tests d'intégration ;

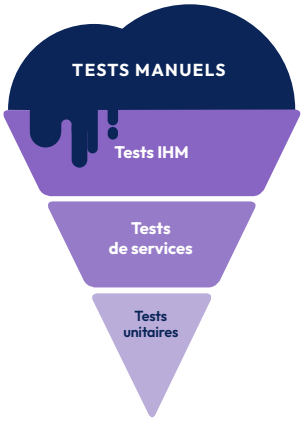
• Les tests d'IHM, d'interface graphique, de bout en bout, fonctionnels ;

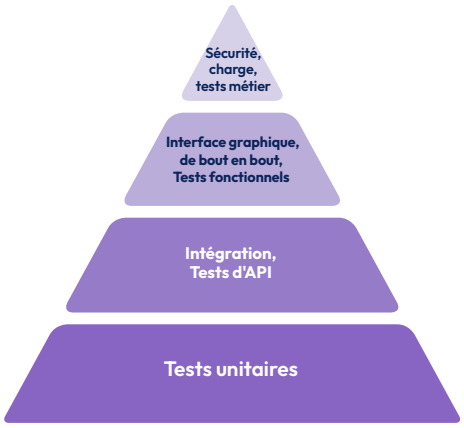
• Les tests métier, de sécurité et de charge.

Les tests unitaires sont faciles à faire, peu coûteux, et nous informent rapidement du résultat : il faut en faire beaucoup.

Les tests d'IHM sont difficiles à faire, car ils parcourent plus de fonctionnalités, plus de cas possibles. Ce type de test est loin de parcourir tous les chemins d'exécution possibles. Ils sont plus coûteux, et lors d'un résultat en erreur, la recherche de la cause est plus difficile, il faut en faire moins. En général, pour ce type de tests, on favorise les cas " simples " dits " passants " ou " happy paths ".

Les personnes qui n'automatisent pas les tests ont tendance à tomber dans l'anti-pattern de l'ice-cream cone. Ils écrivent plus de tests d'IHM que de tests unitaires ; ils inversent la pyramide : les tests coûtent plus cher et donnent moins d'informations sur le code. Par contre, en l'absence de comportement métier, la pyramide inversée se justifie pleinement, voire prendre la forme d'un diamant dans le cas où l'intelligence est portée sur la façon de récupérer et aggréger les données (importance donnée aux tests d'intégration).





Extrait du Culture Code

Pour en savoir plus sur le sujet, le blog octo a dédié 5 articles dont le premier est le suivant : https://blog.octo.com/la-pyramide-des-tests-par-la-pratique-1-5/

Pourquoi faire des tests automatisés, c'est pas fun ?

Parce qu'à l'école nous avons appris à coder, le test est une sanction. Est-ce que les écoles informatiques aujourd'hui apprennent à tester ? Le code est une activité créative alors que le test – réalisé après le développement – est une activité d'observation. Coder, c'est fun parce que c'est de la production alors que tester ça ressemble à "inspecteur des travaux finis". Cette considération est dommage, car un testeur peut sauver un projet.

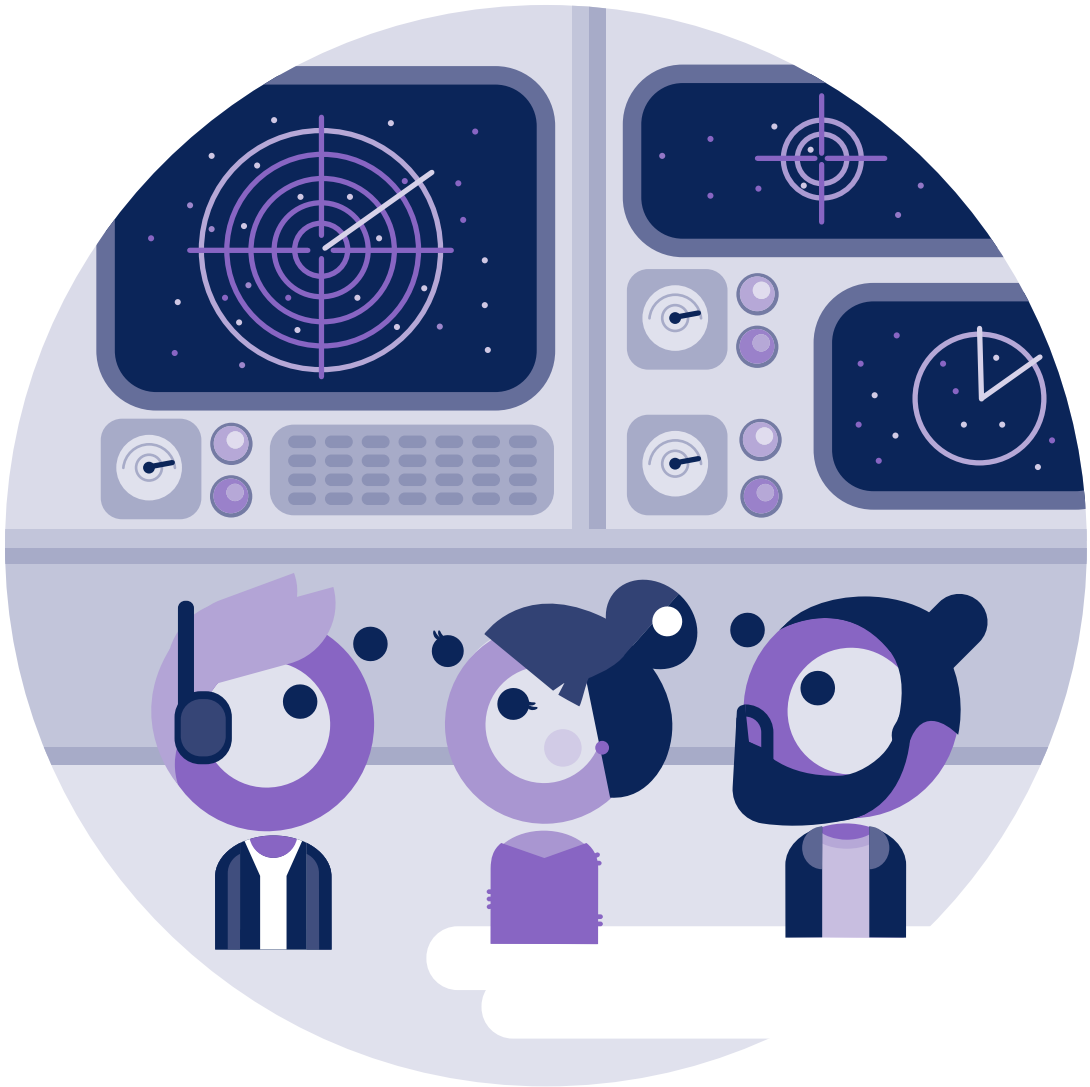
Le testeur apporte une mauvaise nouvelle à l'équipe de réalisation qui n'a plus la disponibilité d'esprit car elle traite déjà un autre sujet. L'interaction apporte plus d'éléments négatifs que positifs. Alors que réalisé en amont, le test apporte du positif à un moment où les développeurs sont au maximum de leur implication intellectuelle sur la fonctionnalité.

Faites attention aux "sprints de tests" ! Un indicateur qui montre un manque de pratique de tests automatisés.

L'activité de test n'est pas valorisée, elle est souvent confiée à des stagiaires ou, au mieux, aux développeurs les moins qualifiés. L'activité de test est vue comme répétitive et ennuyeuse, elle n'est pas confiée aux développeurs expérimentés qui ont besoin d'être créatifs dans leurs activités.

Si le test était réalisé avant le développement, il serait co-construit avec les développeurs et serait une activité tout aussi valorisante que le développement, car il serait inclus dans le processus créatif du développement.

Produire de la qualité, c'est insérer le test à la production, pas au contrôle. À nous de savoir si nous voulons produire de la qualité ou tester la qualité.



C'est ainsi que le groupe s'engage dans la voie des 100 % de couverture de code.

Sortez couverts

Ils entrent enfin dans des sujets concrets, et l'énergie et la tension dans la pièce montent progressivement, à tel point qu'ils en oublient totalement la pause. Tant pis pour le café dont Gaël aurait eu désespérément besoin pour se requinquer et sortir de sa mauvaise humeur.

— Bien ! (!!!), vous serez d'accord avec moi que les tests servent à tester le code, dit Paul pour amorcer son sujet.

D'un ton sarcastique, Gaël paraphrase :

— " Les tests servent à tester ", on n'est pas loin d'une tautologie là, non ?

— Humm oui, merci Gaël pour la justesse de ta remarque, répond Paul calmement. Donc ce qu'il nous faudrait savoir, c'est si tout le code a été testé. Si ce n'est pas le cas, cela veut dire que l'on doit encore ajouter des tests. Et bien figurez-vous que j'ai lu que dans la plupart des langages, on est capable de déterminer l'ensemble des lignes de code exécutées pour chaque test.

Et si l'on met tous les tests bout à bout, on est capable de dire, par ligne de code, si elle a été exécutée par au moins un test.

— Formidable, réagit Farid, tu peux nous montrer un exemple ?

```
1.  import java.util.regex.Matcher;
2.  import java.util.regex.Pattern;
3.
4.
5.  public class Time {
6.    private int hours;
7.    private int minutes;
8.    private static final Pattern pattern =
9.     Pattern.compile("(\\d\\d):(\\d\\d)");
10.
11.   Time(int hours, int minutes) {
12.    this.hours = hours;
13.    this.minutes = minutes;
14. }
15.
16.
17. int getHours() { return hours; }
18. int getMinutes() { return minutes; }
19.
20.
21. public static Time fromString(String timeString) {
22.    Matcher matcher = pattern.matcher(timeString);
23.    matcher.find();
24.    return new Time(Integer.parseInt(matcher.group(1)),
25.     Integer.parseInt(matcher.group(2)));
26.  }
27. }
28.
29.
```


— Naturellement, s'enjoue Paul, voici le code :

Comme vous pouvez le voir facilement, on peut dire que les lignes 19 à 21 ne sont pas testées.

— Mais c'est trop génial, s'exclame Farid.

— Et d'après toi Paul, doit-on faire systématiquement la chasse aux lignes en rouge ?, questionne Chloé.

— C'est une excellente question et j'ai pu lire pas mal de divergences sur ce sujet. Il en est question dans des articles de " couverture de code ", ce n'est ni plus ni moins que le pourcentage de lignes vertes sur le nombre de lignes totales.

Certains affirment qu'une couverture à 100 % est la seule acceptable, d'autres disent que ça demande un effort trop grand et qu'il vaut mieux viser les 80 %. Vu nos récents déboires avec Space Inc., je propose que nous nous montrions irréprochables en visant les 100 %. Quelqu'un y voit-il une objection ?, demande-t-il à toute l'équipe.

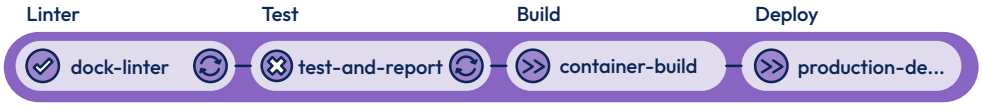
— Mis à part qu'on va passer trois fois plus de temps à tester qu'à coder ? Non, aucune !, assène Gaël d'une humeur exécrable.

— Peut-être n'en serions-nous pas là si nous nous étions un peu plus conformés à l'état du marché pendant l'épreuve, rétorque Paul.

En voulant faire comprendre à Gaël que sa mauvaise humeur commence à l'agacer, Paul ne se doute pas que cette remarque vient de le mortifier au plus profond de lui. Ne sachant plus où se mettre, Gaël préfère faire profil bas pour la suite de la réunion.

Et c'est ainsi que le groupe s'engage dans la voie des 100 % de couverture de code.

— C'est enfin à moi !, s'exclame Farid. Pour faire simple, une intégration continue est un ensemble de traitements qui s'activent, soit cycliquement, soit à chaque fois que le code est poussé dans un repository de source. De nos jours, c'est cette deuxième option qui est retenue la plupart du temps. Concernant les étapes, cela comprend systématiquement l'exécution des tests automatiques. Si un problème survient dans n'importe laquelle des étapes, cela notifie les abonnés. Voici un exemple simple que j'ai monté sur une application Java tournant dans un conteneur Docker[22] :



---

[22] https://en.wikipedia.org/wiki/Docker_(software)

Comme on peut le voir, les règles de lint[23] étaient en succès, mais l'un des tests était en échec. Des questions ?

— Oui, as-tu pu faire un comparatif des différentes intégrations continues du marché ?, demande Paul.

— Très rapidement oui, mais je vais aller droit au but : Space Inc. n'utilise que Gitlab CI[24] ! Fort heureusement, il fait partie des leaders de ce marché. C'est d'ailleurs de là que vient l'exemple que je vous ai montré.

㉓ The computer as master mind - Donald E. Knuth, https://www.cs.uni.edu/~wallingf/teaching/cs3530/resources/knuth-mastermind.pdf
㉔ https://docs.gitlab.com/ee/ci/

Une brève histoire de la CI[25] et de l'agilité

Au début, il n'y avait rien… puis sont arrivées les usines de développement…

En 2002, il fallait pousser son équipe à utiliser un repository de code, ce n'était pas du tout la norme, aujourd'hui, c'est systématique.

Toute l'industrialisation dont nous disposons à ce jour induit un rythme, une urgence plus forte, mais les compétences, la sagesse, des développeurs aujourd'hui est-elle meilleure qu'il y a 20 ans ? Les bonnes pratiques sont-elles intégrées, par exemple Unit Testing ?

L'évolution de l'outillage, notamment les environnements de développement, permettent de plus en plus de choses, sont de plus en plus puissants et intégrés.

L'industrialisation peut concerner les outils de production (tests automatisés, propriété collective du code…) mais aussi la réalisation du produit, **l'industrialisation possible concerne les outils, pas le produit.**

Par exemple :

- Un même cahier des charges donné en entrée est réalisé par deux équipes différentes, il donnera deux produits différents : l'industrialisation du produit n'est pas possible ;
- Le processus industriel CI/CD[25] garantira le passage par les mêmes étapes : l'industrialisation des outils est possible.

[25] https://en.wikipedia.org/wiki/CI/CD

Waouh, s'émerveille Paul, je suis bluffé des avancées qu'on a pu faire en seulement 2 jours. Beau travail à tous.

Ensemble on va plus loin

— Mon deuxième sujet : les frameworks de tests unitaires, complète Farid.

Je vais faire encore plus simple : nous devons développer en Java, "le" framework[26] le plus répandu en Java est JUnit. Et je n'ai vu aucune plainte sérieuse à son sujet sur les forums. Voici un exemple de test que j'ai également réalisé :


```
public class ReadersClubTest {
  private final ReadersClub readersClub = new ReadersClub();
  @Test
  void testAddReader() {
   readersClub.addReader("John Doe");
   List<String> readers = readersClub.getReaders();
   assertEquals(1, readers.size());
   assertEquals("John Doe", readers.get(0));
  }
  @Test
  void testFindReaders() {
```

[26] Framework: ensemble d'outils et de composants logiciels dédié au développement.Définition tirée de https://fr.wikipedia.org/wiki/Framework

```
    readersClub.addReader("John Smith");
    List<String> readers = readersClub.findReaders("Smi");
    assertEquals(1, readers.size());
    assertEquals("John Smith", readers.get(0));
  }
}
```

Et le code associé :


```
public class ReadersClub {
 private final List<String> readers = new ArrayList<>();
 List<String> getReaders() { return readers; }
 void addReader(String reader) {
   String newsReader = reader;
    if(!newsReader.matches("[a-zA-Z\s]+")) {
    throw new IllegalArgumentException(«Reader should only contain letters»);
    }
    readers.add(newsReader);
 }
```

```
List<String> findReaders(String toFind) {
  List<String> result = new ArrayList<>();
  return readers.stream().filter(s ->
   s.contains(toFind)).collect(Collectors.toList());
 }
}
```

Et je vous préviens, on ne se moque pas de mon code ! C'était juste pour l'exemple.

— Waouh, s'émerveille Paul, je suis bluffé des avancées qu'on a pu faire en seulement 2 jours. Beau travail à tous. Gaël, il ne reste plus que toi avec les exemples de construction.

Comme dit le proverbe : " C'est au pied du mur qu'on voit le maçon ". Pas moyen pour Gaël de se défiler. Le tout est de savoir s'il arriverait à faire illusion.

— Premièrement, concernant les scripts, ils consistent tous à :

• Monter l'application ;

• Charger une classe ;

• Exécuter l'une de ces méthodes avec les arguments en entrée du script ;

• Logguer les résultats.

Et concernant les exemples que j'ai utilisés, j'ai chaque fois noté dans un fichier texte :

• Le script à lancer ;

• Les arguments à fournir en entrée ;

• Les résultats attendus dans les logs.

Bien ! (ah ben non voilà que c'est contagieux !!!), si j'ai compris ce que vient de nous présenter Farid, un test automatisé consiste à :

• Monter un environnement de test (pris en charge de manière transparente par le framework de test) ;

• Charger une classe ;

• Exécuter l'une de ces méthodes avec des arguments spécifiques au test ;

• Récupérer les résultats ;

• Faire des assertions sur les résultats.

Les hochements de tête de Farid lui confirment qu'il ne s'est pas fourvoyé.

— Et donc au vu de ça, je pense que cela sera à la fois très simple et très rapide de convertir mes scripts et exemples en tests automatisés. Ce qui veut dire que nous aurons déjà fait le travail de test pour les premiers développements.

— Bien joué Gaël, lui lance Chloé avec un petit sourire.

Les autres aussi ont le sourire. C'est pour le moins surprenant. Il a peut-être finalement une place à se faire dans ce groupe pour le moins étrange.

— Je dirais que c'est l'heure du bilan, annonça Paul.

— Je prends !, dit aussitôt Chloé. Alors nous avons :

C'est l'heure du bilan.

• Un framework de tests : JUnit ;

• Un début d'intégration continue sous GitLab CI ;

• Des tests quasiment prêts pour les premiers développements ;

• Un indicateur et un objectif : 100 % de couverture de code ;

• Une stratégie de test : la pyramide ;

• Et euh… des éléments de réflexion sur la zone d'impact et le dualisme confiance/flexibilité.

— C'est top !, réagit Farid.

— Top ? Tu rigoles, c'est topissime !, éclate Paul avec gaieté. C'est au-delà de toute espérance ! Merci à vous tous pour ce travail formidable. Continuons comme ça et la victoire est assurée ! Midi pile ?! On ne peut pas se quitter comme ça. Je vous invite tous au resto pour fêter ça dans les règles, mettant ainsi fin à cette séance éprouvante.

État de l'Art ou convention d'équipe ?

Nous parlons d'état de l'art comme d'un standard. Ce standard est mondial, et vrai à un instant t. La convention, elle, est locale.

L'État de l'Art est le Saint-Graal à atteindre aux yeux des puristes ; c'est la quête éternelle, ce code légendaire dont on parlera fièrement dans 10 ans, 15 ans… Il représente le summum de la connaissance et des théories du moment et de fait, on l'associe naturellement à un gage de qualité indiscutable et unanime.

Pourtant, les écoles s'opposent souvent et on peut observer des guerres de chapelles entre deux ou trois pratiques ; parfois, c'est simplement parce que quelqu'un voue un véritable culte à une Rock Star du code.

D'ailleurs, un état de l'Art n'est vrai que tant qu'il n'a pas été réfuté, amélioré, bouleversé… Rien n'est éternel, encore moins dans le code.

À quoi ça sert d'avoir un État de l'Art alors ?

Une meilleure question, plus précise : Quelle est l'intention derrière la volonté d'utiliser l'État de l'Art ?

Souvent, on souhaite simplement fédérer une équipe sur des pratiques communes de manière à délivrer plus vite et plus simplement de la valeur.

— Franchement, les points-virgules, c'est has-been, je n'aime pas ce truc de boomers, revendique A.

— C'est peut-être un truc de boomers, mais au moins les fins d'instructions sont claires, défend B.

— Je suis d'accord avec toi, s'exprime C. Par contre, je préfère mettre des tabulations de X espaces.

— Ah non ! On ne va pas rouvrir ce débat, il fallait être embauché quand on en a parlé ! Discutons plutôt sur la convention de nommage des variables, je trouve que le snake_case est vraiment plus lisible !

— Mais on est en Java, la convention du langage c'est le camelCase.

— C'est pas faux…

La plupart du temps, c'est le flop total, on finit tôt ou tard par s'écharper. On impose alors un état de l'Art qui ne correspond à personne et tout devient dur.

Les conventions d'équipe ont la même intention, mais avec des ambitions plus modestes. Il s'agit de se mettre d'accord, en équipe, pour identifier les pratiques les plus efficaces de l'équipe et uniformiser le code sur cette base pour faciliter la livraison de valeur.

Une bonne démarche est de conventionner ce qui est déjà le plus diffusé à l'aide d'une règle de linter[27]. Pour les personnes originales, il faut se recentrer sur l'egoless programming et mettre de l'eau dans son vin.

[27] Linter: outil d'analyse statique de la syntaxe du code.

La bonne nouvelle, c'est qu'on est rarement loin de l'État de l'Art et au quotidien, développer va vite devenir plus confortable pour tout le monde : les revues de codes et la production en pair programming et/ou mob programming se concentrent sur le vrai problème plutôt que sur des futilités. On peut alors créer un cercle vertueux de développement (on parle de Developer Experience).

Et si un jour il faut changer de convention d'équipe parce que A et B sont allés vers d'autres aventures ? Aucun problème puisque le code est homogène ! Dans la plupart des cas, il suffit d'ailleurs de changer la règle de linter qui cristallise la convention et s'il s'agit de changements simples, les linters peuvent corriger auto-magiquement l'ancien code.

«Le bonheur n'est pas au sommet de la montagne mais dans la façon de la gravir», Confucius.

À suivre...

Dans le prochain épisode...

Le monde des tests est vaste et riche en subtilités, il s'oppose à une vision classique du développement. Celle-ci ne laisse pas de place aux tests automatisés, et elle nous accompagne depuis nos débuts dans l'informatique, depuis nos premiers " Hello World ! ".

Nous avons bâti notre savoir-faire de développement sur cette vision classique.

Pour beaucoup, les tests relèvent du superflu. Ils sont négociables, voire renégociables en fonction des impératifs du projet.

Pourtant, tester, c'est une culture ; ce n'est pas seulement répondre à une exigence projet. Tester, et ici apprendre à tester, c'est se donner les moyens de se réinventer continuellement, d'accepter les inconnues pour les traiter afin de vivre sereinement avec la seule constante de la vie : le changement.

Au cours de l'histoire, vous avez rencontré des personnes plus ou moins courageuses, prêtes à se reconstruire, propulsées par le moteur de leurs convictions et de leurs passions.

Nos protagonistes pensent avoir compris l'exigence client et appris à travailler ensemble, autour des tests. Leur bonne étoile semble les accompagner de nouveau.

Mais que va-t-il advenir de leur savoir-faire avec l'introduction de ce nouveau pilier ?

La soif d'apprendre et les convictions de nos aventuriers devront être sans faille pour opérer proprement une transformation exigeante alors que le temps file...

Bibliographie et médiagraphie

- " Unit Testing Principles, Practices, and Patterns " de Vladimir Khorikov
- " The art of unit testing " de Roy Osherove
- " Culture code " de Octo Technology
- Wabi-sabi — Wikipédia
- " Sel Gras Acide Chaleur " de Samin Nosrat
- The computer as master mind - Donald E. Knuth
- Deep Work: Your Weapon for the 21st Century - Parsh Jain
- Maximizing Developer Effectiveness - Martin Fowler
- " The Psychology of Computer Programming " de Gerald M. Weinberg
- Egoless Programming - OCTO Talks !
- re:Work - Guide: Understand team effectiveness
- Patrick Lencioni, Les cinq dysfonctionnements d'une équipe, Un monde différent, Janvier 2005
- Le demi-cercle (épisode 1) - OCTO Talks !
- Ecrire du code propre - Les piliers - OCTO Talks !
- Les artisans codeurs chez OCTO
- " Clean code " de Robert C. Martin
- Le BDD (Behavior Driven Development) - L'essentiel en une image ! - OCTO Talks !

- Fuzzing - Wikipedia
- Software testing - Wikipedia
- Un test peut en cacher un autre - Un peu de théorie - OCTO Talks !
- Un test peut en cacher un autre – Tests unitaires – P1 - OCTO Talks !
- Un test peut en cacher un autre - Tests unitaires - P2 - OCTO Talks !
- Un test peut en cacher un autre - Tests d'intégration - P1 - OCTO Talks !
- Un test peut en cacher un autre - Tests d'intégration - P2 - OCTO Talks !
- La pyramide des tests par la pratique (1/5) - OCTO Talks !
- Remettre la pyramide des tests sur sa base - OCTO Talks !
- " La Guerre des mondes ", film d'H. G. Wells
- " La Haine ", film de Mathieu Kassovitz
- " L'Agence tous risques ", film de Franck Lupo et Stephen J. Cannell
- " Mission impossible ", film de Bruce Geller
- " Objectif Lune ", les aventures de Tintin, bande dessinée d'Hergé
- " RRRrrrr!!! ", film de Alain Chabat
- " Le Livre de la déraison souriante ", livre de Robert Sabatier
- " Highlander ", film de Russell Mulcahy
- Property-Based Testing for everyone by Romeu Moura - Youtube
- " La Gueguerre des étoiles ", sketch de Les Nous C Nous
- " Justice League " et " Injustice League " de DC Comics
- " Star Wars", film de George Lucas


There is a Better Way

OCTO Technology

CABINET DE CONSEIL ET DE RÉALISATION IT

“ Dans un monde complexe aux ressources finies, nous recherchons ensemble de meilleures façons d'agir. Nous œuvrons à concevoir et à réaliser les produits numériques essentiels au progrès de nos clients et à l'émergence d'écosystèmes vertueux”

– Manifeste OCTO Technology -

Dépôt légal : Juillet 2023

Conçu, réalisé et édité par OCTO Technology.
Imprimé par DEJA LINK
ZA de la Cerisaie - 19-27 rue des Huleux - 93240 Stains

Apprivoisez la complexité

Par Christophe Breheret-Girardin, Stéphane Bedeau,
Sylvie Ponthus-Violland

“ Tester, c’est douter ! ” — C’est rigolo, la blague fonctionne. Et en y réfléchissant, peut-être pas. Encore moins la deuxième fois. Quant à la millième fois…

Le sens de cette blague est clair, si nous sommes de bonnes développeuses ou bons développeurs alors nous n’avons pas besoin de perdre du temps à tester.

Cet adage a probablement résonné dans votre esprit lorsque vous avez découvert cet ouvrage. On l’entend systématiquement, il est rarement remis en question.

Dans un contexte de projet, il résonne aussi, et après l’écriture du code, on se précipite vers la livraison.

Cette précipitation est systémique, elle s’inscrit dans la continuité directe des paradigmes du monde qui nous entoure : un monde consommateur et excessif où tout doit aller toujours plus vite pour générer toujours plus de valeur.

Cette précipitation est-elle saine ? Et si tester, c’était mieux faire, plus rapidement ?

À travers une aventure fictive ponctuée d’encarts techniques, ces livres esquissent une philosophie à partir de laquelle on peut forger qualité et efficacité grâce au pragmatisme et à l’humilité.

Ce livre aborde **les tests et leurs déclinaisons, leur intérêt, la place qu’ils prennent dans le monde du logiciel, le lien qui existe avec le métier, leur nature profonde…**

Technology
Part of Accenture

PRIX : 29€ TTC
ISBN 978-2-491672-07-2

9 782491 672072