Coder proprement

Robert C. Martin

Michael C. Feathers Timothy R. Ottinger
Jeffrey J. Langr Brett L. Schuchert
James W. Grenning Kevin Dean Wampler
Object Mentor Inc.

1

Code propre

Si vous lisez ce livre, c’est pour deux raisons : vous êtes programmeur et vous voulez être un meilleur programmeur. C’est parfait, nous avons besoin de meilleurs programmeurs.

Ce livre s'intéresse aux bonnes pratiques de programmation. Il est rempli de code, que examinerons sous plusieurs angles. Nous l'étudierons de haut en bas, de bas en haut et à partir du milieu. Lorsque nous aurons terminé, nous aurons acquis de nombreuses connaissances sur le code, mais le plus important est que nous saurons différencier le bon code et le mauvais code. Nous saurons comment écrire du bon code et comment transformer du mauvais code en bon code.

Il y aura toujours du code

Certains pourraient prétendre qu'un livre qui traite du code est un livre dépassé – le code n'est plus le problème – et qu'il est plus important aujourd'hui de s'intéresser aux modèles et aux exigences. Il a même été suggéré que la fin du code était proche, qu'il serait bientôt généré au lieu d'être écrit, que les programmeurs seraient bientôt inutiles car les directeurs de projets généreraient les programmes à partir des spécifications.

Balivernes ! Nous ne pourrons jamais nous débarrasser du code car il représente les détails des exigences. À un certain niveau, ces détails ne peuvent pas être ignorés ou absents ; ils doivent être précisés. Préciser des exigences à un niveau de détail qui permet à une machine de les exécuter s'appelle *programmer*. Cette spécification *est le code*.

Je m'attends à ce que le niveau d'abstraction de nos langages continue d'augmenter. Je m'attends également à l'augmentation du nombre de langages spécifiques à un domaine. Ce sera une bonne chose. Mais ce n'est pas pour autant que le code disparaîtra. Les spécifications écrites dans ces langages de plus haut niveau et spécifiques à un domaine seront évidemment du code ! Il devra toujours être rigoureux, précis et tellement formel et détaillé qu'une machine pourra le comprendre et l'exécuter.

Ceux qui pensent que le code disparaîtra un jour sont comme ces mathématiciens qui espèrent découvrir un jour des mathématiques qui n'ont pas besoin d'être formelles. Ils espèrent pouvoir trouver une manière de créer des machines qui réalisent ce que nous souhaitons, non ce que nous exprimons. Ces machines devront nous comprendre parfaitement, au point de pouvoir traduire nos besoins exprimés de manière vague en des programmes opérationnels qui répondent précisément à ces besoins.

Cela ne se produira jamais. Même les humains, avec toute leur intuition et leur créativité, ne sont pas capables de créer des systèmes opérationnels à partir des vagues sentiments de leurs clients. Si la spécification des exigences nous a enseigné quelque chose, c'est que les exigences parfaitement définies sont aussi formelles que du code et qu'elles peuvent servir de tests exécutables pour ce code !

N'oubliez pas que le code n'est que le langage dans lequel nous exprimons finalement les exigences. Nous pouvons créer des langages plus proches des exigences. Nous

pouvons créer des outils qui nous aident à analyser et à assembler ces exigences en structures formelles. Mais nous n'enlèverons jamais une précision nécessaire. Par conséquent, il y aura toujours du code.

Mauvais code

Je lisais récemment la préface du livre de Kent Beck, *Implementation Patterns* [Beck07]. Il y est écrit "[...] ce livre se fonde sur un postulat relativement fragile : le bon code a une importance [...]" Je ne suis absolument pas d'accord avec le qualificatif fragile. Je pense que ce postulat est l'un des plus robustes, des plus cautionnés et des plus surchargés de tous les postulats de notre métier (et je pense que Kent le sait également). Nous savons que le bon code est important car nous avons dû nous en passer pendant trop longtemps.

Je connais une entreprise qui, à la fin des années 1980, a développé une application *phare*. Elle a été très populaire, et un grand nombre de professionnels l'ont achetée et employée. Mais les cycles de livraison ont ensuite commencé à s'étirer. Les bogues n'étaient pas corrigés d'une version à la suivante. Les temps de chargement se sont allongés et les crashs se sont multipliés. Je me rappelle avoir un jour fermé ce produit par frustration et ne plus jamais l'avoir utilisé. Peu après, l'entreprise faisait faillite.

Vingt ans plus tard, j'ai rencontré l'un des premiers employés de cette société et lui ai demandé ce qui s'était passé. Sa réponse a confirmé mes craintes. Ils s'étaient précipités pour placer le produit sur le marché, mais avaient massacré le code. Avec l'ajout de nouvelles fonctionnalités, la qualité du code s'est dégradée de plus en plus, jusqu'à ce qu'ils ne puissent plus le maîtriser. *Un mauvais code a été à l'origine de la faillite de l'entreprise*.

Avez-vous déjà été vraiment gêné par du mauvais code ? Si vous êtes un programmeur possédant une quelconque expérience, vous avez déjà dû faire face de nombreuses fois à cet obstacle. Nous donnons même un nom à ce processus : *patauger*. Nous pataugeons dans le mauvais code. Nous avançons laborieusement dans un amas de ronces enchevêtrées et de pièges cachés. Nous nous débattons pour trouver notre chemin, en espérant des indications et des indices sur ce qui se passe. Mais, tout ce que nous voyons, c'est de plus en plus de code dépourvu de sens.

Bien évidemment, vous avez déjà été gêné par du mauvais code. Dans ce cas, pourquoi l'avez-vous écrit ?

Tentiez-vous d'aller vite ? Vous étiez probablement pressé. Vous pensiez sans doute que vous n'aviez pas le temps de faire un bon travail, que votre chef serait en colère si vous preniez le temps de nettoyer votre code. Peut-être étiez-vous simplement fatigué de travailler sur ce programme et souhaitiez en finir. Peut-être avez-vous regardé la liste des autres tâches à effectuer et avez réalisé que vous deviez expédier ce module afin de pouvoir passer au suivant. Nous l'avons tous fait.

Nous avons tous examiné le désordre que nous venions de créer et choisi de le laisser ainsi encore un peu. Nous avons tous été soulagés de voir notre programme peu soigné fonctionner et décidé que c'était toujours mieux que rien. Nous avons tous pensé y revenir plus tard pour le nettoyer. Bien entendu, à ce moment-là nous ne connaissions pas la loi de LeBlanc : *Plus tard signifie jamais.*

Coût total d'un désordre

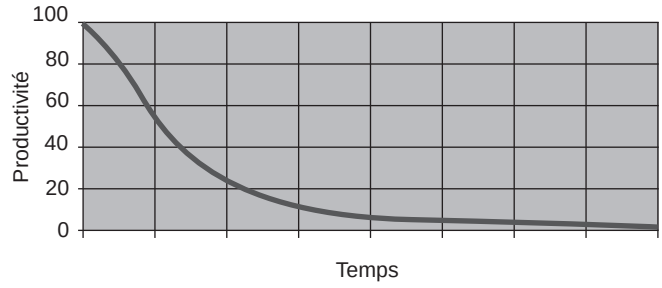
Si vous êtes programmeur depuis plus de deux ou trois ans, vous avez probablement déjà été ralenti par le code négligé d'une autre personne. Le degré de ralentissement peut être important. Sur une année ou deux, les équipes qui ont progressé très rapidement au début d'un projet peuvent finir par avancer à l'allure d'un escargot. Chaque changement apporté au code remet en cause deux ou trois autres parties du code. Aucune modification n'est insignifiante. Tout ajout ou modification du système exige que les enchevêtrements, les circonvolutions et les nœuds soient "compris" afin que d'autres puissent être ajoutés. Au fil du temps, le désordre devient si important, si profond et si grand qu'il est impossible de procéder à une quelconque réorganisation.

Plus le désordre augmente, plus la productivité de l'équipe décroît, de manière asymptotique en s'approchant de zéro. Lorsque la productivité diminue, la direction fait la seule chose qu'elle sache faire : affecter un plus grand nombre de personnes au projet en espérant augmenter la productivité. Mais ces nouvelles personnes ne sont pas versées dans la conception du système. Elles ne savent pas différencier une modification qui correspond à la conception et une modification qui la contrarie. Par ailleurs, elles sont, comme les autres membres de l'équipe, soumises à une forte pression pour améliorer la productivité. Elles ne font qu'augmenter le désordre, en amenant la productivité encore plus près de zéro (voir Figure 1.1).

L'utopie de la grande reprise à zéro

Vient le jour où l'équipe se rebelle. Elle informe la direction qu'elle ne peut pas continuer à développer avec cette odieuse base de code. Elle demande à reprendre à zéro. La

Figure 1.1
La productivité au fil du temps.



direction ne veut pas déployer des ressources sur une toute nouvelle conception du projet, mais elle ne peut pas nier le manque de productivité. Elle finit par se plier aux demandes des développeurs et autorise la grande reprise à zéro.

Une nouvelle équipe d'experts est constituée. Tout le monde souhaite en faire partie car il s'agit d'un nouveau projet. Elle doit repartir de zéro et créer quelque chose de vraiment beau. Mais seuls les meilleurs et les plus brillants sont retenus. Tous les autres doivent continuer à maintenir le système actuel.

Les deux équipes sont alors en course. L'équipe d'experts doit construire un nouveau système qui offre les mêmes fonctionnalités que l'ancien. Elle doit également suivre les modifications constamment apportées à l'ancien système. La direction ne remplacera pas l'ancien système tant que le nouveau n'assurera pas les mêmes fonctions que l'ancien.

Cette course peut durer très longtemps. Je l'ai vue aller jusqu'à dix ans. Le jour où elle est terminée, les membres originels de l'équipe d'experts sont partis depuis longtemps et les membres actuels demandent à ce que le nouveau système soit revu car il est vraiment mal conçu.

Si vous avez déjà expérimenté ce déroulement, même pendant une courte durée, vous savez pertinemment que passer du temps à garder un code propre n'est pas une question de coûts ; il s'agit d'une question de survie professionnelle.

Attitude

Avez-vous déjà pataugé dans un désordre tel qu'il faut des semaines pour faire ce qui devrait prendre des heures ? Avez-vous déjà procédé à un changement qui aurait dû se faire sur une ligne alors que des centaines de modules différents ont été impliqués ? Ces symptômes sont par trop communs.

Pourquoi cela arrive-t-il au code ? Pourquoi un bon code se dégrade-t-il si rapidement en un mauvais code ? Nous avons de nombreuses explications. Nous nous plaignons que les exigences évoluent d'une manière qui contredit la conception initiale. Nous

déplorons que les échéances soient trop courtes pour pouvoir faire les choses bien. Nous médisons les directeurs stupides, les clients intolérants, les types inutiles du marketing et le personnel d'entretien. Mais la faute, cher Dilbert, n'en est pas à nos étoiles, elle en est à nous-mêmes. Nous ne sommes pas professionnels.

La pilule est sans doute difficile à avaler. Comment ce désordre pourrait-il être de *notre* faute ? *Quid* des exigences ? *Quid* des échéances ? *Quid* des directeurs stupides et des types inutiles du marketing ? Ne portent-ils pas une certaine faute ?

Non. Les directeurs et les responsables marketing nous demandent les informations dont ils ont besoin pour définir leurs promesses et leurs engagements. Et, même s'ils ne nous interrogent pas, nous ne devons pas éviter de leur dire ce que nous pensons. Les utilisateurs se tournent vers nous pour valider la manière dont les exigences se retrouveront dans le système. Les chefs de projet comptent sur nous pour respecter les échéances. Nous sommes totalement complices du planning du projet et partageons une grande part de responsabilité dans les échecs ; en particulier si ces échecs sont liés à du mauvais code !

"Mais, attendez !, dites-vous. Si je ne fais pas ce que mon chef demande, je serai licencié." Probablement pas. La plupart des directeurs veulent connaître la vérité, même s'ils ne le montrent pas. La plupart des directeurs veulent du bon code, même lorsqu'ils sont obsédés par les échéances. Ils peuvent défendre avec passion le planning et les exigences, mais c'est leur travail. Le vôtre consiste à défendre le code avec une passion équivalente.

Pour replacer ce point de vue, que penseriez-vous si vous étiez chirurgien et que l'un de vos patients vous demandait d'arrêter de vous laver les mains avant l'intervention car cela prend trop de temps[1] ? Le patient est évidemment le chef, mais le chirurgien doit absolument refuser de se conformer à sa demande. En effet, il connaît mieux que le patient les risques de maladie et d'infection. Il ne serait pas professionnel (sans parler de criminel) que le chirurgien suive le patient.

Il en va de même pour les programmeurs qui se plient aux volontés des directeurs qui ne comprennent pas les risques liés au désordre.

L'énigme primitive

Les programmeurs font face à une énigme basique. Tous les développeurs qui ont quelques années d'expérience savent qu'un travail précédent mal fait les ralentira. Et tous

1. Lorsque le lavage des mains a été recommandé pour la première fois aux médecins par Ignaz Semmelweis en 1847, il a été rejeté car les docteurs étaient trop occupés et n'auraient pas eu le temps de laver leurs mains entre deux patients.

les développeurs connaissent la pression qui conduit au désordre pour respecter les échéances. En résumé, ils ne prennent pas le temps d'aller vite !

Les véritables professionnels savent que la deuxième partie de l'énigme est fausse. Vous ne respecterez pas les échéances en travaillant mal. À la place, vous serez ralenti instantanément par le désordre et vous serez obligé de manquer l'échéance. La seule manière de respecter le planning, ou d'aller vite, est de garder en permanence le code aussi propre que possible.

L'art du code propre

Supposons que vous pensiez que le code négligé est un obstacle important. Supposons que vous acceptiez que la seule manière d'aller vite est de garder un code propre. Alors, vous devez vous demander : "Comment puis-je écrire du code propre ?" Il n'est pas bon d'essayer d'écrire du code propre si vous ne savez pas ce que signifie propre dans ce contexte !

Malheureusement, écrire du code ressemble à peindre un tableau. La majorité d'entre nous sait reconnaître un tableau bien ou mal peint. Cependant, être capable de faire cette différence ne signifie pas être capable de peindre. De même, être capable de différencier le code propre du code sale ne signifie pas savoir écrire du code propre !

Pour écrire du code propre, il faut employer de manière disciplinée une myriade de petites techniques appliquées par l'intermédiaire d'un sens de "propreté" méticuleusement acquis. Cette "sensibilité" au code constitue la clé. Certains d'entre nous sont nés avec, d'autres doivent se battre pour l'acquérir. Non seulement elle nous permet de voir si le code est bon ou mauvais, mais elle nous montre également la stratégie à employer pour transformer un code sale en code propre.

Le programmeur qui ne possède pas cette sensibilité pourra reconnaître le désordre dans un module négligé, mais n'aura aucune idée de ce qu'il faut faire. *A contrario*, un programmeur qui possède cette sensibilité examinera un module négligé et verra les options qui s'offrent à lui. Cette faculté l'aidera à choisir la meilleure solution et le guidera à établir une suite de comportements qui garantissent le projet du début à la fin.

En résumé, un programmeur qui écrit du code propre est un artiste capable de prendre un écran vierge et de le passer au travers d'une suite de transformations jusqu'à ce qu'il obtienne un système codé de manière élégante.

Qu'est-ce qu'un code propre ?

Il existe probablement autant de définitions que de programmeurs. C'est pourquoi j'ai demandé l'avis de programmeurs très connus et très expérimentés.

***Bjarne Stroustrup, inventeur du C++ et auteur du livre* Le Langage C++**

> *J'aime que mon code soit élégant et efficace. La logique doit être simple pour que les bogues aient du mal à se cacher. Les dépendances doivent être minimes afin de faciliter la maintenance. La gestion des erreurs doit être totale, conformément à une stratégie articulée. Les performances doivent être proches de l'idéal afin que personne ne soit tenté d'apporter des optimisations éhontées qui dégraderaient le code. Un code propre fait une chose et la fait bien.*

Bjarne utilise le terme "élégant". Quel mot ! Le dictionnaire de mon MacBook® donne les définitions suivantes : *agréablement gracieux et équilibré dans les proportions ou dans la forme ; agréablement simple et ingénieux*. Vous aurez remarqué l'insistance sur le mot "agréable". Bjarne semble penser qu'un code propre est *agréable* à lire. En le lisant, vous devez sourire, autant qu'en contemplant une boîte à musique bien ouvragée ou une voiture bien dessinée.

Bjarne mentionne également, deux fois, l'efficacité. Cela ne devrait pas nous surprendre de la part de l'inventeur du C++, mais je pense que cela va plus loin que le pur souhait de rapidité. Les cycles gaspillés sont inélégants, désagréables. Notez le terme employé par Bjarne pour décrire les conséquences de cette grossièreté. Il choisit le mot "tenter". Il y a ici une vérité profonde. Le mauvais code a *tendance* à augmenter le désordre ! Lorsque d'autres personnes interviennent sur le mauvais code, elles ont tendances à le dégrader.

Dave Thomas et Andy Hunt expriment cela de manière différente. Ils utilisent la métaphore des vitres cassées[2]. Lorsque les vitres d'un bâtiment sont cassées, on peut penser que personne ne prend soin du lieu. Les gens ne s'en occupent donc pas. Ils acceptent que d'autres vitres soient cassées, voire les cassent eux-mêmes. Ils salissent la façade avec des graffiti et laissent les ordures s'amonceler. Une seule vitre cassée est à l'origine du processus de délabrement.

Bjarne mentionne également que le traitement des erreurs doit être complet. Cela est en rapport avec la règle de conduite qui veut que l'on fasse attention aux détails. Pour les programmeurs, un traitement des erreurs abrégé n'est qu'une manière de passer outre les détails. Les fuites de mémoire en sont une autre, tout comme la concurrence critique

2. **http://www.artima.com/intv/fixit.html**.

et l'usage incohérent des noms. En conséquence, le code propre met en évidence une attention minutieuse envers les détails.

Bjarne conclut en déclarant que le code propre ne fait qu'une seule chose et la fait bien. Ce n'est pas sans raison si de nombreux principes de conception de logiciels peuvent se ramener à ce simple avertissement. Les auteurs se tuent à communiquer cette réflexion. Le mauvais code tente d'en faire trop, ses objectifs sont confus et ambigus. Un code propre est *ciblé*. Chaque fonction, chaque classe, chaque module affiche un seul comportement déterminé, insensible aux détails environnants.

Grady Booch, auteur du livre Object Oriented Analysis and Design with Applications

> *Un code propre est un code simple et direct. Il se lit comme une prose parfaitement écrite. Un code propre ne cache jamais les intentions du concepteur, mais est au contraire constitué d'abstractions nettes et de lignes de contrôle franches.*

Grady souligne les mêmes aspects que Bjarne, mais se place sur le plan de la *lisibilité*. J'adhère particulièrement à son avis qu'un code propre doit pouvoir se lire aussi bien qu'une prose parfaitement écrite. Pensez à un livre vraiment bon que vous avez lu. Les mots disparaissaient pour être remplacés par des images ! C'est comme regarder un film. Mieux, vous voyez les caractères, entendez les sons, ressentez les émotions et l'humour.

Lire un code propre ne sera sans doute jamais équivalent à lire *Le Seigneur des anneaux*. La métaphore littéraire n'en reste pas moins intéressante. Comme un roman, un code propre doit clairement montrer les tensions dans le problème à résoudre. Il doit les mener à leur paroxysme, pour qu'enfin le lecteur se dise "Eh oui, évidemment !" en arrivant à la réponse évidente aux questions.

L'expression "abstractions nettes" employée par Grady est un oxymoron fascinant ! Le mot "net" n'est-il pas un quasi-synonyme de "concret" ? Le dictionnaire de mon MacBook en donne la définition suivante : *d'une manière précise, brutale, sans hésitation et sans ambiguïté.* Malgré cette apparente juxtaposition de significations, les mots portent un message fort. Notre code doit être pragmatique, non spéculatif. Il ne doit contenir que le nécessaire. Nos lecteurs doivent nous sentir déterminés.

"Big" Dave Thomas, fondateur d'OTI, parrain de la stratégie d'Eclipse

> *Un code propre peut être lu et amélioré par un développeur autre que l'auteur d'origine. Il dispose de tests unitaires et de tests de recette. Il utilise des noms significatifs. Il propose une manière, non plusieurs, de réaliser une chose. Ses dépendances sont minimales et explicitement définies. Il fournit une API claire et minimale. Un code doit être littéraire puisque, selon le langage, les informations nécessaires ne peuvent pas toutes être exprimées clairement par le seul code.*

Big Dave partage le souhait de lisibilité de Grady, mais avec une autre formulation. Dave déclare que le code propre doit pouvoir être facilement amélioré par d'*autres* personnes. Cela peut sembler évident, mais il n'est pas inutile de le rappeler. En effet, il existe une différence entre un code facile à lire et un code facile à modifier.

Dave associe la propreté aux tests ! Il y a une dizaine d'années, cela aurait fait tiquer de nombreuses personnes. Mais le développement piloté par les tests a eu un impact profond sur notre industrie et est devenu l'une de nos disciplines fondamentales. Dave a raison. Un code sans tests ne peut pas être propre. Quelles que soient son élégance, sa lisibilité et son accessibilité, s'il ne possède pas de tests, il n'est pas propre.

Dave emploie deux fois le mot *minimal*. Il semble préférer le code court au code long. C'est un refrain que l'on a beaucoup entendu dans la littérature informatique. Plus c'est petit, mieux c'est.

Dave prétend également que le code doit être *littéraire*. Il s'agit d'une référence discrète à la *programmation littéraire* de Knuth [Knuth92]. Le code doit être écrit de sorte qu'il puisse être lu par les humains.

Michael Feathers, auteur de Working Effectively with Legacy Code

> *Je pourrais établir la liste de toutes les qualités que j'ai notées dans un code propre, mais l'une d'elles surpasse toutes les autres. Un code propre semble toujours avoir été écrit par quelqu'un de soigné. Rien ne permet de l'améliorer de manière évidente. Tout a déjà été réfléchi par l'auteur du code et, si vous tentez d'imaginer des améliorations, vous revenez au point de départ, en appréciant le code que l'on vous a laissé – un code laissé par quelqu'un qui se souciait énormément de son métier.*

Un mot : soin. Il s'agit du véritable sujet de cet ouvrage. Un sous-titre approprié pourrait être "comment prendre soin de son code".

Michael a mis le doigt dessus. Un code propre est un code dont on a pris soin. Quelqu'un a pris le temps de le garder simple et ordonné. Il a porté l'attention nécessaire aux détails. Il a été attentionné.

Ron Jeffries, auteur de Extreme Programming Installed *et de* Extreme Programming Adventures in C#

Ron a débuté sa carrière en programmant en Fortran pour Strategic Air Command. Il a écrit du code dans pratiquement tous les langages et pour pratiquement toutes les machines. Nous avons tout intérêt à entendre son point de vue.

> *Ces dernières années, j'ai commencé, et pratiquement réussi, à suivre les règles de code simple de Beck. Par ordre de priorité, un code simple :*
>
> - *passe tous les tests ;*
> - *n'est pas redondant ;*
> - *exprime toutes les idées de conception présentes dans le système ;*

- *minimise le nombre d'entités, comme les classes, les méthodes, les fonctions et assimilées.*

Parmi tous ces points, je m'intéresse principalement à la redondance. Lorsque la même chose se répète de nombreuses fois, cela signifie que l'une de nos idées n'est pas parfaitement représentée dans le code. Je tente tout d'abord de la déterminer, puis j'essaie de l'exprimer plus clairement.

Pour moi, l'expressivité se fonde sur des noms significatifs et je renomme fréquemment les choses plusieurs fois avant d'être satisfait. Avec des outils de développement modernes, comme Eclipse, il est facile de changer les noms. Cela ne me pose donc aucun problème. Cependant, l'expressivité ne se borne pas aux noms. Je regarde également si un objet ou une méthode n'a pas plusieurs rôles. Si c'est le cas d'un objet, il devra probablement être décomposé en deux objets, ou plus. Dans le cas d'une méthode, je lui applique toujours la procédure Extract Method *afin d'obtenir une méthode qui exprime plus clairement ce qu'elle fait et quelques méthodes secondaires qui indiquent comment elle procède.*

La redondance et l'expressivité m'amènent très loin dans ce que je considère être un code propre. L'amélioration d'un code sale en ayant simplement ces deux aspects à l'esprit peut faire une grande différence. Cependant, je sais que je dois agir sur un autre point, mais il est plus difficile à expliquer.

Après des années de développement, il me semble que tous les programmes sont constitués d'éléments très similaires. C'est par exemple le cas de l'opération "rechercher des choses dans une collection". Dès lors que nous avons une base de données d'employés, une table de hachage de clés et de valeurs ou un tableau d'éléments de n'importe quelle sorte, nous voulons constamment rechercher un élément précis dans cette collection. Lorsque cela arrive, j'enveloppe la mise en œuvre particulière dans une méthode ou une classe plus abstraite. J'en retire ainsi plusieurs avantages.

Je peux alors implémenter la fonctionnalité avec quelque chose de simple, par exemple une table de hachage, mais, puisque toutes les références à cette recherche sont désormais couvertes par la petite abstraction, je peux modifier l'implémentation à tout moment. Je peux avancer plus rapidement, tout en conservant une possibilité de modifications ultérieures.

Par ailleurs, l'abstraction de collection attire souvent mon attention sur ce qui se passe "réellement" et m'empêche d'implémenter un comportement de collection arbitraire alors que je souhaite simplement disposer d'une solution me permettant de trouver ce que je recherche.

Redondance réduite, haute expressivité et construction à l'avance d'abstractions simples. Voilà ce qui, pour moi, permet d'obtenir un code propre.

En quelques paragraphes courts, Ron a résumé le contenu de cet ouvrage. Tout se rapporte aux points suivants : pas de redondance, une seule chose, expressivité et petites abstractions.

Ward Cunningham, inventeur des wikis, inventeur de Fit, co-inventeur de l'eXtreme Programming. Force motrice derrière les motifs de conception. Leader des réflexions sur Smalltalk et l'orienté objet. Parrain de tous ceux qui prennent soin du code.

> *Vous savez que vous travaillez avec du code propre lorsque chaque méthode que vous lisez correspond presque parfaitement à ce que vous attendiez. Vous pouvez le qualifier de beau code lorsqu'il fait penser que le langage était adapté au problème.*

Ces déclarations sont caractéristiques de Ward. Vous les lisez, hochez la tête, puis passez au sujet suivant. Elles paraissent si justes et si évidentes qu'elles se présentent rarement comme quelque chose de profond. Vous pourriez penser qu'elles correspondent presque parfaitement à ce que vous attendiez. Cependant, étudions-les de plus près.

"... presque parfaitement à ce que vous attendiez." Quand avez-vous pour la dernière fois rencontré un module qui correspondait presque parfaitement à ce que vous attendiez ? Il est plus probable que les modules que vous examinez soient déroutants, compliqués ou embrouillés. La règle n'est-elle pas bafouée ? N'êtes-vous pas habitué à vous battre pour tenter de suivre les fils du raisonnement qui sortent du système global et se faufilent au sein du module que vous lisez ? Quand avez-vous pour la dernière fois lu du code et hoché la tête comme pour les déclarations de Ward ?

Ward explique que vous ne devez pas être surpris lorsque vous lisez du code propre. Vous ne devez même pas faire un tel effort. Vous le lisez et il correspond presque parfaitement à ce que vous attendiez. Il est évident, simple et incontestable. Chaque module plante le décor pour le suivant. Chacun vous indique comment sera écrit le suivant. Un programme qui affiche une telle propreté est tellement bien écrit que vous ne vous en apercevrez même pas. Le concepteur l'a rendu ridiculement simple, comme c'est le cas des conceptions exceptionnelles.

Quid de la notion de beauté mentionnée par Ward ? Nous nous sommes tous insurgés contre le fait que nos langages n'étaient pas adaptés à nos problèmes. Cependant, la déclaration de Ward nous retourne nos obligations. Il explique qu'un beau code *fait*

penser que le langage était adapté au problème ! Il est donc de *notre* responsabilité de faire en sorte que le langage semble simple ! Attention, les sectaires du langage sont partout ! Ce n'est pas le langage qui fait qu'un programme apparaît simple. C'est le programmeur qui fait que le langage semble simple !

Écoles de pensée

Et de mon côté (Oncle Bob) ? Quel est mon point de vue sur le code propre ? Cet ouvrage vous expliquera, avec force détails, ce que mes compatriotes et moi-même pensons du code propre. Nous vous dirons ce qui nous fait penser qu'un nom de variable, qu'une fonction, qu'une classe, etc. est propre. Nous exposerons notre avis comme une certitude et n'excuserons pas notre véhémence. Pour nous, à ce stade de nos carrières, il est irréfutable. Il s'agit de notre *école de pensée* quant au code propre.

Les pratiquants d'arts martiaux ne sont pas tous d'accord sur le meilleur art martial ou la meilleure technique d'un art martial. Souvent, un maître forme sa propre école de pensée et réunit des étudiants pour la leur enseigner. Il existe ainsi le *Gracie Jiu Jitsu*, fondé et enseigné par la famille Gracie au Brésil, le *Hakkoryu Jiu Jitsu*, fondé et enseigné par Okuyama Ryuho à Tokyo, le *Jeet Kune Do*, fondé et enseigné par Bruce Lee aux États-Unis.

Dans chaque école, les étudiants s'immergent dans les enseignements du fondateur. Ils se consacrent à l'apprentissage de la discipline du maître, souvent en excluant celles des autres maîtres. Ensuite, alors qu'ils progressent au sein de leur art, ils peuvent devenir les étudiants de maîtres différents, élargissant ainsi leurs connaissances et leurs pratiques. Certains affinent leurs techniques, en découvrent de nouvelles et fondent leurs propres écoles.

Aucune de ces écoles n'a, de manière absolue, raison, même si, au sein d'une école particulière, nous agissons comme si les enseignements et les techniques étaient les seules bonnes. Il existe en effet une bonne manière de pratiquer le Hakkoryu Jiu Jitsu ou le Jeet Kune Do. Mais cette justesse au sein d'une école n'invalide en rien les enseignements d'une autre école.

Ce livre doit être pris comme la description de l'*École Object Mentor du code propre*. Les techniques et les enseignements qu'il contient sont notre manière de pratiquer notre

art. Nous sommes prêts à affirmer que, si vous suivez notre enseignement, vous apprécierez les avantages dont nous avons bénéficié et vous apprendrez à écrire du code propre et professionnel. Mais n'allez pas croire que nous avons absolument raison. D'autres écoles et d'autres maîtres revendiquent autant que nous leur professionnalisme. Il pourrait vous être nécessaire d'apprendre également auprès d'eux.

Quelques recommandations données dans cet ouvrage peuvent évidemment être sujettes à controverse. Vous ne serez sans doute pas d'accord avec certaines d'entre elles, voire y serez totalement opposé. Pas de problème. Nous ne prétendons pas constituer l'autorité finale. Néanmoins, nous avons réfléchi longtemps et durement à ces recommandations. Elles découlent de plusieurs dizaines d'années d'expérience et de nombreux processus d'essais et d'erreurs. Que vous soyez d'accord ou non, il serait dommage que vous ne connaissiez pas, et ne respectiez pas, notre point de vue.

Nous sommes des auteurs

Le champ `@author` de Javadoc indique qui nous sommes : les auteurs. Les auteurs ont pour caractéristique d'avoir des lecteurs. Les auteurs ont pour *responsabilité* de bien communiquer avec leurs lecteurs. La prochaine fois que vous écrirez une ligne de code, n'oubliez pas que vous êtes un auteur qui écrit pour des lecteurs qui jugeront votre travail.

Vous pourriez vous demander dans quelle mesure un code est réellement lu. L'effort principal ne se trouve-t-il pas dans son écriture ?

Avez-vous déjà rejoué une session d'édition ? Dans les années 1980 et 1990, nous utilisions des éditeurs, comme Emacs, qui conservaient une trace de chaque frappe sur le clavier. Nous pouvions travailler pendant une heure et rejouer ensuite l'intégralité de la session d'édition, comme un film en accéléré. Lorsque je procédais ainsi, les résultats étaient fascinants.

Une grande part de l'opération de relecture était constituée de défilements et de déplacements vers d'autres modules !

> *Bob entre dans le module.*
>
> *Il se déplace vers la fonction à modifier.*
>
> *Il marque une pause afin d'étudier ses possibilités.*
>
> *Oh, il se déplace vers le début du module afin de vérifier l'initialisation d'une variable.*
>
> *Il retourne à présent vers le bas et commence à saisir.*
>
> *Oups, il est en train d'effacer ce qu'il a tapé !*

Il le saisit à nouveau.

Il l'efface à nouveau !

Il saisit une partie d'autre chose, mais finit par l'effacer !

Il se déplace vers le bas, vers une autre fonction qui appelle la fonction en cours de modification afin de voir comment elle est invoquée.

Il revient vers le haut et saisit le code qu'il vient juste d'effacer.

Il marque une pause.

Il efface à nouveau ce code !

Il ouvre une nouvelle fenêtre et examine une sous-classe.

Est-ce que cette fonction est redéfinie ?

...

Vous voyez le tableau. Le rapport entre le temps passé à lire et le temps passé à écrire est bien supérieur à 10:1. Nous lisons *constamment* l'ancien code pour écrire le nouveau.

Puisque ce rapport est très élevé, la lecture du code doit être facile, même si son écriture est plus difficile. Bien entendu, il est impossible d'écrire du code sans le lire. Par conséquent, *en rendant un code facile à lire, on le rend plus facile à écrire*.

Vous ne pouvez pas échapper à cette logique. Vous ne pouvez pas écrire du code si vous ne pouvez pas lire le code environnant. Le code que vous écrivez aujourd'hui sera difficile ou facile à écrire selon la difficulté de lecture du code environnant. Par conséquent, si vous souhaitez aller vite, si vous voulez terminer rapidement, si vous voulez que votre code soit facile à écrire, rendez-le facile à lire.

La règle du boy-scout

Il ne suffit pas de bien écrire le code, il doit *rester propre* avec le temps. Nous avons tous vu du code se dégrader au fil du temps. Nous devons jouer un rôle actif pour empêcher cette dégradation.

Les boy-scouts ont une règle simple que nous pouvons appliquer à notre métier :

Laissez le campement plus propre que vous ne l'avez trouvé en arrivant.[3]

3. Adaptée du message d'adieu de Robert Stephenson Smyth Baden-Powell aux scouts : "Essayez de laisser ce monde un peu meilleur que vous ne l'avez trouvé..."

Si nous enregistrons tous un code un peu plus propre que celui que nous avons chargé, le code ne peut tout simplement pas se dégrader. Le nettoyage n'est pas nécessairement important. Trouver un meilleur nom pour une variable, découper une fonction qui est un peu trop longue, supprimer une légère redondance, nettoyer une instruction `if` composée.

Imaginez un projet dont le code s'améliore simplement avec le temps. Pensez-vous que toute autre évolution est professionnelle ? L'amélioration perpétuelle n'est-elle pas un élément intrinsèque du professionnalisme ?

Préquel et principes

Cet ouvrage est, de différentes manières, un "préquel" à celui de 2002 intitulé *Agile Software Development: Principles, Patterns, and Practices* (PPP). Le livre PPP se focalise sur les principes de la conception orientée objet et sur les pratiques employées par les développeurs professionnels. Si vous ne l'avez pas encore lu, consultez-le après celui-ci et vous constaterez qu'il en constitue une suite. Si vous l'avez déjà lu, vous verrez que bien des opinions émises dans cet ouvrage trouvent écho dans celui-ci au niveau du code.

Dans ce livre, vous rencontrerez quelques références à différents principes de conception. Il s'agit notamment du principe de responsabilité unique (SRP, *Single Responsibility Principle*), du principe ouvert/fermé (OCP, *Open Closed Principle*) et du principe d'inversion des dépendances (DIP, *Dependency Inversion Principle*). Tous ces principes sont décrits en détail dans PPP.

Conclusion

Les livres d'art ne promettent pas de vous transformer en artiste. Ils ne peuvent que vous apporter les outils, les techniques et les processus de réflexion employés par d'autres artistes. Il en va de même pour cet ouvrage. Il ne promet pas de faire de vous un bon programmeur ou de vous donner cette "sensibilité au code". Il ne peut que vous montrer les méthodes des bons programmeurs, ainsi que les astuces, les techniques et les outils qu'ils utilisent.

Tout comme un livre d'art, celui-ci est rempli de détails. Il contient beaucoup de code. Vous rencontrerez du bon code et du mauvais code. Vous verrez du mauvais code se transformer en bon code. Vous consulterez des listes d'heuristiques, de disciplines et de techniques. Vous étudierez exemple après exemple. Ensuite, c'est à vous de voir.

Connaissez-vous cette blague du violoniste qui s'est perdu pendant qu'il se rendait à son concert ? Il arrête un vieux monsieur au coin de la rue et lui demande comment faire pour aller au Carnegie Hall. Le vieux monsieur regarde le violoniste et le violon replié sous son bras, puis lui répond : "Travaille, mon garçon. Travaille !"