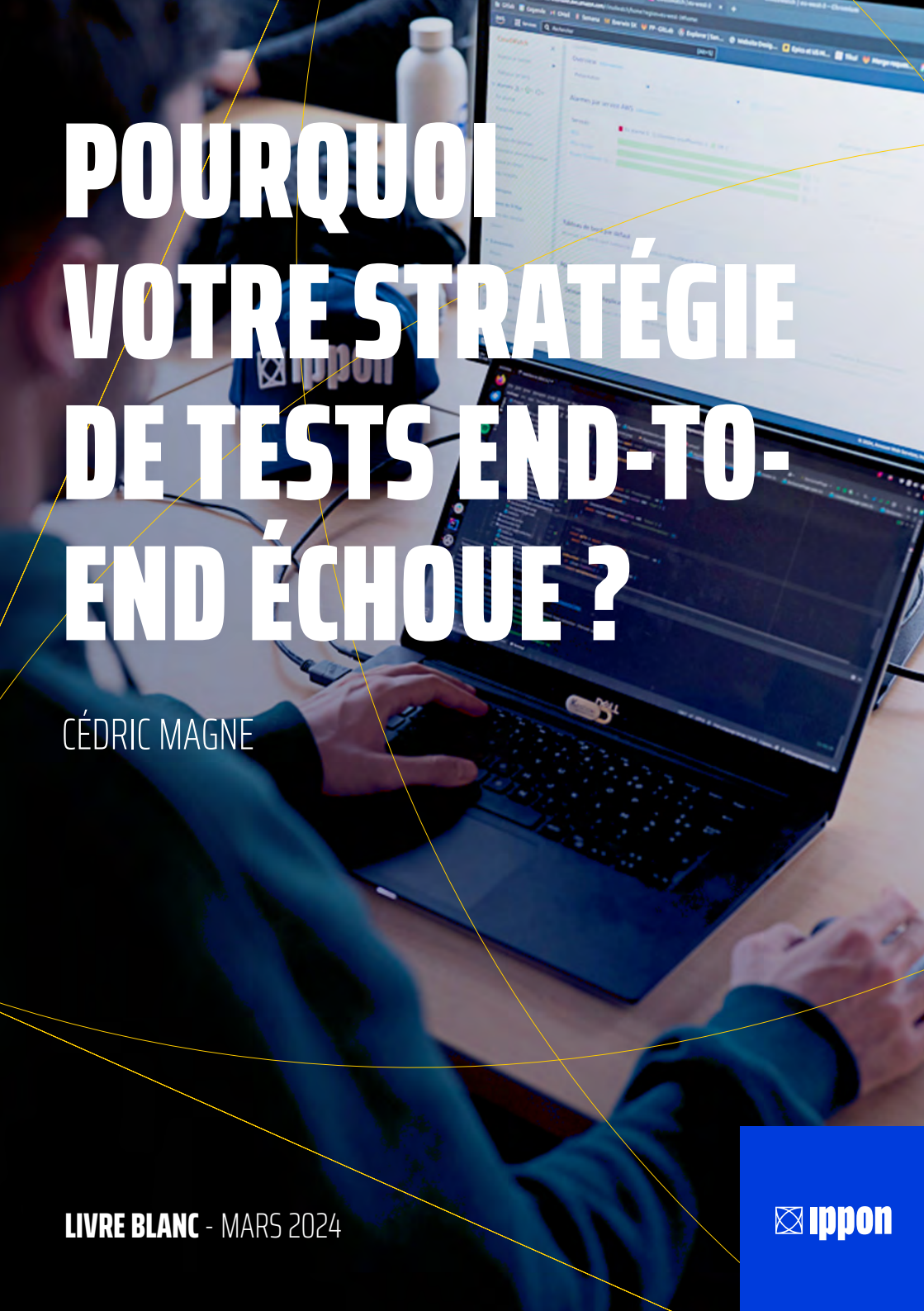


POURQUOI VOTRE STRATÉGIE DE TESTS END-TO-END ÉCHOUE ?

CÉDRIC MAGNE

LIVRE BLANC - MARS 2024

ippon

SOMMAIRE

CÉDRIC MAGNE

Je suis Practice Leader Software Engineering des équipes Ippon Technologies, c'est-à-dire leader du corps de métier des développeuses et développeurs dans la quête de tout ce qui contribue à la cohérence de nos pratiques et de notre bonheur. Après une dizaine d'années de développement à construire des produits dans le monde de l'assurance maladie, du diagnostic médical et du retail, découvrant les mêmes paysages criblés de bugs d'une manière constante à en baisser les bras, j'ai découvert trois consultantes et consultants Ippon qui m'ont fait me questionner sur mes pratiques de développement : "Est-ce que la non-qualité informatique est normale ? Et si j'étais passé à côté de l'essentiel ? Et s'il existait des pratiques permettant d'augmenter drastiquement la qualité de manière maîtrisée ?"

Depuis quatre ans, je me suis plongé dans le monde du "craft", cherchant à améliorer mes compétences en m'appuyant sur la communauté de développement. L'artisanat résonne dans ma vie quotidienne, m'inspirant à explorer des parallèles avec d'autres métiers pour enrichir notre vision du développement logiciel. Ce domaine a profondément influencé ma façon de penser, d'apprendre, de partager et d'aborder la création de valeur.

PREAMBULE

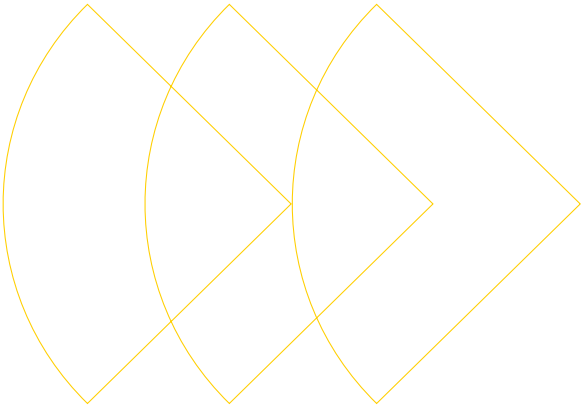
Ce livre est rédigé en mon nom, mais il représente d'une certaine façon un compilé d'apprentissages que j'ai reçu en tant que développeur des équipes Ippon Technologies, pour lesquels je remercie très sincèrement mes collègues de ces quatre dernières années, ainsi que des nombreux partages organisés par les Software Crafters de Lyon, qui contribuent très largement à l'apprentissage collectif que notre corps de métier a la joie de recevoir, gratuitement et sans condition.

Je formulerai un remerciement tout spécial à Hippolyte DURIX, qui a su incarner ce que je décrirais aboutir à une "qualité" dans la suite de ce livre, pendant une année de conférences en duo, à rejouer pas moins de dix fois ce live-coding "Du Bonheur dans le Craft", conférence jouée au Devoxx 2023. Il est un des piliers silencieux de l'artisanat logiciel, et un mentor hors pair.

Aux côtés de mes collègues, j'ai pu développer bien des applications, chacune d'entre elles étant une continuité d'apprentissage vis-à-vis de la précédente. Mais une constante était présente : l'équipe était constituée de passionnés remettant sans cesse en question ses pratiques, prenant toute critique de manière constructive, peu importe le niveau de séniorité de la personne formulant la critique, leur permettant d'aboutir à une pratique tendant vers le zéro bug, avec une réduction drastique des coûts de run, et une réduction drastique des coûts de build que j'avais connu dans mes précédentes vies.

Le développement logiciel souffre de tant de problèmes de qualité, de problèmes de maintenabilité, de convictions mal placées menant à des décisions parfaitement inefficaces, que j'aimerais à travers ce livre écrire une recette de cuisine pour tenter de systématiser l'atteinte d'une qualité "convenable". Diable, ça y est, je ne peux plus faire marche arrière, j'ai osé envisager qu'il puisse exister des façons de faire menant à des résultats plus stables, sereins, souhaitables, ou tout simplement moins coûteux. Irais-je jusqu'à nier avec dogmatisme l'argument confortable ayant dominé l'informatique depuis ces vingt dernières années tout au moins : "ça dépend du contexte" ?

Mais avant d'en arriver là, répondons à la première question faisant l'objet premier de la rédaction de ce livre.



01 POURQUOI VOTRE STRATÉGIE DE TESTS END-TO-END ÉCHOUE ?

En quelques mois j'ai eu l'occasion d'accompagner déjà quatre entreprises ayant la même problématique commençant par :

"NOTRE ÉQUIPE QA (QUALITY ASSURANCE) A MIS EN PLACE UN CHANTIER DE TESTS END-TO-END MAIS [...]"

et finissant par l'une des assertions suivantes :

- "[...] la qualité de nos développements n'a pas augmenté"
- "[...] la maintenance de ces tests est impossible"
- "[...] ils prennent beaucoup trop longtemps à s'exécuter"

Je vais vous détailler ma lecture de la situation, et ensuite parler de qualité au sens large puis au sens pratique. Commençons par un bon vieux "pourquoi ?"

POURQUOI LES QA METTENT EN PLACE DES TESTS END-TO-END ?

La réponse immédiate peut paraître simple : un effet Fordisme[1] de séparation des responsabilités, une approche autonome facile, un "quick-win". Pour autant, une approche pleine de bonne volonté de la part des QA.

Je vais repartir d'une question "provoc" : pourquoi a-t-on des QA sur nos projets agiles[2] ? Soit parce qu'on ne s'est pas re-questionnés depuis les cycles en V[3] et il a été plus commode de garder l'organisation actuelle dans la plupart des entreprises ayant de l'âge. Soit car l'équipe de développement n'arrive pas à nous rassurer quant à la qualité de son livrable. Soit les deux.

1. Fordisme : modèle d'organisation du travail par la division scientifique des responsabilités. Essor du modèle Ford dans la réduction drastique des coûts de construction automobile.
https://fr.wikipedia.org/wiki/Fordisme

2. Agilité : **https://manifesteagile.fr/**

3. Cycle en V : modèle en V ou "waterfall" privilégiant une conception approfondie avant implémentation, puis une phase de test vérifiant et validant chaque point de la conception.
https://fr.wikipedia.org/wiki/Cycle_en_V

Quand un·e QA est recruté·e, son job et son plaisir va être d'aider à tirer la qualité vers le haut. "Par quel bout prendre le chantier ?" De nombreux QA ayant des quotidiens très éloignés de ceux des développeur·euses, à l'écart des relectures de code et des demandes d'aide dans l'implémentation des fonctionnalités, la Loi de Conway[1] intervient ici : *"Toute organisation qui conçoit un système, au sens large, concevra une structure qui sera la copie de la structure de communication de l'organisation."*
La mission de l'équipe QA semble autre que la mission de l'équipe de devs.
Exactement comme la mission des devs semble différente de la mission des ops, donnant naissance à la nécessité du mouvement DevOps.

Une solution pouvant être qualifiée d'héroïque émerge : "on prend le chantier nous-mêmes."

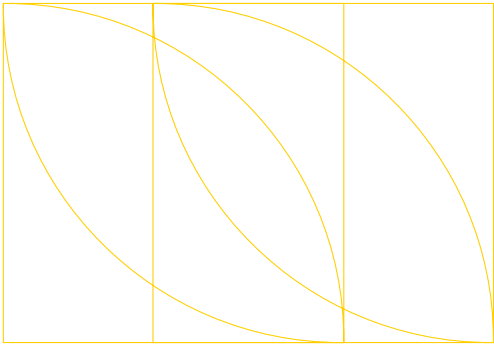
1. Loi de Conway : loi mettant en relation le reflet de l'organisation dans l'architecture mise en œuvre. **https://fr.wikipedia.org/wiki/Loi_de_Conway**

Prendre un chantier en autonomie a le mérite de gaspiller le moins d'énergie à court-terme, générant le moins de charge cognitive pour l'organisation, et d'effort d'alignement entre des visions différentes de la "qualité". Loin des lignes de code aux mains des équipes de développement, on prend le système en boîte noire pour valider son comportement global, et donc valider les critères de validation : plus qu'à sélectionner le bon outil pour implémenter des premiers scénarios de validation, écrire les premiers tests end-to-end, et planifier leur exécution quotidienne en guise de non-régression.

Afin d'analyser la situation et les impasses de cette réponse, nous allons nous baser sur plusieurs prismes de lecture de la pyramide de tests. Revenons sur ce concept qui fait couler beaucoup d'encre d'ailleurs. La pyramide de tests existe sous deux déclinaisons majeures :

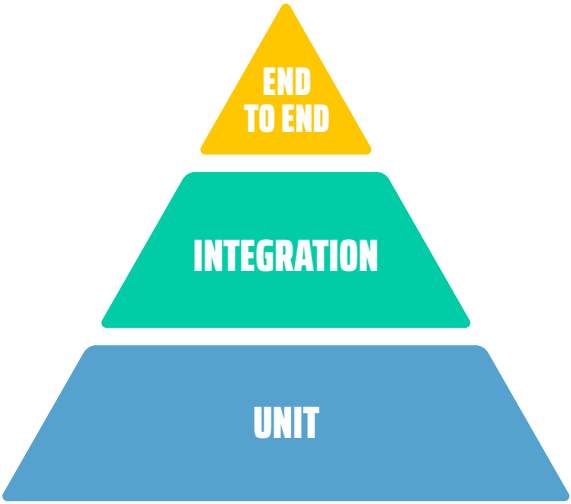
1. L'**ISTQB** (International Software Testing Qualifications Board), principale formation et reconnaissance certifiante du monde de la QA, reprend la théorisation de la pyramide de test par Mike Cohn dans son livre "Succeeding with Agile: Software Development Using Scrum". Elle se découpe en trois niveaux : les tests unitaires, les tests d'intégration, et les tests end-to-end.
2. Les pratiques issues du craftsmanship tendent à varier les subtilités et définir davantage de niveaux : les tests unitaires, les tests d'intégration, les tests de composant, les tests de contrat, les tests end-to-end, les tests exploratoires [...] et à faire apparaître des notions de tests sociaux par Jay Field dans son livre "Working Effectively with Unit Tests"

Et en réalité ces mêmes descriptions sont très incomplètes voire fausses tellement la notion a évolué et n'a pas convergé. Des lectures différentes venant ajouter en confusion sur la définition des termes, que je vous propose de mettre de côté pour l'instant, mais qui pour autant se rejoignent sur la notion de tests end-to-end, étant des tests exécutés sur un environnement déployé, au travers de toute la stack, le plus souvent par des tests UI avec des frameworks tels que Cypress[1] et Playwright[2].

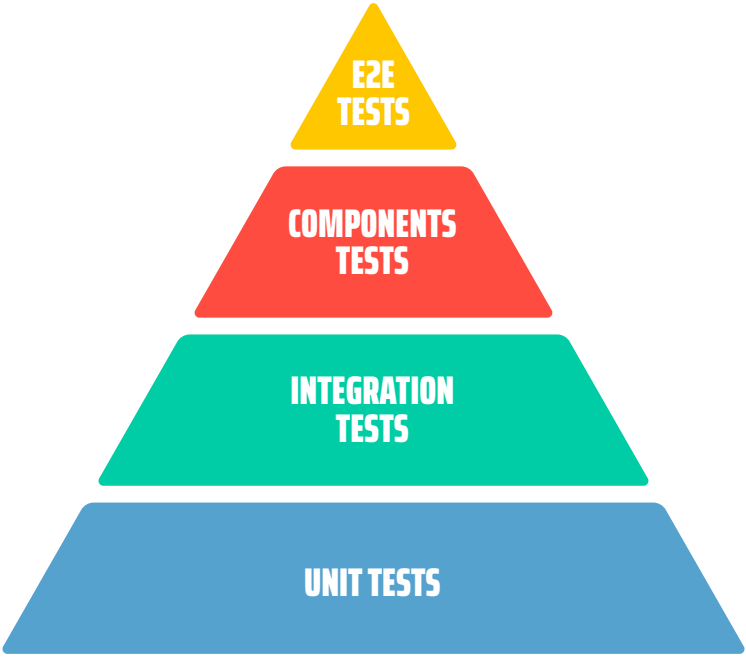


1. Cypress : framework de test frontend permettant de mettre en œuvre des tests de composant ou des tests end-to-end. **https://www.cypress.io/**

2. Playwright : cf Cypress. **https://playwright.dev/**

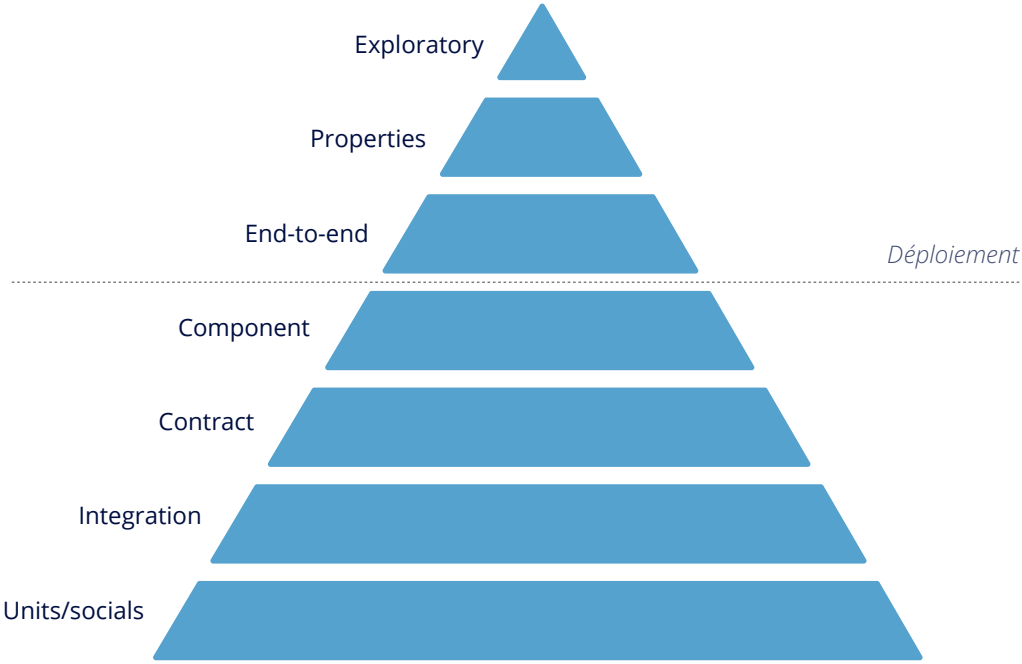


Pyramide de test décrite par Mike Cohn dans “Succeeding with Agile: Software Development Using Scrum”.



Pyramide de test agrémentée des tests de composant largement utilisés dans la culture craft.

La pyramide de tests va nous donner toutes les réponses à l'échec des stratégies end-to-end (e2e) massives. Première chose, elle permet de comprendre une rupture claire dans le cycle de développement, avec une notion de "déploiement" :



Dans le cycle de développement, on y analyse deux sous-parties :

— **le bas de la pyramide** : je suis dev, en local je lance mes tests unitaires jusqu'à mes tests de composants, de manière isolée. Dans un écosystème backend Java et frontend TS, on aurait du JUnit, du Cucumber, du Jest, du Cypress…

— **le haut de la pyramide** : j'ai besoin de tous les composants déployés, c'est quand je push sur ma MR/PR que l'histoire aura une chance de commencer. Soit sur ma MR/PR[1] les e2e vont se lancer, soit c'est au merge, soit une fois par jour… dans tous les cas, c'est hors "local", et ici commence le premier biais.

1. MR/PR : Merge Request et Pull Request, mécanismes de relecture et validation des changements sur le code source avant intégration dans l'arborescence principale.

On a tendance à ainsi épouser la séparation des responsabilités qui en découle :

— les devs s'occupent d'améliorer la qualité du code, à comprendre "ce qu'on ne peut pas toucher car c'est votre repo de code, votre workflow en local"

— les QA s'occupent des environnements déployés : lançons un chantier de tests end-to-end.

La frontière est franche, claire, précise. La Loi de Conway s'applique parfaitement, et tous les travers qui vont en découler. La posture même de traiter la qualité avec Fordisme est vouée à l'échec, parce que la qualité est une résultante de la construction du produit.

En transition, traiter "la qualité" sans tenter de définir ce que le mot vient abstraire, c'est oublier une belle partie de l'histoire : la maintenabilité et l'évolutivité. La **maintenabilité** c'est notre aptitude à maintenir notre produit en état de marche, à moindre coût. L'évolutivité est notre aptitude à poursuivre les améliorations sur le produit, tout en maîtrisant un coût à hauteur de la valeur apportée, autrement-dit en limitant ces coûts. Venons-en à ce coût :

POURQUOI EST-IL CHRONOPHAGE DE MAINTENIR CES TESTS ?

Plus on monte haut dans la pyramide de tests, plus les tests couvrent de fonctionnalités et de liants technologiques. Un test end-to-end va donc couvrir directement une grande partie fonctionnelle, et par effet de bord, la bonne communication entre tous les composants technologiques : la connexion à la base de données, les ouvertures réseau, l'authentification sur une API, une partie du contrat entre le frontend et le backend [...] Ils ont l'air délicieux de prime abord ! Et c'est plutôt vrai, ils le sont. Dans la modération. Mangez un Paris-Brest vous serez ravi·es, mangez en 72 d'une traite et vous commencerez à avoir une émotion différente...

Plus on monte haut dans cette pyramide, plus il devient complexe d'écrire et de maintenir les tests. Un test unitaire s'écrit en quelques secondes. Un test de composants en quelques minutes, on y cible souvent des appels API ou de services. Un test e2e prendra lui quelques dizaines de minutes voire de l'ordre de l'heure : on voudra cibler un parcours utilisateur·rice. Et ces temps sont très dépendants du contexte et des exemples déjà existants, l'ordre de grandeur est le jour dans de nombreuses organisations.

Imaginez que votre stratégie soit de couvrir toutes les règles métier, leur combinatoire, et les tests aux limites par cette typologie de tests ? Le temps d'écriture va être abominable, mais le temps de maintenance va être véritablement pire. Nous l'avons vu plus haut : il faudra attendre la fin du déploiement pour corriger nos erreurs. Toute l'organisation perdra un temps monstrueux dans des allers-retours entre du code local et un environnement déployé. **Et ici nous arrivons sur la notion la plus importante de la pyramide de tests : la boucle de feedback.**

FEEDBACK, LE RETOUR D'EXPÉRIENCE PERÇU PAR LES ÉQUIPES

Colin Damon l'aborde déjà dans son article "80 ou 90% de couverture de tests pour un nouveau projet?"[1]. Pour toucher cette notion de boucle de feedback, nous allons devoir nous intéresser à la DevX, la Developer eXperience.

Le quotidien de dev, c'est de faire émerger des comportements métier à partir d'items ou de 'user stories' [...]. Chaque item dans le sprint backlog est théoriquement découpé pour apporter de la valeur rapidement. Son développement est un changement qui répond aux critères de validation : des règles métier strictes, avec des limites et des invalidations. Ainsi qu'un comportement métier global, avec potentiellement un impact sur une autre fonctionnalité.

Si on part dans un cycle de développement de plusieurs dizaines de minutes, voire heures, voire jours, avant d'avoir la certitude que notre développement répond aux critères d'acceptance, on s'expose au risque quasi systématique de devoir corriger des erreurs, avec une efficacité organisationnelle très pauvre, dans un ordre du pire au mieux :

nous avons un bug en production : nos utilisateur·rices perdent du temps et de la satisfaction, l'équipe entière perd du temps, on priorise, on rédige des tickets [...]

notre QA a détecté une erreur avant que la feature parte en prod' : on rédige un ticket ou on ping le ou la dev, on priorise, notre cerveau switche de contexte, on ouvre une nouvelle MR/PR, on corrige, et c'est reparti

nos tests end-to-end ont détecté une erreur : on est déjà passé sur un autre sujet, on n'allait pas attendre à ne rien faire d'avoir le feedback des tests. En théorie on reçoit une alerte quelque part disant qu'un test est KO, l'équipe s'affole, on cible vite le problème, on corrige [...]

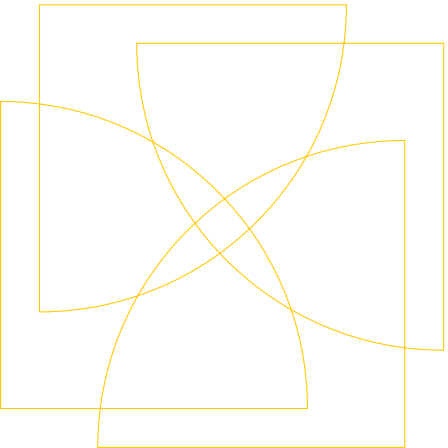
nos tests de composant ont détecté une erreur : on lance les tests, on se fait couler un café, le test de composant cible le problème, et on corrige.

nos tests unitaires ont détecté une erreur : on lance les tests, on se lève pour se faire couler un café... ah bon ben on se rassoit, on a déjà le feedback.

1. **https://blog.ippon.fr/2019/07/22/80-ou-90-de-couverture-de-tests/**

Une lecture de la pyramide de tests serait d'en déterminer le nombre de tests par niveau : il y a une part de vérité liée au feedback, mais il n'y a pas de calcul mathématique du nombre de tests par niveau. Pour prendre un seul exemple, dans une implémentation backend Spring/Cucumber, un test de composant en boîte noire aura un feedback moins bon qu'une implémentation frontend Angular/Cypress. Il sera donc préférable de systématiquement réadapter la stratégie de tests au feedback que cette stratégie vous procure.

Dans des pratiques comme **Test-Driven Development**[1], on aborde un problème à la fois. Le feedback est constant : chaque minute passée est une assurance vers une complétion sans erreur de notre développement (enfin, personne n'est infaillible), grâce aux différentes typologies de tests. On avance, une règle fonctionnelle après l'autre, et étape par étape, on finit notre cycle de développement. 0 temps de perdu, 0 douleur organisationnelle, une DevX absolue. **Rien que le geste de lancer l'application en local devient un non-sens.**



1. **https://blog.ippon.fr/2023/01/18/mon-apprentissage-du-tdd/**

VERS UN ÉVEIL DU RÔLE DE QA

Ne vous méprenez pas : j'aime les tests end-to-end. Mais comme le Paris-Brest. Un seul parcours critique de bout-en-bout, il est possible que ça suffise. Deux à la rigueur. Mais surtout, on compte sur eux en dernier recours.
Car **détecter un problème en end-to-end est une victoire pour la prod', mais un échec pour l'efficacité de l'équipe**. Il est important de chercher la confiance presque absolue en la pertinence de nos tests "locaux" (unitaires à composants). C'est notre ceinture de sécurité. Le end-to-end final, c'est la bretelle.

Alors si on n'écrit pas 72 end-to-end, à quoi vont servir nos QA ? Et si le rôle des QA n'était pas d'être des héro·ïnes de la qualité mais des coachs et des accélérateur·rices de la qualité ?
Comme la culture DevOps nous enseigne que les ops doivent être au contact des équipes dans le quotidien pas pour "faire tout seul" mais pour "faire ensemble". Comme Scrum nous enseigne que les Scrum Master sont des facilitateur·rices de l'agilité, devant accompagner leurs équipes dans l'adoption de pratiques agiles.

Il me paraît essentiel que de nombreux·ses QA réinventent leur rôle : au sein des équipes, à coacher la pyramide de tests, à coacher TDD, à coacher la boucle de feedback, à acquérir les compétences techniques nécessaires pour bootstrapper la mise en place de Cucumber et de Cypress qui est souvent un véritable frein pour les équipes, à accélérer la mise en place de tests de composants par exemple en pair-programming ou mob-programming avec les devs.
À se tourner également vers les Product Owner pour faciliter l'émergence d'une compréhension unifiée des concepts au sein du produit et de l'équipe, par exemple avec un langage omniprésent[1]. Et je n'ai même pas abordé la notion de qualité et maintenabilité du code livré ici, mais que des tests. Il reste donc véritablement une belle marge de manœuvre pour couvrir ce mot "qualité".

1. Langage omniprésent communément anglicisé "Ubiquitous language" : la construction d'un vocabulaire unifié afin de s'aligner sur une même définition des termes, en approfondissant leur sens avec une rigueur permettant de soulever des incohérences ou des divergences de compréhension.
https://martinfowler.com/bliki/UbiquitousLanguage.html

02 QUALITÉ MÉTAPHORIQUE

FAITES BOUGER UN MIKADO ET ALLEZ EN PRISON SANS PASSER PAR LA CASE DÉPART

Dans notre corps de métier nous utilisons la métaphore du plat de spaghetti pour représenter le nouage complexe de nos solutions informatiques. Or dans mon imaginaire, le chaos du plat de spaghetti est parfaitement désirable et en réalité me fait envie. Un plat de spaghetti bien rangé bien propre et sans aucun entremêlage ne serait-il pas morne et dénué de sens ?
Un plat de spaghetti est une création artistique telle qu'en est la cuisine, un art des cinq sens.
Notre profession est avant tout celle de la construction, telle des charpentier·ères. Nous ne vivons pas dans l'émotion intense et éphémère, mais dans des "dogmes" souhaitables : l'absence de défaut, l'évolutivité, la maintenabilité, la robustesse [...] Bien qu'amusants car les règles s'y prêtent, les jeux de Mikado et de Jenga sont un bien meilleur reflet des applications informatiques et des sentiments que nous provoquons à travers les repositories de code source chez les développeuses et développeurs : le challenge, la crainte, l'absence de contrôle, l'échec.
Nous évoluons très largement dans un monde de l'effet de bord, de l'effet Domino, de l'effet papillon, de la régression. Les katas de code sont de très bons moyens d'appréhender les façons qualitatives pour remédier à ces problèmes, et à l'image du Mikado et du Jenga, ils trouvent leur amusement dans le fait qu'ils soient parfaitement décorrélés de la production, de la pression négative, et des coûts en tout genre.

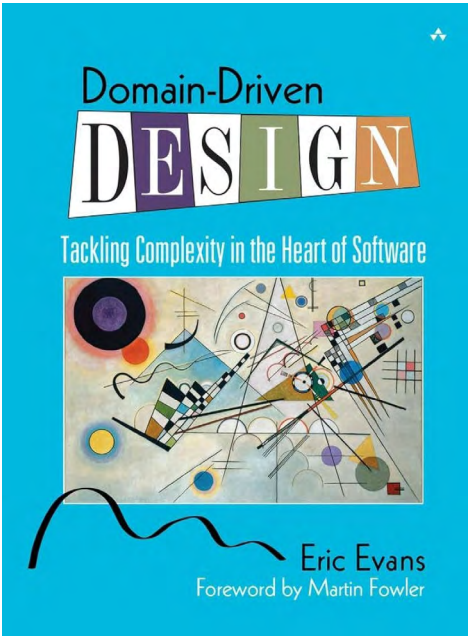
Est-ce qu'ils seraient autant pratiqués si la conséquence de faire bouger un Mikado ne serait pas de passer la main à l'autre joueur, mais de devoir effectuer une réunion de crise pour comprendre les tenants et aboutissants de notre échec ? Tracer scrupuleusement un ticket dans notre journal des incidents (backlog) ? Recevoir un courrier de critique dans notre boîte aux lettres (note sur les apps) ? Qu'est-ce que cela donnerait si changer les tuiles d'une toiture pouvait provoquer des fuites d'eau au rez-de-chaussée, un effondrement de la maison, ou des court-circuits à chaque interrupteur ?

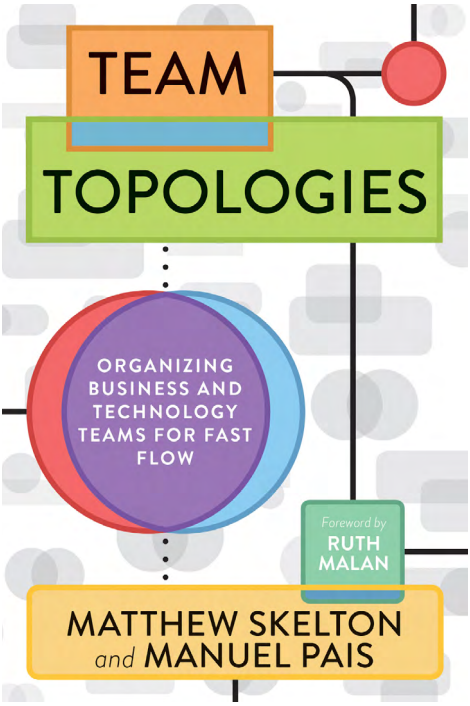
DÉCOUPER EN PLEIN DE PETITS TAS DE MIKADO ?

Le Mikado est également intéressant dans son lien avec la charge cognitive théorisée par John Sweller. La mémoire de travail gère une quantité d'informations très limitée du nombre magique "sept, plus ou moins deux" de George Armitage Miller. Notre échec au Mikado réside en partie dans notre inhabileté (autrement dit nous avons des gros doigts), mais également dans le fait que notre vue et notre cerveau sont en incapacité de nous prévenir de l'échec imminent. Nous prenons des risques involontaires et volontaires à la fois, car "c'est trop".

Pourtant, arrivé aux cinq derniers Mikado (oui cinq, moi aussi j'ai le droit à mon nombre magique, et si vous le souhaitez plus ou moins deux...), le jeu s'accélère, les doutes s'estompent, on gagne en confiance, l'effet de bord peut arriver mais est discernable très aisément, nos risques deviennent à 100% volontaires car nous ne souhaitons pas laisser la partie facile à notre adversaire, à nous de tout rafler.

Et si nous ne jouions qu'avec des paquets de cinq bâtonnets plutôt que quarante-et-un ? sept tas de cinq, et un tas de six. Et bien ce serait profondément facile à tel point qu'il faudrait réinventer de nouvelles règles pour ajouter en challenge.





C'est ce que viennent prouver des livres comme **"Team Topologies"** de *Matthew Skelton* et *Manuel Pais*, ainsi que "Domain-Driven Design: Tackling Complexity in the Heart of Software" d'*Eric Evans*.
Il est plus facile d'aborder plusieurs problèmes simples ou compliqués, qu'un problème complexe ou impossible.
Merci Captain Obvious.

Une façon de gagner en sérénité passe par les pratiques de conception stratégiques et tactiques. Je ne parle pas de "document de conception", ou de "phase de conception", mais de finalité de cette conception.

C'est ce que les corps de métiers du bois ont compris depuis des temps immémoriaux. Notre corps de métier est jeune, de l'ordre de quelques dizaines d'années, et notre communauté échoue majoritairement à le comprendre. Il est absolument certain que des charpentier·ères et menuisier·ères sortent d'école en sachant appliquer des tenons et mortaises pour assembler qualitativement du bois. Pourtant nos écoles échouent à former notre corps de métier aux patterns d'assemblage des solutions informatiques : comment faire cohabiter un morceau de code métier avec Vue, Spring, PostgreSQL, AWS, ou l'API d'un partenaire ? Pour chacune de ces situations, il y a pourtant une recette de cuisine.

Mince je suis revenu à la cuisine : à l'image de la cuisine, il peut y avoir plusieurs recettes, mais à l'inverse de la cuisine, nous n'avons pas **besoin** d'avoir plusieurs recettes tant que l'objectif final de robustesse, d'évolutivité et de sérénité est assouvi. Disclaimer : je ne pointe pas du doigt l'école, c'est un problème systémique dans laquelle l'école arrive en bout de chaîne. Si la communauté adopte une pratique, alors l'école vient à former par effet de bord ses étudiant·es, car les professionnel·les apportent alors ces connaissances en cours.

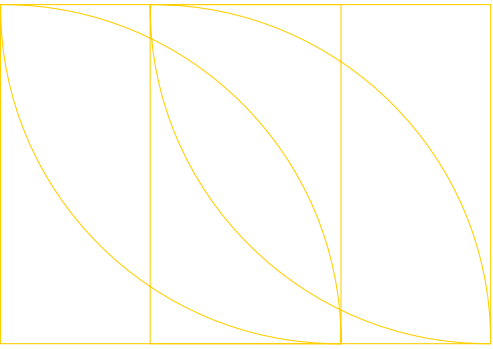
La même chose est arrivée sur l'intégration continue (CI) autour de 2015. En 2010 ou avant, les étudiant·es ne savaient pas ce qu'étaient GitLab CI, GitHub Actions, Jenkins [...] Pourtant à présent nous n'avons pas un seul entretien d'embauche où ce concept et cette mise en pratique n'a pas été déjà abordée par la recrue. L'école est globalement un reflet de la communauté avec 10 ans de retard.

*Extrait de notre conférence avec **Hippolyte DURIX** du Bonheur dans le Craft, Devoxx 2023, mettant en parallèle les corps de métier du bois, avec le corps de métier du développement logiciel.*

LA RÉSIGNATION DE NOTRE MÉTIER FACE À L'ABSENCE DE QUALITÉ

Dans l'informatique la non-qualité est devenue (ou est restée) la norme. Il est normal pour une équipe de corriger des bugs, plusieurs par semaine, voire par jour. Se rapprocher du zéro bug semble être devenu une chimère pour beaucoup, au point où le fait de l'aborder provoque des réactions à la limite de la provocation ou du rire nerveux. Que ce soit en soutenance auprès des clients "Le rêve, j'aimerais bien [...]", ou au sein des équipes "C'est beau l'utopie [...]".

Plusieurs expériences professionnelles m'ont plongé dans les abaques de chiffrage des ESN géantes où le coût de correction des bugs est automatiquement calculé à 10% des coûts de build. Le plus triste est que sa mise en application démontrait la véracité du modèle pour de nombreux projets. Voire parfois le dépassement budgétaire sur les défauts.



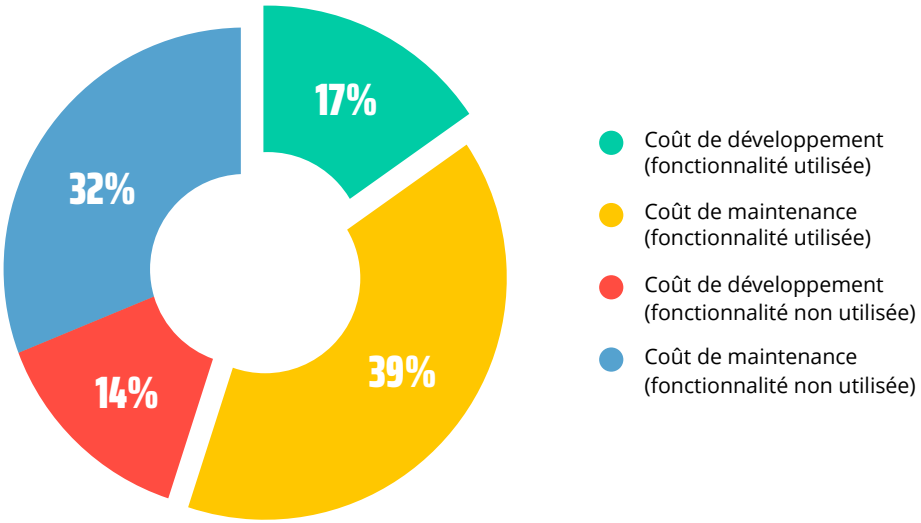
Pire, les coûts de maintenance (bugs, incidents, montées de version...) dans l'informatique représentent en moyenne 70% du budget alloué au développement au sens large (source **TechnoSys**[1] et source **geeksforgeeks**[2]), contre 30% alloué au développement des nouvelles fonctionnalités.

Sombrons davantage, si l'on recroise avec la sobriété, base première de l'éco-conception, 45% des fonctionnalités informatiques ne sont jamais utilisées (source **GreenIT.fr**[3]).

Poursuivons l'énumération, il m'a été courant de participer à des organisations où une personne devait être consacrée au "run" (traitement des incidents, des bugs, déblocage des clients [...]). J'ai été frappé par les élans d'imagination mis en place par les équipes pour atténuer leurs douleurs. Dans le pire des cas, la gestion de projet dédiait des personnes au correctif des bugs sur plusieurs mois. Dans le meilleur des cas, les développeurs mettaient en place des rotations hebdomadaires ou quotidiennes pour se répartir la charge.

De quoi motiver la nécessité de s'attarder à ce fatalisme non ? Est-ce qu'il est utopiste de tendre vers un 0 bug ?

1. **https://devtechnosys.com/insights/software-maintenance-costs/**
2. **https://www.geeksforgeeks.org/cost-and-efforts-of-software-maintenance/**
3. **https://www.greenit.fr/2014/02/21/sobriete-fonctionnelle-la-cle-de-l-ecoconception-des-logiciels/**



Or de nombreux indices se glissent dans le **manifeste Agile**[1] et le **manifeste Craftsmanship**[2], ainsi que tous les partages de connaissance qui en ont découlés.

Ces deux mots permettent dans la théorie de se consacrer sur les 17% de budget utile, et d'amoindrir considérablement les autres parts de gaspillage. Agilité : "build the right thing", Craftsmanship : "build the thing right". Ce croisement de culture permet de rappeler l'importance de faire les choses bien, dès le début, via le gain en coût représenté dans sa globalité. Il ne devrait pas exister de "quick & dirty" (*ou alors très à la marge sur des cas parfaitement maîtrisés, où les coûts induits par la refonte future ou par la perte d'image de marque lors de dysfonctionnements sont mis sur la table avec les décideurs métier : "est-ce que vous acceptez d'avoir 10 bugs en production par semaine ?"*). La qualité n'est pas un surcoût, c'est parfaitement le contraire : les études précédentes le démontrent sur les coûts de maintenance excessivement hauts induits par la non-qualité des approches courantes, de plus l'utilisation d'approches qualitatives n'apporte aucun surcoût sauf celui de l'apprentissage en lui-même, comme toute approche ou toute technologie. Il convient donc de mettre en place des approches focalisées sur la valeur utilisateur (détecter ce qui sert réellement, se confronter au terrain), et des approches focalisées sur l'évolutivité et la qualité logicielle pour écraser les coûts de maintenance et de traitement des bugs. Stop à la complexité accidentelle, oui à la qualité non-accidentelle.

1. **https://agilemanifesto.org/**
2. **https://manifesto.softwarecraftsmanship.org/**

03 UNE RECETTE DE CUISINE POUR RÉDUIRE LA COMPLEXITÉ

REVENONS UNE DERNIÈRE FOIS À LA CUISINE. IL EST TEMPS D'ENVISAGER DANS L'INFORMATIQUE QU'IL PUISSE EXISTER DES RECETTES DE CUISINE.

Nous avons trop fait aléatoirement, sans capitaliser sur les connaissances de nos prédécesseurs qui nous ont légué des boîtes à outils et approches multi-générationnelles et multi-technologiques comme Domain-Driven Design, Test-Driven Development, Behavior-Driven Development, Architecture Hexagonale, Rich Domain, Pyramide de tests [...].

Arrêtons les POJO `@Data` `@Entity` `@Toutelaterre` `@ManyToMany` nous laissant à la merci des sacs de noeuds empêchant toute évolutivité souple.
Arrêtons de mélanger les technologies devant être montées de version tous les mois et le comportement métier de l'application garantit à nos utilisateurs. Une montée de version Spring Boot, JPA ou Jackson ne devrait jamais risquer de faire régresser le fait qu'un virement bancaire ait un émetteur et un destinataire.
Arrêtons les monolithes distribués qui alourdissent les coûts d'infrastructure et les coûts de refactoring.

Vous trouverez très certainement d'autres recettes de cuisine, meilleures je le souhaite ! Ou mieux adaptées à votre contexte ! Celle-ci a le mérite d'être relativement stable dans le temps, donc avec un faible risque d'obsolescence dans l'espace d'une carrière, les concepts étant les mêmes depuis le début des années 2000, et a été éprouvée sur au moins 5 projets avec des résultats identiques. *J'ajouterais un disclaimer : les ustensiles mis en avant dans la recette peuvent sembler difficiles d'accès pour certaines personnes, toutefois ma propre expérience me fait largement relativiser dans l'effort nécessaire à l'apprentissage d'une approche comme l'architecture hexagonale, vis-à-vis de l'apprentissage de n'importe quelle technologie comme Angular.*

Résultats escomptés :

Coût de run <1%

Max 1 à 2 bugs par mois

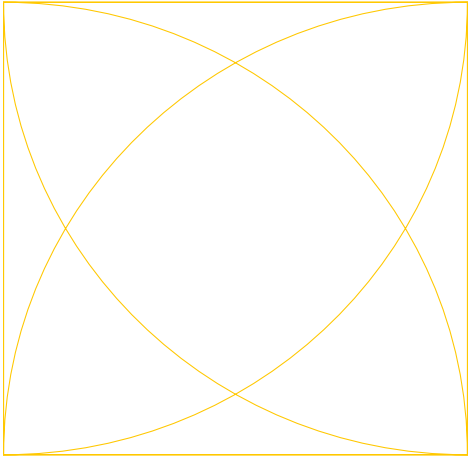
100% de code coverage pertinent

Ingrédients minimaux à une réussite optimale :	Ustensiles souhaitables :
Un lead dev “craft” souhaitant accompagner son équipe	Architecture hexagonale (variante possible vers la Clean architecture, même principe sous-jacent de séparation des responsabilités)
Des développeurs ayant une forte volonté d’apprendre à maîtriser la qualité	Domain-Driven Design : Bounded Context, Value Object, Entity, Aggregate
Une autonomie de l’équipe de développement sur ses méthodes de travail et ses choix de conception	Test-Driven Development
	Pyramide de tests : Tests unitaires, Tests sociaux, Tests d’intégration, Tests de composant
	EventStorming

Étapes

01

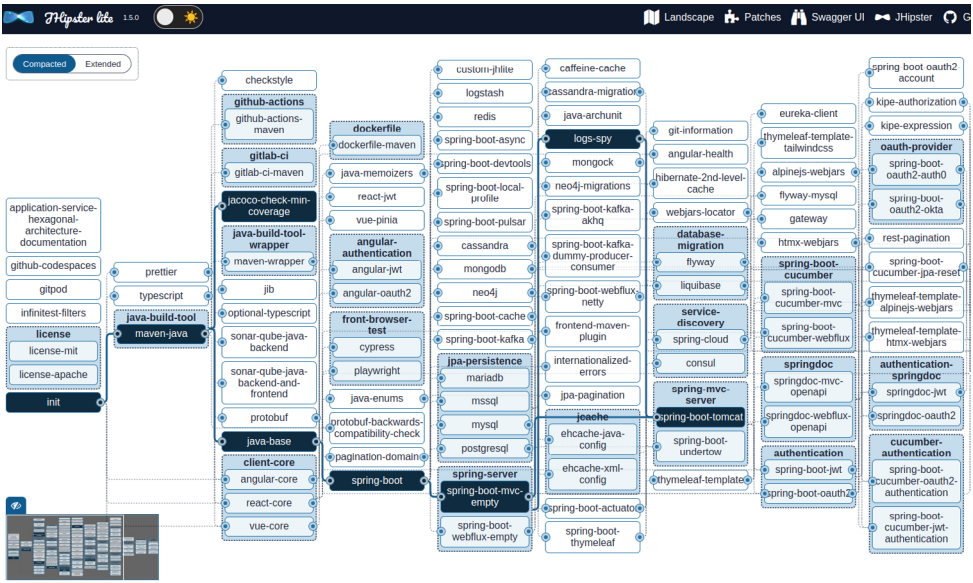
Organisez un EventStorming. Même organisé trop tard, l'EventStorming est redoutable pour monter rapidement en connaissance sur le problème à résoudre, aligner et impliquer toutes les personnes de l'équipe, ainsi que pour détourer une première représentation des bounded contexts permettant de "commencer quelque part". Spoiler alert : les bounded contexts détourés seront très probablement faux, ce n'est pas grave. Ils se découvrent réellement pendant l'implémentation, mais il est primordial de considérer que la réussite d'une conception est reflétée par la facilité qu'on a à refactorer le code pour incarner les principes d'architecture émergente et d'architecture vivante. Il est vital de savoir changer son implémentation. Souvent. Le début de l'immobilisme "parce que c'est trop compliqué" "parce que ça prend trop de temps" marquera probablement la descente aux enfers vers le concept de dette technique non assumée.



02

Ouvrez le générateur **JHipster-Lite**[1] et générez une première application héritant de toute la configuration de tests de composant, unitaire, d'architecture, coverage, avec un exemple de fonctionnalité en archi hexagonale. Si votre écosystème n'est pas supporté par JHipster-Lite, alors recréez par vous-même la configuration nécessaire. Tests de composant obligatoires.

1. **https://lite.jhipster.tech/landscape**



03

En Gherkin (langage naturel), écrivez votre premier comportement global de l'application. Ignorez dans la rédaction l'utilisateur connecté, rédigez en point de vue de l'application, et n'introduisez absolument jamais le moindre détail technique. "Etant donné que l'émetteur dispose d'argent sur son compte, Lorsqu'un virement bancaire est effectué, Alors le solde du créditeur est crédité". Surtout pas "Etant donné que Jack est connecté en tant qu'utilisateur Et que Jack a 200€ sur son compte, Lorsque Jack fait un virement vers Daniel Et que le virement répond HTTP 200 OK, Alors Daniel consulte ses comptes Et Daniel a 200€ sur son compte". Mais partons sur du jardinage pour un peu de fraîcheur et un second exemple de rédaction :

```
Feature: Gardening is extraordinary, things happen.

Scenario: Sowing a crop should give a vegetable in my garden
   When a crop is sown
   Then the garden has a vegetable planted
```

04

A cette étape vous avez un test de comportement non implémenté. Implémentez-le en Cucumber sur vos appels HTTP si c'est un backend ou Cypress sur votre parcours frontend (ou autre technologie de test de composant au sens "boite noire" et non au sens du composant graphique frontend). Le test de comportement est rouge quand vous l'exécutez, c'est normal, vous n'avez fait aucun code associé pour l'instant. Nous entrons alors dans le domaine de l'Outside-In TDD[1] : nous avons défini le comportement global de la fonctionnalité (outside), et pouvons à présent nous plonger dans son implémentation interne (inside).

1. Outside-In TDD : Pratique du TDD permettant de définir le comportement d'une fonctionnalité vue de l'extérieur, et donc comme un utilisateur devrait la percevoir, avant de se plonger l'implémentation sous-jacente qui aidera à faire émerger toutes les bonnes questions aux limites du cas nominal. Outside-In TDD est une pratique en opposition à Inside-Out TDD (ne pas comprendre "opposition" au sens de "conflit", mais au sens de la démarche de découverte du besoin et d'émergence de son implémentation.
https://cucumber.io/blog/hiptest/why-you-should-be-using-outside-in-development/

```
package opm.garden.permaculture.primary;

import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import opm.garden.cucumber.CucumberRestTemplate;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;

import java.util.List;
import java.util.Map;

import static opm.garden.cucumber.CucumberAssertions.*;

public class GardenSteps {
 private static final String CROP_TEMPLATE = """
   {
     "variety": "{VARIETY}"
   }
   """;

 @Autowired
 private CucumberRestTemplate rest;

 @When("a crop is sown")
 public void sowACrop() {
   String payload = CROP_TEMPLATE.replace("{VARIETY}", "salad");

   rest.post("/api/crops", payload);

   assertThatLastResponse().hasHttpStatus(HttpStatus.CREATED);
 }

 @Then("the garden has a vegetable planted")
 public void shouldHaveASeedling() {
   rest.get("/api/vegetables");

   assertThatLastResponse().hasOkStatus().hasElement("$.vegetables")
     .containingExactly(List.of(Map.of("variety", "salad")));
 }
}
```

05 Implémentez l'infrastructure primaire de votre architecture hexagonale, c'est-à-dire la partie HTTP ou composant graphique.

```
package opm.garden.permaculture.primary;

import io.swagger.v3.oas.annotations.Operation;
import io.swagger.v3.oas.annotations.responses.ApiResponse;
import opm.garden.permaculture.application VegetablesApplicationService;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.annotation.Validated;
import org.springframework.web.bind.annotation.*;


@RestController
@RequestMapping("/api")
class GardenResource {

 private final VegetablesApplicationService vegetables;

 public GardenResource(VegetablesApplicationService vegetables) {
   this.vegetables = vegetables;
 }

 @PostMapping("/crops")
 @Operation(summary = "Sow a crop in the garden")
 @ApiResponse(description = "Crop has been sowed", responseCode = "201")
 @ResponseStatus(HttpStatus.CREATED)
 void sowACrop(@Validated @RequestBody HttpCrop cropToSow) {
   vegetables.sow(cropToSow.toDomain());
 }

 @GetMapping("/vegetables")
 @Operation(summary = "See all vegetables of the garden")
 @ApiResponse(description = "Collection of found vegetables",
 responseCode = "200")
 ResponseEntity<RestVegetables> getGardenVegetables() {
   return ResponseEntity.ok(RestVegetables.from(vegetables.all()));
 }
}
```

06

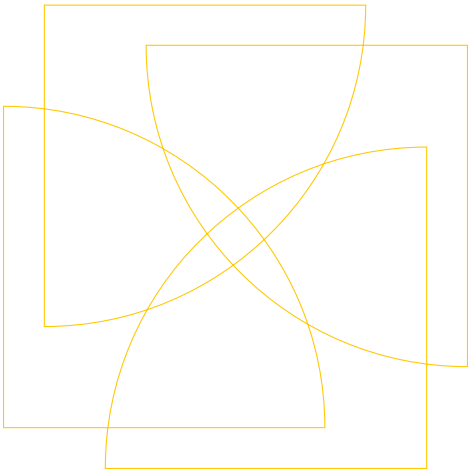
Ne laissez absolument pas de responsabilité fuiter entre les classes. Ici dans une architecture hexagonale, l'infrastructure dépend du domaine métier. Le domaine métier ne doit pas avoir la moindre chance de laisser comprendre l'implémentation technique : API HTTP, listener AWS SQS, base de données Postgresql, tout ceci n'est que détail d'implémentation et ne doit jamais transparaître dans votre domaine. Les Data Transfer Object (DTO) sont responsables de se convertir en objet du domaine :

```
package opm.garden.permaculture.primary;

import io.swagger.v3.oas.annotations.media.Schema;
import opm.garden.permaculture.domain.Crop;
import opm.garden.permaculture.domain.Variety;


@Schema(name = “Crop”, description = “This is you crop, sow it with
pleasure!”)
public class HttpCrop {

 @Schema(description = “Is it garlic or tomato? Which variety would you
like to sow?”)
 private String variety;

 public void setVariety(String variety) {
   this.variety = variety;
 }

 public Crop toDomain() {
   return new Crop(Variety.of(variety));
 }
}
```

```
package opm.garden.permaculture.primary;

import opm.garden.permaculture.domain.Vegetable;

import java.util.Collection;

class RestVegetables {

 private final Collection<Vegetable> vegetables;

 private RestVegetables(Collection<Vegetable> vegetables) {
   this.vegetables = vegetables;
 }

 public static RestVegetables from(Collection<Vegetable> vegetables) {
   return new RestVegetables(vegetables);
 }

 public Collection<Vegetable> getVegetables() {
   return vegetables;
 }
}
```

07

Poursuivez l'implémentation au niveau du service métier. Dans un premier temps ce serait une architecture hexagonale très simplifiée. Imaginez que si un utilisateur demande en temps réel à planter une graine dans son potager via la ressource HTTP, ou si c'est un batch asynchrone dépilé via l'écoute de messages sur une file SQS, ceci ne change en rien le service métier sous-jacent : on plante une graine, et la graine doit être tout aussi valide fonctionnellement.



```java
package opm.garden.permaculture.application;

import opm.garden.permaculture.domain.Crop;
import opm.garden.permaculture.domain.Vegetable;
import opm.garden.permaculture.domain.VegetablesPort;
import org.springframework.stereotype.Service;

import java.util.Collection;

@Service
public class VegetablesApplicationService {
 private final VegetablesPort vegetables;

 public VegetablesApplicationService(VegetablesPort vegetables) {
   this.vegetables = vegetables;
 }

 public void sow(Crop cropToSow) {
   vegetables.save(cropToSow);
 }

 public Collection<Vegetable> all() {
   return vegetables.all();
 }
                                                                  }
```

08 Créez une interface métier qui permettra de définir les attendus fonctionnels : peu importe l'implémentation, qu'il s'agisse d'une base de données Postgresql, d'une base de données MongoDB, de l'appel d'un fournisseur de potager externe MyBeautifulPotager, ou d'un simple stockage en mémoire, rien ne doit transpirer dans cette interface.

```java
package opm.garden.permaculture.domain;

import java.util.Collection;

public interface VegetablesPort {
 void save(Vegetable vegetable);

 Collection<Vegetable> all();
}
```

09

Implémentez le plus simplement possible l'adaptateur secondaire, c'est-à-dire l'implémentation de l'interface, ici avec un stockage en mémoire.

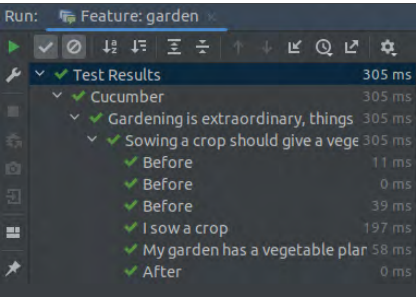
```java
package opm.garden.permaculture.secondary;

import opm.garden.permaculture.domain.Vegetable;
import opm.garden.permaculture.domain.VegetablesPort;
import org.springframework.stereotype.Component;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;

@Component
public class InMemoryVegetablesAdapter implements VegetablesPort {
 private final Collection<Vegetable> vegetables = new ArrayList<>();

 @Override
 public void save(Vegetable vegetable) {
   vegetables.add(vegetable);
 }

 @Override
 public Collection<Vegetable> all() {
   return Collections.unmodifiableCollection(vegetables);
 }
}
```

09

L'implémentation de Crop est celle que vous souhaitez, ici une étape de trop est très clairement proposée, celle d'ajouter une abstraction si tôt, mais je souhaitais que le mot "vegetable" du test de composant se retrouve dans mon implémentation initiale car c'est ce langage omniprésent qui guidera la compréhension de la solution.

```
package opm.garden.permaculture.domain;

import opm.garden.error.domain.Assert;

public final class Crop extends Vegetable {

 public Crop (Variety variety) {
   super(variety);
   Assert.notNull("variety", variety);
 }
}
```

```
package opm.garden.permaculture.domain;

public abstract class Vegetable {
 private final Variety variety;

 protected Vegetable(Variety variety) {
   this.variety = variety;
 }

 public String getVariety() {
   return variety.get();
 }
}
```



10

Lancez le test de composant, il doit être passant à ce stade. C'est-à-dire que la première itération de l'architecture complète et du comportement en boîte noire de votre fonctionnalité est parfaitement valide.

11

Partez à volonté dans le “In” de l’Outside-In TDD. C’est-à-dire qu’il est enfin temps de définir des vraies règles métier de validation et de compléter le modèle pour qu’il soit riche et qu’il s’approche de l’exhaustivité du souhait de l’utilisateur. Premièrement rattrapez le retard sur la combinatoire et la spécification vivante, même si elle était couverte par effet de bord par le test de composant, ces étapes sont normalement définies plus tôt dans la construction.

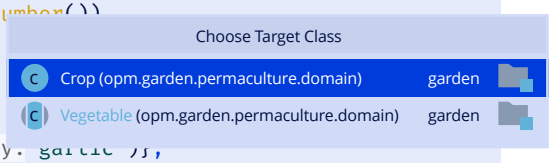
```
package opm.garden.permaculture.domain;

import opm.garden.error.domain.MissingMandatoryValueException;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.NullAndEmptySource;
import org.junit.jupiter.params.provider.ValueSource;

import static org.assertj.core.api.Assertions.*;

class CropTest {

 @Test
 void shouldNotBuildACropWithoutItsVariety() {
   assertThatThrownBy(() → new Crop(null))
     .isExactlyInstanceOf(MissingMandatoryValueException.class)
     .hasMessageContaining(“variety”);
 }

 @ParameterizedTest
 @NullAndEmptySource
 @ValueSource(strings = {“ “, “\t”})
 void shouldNotBuildAnEmptyVariety(String variety) {
   assertThatThrownBy(() → Variety.of(variety))
     .isExactlyInstanceOf(MissingMandatoryValueException.class)
     .hasMessageContaining(“variety”);
 }
}
```

12

Envisagez du réel TDD, c'est-à-dire l'introduction d'un comportement métier vous amenant à pousser votre implémentation.

Exemple de nouvelle fonctionnalité sur le nombre de feuilles :

```
@Test
void shouldGenerateItsOwnNumberOfLeavesWhenSowed() {
  assertThat(garlicCrop).getLeavesNumber())
 }

private static Crop garlicCrop() {
  return new Crop(Variety.of(variety. garlic ));
 }
```




```
class CropTest {
  ...

 @Test
 void shouldGenerateItsOwnNumberOfLeavesWhenSowed() {
  assertThat(garlicCrop().getLeavesNumber()).isPositive();
 }

 private static Crop garlicCrop() {
  return new Crop(Variety.of("garlic"));
 }
}
```

```
public final class Crop extends Vegetable {
  ...

 public Integer getLeavesNumber() {
  return 1;
 }
}
```

Une étape de plus qui nous pousse à gérer une combinatoire plus importante :

```
class CropTest {
  ...

 @Test
 void mintLeavesShouldGrowInPair() {
  assertThat(mintCrop().getLeavesNumber() % 2).isZero();
 }

 private static Crop mintCrop() {
  return new Crop(Variety.of("mint"));
 }
}
```

```
public final class Crop extends Vegetable {
  ...

 public Integer getLeavesNumber() {
   if (getVariety().equals("mint")) {
     return 2;
   }
   return 1;
 }
}
```

Puis de refactorer votre test vers une étape finale faisant apparaître un nouveau mot de vocabulaire souhaitable pour la compréhension du comportement de votre code.

```
class CropTest {
  ...

 @ParameterizedTest
 @ValueSource(strings = {"mint", "basil", "thyme", "oregano"})
 void lamiaceaeLeavesShouldGrowInPair(String lamiaceae) {
   assertThat(new Crop(Variety.of(lamiaceae)).getLeavesNumber() %
   2).isZero();
 }
}
```

```
public final class Crop extends Vegetable {
 ...

 public Integer getLeavesNumber() {
   if (isLamiaceae()) {
     return 2;
   }
   return 1;
 }

 private boolean isLamiaceae() {
   return List.of("mint", "basil", "thyme", "oregano").
   contains(getVariety());
 }
```

Les lamiacées

On les reconnaît par :

- *leurs tiges généralement quadrangulaires (présentant 4 angles et 4 faces)*
- ***leurs feuilles opposées décussées***
- *leurs inflorescences groupées autour de l'axe et sur le même plan*
- *le fait qu'elles sont souvent aromatiques*

La classe gérant les variétés doit posséder à terme cette responsabilité, mais l'exemple s'arrête ici.

13 Lancez vos tests et constatez que la stack JHipster-Lite a bien vérifié le coverage.

```
INFO] --- jacoco-maven-plugin:0.8.8:report-integration (post-
integration-tests) @ garden ---
[INFO] Loading execution data file /home/onepunchmagne/code/garden/
target/jacoco-it.exec
[INFO] Analyzed bundle 'garden' with 41 classes
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.8:merge (merge) @ garden ---
[INFO] Loading execution data file /home/onepunchmagne/code/garden/
target/jacoco/allTest.exec
[INFO] Loading execution data file /home/onepunchmagne/code/garden/
target/jacoco-it.exec
[INFO] Loading execution data file /home/onepunchmagne/code/garden/
target/jacoco.exec
[INFO] Writing merged execution data to /home/onepunchmagne/code/garden/
target/jacoco/allTest.exec
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.8:report (post-merge-report) @ garden
---
[INFO] Loading execution data file /home/onepunchmagne/code/garden/
target/jacoco/allTest.exec
[INFO] Analyzed bundle 'garden' with 41 classes
[INFO]
[INFO] --- jacoco-maven-plugin:0.8.8:check (check) @ garden ---[INFO]
Loading execution data file /home/onepunchmagne/code/garden/target/
jacoco/allTest.exec
[INFO] Analyzed bundle 'garden' with 41 classes
[INFO] All coverage checks have been met.
```

```
<execution>
 <id>check</id>
 <goals>
   <goal>check</goal>
 </goals>
 <configuration>
   <dataFile>target/jacoco/allTest.exec</dataFile>
   <rules>
     <rule>
       <element>CLASS</element>
       <limits>
         <limit>
           <counter>LINE</counter>
           <value>COVEREDRATIO</value>
           <minimum>1.00</minimum>
         </limit>
         <limit>
           <counter>BRANCH</counter>
           <value>COVEREDRATIO</value>
           <minimum>1.00</minimum>
         </limit>
       </limits>
     </rule>
   </rules>
 </configuration>
</execution>
```

100% de code coverage garanti, pertinent, sans aucun surcoût bien au contraire, que du temps libéré à ne pas traiter de bugs, d'issues, à attendre qu'une personne dédiée au test ne valide l'implémentation, à lancer son application en local et tester manuellement.

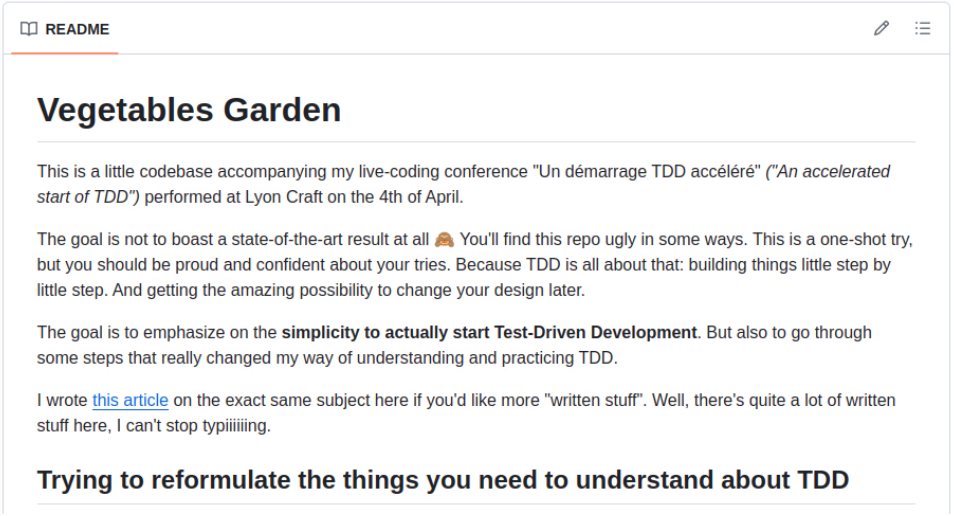
Si les bugs surviennent à nouveau, c'est que le problème est ailleurs :

— Si le besoin est mal compris à l'origine, il y a peu de chance que l'implémentation soit correcte. C'est chose banale à signaler, mais de nombreux problèmes dont j'ai été témoin relevaient d'un manque d'alignement. On sort d'une session de "backlog refinement" avec le cerveau qui va exploser, "Tout le monde a compris ?" "Yes !" Et le lendemain la merge request est à côté de la plaque. Transmettre l'intention métier et se l'approprier n'est pas la chose la plus simple de notre métier. D'où des outils comme l'EventStorming, le Behavior-Driven Development, le pair et le mob programming [...]

— Il est également possible que les bugs surviennent lors d'un angle mort de la couverture de test. J'ai pu voir a minima 3 angles morts dans certaines équipes :
  - Pas assez de TDD = du coverage a posteriori = des tests non pertinents, certains cas de mapping sont loupés ou mal couverts.
  - Des tests de composants aux frontières des bounded context : la communication entre bounded context est en événementiel, il n'y a ni test end-to-end sur ce scénario ni test de composant, et on ne s'aperçoit pas que l'on aurait dû propager l'événement "graine plantée" qui est écoutée par la fonctionnalité d'inventaire des graines restantes.
  - Les tests d'intégration au niveau de la base de données peuvent être pénibles, des utilitaires de base de données comme Liquibase et Spring JPA ne détectent pas de problème à la compilation car le liant est en chaîne de caractères brute, et les spécifications de recherche des repositories ne retournent aucun résultat.

Pour chacun de ces problèmes, une résolution de problème en équipe avec la méthode des 5 “Pourquoi[1]” devrait amener sur une itération souhaitable et maîtrisée. En quelques itérations, l’équipe devrait avoir comblé tous les trous dans sa raquette.

Une autre version de ce travail est disponible sur GitHub entrant bien plus en détail sur la notion de TDD : **https://github.com/onepunchmagne/garden**


README

Vegetables Garden

This is a little codebase accompanying my live-coding conference "Un démarrage TDD accéléré" *("An accelerated start of TDD")* performed at Lyon Craft on the 4th of April.

The goal is not to boast a state-of-the-art result at all 🙈 You'll find this repo ugly in some ways. This is a one-shot try, but you should be proud and confident about your tries. Because TDD is all about that: building things little step by little step. And getting the amazing possibility to change your design later.

The goal is to emphasize on the **simplicity to actually start Test-Driven Development**. But also to go through some steps that really changed my way of understanding and practicing TDD.

I wrote this article on the exact same subject here if you'd like more "written stuff". Well, there's quite a lot of written stuff here, I can't stop typiiiiing.

Trying to reformulate the things you need to understand about TDD

1. **https://fr.wikipedia.org/wiki/Cinq_pourquoi**

CONCLUSION

Bien que la qualité du code soit un élément majeur dans la réduction des coûts, le dernier signe d'importance est la capacité à réagir aux problèmes survenant en production :

— Se forger un regard pertinent sur la production, un regard aux mains de l'équipe de développement, permettant de surveiller et remonter toute alerte nécessitant correction. La culture DevOps est un élément clé à cette réussite, avec des Ops accompagnant l'équipe de développement dans son autonomie de surveillance de la production

— Être en capacité de réagir aux incidents, c'est-à-dire avec toute l'autonomisation et les pratiques permettant une livraison d'un correctif en un temps maîtrisé, et sans craindre de régression. Les tests pertinents apportés par une pratique du TDD, mais également toute la CI et CD permettant une livraison automatisée et fiabilisée des environnements.

Cette double composante décrite de la qualité "pendant le développement" et de la qualité "en situation de production", l'équipe se met en posture de réduire considérablement ses coûts de run (le traitement des bugs devient une activité rare), d'augmenter l'image de marque du produit, et donc la confiance de ses utilisateurs.

Alors, en cuisine ?

Crédits photo :

Photos de **François Delauney**
Photo geste stop généré par **midjourney**
Photo de **Oktavisual Project** sur **Unsplash**
Photos de **Engin Akyurt** sur **Pixabay**
Photos de **Willi Heidelbach** sur **Pixabay**
Photo geste stop généré par **midjourney**

ippon

La stratégie de tests basée sur les scénarios de bout en bout peut s'avérer chronophage et coûteuse, entravant le développement et la vélocité des équipes. Et si nous devions totalement désaxer notre réflexion pour atteindre les clés de compréhension ? L'immersion à travers le quotidien des testeurs et des développeur·euses devrait nous faire réaliser nos défauts.
Prenons un pas de recul, ou deux, et envisageons une recette de cuisine nous permettant de nous rapprocher du zéro bug par d'autres moyens.

www.ippon.fr
blog.ippon.fr
contact@ippon.fr
+33 1 46 12 48 48