Le bon code et le mauvais code

Modèle de plan assurance qualité logicielle

Joël Abenhaïm, 10x développeur

https://www.linkedin.com/in/joel-abenhaim-a514404a

Sommaire

Introduction : Le bon code et le mauvais code

Tout le monde connaît le sketch des Inconnus au sujet du bon chasseur et du mauvais chasseur. Et bien pour le code, c'est à peu près cette histoire: Il y a le bon code qui s'exécute et fait un truc, et puis il y a le mauvais code qui s'exécute et fait exactement le même truc, sauf que c'est pas pareil ! C'est marrant comme intro je trouve, mais avant de parler de ça, laissez-moi vous raconter d'où je viens.

J'ai appris l'informatique tout seul, étant un enfant, chez moi, sans enseignement. Je recopiais des programmes en GW-Basic que je trouvais dans des livres et petit à petit j'ai compris comment ils fonctionnaient. Puis j'ai commencé à les modifier, puis à écrire les miens les années passant, de plus en plus complexes, de plus en plus grands. J'écrivais les programmes n'importe comment, en toute innocence : « *Si ça fonctionnait, c'était forcément parfait du premier coup, pensais-je.* »

Ce que j'ignorais, et que j'ai mis de nombreux projets ratés à comprendre, c'est que si deux codes fonctionnent de la même façon à l'exécution, l'un des deux est plus lisible que l'autre, plus maintenable que l'autre, plus valable que l'autre ! Et ce différentiel augmente avec la taille du code, je l'ai appris à mes dépends. J'ai raté de nombreux gros projets. Dans certains cas, je m'étais trompé de techno – Aujourd'hui on parlerait de stack –, et dans d'autres cas mes programmes avaient atteint une taille critique mais n'avait pas de plan d'assurance qualité.

Cette notion de taille critique est très importante. Au delà d'un certain nombre de lignes de code, environ 50.000, ou d'un certain nombre de fichiers source, environ 500, on ne peut plus se permettre de coder n'importe comment. Aucune mémoire humaine, y compris collective, n'est assez grande pour ranger tous les pièges et tous les effets de bord que contient un code de cette taille. Un projet réalisé librement par des développeurs senior contient malheureusement un certain nombre de défauts de qualité qui ralentissent la maintenance et l'ajout de fonctionnalités.

Alors comment faire ? Il y a des méthodes pour s'y retrouver. Ces méthodes ne sont pas enseignées à l'université, ni dans les grandes écoles, car ceux qui connaissent ces techniques, dont je fais partie, ne font pas d'enseignement ; ils créent des logiciels extraordinaires. Dans ce document, je vais lister quelques unes de ces méthodes, puis je vais en décrire quelques avantages, et finalement je vais essayer d'expliquer dans quelle mesure elles peuvent être appliquées à vos nouveaux projets, mais aussi à vos projets existants.

Un bon code est trouvable

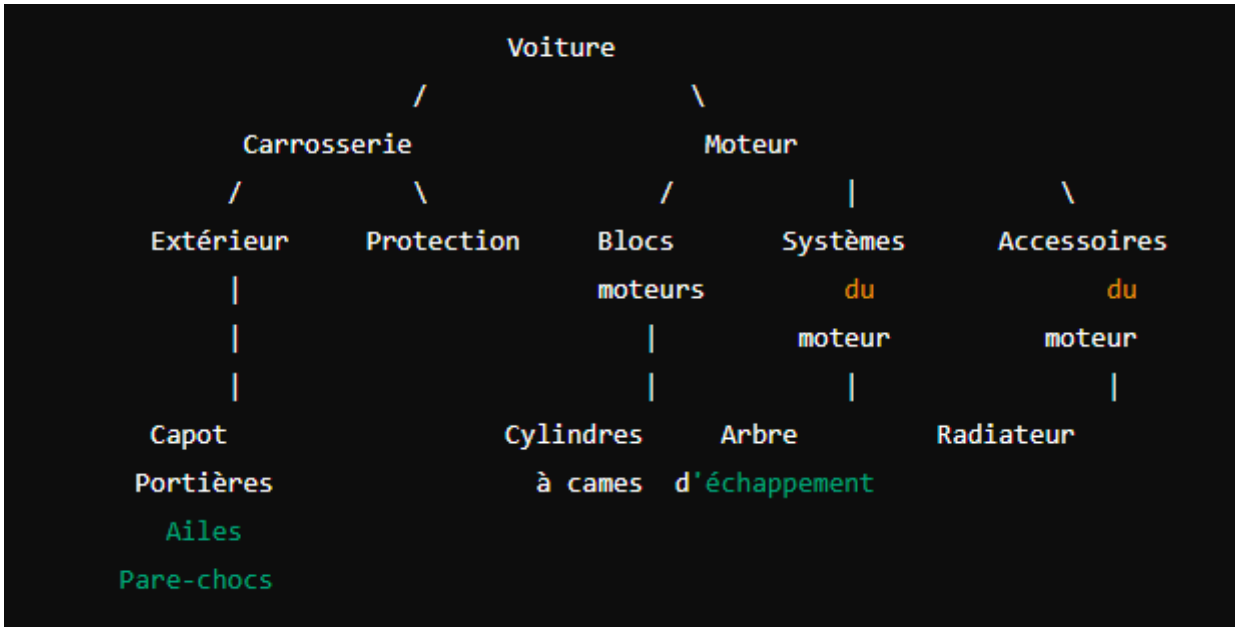
C'est bête à dire, mais la première chose avant de se demander si le code est bien écrit, c'est qu'on puisse le retrouver. Et c'est le défaut de qualité le plus fréquent dans un projet mal réalisé.

On part du principe que l'on veut corriger ou modifier une fonctionnalité, un écran ou une zone d'un écran. Ce qu'on veut c'est rapidement pouvoir déterminer quel ensemble de fichiers source il va falloir modifier pour accomplir la tâche. En général, on a besoin de modifier entre 1 et 10 fichiers, mais dans certains cas pour du refactor il va falloir en modifier un très grand nombre.

Pour pouvoir être retrouvé rapidement, le code a besoin d'être rangé dans un certain ordre. Voici quelques méthodes, non exhaustives, qui permettent de classer le code :

- L'architecture Micro-services est une des façons de catégoriser le code. Chaque micro-service fait une chose particulière, et c'est un premier niveau de classement. On peut aussi utiliser une architecture monolithique avec des plugins ou services, qui classeront tout aussi bien le code. Le choix entre l'utilisation ou non de micro-services n'a pas réellement d'incidence sur le classement par rapport au classement par plugin. C'est plutôt une question de stratégie au niveau gestion des équipes projet.

- La structure des répertoires est une des clés du classement du code. En utilisant un grand nombre de répertoires, et en créant une arborescence profonde, on facilite la recherche d'un morceau du programme.



- Pareillement, on devrait séparer le code dans de nombreux fichiers. Il ne faut pas trop en avoir sinon on aura du code « spaghetti » qui est difficile à lire. Mais il faut bien séparer le

source du programme en fichiers d'environ quelques centaines de lignes si possible. Il faut que chaque fichier ait une étendue fonctionnelle assez grande pour qu'on n'ait pas à ouvrir trop de fichiers à chaque intervention, et assez petite pour qu'il ne soit pas trop compliqué à modifier. Et bien entendu, chaque nom de fichier doit être choisi soigneusement pour qu'on sache ce qu'il fait directement en lisant son nom.

- Le formatage des lignes du code est une autre façon de le classer. Par exemple en utilisant un formateur automatique comme Prettier, toutes les lignes de code auront la même forme, et il sera possible de faire des recherches textuelles ou par expressions régulières, sans manquer aucun élément.

- Le typage est très important pour classer le code. A partir des types d'un programme : Interfaces, classes, héritage, composition, appels de variables, constantes, fonctions typées, un éditeur de code moderne comme par exemple Visual Studio Code est capable de permettre au développeur de naviguer facilement dans le programme.

- Pour les noms des composants du front-end, que l'on utilise du javascript vanilla, ou des outils comme Angular, React, Vue, il faut pouvoir retrouver facilement les éléments, en les nommant de la même façon que les éléments du Dom, ou de la même façon que les éléments du framework. On dispose souvent d'une extension pour le navigateur qui permet de déboguer les vues de l'interface en fonction du framework utilisé, et le nom de chaque composant doit pouvoir être retrouvé facilement dans le programme à l'aide de son nom de fichier ou de classe ou de fonction. Il faut utiliser le même nom partout pour chaque élément.

  C'est une erreur que j'ai vue dans un projet, ou les développeurs avaient nommés des composants d'interface avec des noms sortis de nulle part, par exemple *project-button*. Pour retrouver le code source des éléments visibles à l'écran, c'était extrêmement difficile et il fallait fouiller et presque faire des investigations. Ce problème aurait été évité en nommant correctement les vues et les classes de la même façon.

Un bon code est typé

De nombreux projets web et node.js sont écrits en javascript qui est un langage non typé. Et c'est bien dommage ! C'est un standard de l'industrie aujourd'hui, et l'industrie a tort. Les entreprises considèrent que leurs développeurs connaissent javascript, et que les faire basculer sur typescript est risqué. Mais ce qui est risqué, c'est de se passer du typage.

Le typage a été inventé en Fortran il y a plus de 50 ans. C'est indispensable pour un travail de programmation moderne. Cela permet :

- de définir solidement les structures des objets et types de données, qui sont la base de l'algorithmique
- de détecter certaines erreurs avant qu'elles n'arrivent
- de guider les personnes suivantes qui devront intervenir sur une partie, car le typage empêche de mal utiliser un composant
- de modifier un code sans avoir peur de tout casser
- de refactor facilement une partie du logiciel
- de faire de la planification semi automatique de la programmation, en modifiant un type et en regardant le compilateur trouver ce qui a cassé, et ce qu'il va falloir modifier pour faire le travail
- c'est aussi une documentation intégrée, dont les éditeurs de code modernes se servent pour faciliter la tâche du programmeur

Le plus triste, c'est que quand on programme en JavaScript, on fait à chaque fois le typage mentalement. Mais comme on a un langage non typé, on ne peut pas l'écrire dans le programme et l'information est perdue dans la tête de l'auteur. Tous les programmeurs suivants, y-compris soi-même, devront refaire indéfiniment le même travail intellectuel, pour rien.

Quand on type un programme avec Typescript, on a le choix de typer chaque zone du programme avec différentes intensités.

- Dans certains cas, il faut typer normalement. Principalement on va typer les arguments des fonctions et leur retour, les propriétés des classes, et la structure des types et interfaces.
- Dans le cas ou on type des points d'entrée des modules, il faut typer très fort, parfois même en utilisant des types génériques paramétrés. Car on veut aider le programmeur qui va invoquer un module qu'il n'a pas écrit. On veut lui éviter d'avoir des erreurs, et lui éviter le besoin d'aller lire notre code interne du module.
- Dans d'autres cas, on ne veut pas typer du tout. On peut le faire avec any ou unknown. C'est utile quand le code est alourdi inutilement, et illisible. Ou quand on est en train de typer un code legacy javascript petit à petit.

Un bon code est commenté

Il existe depuis un certain nombre d'années une théorie, je dirais une mode, qui séduit environ un tiers des développeurs informatique. D'après eux, il ne faudrait pas écrire de commentaires dans le code source des programmes. Les raisons principales avancées sont :

- On va mettre à jour le code mais pas les commentaires.
- Si le code est assez clair, on n'a pas besoin de commentaire. Si on ressent le besoin de mettre un commentaire, c'est parce qu'il faut reprendre le code pour qu'il soit plus clair, et retirer ensuite le commentaire.

Je réfute intégralement cette théorie, pour les raisons suivantes :

- Je passe rapidement sur la première raison en dit long sur le manque de sérieux des personnes qui pensent qu'on ne va pas mettre à jour les commentaires.
- La seconde raison est mauvaise car ce qui va se passer en réalité, c'est que le développeur ne va pas écrire de commentaire, et en prime son code ne sera pas clair.

Ensuite j'ajouterai les arguments suivant :

- Quand on écrit un programme, on a une logique dans la tête. Si on n'écrit pas de commentaire, cette logique est perdue. On va devoir la retrouver à chaque fois qu'on va travailler sur ce programme. On aurait pu cristalliser dans le programme sa logique, en l'écrivant dans des commentaires, une fois pour toutes, et gagner ce temps de répétition.
- Le code informatique n'est pas fait pour être lisible à la base. C'est un langage que comprend l'ordinateur, qui lui sert de liste d'instructions à exécuter. On peut effectivement et on doit le rendre le plus lisible possible en choisissant bien les identificateurs et le flux du programme. Mais il ne faut pas oublier qu'à la base, c'est du code informatique. Et que si les ordinateurs sont faits pour comprendre le langage informatique, les développeurs, humains eux, sont faits pour comprendre le langage humain.
- Enfin, un code sans commentaire constitue de fait un syndicat. Car l'équipe qui a écrit ce code est la seule à pouvoir le modifier facilement. Si leur code est suffisamment long et suffisamment sale, ils auront le pouvoir de défier leur management.

Mon père, qui travaillait chez Framatome, m'a raconté l'histoire d'un ingénieur qui avait écrit un programme en assembleur servant à simuler des équations de réactions nucléaires. Il avait écrit le programme sur le support informatique de l'entreprise. Mais il avait écrit les commentaires séparément dans un petit carnet, avec les numéros de lignes. Ce carnet lui appartenait personnellement. Il a pu continuer ainsi pendant une dizaine d'années, avec ses collègues, sans rien faire de plus, et sa Direction était pétrifiée.

Un bon code est symétrique, autant que possible

Pour rendre le code facile à maintenir, il est important de le garder symétrique. Cela signifie utiliser des structures, des patterns et des conventions de manière cohérente tout au long du projet. Un code symétrique est plus lisible, plus compréhensible et plus simple à gérer. Lorsqu'on modifie une partie du code, il est crucial de répercuter cette modification partout où c'est nécessaire. Cela garantit une cohérence et une maintenabilité du code.

Mais parfois, on manque de temps pour faire ça de manière exhaustive. Dans ces moments-là, les développeurs doivent trouver un équilibre entre la rapidité et la qualité. Même si on manque de temps, il est crucial de documenter les modifications faites et de noter celles à faire plus tard. Cela aide à garder une traçabilité et à planifier des améliorations futures.

Aucune règle en assurance qualité logicielle n'est absolue. La programmation est plus un art qu'une science. Les développeurs doivent faire preuve de jugement et s'adapter aux circonstances spécifiques de chaque projet. Savoir quand et comment appliquer les principes de qualité est ce qui distingue les développeurs expérimentés des cracks.

Un bon code est factorisé, mais pas compressé

Un bon code doit être factorisé, c'est-à-dire que les parties répétitives du code doivent être mises en fonctions ou modules réutilisables. Cependant, il ne doit pas être compressé au point de devenir illisible.

Factoriser le code consiste à regrouper les parties similaires ou identiques en une seule fonction. Cela permet de réduire les répétitions, ce qui rend la maintenance plus facile et réduit le risque d'erreur. Par exemple, si vous avez une formule mathématique utilisée plusieurs fois dans votre code, il vaut mieux la mettre dans une fonction dédiée plutôt que de copier-coller la formule partout avec de légères modifications.

Non factorisé	Factorisé

`let result1 = (a + b) * (c - d);`
`let result2 = (a + b) * (e - f);`
`let result3 = (a + b) * (g - h);`
	`function calculate(a: number, b: number, x: number, y: number): number {`
	`  return (a + b) * (x - y);`
	`}`
	
	`let result1 = calculate(a, b, c, d);`
	`let result2 = calculate(a, b, e, f);`
	`let result3 = calculate(a, b, g, h);`

Si plus tard la formule doit changer, il suffira de modifier la fonction *calculate* et toutes les occurrences seront mises à jour automatiquement. C'est une des bases de la programmation.

Mais attention, il est important de ne pas se laisser tenter par des pseudo-optimisations qui rendent le code plus complexe sans apporter de bénéfices significatifs. Par exemple, compresser 20 lignes de code en une seule ligne utilisant des fonctions comme Array.reduceRight, Array.zip, ou des triple map avec des opérateurs bitwise, peut rendre le code illisible et difficile à maintenir.

Certains programmeurs se croient plus intelligents parce qu'on ne comprend pas ce qu'ils écrivent. Ils ont seulement prouvé qu'ils n'avaient pas l'expérience d'avoir maintenu leurs programmes sur le long terme.

Code trop compressé	Code normal

`let result = data.map(x => x * 2)`
`.filter(x => x % 3 === 0)`
`.reduce((acc, x) => acc + x, 0)`
`.toString(16)`
`.split('')`
`.map(c => c.charCodeAt(0))`
`.reduce((acc, x) => acc ^ x, 0);`
	`// Doubler chaque élément`
	`let doubled = data.map(x => x * 2);`
	
	`// Filtrer les éléments divisibles par 3`
	`let divisibleByThree = doubled.filter(x => x % 3 === 0);`
	
	`// Réduire en additionnant tous les éléments`
	`let sum = divisibleByThree.reduce((acc, x) => acc + x, 0);`
	
	`// Convertir en hexadécimal`
	`let hexString = sum.toString(16);`
	
	`// Convertir en tableau de caractères`
	`let charArray = hexString.split('');`
	
	`// Convertir en codes de caractères`
	`let charCodes = charArray.map(c => c.charCodeAt(0));`
	
	`// Réduire avec un XOR`
	`let finalResult = charCodes.reduce((acc, x) => acc ^ x, 0);`

Un bon code est lisible facilement

Il existe plusieurs façons de programmer la même fonctionnalité. Il faut choisir la façon la plus lisible.

Mais il est difficile de décrire comment exactement un code doit être plus lisible. Déjà il faut partir du principe qu'il doit l'être, ce qui n'est pas évident pour tout le monde. C'est un problème de priorité. Il faut suivre le plus souvent cette séquence de priorité :

- La première des priorité, c'est que le logiciel doit faire ce qu'il est censé faire, et sans erreur.
- La seconde priorité, c'est que le code soit lisible et modifiable facilement. Si on a le choix entre plusieurs façons d'écrire un même sous-programme, qui ont le même résultat à l'exécution, il faut choisir celle qui est la plus simple à lire, à comprendre, et à modifier.

  Comment choisir ? On peut utiliser une de ces deux méthodes :

  - Si on m'efface la mémoire, et que je dois ensuite modifier ce code, quelle écriture me permettra de le faire le plus efficacement.
  - Si j'abandonne ce projet et que je le reprends dans dix ans, quelle écriture me permettra de reprendre le projet dans les meilleures conditions.
- Tout le reste : La vitesse d'exécution, le nombre de lignes, la taille du code, etc. n'a aucune importance en comparaison avec la lisibilité. Ces critères sont à bannir, sauf si il y a une contrainte particulière liée aux spécifications, ou un blocage, ou une contrainte temps réel, ou de prix du matériel.
  - Il (ne) faut (pas) optimiser la vitesse du code. On n'a pas à optimiser 10 ms sur un clic, on s'en fiche. On optimise seulement un truc qui tourne régulièrement sur le serveur, un truc qui ralentit la réponse au client, qui ralentit un processus batch
  - Il (ne) faut (pas) optimiser la taille du code. Si le code est trop gros, il faut utiliser un compresseur comme UglifyJs. On peut aussi mieux découper les bundles délivrés au client par http. Les outils modernes comme NextJs se chargent de faire ce travail, mais il peut être configuré manuellement par exemple dans Webpack.

Pourquoi est-ce qu'on devrait faire passer la maintenabilité avant la plupart des autres critères ?

- Si le code est maintenable, mais qu'il a des bugs, on corrigera facilement les bugs.
- Si le code est maintenable, mais qu'il manque des fonctionnalités, on ajoutera facilement les fonctionnalités.
- Si le code est maintenable, et qu'on veut aller plus vite, on ajoutera facilement des développeurs.
- Mais si le code n'a pas de bugs, qu'il a énormément de fonctionnalités, et qu'on a déjà plein de développeurs, il faudra tout de même le modifier, et il faudra tout de même remplacer des

développeurs. Dans ce cas, on aura énormément de mal à le faire. Ça coûtera très cher !

Pour faire un code lisible, on peut utiliser quelques principes :

- Un module, une classe, une méthode, un paragraphe, doit faire une seule chose, le faire clairement, avec une méthodologie simple. Les étapes des algorithmes sont bien séparées. On ne cherche pas à faire deux choses en même temps.
- Il faut séparer les paragraphes avec des lignes vides. Pour aérer le texte. Comme pour un document.
- Il faut des commentaires utiles et nombreux.
- Il faut expliquer les pièges avec des commentaires. Ou éviter les pièges avec du typage fort.

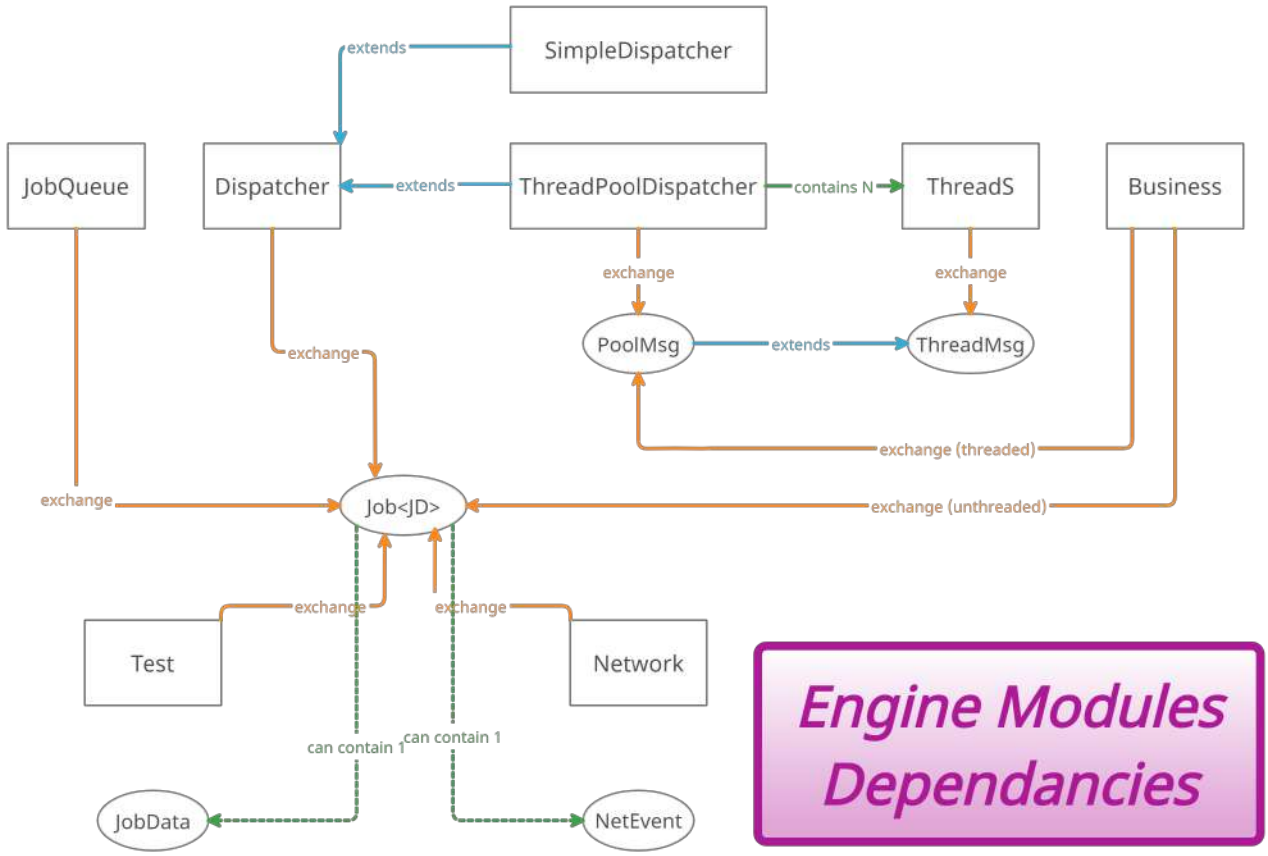
Le principe de base : Un programme est lisible si je peux le maintenir après qu'on m'a effacé la mémoire.

Un bon code est architecturé

Une bonne architecture logicielle repose sur une structuration réfléchie des types et des composants. La structure des types est la base de tout algorithme et architecture. C'est à partir de cette base que l'on construit le reste du programme, en veillant à ne pas adapter les types au code de manière arbitraire.

Les composants du programme doivent être isolés les uns des autres et posséder des interfaces claires et minimales pour les entrées et les sorties. Cela permet non seulement de faciliter la maintenance et l'évolution du code.

Voici par exemple un diagramme représentant le moteur central du logiciel Player22. Constatez qu'on ne code pas au fur et à mesure, comme ça vient. Au contraire, tout est planifié, et rien n'est laissé au hasard. Ce diagramme a été créé avant le code du moteur, ce qui a rendu le moteur solidement architecturé.



Il faut d'autre part limiter les effets de bord. Un effet de bord se produit lorsqu'une fonction ou une méthode modifie un état externe à son propre contexte, ce qui peut provoquer des comportements inattendus. Par exemple, si nous avons une variable a qui est toujours égale à 1, et que dans le programme, nous remplaçons la condition *si a=1 et b=2* par *si b=2*, en supposant que a sera toujours égale à 1, nous créons un effet de bord. Dès que a sera modifié, le code cassera. Il est donc crucial d'éviter ces simplifications et de maintenir les conditions complètes pour assurer la stabilité du code.

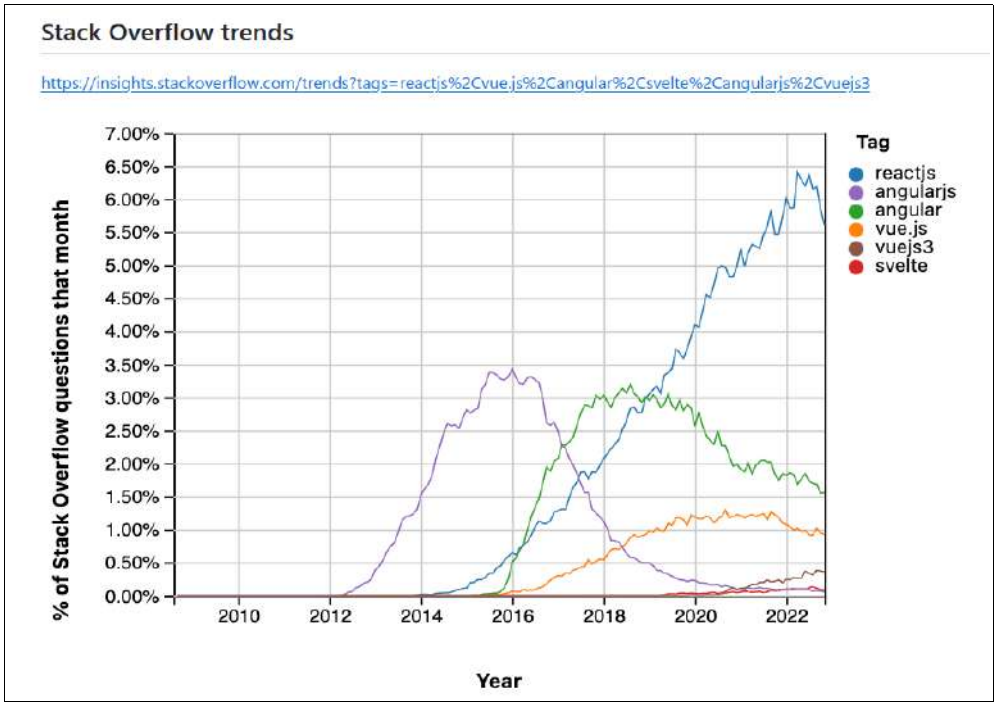
Il ne faut pas non plus exagérer. Il est essentiel de trouver un équilibre entre ajouter suffisamment de code pour éviter les effets de bord et ne pas en rajouter trop au point de négliger les spécifications. L'expérience du développeur est la clé pour deviner ce qui risque de changer et ce qui restera constant. Il pourra ajouter des zones flexibles avec de l'over-engineering là où c'est nécessaire, et écrire des constantes et du code fixe là où il pense que ça ne bougera pas.

Un bon code utilise des technologies récentes et adaptées

Pour m'expliquer ici, je vais prendre ici le contre-exemple d'une entreprise fictive que nous appellerons PortailImmobilier.com. C'est un cas d'école de dette technique due au choix de stack.

PortailImmobilier.com est écrit avec du JavaScript, non typé et archaïque. Le front-end utilise du Jade, qui est une horreur. Le jade est en train d'être traduit en Vue2, qui est presque aussi mauvais que Jade. En plus la traduction en Vue2 est bâclée et injecte du Jade à l'intérieur.

Voici ce que vaut la technologie Vue sur le marché. Elle ne vaut pas grand-chose comme vous pouvez le constater sur le graphique suivant. Et le plus triste, c'est qu'elle ne valait déjà pas grand-chose quand elle a été choisie en 2020 par les équipes de PortailImmobilier.com pour remplacer la technologie Jade. Sur le même graphique, on voit clairement que React se détache du lot.



Source : https://gist.github.com/tkrotoff/b1caa4c3a185629299ec234d2314e190

Par contre, le back-end utilise Express, qui n'est pas mal du tout en l'état, même si NestJs et surtout NextJs sont nettement mieux aujourd'hui. En ce qui concerne la base de données en MongoDb, cela reste un excellent choix jusqu'à présent, ainsi qu'Elastic Search.

Le système de communication entre micro-services est bricolé en Redis alors que le standard de l'industrie est clairement RabbitMq, voire Kafka. Je ne veux pas être injuste, et je reconnais que ce choix était intéressant à l'époque ou il a été fait, mais aujourd'hui c'est de la dette technologique.

Un bon code utilise des technologies standard, leaders du marché

Quand on utilise une technologie standard, leader du marché, on a plein d'avantages gratuits, et plein de problèmes résolus gratuitement.

- On trouve facilement de l'aide et des exemples de code.
- On corrige facilement les erreurs en tapant le texte de l'erreur sur Google et en visitant des forums tels que Stackoverflow, Reddit ou Serverfault.
- On a souvent droit à une maintenance de qualité. Et même des solutions de migration de version bien ficelées et faciles à appliquer. C'est parce que tellement de business importants l'utilisent que tout est fait de façon sérieuse.
- On donne de la valeur à nos collaborateurs. Un développeur qui a une expérience NodeJs Typescript React a une plus grande valeur sur le marché, et date de désuétude est plus tardive.
- On recrute plus facilement les meilleurs profils, car ils veulent travailler sur les technologies standard et leaders du marché.
- On onboard plus facilement technologiquement. Car il existe des tutoriels qui expliquent exactement comment on fait : On fait du standard ! Il existe même des écoles d'informatique qui apprennent directement ces technologies aux étudiants.

Je le dis par expérience : Souvent quand on choisit une technologie outsider, et on le fait après avoir analysé les forces et faiblesses des alternatives, on va le payer quelques années plus tard. Le leader finira par rattraper tous les autres, et même par racheter leurs ingénieurs, et la technologie outsider évoluera dans le mauvais sens.

Pour choisir un élément du stack, une dépendance, une librairie, il faut se renseigner sur les forums spécialisés, lire les avis et débats entre défenseurs de telle et telle option. Il faut regarder les comparatifs de nombre d'étoiles sur GitHub par exemple, ou la popularité des mots clés sur StackOverflow. Il faut regarder les dernières dates de modification du projet, le nombre de mainteneur et leur crédibilité, le nombre de bugs non corrigés. Il faut vérifier la santé de l'entreprise ou structure qui chapeaute le projet, son historique d'abandon de projets, sa politique de compatibilité ascendante, son modèle économique.

Bref il ne faut pas choisir les dépendances sur un coup de cœur. Il faut faire ses devoirs.

Autres considérations

La qualité d'un logiciel ne dépend pas seulement du code. Il y a d'autres aspects tout aussi importants à prendre en compte pour garantir un logiciel fiable.

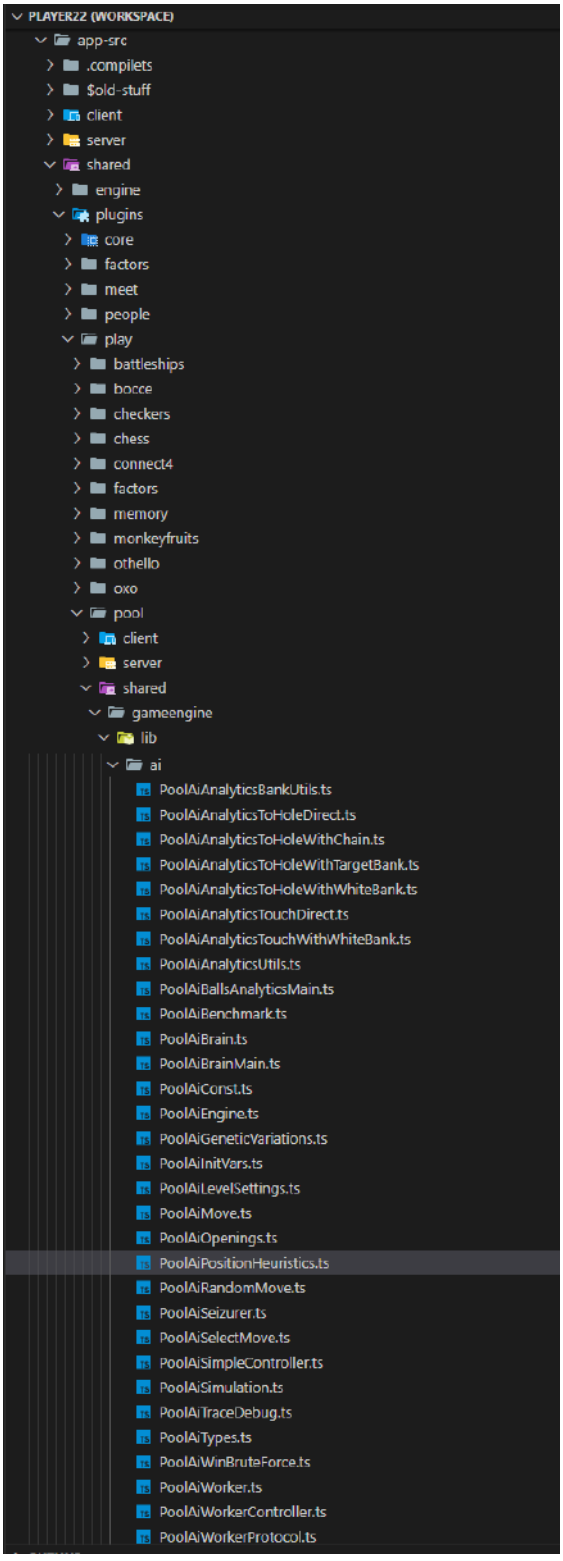
Par exemple, les tests unitaires, les tests d'intégration et les tests end-to-end sont cruciaux pour vérifier que le logiciel fonctionne correctement après des modifications, mais ce document ne couvre pas ces sujets.

Il en va de même pour les pratiques DevOps, comme la mise en place d'un pipeline CI/CD et les livraisons fréquentes (deliver often). Ces éléments sont essentiels pour la continuité et la stabilité du développement, mais ils ne sont pas inclus dans ce document.

Même si ces aspects ne sont pas abordés ici, puisque ce document traite spécifiquement du code de qualité, ils sont des éléments importants pour assurer la qualité globale du logiciel.

Comparatif : code de « 10x développeur »

Voici comment j'ai rangé les répertoires et fichiers, par exemple pour stocker les fichiers d'intelligence artificielle du jeu de snooker, sur le programme Player22 :



On peut remarquer plusieurs choses :

- Premièrement la profondeur des dossiers, signe d'un classement fin, qui permettra de naviguer dans le code. Puisqu'avant toute chose, il faut pouvoir retrouver le code. Sur cet exemple, le fichier concernant l'IA sur lequel nous allons zoomer est dans le chemin

  shared\plugins\play\pool\shared\gameengine\lib\ai\PoolAiPositionHeuristics.ts

  Plus on fait de petits répertoires, mieux on classe ses fichiers, et plus facilement on les retrouvera. Ici on 9 niveaux dans l'arborescence de sous-répertoire, qui permettent de classer parfaitement le fichier *PoolAiPositionHeuristics.ts*.
- Ensuite vous pouvez noter que pour une simple fonction, on a une multitude de fichiers, J'ai compté 31 fichiers, seulement pour l'IA d'un jeu. Le fait de découper un fichier en plusieurs fichiers de 100 à 500 lignes est positif car encore une fois on retrouvera facilement le code qu'on cherche. C'est une architecture qui fonctionne comme une bibliothèque. Dans une bibliothèque, on trouve ce que l'on cherche. Les livres sont classées par thème, par époque, par auteur...
- Regardez les noms des fichiers. Chaque fichier est préfixé par le nom du module « PoolAi ». Ce n'est pas obligatoire, mais ça aide à travailler et à savoir ce qu'on est en train de faire quand on a plein de fichiers ouverts. La suite du nom de chaque fichier est explicite. On ne les a pas choisis n'importe comment. Le nom d'un fichier doit permettre de savoir ce qu'on a mis dedans. Et le fichier ne doit pas être trop gros, ni trop petit, faire en moyenne quelques centaines de lignes, et il doit traiter un seul sujet et rien d'autre, et le faire de façon isolée. Le CLASSEMENT d'un code de grande taille est LA CHOSE LA PLUS IMPORTANTE dans un plan assurance qualité. Le plus important, c'est de pouvoir retrouver ce qu'on cherche.
- Bien sur, il y a d'autres façons de faire ce classement. Mais ce qui est important, c'est de trouver une façon logique, et de prendre le temps de reclasser régulièrement une partie dès qu'elle grossit. Il faut faire du refactoring, très régulièrement, et ne pas être paresseux.

Maintenant intéressons-nous au fichier *PoolAiPositionHeuristics.ts,* et en particulier à la fonction *evalPositionPocketFactor()*.

```ts
/**
 * Eval a position's pocket factor (ppf).
 *     + The pocket factor is a heuristic representing the advantageous placement of the target balls on the pool table.
 *     + It is obtained by moving the white ball virtually to a number of pre-dertimined significant positions,
 *       and couting in how many holes how many target balls can be pocketed directly.
 *     + A formula is applied and this number is brought to 0..1
 *     + It is possible to compute this factor for any of the 2 players.
 *     + It is possible to include or not the black ball in this factor's evaluation.
 *       It should be included if it's the ai's turn to play, so it could win without losing turn to play when it's time to pocket black.
 *       It should not be included if the ai is going to lose the turn to play, so it won't help the opponent to have an easy black ball.
 *     + Cost: 1ms for an average position.
 */

private static readonly WANTED_WHITE_POSITIONS : Array<[ float, float ]> = [ // % of width, % of height
                    [ 1/2, 1/10 ],
    [ 1/5, 1/5 ],                       [ 4/5, 1/5 ],
                    [ 1/2, 1/2  ],
    [ 1/5, 4/5 ],                       [ 4/5, 4/5 ],
                    [ 1/2, 9/10 ],
];
private static readonly DIRECTIONS : Array<[ int, int ]> = [ // delta sign hor, delta sign ver
    [ -1, -1 ], [ 0, -1 ],  [ +1, -1 ],
    [ -1,  0 ],             [ +1,  0 ],
    [ -1, +1 ], [ 0, +1 ],  [ +1, +1 ],
];

public evalPositionPocketFactor(v : PoolAiVars, args : {
    arrTargetBall : Array<PoolBall>,
    includeBlack : boolean,
}) : float {
    // init vars: save white position
    const whitePosOrigX : float = v.ballsAnalysis.whiteBallPos.x_;
    const whitePosOrigY : float = v.ballsAnalysis.whiteBallPos.y_;
    // init vars: add black to target balls if required
    const arrTargetBall : Array<PoolBall> = ArrayUtils.arrayCloneNotDeep(args.arrTargetBall);
    if (args.includeBlack && ! arrTargetBall.some((curPoolBall : PoolBall) : boolean => curPoolBall.ballColor == POOL_BALL_COLOR.BLACK)) {
        arrTargetBall.push(v.board.blackBall);
    }
    // init vars: counts
    let totalWhitePositions : int = 0;
    let totalCountPocketableInHoles : int = 0;

    // for each wanted white position
    for (const wantedWhitePosition of PoolAiPositionHeuristics.WANTED_WHITE_POSITIONS) {
        const wantedXWhite : float = v.board.tableBounds.left_ + wantedWhitePosition[0] * v.board.tableBounds.width_;
        const wantedYWhite : float = v.board.tableBounds.top_ + wantedWhitePosition[1] * v.board.tableBounds.height_;

        // find a position for white, at wanted position if possible, otherwise close to it
        // first try the wanted position
        const vPosWhite = new Vect(wantedXWhite, wantedYWhite);
        let isPosWhiteOk : boolean = v.board.isBallPosOk({ vPos: vPosWhite, boundsTolerance: 0, doAutofixBoundsLimits: false });
        // if wanted position is occupied: try to place the white close to it, in each direction
        if (! isPosWhiteOk) {
            for (let distFactor : int = 1; distFactor <= 2; distFactor++) {
                for (const direction of PoolAiPositionHeuristics.DIRECTIONS) {
                    vPosWhite.x_ = wantedXWhite + direction[0] * distFactor * PoolBoard.BALL_SIZE;
                    vPosWhite.y_ = wantedYWhite + direction[1] * distFactor * PoolBoard.BALL_SIZE;
                    isPosWhiteOk = v.board.isBallPosOk({ vPos: vPosWhite, boundsTolerance: 0, doAutofixBoundsLimits: false });
                    if (isPosWhiteOk) break;
                }
                if (isPosWhiteOk) break;
            }
        }

        // if we could find a position for white near the wanted position (if we couldn't, we simply skip this position)
        if (isPosWhiteOk) {
            totalWhitePositions++;
            // place the white on the computed position
            v.ballsAnalysis.whiteBallPos.setInPlace(vPosWhite);
            // for each target ball
            for (const targetPoolBall of arrTargetBall) {
                // count the number of holes the ball can be directly pocketed into
                const countPocketableInHoles : int = this._ai.anaHoleDirect.computeBallAnalysisToHoleDirect(v, targetPoolBall, true);
                totalCountPocketableInHoles += countPocketableInHoles;
            }
        }
    }

    // put back the white ball where it was
    v.ballsAnalysis.whiteBallPos.x_ = whitePosOrigX;
    v.ballsAnalysis.whiteBallPos.y_ = whitePosOrigY;

    // compute factor
    const positionPocketFactor : float =
          totalCountPocketableInHoles
        / totalWhitePositions
        / arrTargetBall.length
        / v.board.holes.length
    ;
    assert2(positionPocketFactor <= 1);
    return positionPocketFactor;
}

```

Je choisis cette fonction car c'est le cœur de l'IA du jeu de snooker, celle qui peut *sentir* si une position est plus avantageuse stratégiquement qu'une autre, et que son calcul est immensément complexe. Pourtant vous allez constater que même sans être informaticien, on devine ce qu'elle fait, et comment elle le fait. C'est du code 10x! En écrivant la totalité d'un programme ainsi, on facilite tellement le travail de maintenance qu'on fait tout 10 fois plus vite. Le code est aéré car qui a envie de lire des paragraphes collés sans espaces pour comprendre comment c'est découpé ? Le code est typé car on veut savoir sur quoi on travaille. Le code est commenté car au moment ou on l'écrit, on sait ce qu'on fait, et il faut noter l'information DANS LE CODE et ne pas la garder dans sa tête. Le code est architecturé et chaque paragraphe fait une chose, une seule, et ne déborde pas. La pensée cartésienne est toujours meilleure que le fouillis et le raccourci.

Le 10x développeur n'est pas un génie. Ce n'est pas quelqu'un qui écrit dix fois plus vite, car en fait il écrit moins vite. Ce n'est pas quelqu'un qui livre dix fois plus vite, car en fait il refactor et livre moins vite. Ce n'est pas quelqu'un qui pense plus vite, car il se pose des questions supplémentaires et pense moins vite. Mais c'est quelqu'un qui s'applique plus que les autres, pour produire du code lisible. Ce code sera plus facile à trouver, à comprendre, à modifier. Ce code sera typé et évitera les fautes idiotes. Ce code occasionnera moins de bugs et de régressions. Ce code permettra, sur le long terme, de travailler dix fois plus vite. Si ce n'est cent fois !

Tandis que le programmeur paresseux, qui livre vite et va boire son café, passera l'éternité à corriger les mêmes erreurs, et à ré-apprendre ce qu'il aurait pu noter dans la documentation du typage et des commentaires. Le 10x développeur, lui, investit mieux son temps, celui de l'équipe, et gagne sur le long terme. Et n'importe qui peut apprendre à faire ça s'il est drivé correctement et qu'il a envie de progresser.

Comparatif : code de développeur senior « qui fonctionne »

Après avoir étudié un exemple de code 10x, comparons-le avec le code « qui fonctionne » d'un développeur senior. Sur l'exemple suivant, pouvez-vous deviner ce qu'il y a dans le répertoire et à quoi ça sert ?

Moi non, je n'y arrive pas. Alors comment faire à votre avis ? Et bien il faut galérer et déboguer et y passer des semaines, des mois, et on comprendra à peine ce qu'il y a dedans car tout est mélangé et mal nommé. Pourtant ce code fonctionne aussi bien que le précédent, quand on regarde le résultat de l'exécution dans le navigateur. Il y a le bon code, et il y a le mauvais code, mais c'est impossible à voir si on regarde seulement le résultat. C'est le moteur, caché sous le brillant capot, qui pose problème et qui nécessitera tôt ou tard des réparations.

Maintenant zoomons dans le fichier *dataMixer.js* et regardons ce qu'il contient :

```
const _ = require('lodash');
const ultimateMixerFunction = () => {
  const mixData = (a, b) => _.flatten(_.zip(a, b).map(x =>
    x.map(y => _.isArray(y) ? y : _.times(_.random(1, 3), () => y)))
    .map(z => _.shuffle(z.flat())).reduce((m, n) => {
    let p = _.filter(m, q => q !== n);
    return _.concat(p, n);
  }, []);
  const processArray = x => _.chain(x).groupBy(y => typeof y)
    .mapValues(z => _.flatten(z)).values().flatten().value();
  const dataMixer = (arr1, arr2) => {
    let mixed1 = mixData(arr1, processArray(arr2));
    let mixed2 = mixData(processArray(arr1), arr2);
    return _.union(mixed1, mixed2);
  };
  const randomizeArray = arr => _.shuffle(_.flatten(arr.map(x =>
    _.isArray(x) ? x : [x])));
  const finalizeMix = (d1, d2) => dataMixer(randomizeArray(d1), randomizeArray(d2));
  const mixAndMatch = (x, y) => finalizeMix(processArray(x), processArray(y));
  const ultimateMixer = (input1, input2) => mixAndMatch(_.shuffle(input1), _.shuffle(input
  return { ultimateMixer };
};
module.exports = ultimateMixerFunction();
```

Vous arrivez à comprendre quelque chose vous ? Moi non, et pourtant j'ai 35 ans d'expérience en programmation. Et si on me demandait de modifier ce code, est-ce que j'y arriverais ? Et à votre avis, combien de temps ça me prendrait ?

Conséquences du défaut de maintenabilité

C'est simple, quand un programme n'est pas assez maintenable :

- On perd du temps.
- On perd de l'argent.
- On a beaucoup de bugs de régression. Dès qu'on touche un truc à gauche, ça pète à droite.
- On décourage les développeurs.
- On va lentement dans l'ajout de fonctionnalités.
- On a du mal à ajouter des développeurs dans l'équipe.
- On a du mal à remplacer des développeurs qui partent.

On finira par jeter le programme et le ré-écrire. Ou bien par le rénover et ça coûtera un bras. C'est inévitable.

Conclusion : Ce qui est faisable concrètement dans votre entreprise

Si vous désirez ajouter un certain niveau d'assurance qualité logicielle dans votre entreprise, le critère déterminant est : Commencez-vous de zéro ou devez-vous améliorer un projet existant ?

Le second critère est la nécessité de maintenance : Si on est éditeur de logiciel ou client final, c'est très important de faire de la qualité. Puisque si on doit maintenir le code nous-mêmes, il vaut mieux s'assurer de la qualité dès le début. A l'inverse, si on est prestataire avec un contrat, et qu'une fois qu'on livre le projet à un client, on n'aura plus à s'en occuper, on peut faire les choses avec moins de qualité.

Ensuite, il faut définir jusqu'à quel point on veut de la qualité : est-ce qu'on est prêt à y investir du temps et de l'argent, ou on préfère livrer rapidement quitte à avoir des problèmes plus tard, ce qui peut être une stratégie gagnante par exemple pour une start-up.

Si on pense qu'on devra maintenir, traduire ou réécrire le code plus tard, il vaut mieux investir dans la qualité dès le début. Si on est pressé et qu'on veut juste sortir un produit pour battre les concurrents, on pourrait faire quelque chose de rapide et pas très propre. Mais attention, la dette technique qui en résulte est souvent difficile à gérer.

Pour un projet *from scratch*, il suffit de suivre les bonnes pratiques. Pour rénover un projet existant, il faut le découper, le relire, le commenter, le typer, et faire ça petit à petit, par module, en commençant par le plus urgent.

Il faut aussi prévenir les décideurs qu'il faudra payer du travail en plus, mais qu'on en tirera des bénéfices après. Et il faut expliquer aux développeurs, qui pourraient être vexés de voir leur code modifié, que c'est aussi une opportunité d'apprendre. Et en échangeant les connaissances, tout le monde s'améliore.

En conclusion, même si l'assurance qualité logicielle demande un effort initial, elle offre des avantages durables pour l'entreprise et pour les développeurs.

License Creative commons

Creative Commons Legal Code

CC0 1.0 Universal

    CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE
    LEGAL SERVICES. DISTRIBUTION OF THIS DOCUMENT DOES NOT CREATE AN
    ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS
    INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES
    REGARDING THE USE OF THIS DOCUMENT OR THE INFORMATION OR WORKS
    PROVIDED HEREUNDER, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM
    THE USE OF THIS DOCUMENT OR THE INFORMATION OR WORKS PROVIDED
    HEREUNDER.

Statement of Purpose

The laws of most jurisdictions throughout the world automatically confer exclusive Copyright and Related Rights (defined below) upon the creator and subsequent owner(s) (each and all, an "owner") of an original work of authorship and/or a database (each, a "Work").

Certain owners wish to permanently relinquish those rights to a Work for the purpose of contributing to a commons of creative, cultural and scientific works ("Commons") that the public can reliably and without fear of later claims of infringement build upon, modify, incorporate in other works, reuse and redistribute as freely as possible in any form whatsoever and for any purposes, including without limitation commercial purposes. These owners may contribute to the Commons to promote the ideal of a free culture and the further production of creative, cultural and scientific works, or to gain reputation or greater distribution for their Work in part through the use and efforts of others.

For these and/or other purposes and motivations, and without any expectation of additional consideration or compensation, the person associating CC0 with a Work (the "Affirmer"), to the extent that he or she is an owner of Copyright and Related Rights in the Work, voluntarily elects to apply CC0 to the Work and publicly distribute the Work under its terms, with knowledge of his or her Copyright and Related Rights in the Work and the meaning and intended legal effect of CC0 on those rights.

1. Copyright and Related Rights. A Work made available under CC0 may be protected by copyright and related or neighboring rights ("Copyright and Related Rights"). Copyright and Related Rights include, but are not limited to, the following:

  i. the right to reproduce, adapt, distribute, perform, display, communicate, and translate a Work;
 ii. moral rights retained by the original author(s) and/or performer(s);
iii. publicity and privacy rights pertaining to a person's image or likeness depicted in a Work;
 iv. rights protecting against unfair competition in regards to a Work, subject to the limitations in paragraph 4(a), below;
  v. rights protecting the extraction, dissemination, use and reuse of data in a Work;
 vi. database rights (such as those arising under Directive 96/9/EC of the European Parliament and of the Council of 11 March 1996 on the legal protection of databases, and under any national implementation thereof, including any amended or successor version of such directive); and
vii. other similar, equivalent or corresponding rights throughout the world based on applicable law or treaty, and any national implementations thereof.

2. Waiver. To the greatest extent permitted by, but not in contravention of, applicable law, Affirmer hereby overtly, fully, permanently, irrevocably and unconditionally waives, abandons, and surrenders all of Affirmer's Copyright and Related Rights and associated claims and causes of action, whether now known or unknown (including existing as well as future claims and causes of action), in the Work (i) in all territories worldwide, (ii) for the maximum duration provided by applicable law or treaty (including future time extensions), (iii) in any current or future medium and for any number of copies, and (iv) for any purpose whatsoever,

including without limitation commercial, advertising or promotional purposes (the "Waiver"). Affirmer makes the Waiver for the benefit of each member of the public at large and to the detriment of Affirmer's heirs and successors, fully intending that such Waiver shall not be subject to revocation, rescission, cancellation, termination, or any other legal or equitable action to disrupt the quiet enjoyment of the Work by the public as contemplated by Affirmer's express Statement of Purpose.

3. Public License Fallback. Should any part of the Waiver for any reason be judged legally invalid or ineffective under applicable law, then the Waiver shall be preserved to the maximum extent permitted taking into account Affirmer's express Statement of Purpose. In addition, to the extent the Waiver is so judged Affirmer hereby grants to each affected person a royalty-free, non transferable, non sublicensable, non exclusive, irrevocable and unconditional license to exercise Affirmer's Copyright and Related Rights in the Work (i) in all territories worldwide, (ii) for the maximum duration provided by applicable law or treaty (including future time extensions), (iii) in any current or future medium and for any number of copies, and (iv) for any purpose whatsoever, including without limitation commercial, advertising or promotional purposes (the "License"). The License shall be deemed effective as of the date CC0 was applied by Affirmer to the Work. Should any part of the License for any reason be judged legally invalid or ineffective under applicable law, such partial invalidity or ineffectiveness shall not invalidate the remainder of the License, and in such case Affirmer hereby affirms that he or she will not (i) exercise any of his or her remaining Copyright and Related Rights in the Work or (ii) assert any associated claims and causes of action with respect to the Work, in either case contrary to Affirmer's express Statement of Purpose.

4. Limitations and Disclaimers.

a. No trademark or patent rights held by Affirmer are waived, abandoned, surrendered, licensed or otherwise affected by this document.
b. Affirmer offers the Work as-is and makes no representations or warranties of any kind concerning the Work, express, implied, statutory or otherwise, including without limitation warranties of title, merchantability, fitness for a particular purpose, non infringement, or the absence of latent or other defects, accuracy, or the present or absence of errors, whether or not discoverable, all to the greatest extent permissible under applicable law.
c. Affirmer disclaims responsibility for clearing rights of other persons that may apply to the Work or any use thereof, including without limitation any person's Copyright and Related Rights in the Work. Further, Affirmer disclaims responsibility for obtaining any necessary consents, permissions or other rights required for any use of the Work.
d. Affirmer understands and acknowledges that Creative Commons is not a party to this document and has no duty or obligation with respect to this CC0 or use of the Work.