

REFACTORING

**Comment améliorer
le code existant**

Martin Fowler
avec la collaboration de Kent Beck

Préface d'Erich Gamma
Traduit de l'anglais par Dominique Maniez

DUNOD

Photo de couverture : OlegAlbinsky-iStock.

Authorized translation from the English language edition, entitled *Refactoring: Improving the Design of Existing Code*, 2nd edition, by Martin FOWLER, published by Pearson Education, Inc, publishing as Addison-Wesley Professional, Copyright © 2019 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc. French Edition Language published by Dunod, copyright © 2019.

Toutes les marques citées dans cet ouvrage sont des marques déposées par leurs propriétaires respectifs.

<p>Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.</p> <p>Le Code de la propriété intellectuelle du 1^{er} juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements</p>	<p>d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.</p> <p>Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).</p>
	

© Dunod, 2019

11 rue Paul Bert, 92240 Malakoff



www.dunod.com

ISBN 978-2-10-080116-9

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2^o et 3^o a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).

Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

TABLE DES MATIÈRES

Préface de la première édition	IX
Avant-propos	XI
Note du traducteur	XIX
 Refactoring : exemple introductif	1
1.1 Programme d'exemple.....	2
1.2 Commentaires sur le programme d'exemple.....	4
1.3 La première étape du refactoring.....	6
1.4 Décomposition de la fonction statement.....	7
1.5 Suppression de la variable play.....	12
1.6 Extraction de crédits de volume.....	17
1.7 Suppression de la variable format.....	19
1.8 Suppression de la totalisation des crédits de volume.....	21
1.9 Constat : beaucoup de fonctions imbriquées.....	26
1.10 Fractionnement des phases de calcul et de formatage.....	28
1.11 Constat : séparation en deux fichiers (et deux phases).....	36
1.12 Réorganisation des calculs par type.....	39
1.13 Création d'un calculateur de représentations.....	41
1.14 Déplacement des fonctions dans le calculateur.....	42
1.15 Rendre le calculateur de représentations polymorphe.....	44
1.16 Constat : création des données avec le calculateur polymorphe.....	47
1.17 Réflexions finales.....	49
 Principes du refactoring	51
2.1 Définition du refactoring.....	51
2.2 Les deux casquettes.....	52
2.3 Pourquoi devons-nous faire du refactoring?.....	53
2.4 Quand devons-nous faire du refactoring?.....	56
2.5 Problèmes liés au refactoring.....	63
2.6 Refactoring, architecture et YAGNI.....	70
2.7 Refactoring et processus de développement de logiciels plus larges.....	71
2.8 Refactoring et performances.....	73
2.9 D'où vient le refactoring?.....	75
2.10 Refactorings automatisés.....	77
2.11 Aller plus loin.....	79

3	Quand le code sent mauvais	81
3.1	Nom mystérieux.....	82
3.2	Code dupliqué.....	82
3.3	Fonction longue.....	83
3.4	Longue liste de paramètres.....	84
3.5	Données globales.....	85
3.6	Données mutables.....	86
3.7	Changement divergent.....	87
3.8	Shotgun surgery (chirurgie dispersée).....	87
3.9	Feature envy (jalousie de fonctionnalité).....	88
3.10	Amas de données.....	89
3.11	Obsession des types primitifs.....	89
3.12	Instructions switch répétées.....	90
3.13	Boucles.....	90
3.14	Élément paresseux.....	91
3.15	Généralité spéculative.....	91
3.16	Champ temporaire.....	92
3.17	Chaînes de messages.....	92
3.18	Middle man (intermédiaire).....	92
3.19	Insider trading (délit d'initié).....	93
3.20	Grande classe.....	93
3.21	Classes alternatives avec différentes interfaces.....	94
3.22	Classe de données.....	94
3.23	Refused bequest (legs refusé).....	95
3.24	Commentaires.....	95
4	Création des tests	97
4.1	La valeur des tests automatisés.....	97
4.2	Exemple de code à tester.....	100
4.3	Un premier test.....	103
4.4	Ajout d'un autre test.....	106
4.5	Modification du dispositif.....	108
4.6	Sonder les limites.....	109
4.7	Les tests sont beaucoup plus que cela.....	112
5	Présentation du catalogue	115
5.1	Format des refactorings.....	115
5.2	Le choix des refactorings.....	117
6	Premier ensemble de refactorings	119
6.1	Extraire fonction.....	120
6.2	Incorporer fonction.....	130
6.3	Extraire variable.....	133



6.4	Incorporer variable.....	137
6.5	Modifier déclaration de fonction.....	138
6.6	Encapsuler variable.....	146
6.7	Renommer variable.....	151
6.8	Introduire objet comme paramètre.....	153
6.9	Combiner fonctions en classe.....	157
6.10	Combiner fonctions en transformation.....	162
6.11	Fractionner phase.....	167
7	Encapsulation.....	173
7.1	Encapsuler enregistrement.....	174
7.2	Encapsuler collection.....	182
7.3	Remplacer primitive par objet.....	186
7.4	Remplacer variable temporaire par requête.....	190
7.5	Extraire classe.....	193
7.6	Incorporer classe.....	197
7.7	Masquer délégué.....	200
7.8	Supprimer intermédiaire.....	202
7.9	Remplacer algorithme.....	204
8	Déplacement des fonctionnalités.....	207
8.1	Déplacer fonction.....	208
8.2	Déplacer champ.....	217
8.3	Déplacer instructions dans fonction.....	223
8.4	Déplacer instructions vers appelants.....	227
8.5	Remplacer code incorporé par appel de fonction.....	232
8.6	Déplacer instructions.....	233
8.7	Fractionner boucle.....	238
8.8	Remplacer boucle par pipeline.....	241
8.9	Supprimer code mort.....	247
9	Organisation des données.....	249
9.1	Fractionner variable.....	250
9.2	Renommer champ.....	254
9.3	Remplacer variable dérivée par requête.....	258
9.4	Changer référence en valeur.....	262
9.5	Changer valeur en référence.....	265
10	Simplification de la logique conditionnelle.....	269
10.1	Décomposer condition.....	270
10.2	Consolider expression conditionnelle.....	272
10.3	Remplacer conditions imbriquées par clauses de garde.....	275
10.4	Remplacer condition par polymorphisme.....	280

10.5	Introduire cas spécial.....	298
10.6	Introduire assertion.....	311
11	Refactoring des API	315
11.1	Séparer requête du modificateur.....	316
11.2	Paramétrer fonction.....	319
11.3	Supprimer argument de drapeau.....	323
11.4	Préserver totalité objet.....	328
11.5	Remplacer paramètre par requête.....	333
11.6	Remplacer requête par paramètre.....	336
11.7	Supprimer setter.....	340
11.8	Remplacer constructeur par fonction factory.....	342
11.9	Remplacer fonction par commande.....	344
11.10	Remplacer commande par fonction.....	351
12	Gestion de l'héritage	355
12.1	Remonter méthode.....	356
12.2	Remonter champ.....	359
12.3	Remonter corps du constructeur.....	360
12.4	Abaisser méthode.....	364
12.5	Abaisser champ.....	365
12.6	Remplacer code de type par sous-classes.....	365
12.7	Supprimer sous-classe.....	373
12.8	Extraire superclasse.....	379
12.9	Réduire hiérarchie.....	384
12.10	Remplacer sous-classe par délégué.....	385
12.11	Remplacer superclasse par délégué.....	404
	Bibliographie	411
	Index	415

Pour Cindy
— *Martin*

PRÉFACE DE LA PREMIÈRE ÉDITION

Le « refactoring » a été conçu par la communauté Smalltalk, mais il n'a pas fallu longtemps pour qu'il trouve sa place parmi les autres langages de programmation. Comme le refactoring fait partie intégrante du développement des frameworks, le terme apparaît rapidement dans les conversations lorsque les créateurs de frameworks parlent de leur métier. Il est évoqué quand ils affinent leurs hiérarchies de classes et quand ils s'extasient sur le nombre de lignes de code qu'ils ont supprimées. Les développeurs savent qu'un framework ne sera pas parfait du premier coup, et qu'il doit évoluer au fur et à mesure qu'ils acquièrent de l'expérience. Ils savent aussi qu'ils passeront plus temp à lire le code et à le modifier qu'ils n'en ont passé à l'écrire. Le refactoring est la clé pour conserver un code lisible et modifiable, notamment pour les frameworks, mais aussi pour les logiciels en général.

Quelle est la nature du problème ? Tout simplement que le refactoring est risqué, car il nécessite des modifications du code en production, ce qui peut introduire des bugs subtils. S'il n'est pas fait correctement, le refactoring peut vous retarder pendant des jours, voire des semaines. Et le refactoring est encore plus risqué lorsqu'il est pratiqué de façon informelle. Vous commencez à creuser dans le code et vous découvrez rapidement de nouvelles possibilités de changement, si bien que vous creusez plus profond. Et plus vous creusez, plus vous trouvez des choses intéressantes, ce qui entraîne d'autres modifications du code... Finalement, vous finissez par creuser un trou dont vous n'arrivez pas à vous extraire. Pour éviter de creuser votre propre tombe, le refactoring doit être fait de manière systématique. Quand nous avons écrit *Design Patterns*, nous avons souligné que les modèles de conception fournissent des cibles pour les refactorings. Cependant, l'identification des cibles ne constitue qu'une partie du problème et la transformation de votre code pour en arriver là est un autre défi.

Martin Fowler et les auteurs qui lui ont emboîté le pas apportent une contribution inestimable au développement des logiciels orientés objet en attirant l'attention sur le processus de refactoring. Ce livre explique les principes et les bonnes pratiques du refactoring, en indiquant le moment opportun et l'emplacement où vous devez commencer à analyser votre code pour l'améliorer. Un catalogue complet de refactorings constitue le cœur de ce livre. Chaque refactoring décrit la finalité et le mécanisme d'une transformation du code qui a fait ses preuves. Certains des refactorings, comme Extraire méthode ou Déplacer champ, peuvent sembler évidents.

Mais détrompez-vous ! La compréhension du mécanisme de tels refactorings est la clé de la réorganisation du code bien conduite. Les refactorings de ce livre vous aideront à modifier votre code par petites étapes, ce qui réduit ainsi les risques de



Préface de la première édition

modification de votre conception. Vous ajouterez rapidement ces refactorings et leurs noms à votre lexique de développement.

Ma première expérience de refactoring bien conduit, « une étape à la fois », s'est produite quand je programmais à 30 000 pieds d'altitude, dans un avion, avec Kent Beck. Il s'était assuré que nous appliquions les refactorings du catalogue de ce livre, étape par étape. J'ai été étonné de voir le succès d'une telle pratique. Non seulement j'avais plus confiance dans le code qui en résultait, mais je me sentais aussi moins stressé. Je vous recommande fortement d'essayer ces refactorings car votre code et vous-même, vous vous sentirez beaucoup mieux.

— *Erich Gamma, Object Technology International, janvier 1999*

AVANT-PROPOS

Il était une fois, un consultant qui expertisait un projet de développement afin d'examiner une partie du code. En parcourant la hiérarchie des classes qui constituait le cœur du système, le consultant la trouva plutôt désordonnée. Les classes de niveau supérieur formulaient certaines hypothèses sur le fonctionnement des classes, hypothèses qui étaient incorporées dans du code hérité. Cependant, ce code ne convenait pas à toutes les sous-classes, et il avait été surchargé outre mesure. De légères modifications à la superclasse auraient considérablement réduit la nécessité de ces surcharges. Dans d'autres endroits, une intention de la superclasse n'avait pas été correctement comprise, et le comportement de la superclasse avait été dupliqué. Dans d'autres endroits encore, plusieurs sous-classes faisaient la même chose avec un code qui aurait pu être clairement déplacé vers le haut de la hiérarchie.

Le consultant recommanda au chef de projet d'examiner et de nettoyer le code, mais ce dernier n'était pas très enthousiaste. Le code semblait fonctionner et il y avait des pressions considérables sur les échéances. Le chef de projet décida donc de régler les problèmes plus tard.

Le consultant avait également indiqué la situation aux programmeurs travaillant sur la hiérarchie. Les programmeurs étaient passionnés et ils ont bien vu le problème car, parfois, une nouvelle paire d'yeux est nécessaire pour repérer une difficulté. Les programmeurs passèrent donc un jour ou deux à nettoyer la hiérarchie. Quand cela fut terminé, ils avaient supprimé la moitié du code de la hiérarchie sans réduire ses fonctionnalités. Ils étaient très contents du résultat et ils trouvèrent qu'il était devenu plus facile et plus rapide à la fois d'ajouter de nouvelles classes et d'utiliser les classes du reste du système.

Le chef de projet n'était pas satisfait. Le calendrier était serré et il y avait beaucoup de travail à faire. Ces deux programmeurs avaient passé deux jours à faire un travail qui n'ajoutait rien aux nombreuses fonctionnalités que le système devait livrer dans quelques mois. L'ancien code avait très bien fonctionné. Certes, le design était un peu plus « pur » et un peu plus « propre », mais le projet devait livrer du code qui fonctionnait, et non pas du code qui plairait à un universitaire. Le consultant suggéra qu'un nettoyage similaire soit effectué sur d'autres parties centrales du système, ce qui arrêterait le projet pendant une semaine ou deux. Le but était de rendre le code plus joli, et non pas d'ajouter une fonctionnalité.

Que pensez-vous de cette histoire ? Pensez-vous que le consultant a eu raison de suggérer d'autres nettoyages ? Ou suivez-vous ce vieil adage en vogue chez les ingénieurs qui stipule, « si ça marche, ne le réparez pas » ?

Je dois admettre un certain parti pris car le consultant en question, c'était moi ! Six mois plus tard, le projet a échoué, en grande partie parce que le code était trop complexe à déboguer ou à optimiser afin d'atteindre des performances acceptables.

Le consultant Kent Beck a été mandaté pour redémarrer le projet, exercice qui consiste à réécrire presque tout le système à partir de zéro. Il a fait plusieurs choses différemment, mais l'un des changements les plus importants a été d'insister sur le nettoyage en continu du code grâce au refactoring. L'efficacité améliorée de l'équipe, ainsi que le rôle joué par le refactoring, sont les éléments qui m'ont inspiré pour écrire la première édition de ce livre, afin que je puisse transmettre les connaissances que Kent et d'autres ont acquises en utilisant le refactoring pour améliorer la qualité du logiciel.

Depuis, le refactoring fait partie du vocabulaire des programmeurs et la première édition de ce livre a plutôt bien résisté au temps. Cependant, dix-huit ans est un âge avancé pour un livre de programmation, et j'ai donc senti qu'il était temps de le retravailler. Ce processus m'a fait réécrire à peu près toutes les pages du livre, mais dans un sens, il y a eu très peu de changement. L'essence du refactoring est identique car la plupart des refactorings importants ne bouleversent pas les choses. J'espère néanmoins que la réécriture de ce livre aidera plus de gens à apprendre à réaliser des refactorings efficaces.

◆ ***Qu'est-ce que le refactoring ?***

Le refactoring est le processus de modification d'un système logiciel d'une manière qui ne modifie pas le comportement externe du code tout en améliorant sa structure interne. Il s'agit d'une méthode rigoureuse pour nettoyer le code qui minimise les risques d'introduire des bugs. Essentiellement, lorsque vous faites du refactoring, vous améliorez la conception du code après son écriture.

« Améliorer la conception après son écriture » est une drôle de tournure de phrase. Pendant une grande partie de l'histoire du développement de logiciels, la plupart des gens croyaient que l'on conçoit d'abord, et que l'on code seulement après que la conception est terminée. Au fil du temps, le code sera modifié, et l'intégrité du système (sa structure en fonction de cette conception) s'estompe progressivement. Le code sombre lentement en passant de l'ingénierie au bidouillage.

Le refactoring est à l'opposé de cette pratique. Avec le refactoring, nous pouvons partir d'une conception mauvaise, voire chaotique, et la retravailler afin d'obtenir un code bien structuré. Chaque étape est simple, voire simpliste. Je déplace un champ d'une classe à l'autre, j'extrais un peu de code d'une méthode pour en faire sa propre méthode, ou je remonte ou descends un code dans une hiérarchie. Toutefois, l'effet cumulatif de ces petits changements peut améliorer radicalement la conception. C'est l'inverse exact de la notion de pourrissement logiciel.

Avec le refactoring, l'équilibre du travail change. J'ai constaté que la conception, plutôt que de se produire en amont, est continue pendant le développement. En construisant le système, j'apprends à améliorer la conception. Le résultat de cette interaction est un programme dont la conception reste bonne au fur et à mesure que le développement se poursuit.

◆ **Contenu du livre**

Ce livre est un guide du refactoring et il est destiné aux programmeurs professionnels. Mon but est de vous montrer comment pratiquer le refactoring d'une manière contrôlée et efficace. Vous apprendrez à réorganiser vos programmes sans introduire de bugs dans le code, tout en améliorant méthodiquement sa structure.

Traditionnellement, un livre commence par une introduction et je suis d'accord avec ce principe, mais j'ai du mal à présenter le refactoring avec des propos généraux. Je commence donc par un exemple et le chapitre 1 prend un petit programme comportant quelques défauts de conception courants et le réorganise en un programme qui est plus facile à comprendre et à modifier. Cela vous montrera à la fois le processus de refactoring et un certain nombre de refactorings utiles. C'est le chapitre clé à lire si vous voulez comprendre ce qu'est vraiment le refactoring.

Dans le chapitre 2, je traite des principes généraux du refactoring, et de certaines définitions, ainsi que des raisons qui peuvent vous pousser à entreprendre le refactoring. J'esquisse quelques-uns des défis que pose le refactoring. Dans le chapitre 3 que j'ai écrit avec Kent Beck, nous vous aidons à trouver les mauvaises odeurs dans le code et à les combattre grâce à des refactorings. Les tests jouent un rôle très important dans le refactoring, de sorte que le chapitre 4 décrit comment les intégrer dans le code.

Le cœur du livre, qui est constitué par le catalogue des refactorings, occupe le reste du volume. Bien qu'il ne s'agisse en aucun cas d'un catalogue complet, il couvre les refactorings essentiels dont la plupart des développeurs auront probablement besoin. Ce catalogue est né des notes que j'ai prises en apprenant à réaliser des refactorings à la fin des années 1990, et j'utilise encore ces notes maintenant car je ne me souviens pas de tout. Quand je veux faire un refactoring, comme Fractionner phase, le catalogue me rappelle comment réaliser cela d'une manière sûre, étape par étape. J'espère que c'est la partie du livre que vous fréquenteriez souvent.

L'aventure continue sur le Web

Le Web a eu un impact énorme sur notre société, et cela affecte particulièrement la façon dont nous recueillons l'information. Quand j'ai écrit la première édition de ce livre, la plupart des connaissances sur le développement de logiciels se trouvaient dans les livres, alors que maintenant, je recueille la plupart de mes informations en ligne. Cet état de fait constitue un défi pour les auteurs comme moi : les livres vont-ils continuer à jouer un rôle, et à quoi doivent-ils ressembler ?

Je crois que les livres comme celui-ci ont encore un rôle à jouer, mais ils doivent évoluer. La valeur d'un livre est constituée par un vaste ensemble de connaissances rassemblées de façon cohérente. En écrivant ce livre, j'ai essayé de couvrir de nombreux refactorings différents et de les organiser d'une manière cohérente et intégrée.

Mais cet ensemble intégré est une œuvre littéraire abstraite qui, bien qu'elle se présente traditionnellement sous la forme d'un livre papier, n'a pas besoin de cette

représentation à l'avenir. La plupart de l'industrie du livre voit encore le livre papier comme la représentation principale, et alors que nous avons adopté avec enthousiasme les ebooks, ils ne sont que des représentations électroniques d'une œuvre originale basée sur la structure d'un livre papier.

Avec ce livre, j'explore une approche différente. La forme canonique de ce livre est son site web ou son édition web. L'accès à l'édition web est inclus avec l'achat des versions imprimées ou des versions électroniques (voir les détails ci-dessous pour l'enregistrement de votre produit sur InformIT). Le livre papier est une sélection des informations du site Web, qui sont organisées d'une manière qui a du sens pour l'impression. Cet ouvrage n'essaie pas d'inclure tous les refactorings du site Web, d'autant plus qu'il est possible que j'ajoute à l'avenir des refactorings à l'édition web canonique. De la même manière, le livre électronique est une représentation différente du livre web qui peut ne pas inclure le même ensemble de refactorings que le livre imprimé; après tout, les ebooks ne s'alourdissent pas quand on leur ajoute des pages et ils peuvent être facilement mis à jour après qu'ils ont été achetés.

Je ne sais pas si vous lisez l'édition web en ligne, un ebook sur votre téléphone, un exemplaire papier, ou bien une autre forme que je ne peux pas encore imaginer alors que j'écris ces lignes. Je fais de mon mieux pour que ce travail soit utile, quelle que soit la façon dont vous voulez l'exploiter.

Pour accéder à l'édition web canonique et aux mises à jour ou bien aux corrections dès qu'elles sont disponibles, enregistrez votre exemplaire de cet ouvrage sur le site InformIT. Pour commencer le processus d'inscription, rendez-vous sur <http://www.informit.com/register> et connectez-vous (ou bien créez un compte si vous n'en avez pas). Dans la zone de texte, saisissez l'ISBN du livre, 9780134757599, et cliquez sur le bouton Submit. On vous posera une question qui garantit que vous possédez bien un exemplaire du livre imprimé ou bien du livre électronique. Une fois que vous avez enregistré avec succès votre exemplaire, ouvrez l'onglet « Digital Purchases » sur la page de votre page compte (Account) et cliquez sur le lien Launch pour ouvrir dans une nouvelle fenêtre l'édition web de cet ouvrage.

Exemples JavaScript

Comme dans la plupart des domaines techniques du développement de logiciels, les exemples de code sont très importants pour illustrer les concepts. Cependant, les refactorings se ressemblent, quel que soit le langage utilisé. Il y aura parfois des choses particulières auxquelles un langage m'oblige à prêter attention, mais les éléments fondamentaux des refactorings restent identiques.

J'ai choisi JavaScript pour illustrer ces refactorings, car j'avais le sentiment que ce langage serait lisible par le plus grand nombre de personnes. Vous ne devriez pas cependant trouver difficile l'adaptation des refactorings à n'importe quel autre langage que vous utilisez. J'essaie de ne pas utiliser les éléments les plus compliqués de ce langage, de telle sorte que vous devriez être en mesure de comprendre

les refactorings, même avec une connaissance minimale de JavaScript. Mon utilisation de JavaScript n'est certainement pas une approbation de ce langage.

Bien que j'utilise JavaScript pour mes exemples, cela ne signifie pas que les techniques de ce livre sont limitées à JavaScript. La première édition de ce livre utilisait Java, et de nombreux programmeurs l'ont trouvé utile, même s'ils n'ont jamais écrit une seule classe en Java. Au départ, je me suis amusé à illustrer cette généralité en utilisant une douzaine de langages différents pour les exemples, mais j'ai estimé que ce serait trop déroutant pour le lecteur. Pourtant, ce livre est écrit pour les programmeurs utilisant n'importe quel langage. En dehors des sections d'exemple, je ne fais aucune hypothèse sur le langage que vous utilisez et je m'attends à ce que le lecteur assimile mes commentaires généraux et les applique au langage qu'il emploie. En effet, j'espère que les lecteurs prendront les exemples JavaScript et les adapteront à leur langage.

Cela signifie qu'en dehors des discussions sur des exemples spécifiques, quand je parle de « classe », de « module », de « fonction », etc., j'utilise ces termes dans un sens général, qui n'est pas spécifique au modèle du langage JavaScript.

Le fait que j'utilise JavaScript comme langage d'exemple signifie également que j'essaie d'éviter les styles de programmation JavaScript qui seront moins familiers à ceux qui ne sont pas des développeurs JavaScript chevronnés. Ce n'est pas un livre de « refactoring du code JavaScript », mais plutôt un livre de refactoring généraliste qui utilise des exemples en JavaScript. Il existe de nombreux refactorings intéressants qui sont spécifiques à JavaScript (comme le refactoring des callbacks, des promesses, des `async/await`), mais ils dépassent le cadre de ce livre.

◆ ***À qui est destiné ce livre ?***

J'ai conçu ce livre en pensant aux programmeurs professionnels, c'est-à-dire à tous ceux qui écrivent des logiciels pour gagner leur vie. Les exemples et les discussions comprennent beaucoup de code à lire et à comprendre. Les exemples sont en JavaScript, mais devraient être applicables à la plupart des langages. Je pars du principe que le lecteur a déjà une certaine expérience de la programmation pour apprécier ce livre, mais je ne suppose pas qu'il soit expert.

Bien que la cible principale de ce livre soient les développeurs qui cherchent à en apprendre davantage sur le refactoring, ce livre sera également précieux pour ceux qui ont déjà des notions sur le refactoring, et il peut notamment être utilisé comme une aide à l'enseignement. Dans ce livre, j'ai consacré du temps à expliquer le fonctionnement de différents refactorings, afin qu'un développeur expérimenté puisse utiliser ces ressources pour former ses collègues.

Bien qu'il soit axé sur le code, le refactoring a un impact important sur la conception du système. Il est essentiel que les designers et architectes chevronnés comprennent les principes du refactoring et les utilisent dans leurs projets. Le refactoring sera mieux présenté par un développeur respecté et expérimenté. Un tel développeur comprendra mieux les principes qui sous-tendent le refactoring et adaptera

ces principes à son environnement spécifique. C'est particulièrement vrai lorsque vous utiliserez un langage autre que JavaScript, parce que vous devrez adapter les exemples que j'ai donnés à d'autres langages.

Voici comment tirer le meilleur parti de ce livre sans en lire l'intégralité.

- ✓ Si vous voulez comprendre ce qu'est le refactoring, lisez le chapitre 1 (l'exemple devrait clarifier le processus).
- ✓ Si vous voulez comprendre pourquoi vous devez réorganiser votre code, lisez les deux premiers chapitres. Ils vous enseigneront ce qu'est le refactoring et pourquoi vous devez vous y mettre.
- ✓ Si vous voulez trouver où vous devez entreprendre le refactoring, lisez le chapitre 3 qui vous indique les signes qui suggèrent la nécessité d'un refactoring.
- ✓ Si vous voulez réellement faire le refactoring de votre code, lisez les quatre premiers chapitres intégralement, puis parcourez le catalogue. Lisez suffisamment le catalogue pour savoir ce qu'il contient à peu près. Vous n'avez pas à comprendre tous les détails, mais lorsque vous avez réellement besoin d'effectuer un refactoring, lisez son contenu en détail afin de vous aider à l'employer. Le catalogue est une section de référence, et il est donc peu probable que vous le lirez d'une traite.

Une partie importante de l'écriture de ce livre a consisté à nommer les différents refactorings. La terminologie nous aide à communiquer, de telle sorte que lorsqu'un développeur conseille à un autre d'extraire un peu de code d'une fonction, ou de fractionner certains calculs en phases distinctes, ils comprennent tous les deux les références à Extraire fonction et à Fractionner phase. Ce vocabulaire permet également de sélectionner des refactorings automatisés.

◆ *Rendre à César*

Je dois avouer dès le début de cet ouvrage que j'ai une dette immense envers ceux dont le travail dans les années 1990 a permis le développement du refactoring. C'est en apprenant de leur expérience que j'ai trouvé l'inspiration pour écrire la première édition de ce livre, et bien que de nombreuses années se soient écoulées, il est important de reconnaître l'intérêt des fondations qu'ils ont posées. Idéalement, c'est l'un d'entre eux qui aurait dû écrire cette première édition, mais j'ai fini par être celui qui a eu le temps et l'énergie de le faire.

Ward Cunningham et **Kent Beck** ont été les deux premiers promoteurs du refactoring. Ils l'ont utilisé comme base de développement au tout début et ont adapté leurs processus de développement pour en tirer parti. En particulier, c'est ma collaboration avec Kent qui m'a montré l'importance du refactoring, et il constitue ma principale source d'inspiration pour ce livre.

Ralph Johnson dirige un groupe à l'Université de l'Illinois Urbana-Champaign qui est remarquable pour ses contributions pratiques à la technologie des objets. Ralph a longtemps été un champion du refactoring, et plusieurs de ses étudiants ont accompli un travail fondamental au début dans ce domaine. **Bill Opdyke** a élaboré le premier travail écrit détaillé sur le refactoring dans sa thèse de doctorat. **John Brant** et **Don**

Roberts sont allés au-delà de l'écriture de mots, et ils ont créé le premier outil de refactoring automatisé, le *Refactoring Browser*, qui fonctionne sur les programmes Smalltalk.

Beaucoup de gens ont progressé dans le domaine du refactoring depuis la première édition de ce livre. En particulier, le travail de ceux qui ont ajouté des refactorings automatisés aux outils de développement a énormément contribué à faciliter la vie des programmeurs. Il est facile pour moi de tenir pour acquis que je peux renommer une fonction largement utilisée grâce à une simple séquence de touches, mais cette facilité repose sur les efforts des équipes développant des IDE dont le travail nous aide tous.

◆ **Remerciements**

Même avec toutes ces recherches dont je me suis inspiré, j'ai eu encore besoin de beaucoup d'aide pour écrire ce livre. La première édition doit beaucoup à l'expérience et aux encouragements de **Kent Beck**. Il m'a d'abord initié au refactoring, m'a incité à commencer à écrire des notes pour enregistrer des refactorings, et m'a aidé à les formaliser afin d'obtenir une prose limpide. C'est lui qui a eu l'idée des odeurs de code. J'ai souvent l'impression qu'il aurait écrit la première édition de ce livre mieux que je ne l'ai fait, si nous n'avions pas été en train de rédiger le livre qui jetait les bases de *l'Extreme Programming*.

Tous les auteurs de livres techniques que je connais mentionnent la dette qu'ils ont à l'égard des relecteurs techniques. Nous avons tous écrit des œuvres comportant de grosses erreurs qui n'ont été détectées que par nos pairs œuvrant en tant que relecteurs. Je ne fais pas beaucoup de relecture technique moi-même, en partie parce que je ne pense pas être très bon dans ce domaine, et j'ai donc beaucoup d'admiration pour ceux qui s'en occupent. Comme on gagne une misère à relire le livre de quelqu'un d'autre, c'est donc un grand acte de générosité.

Quand j'ai commencé à travailler sérieusement sur le livre, j'ai créé une liste de diffusion de consultants pour qu'ils m'adressent leurs commentaires. Au fur et à mesure que j'avais, j'ai envoyé des ébauches de nouveaux documents à ce groupe et je leur ai demandé leur avis. Je tiens à remercier les personnes suivantes pour avoir posté leurs commentaires sur la liste de diffusion: **Arlo Belshee, Avdi Grimm, Beth Anders-Beck, Bill Wake, Brian Guthrie, Brian Marick, Chad Wathington, Dave Farley, David Rice, Don Roberts, Fred George, Giles Alexander, Greg Doench, Hugo Corbucci, Ivan Moore, James Shore, Jay Fields, Jessica Kerr, Joshua Kerievsky, Kevlin Henney, Luciano Ramalho, Marcos Brizenno, Michael Feathers, Patrick Kua, Pete Hodgson, Rebecca Parsons** et **Trisha Gee**.

Dans ce groupe, je voudrais particulièrement souligner l'aide spéciale que j'ai obtenue sur JavaScript de la part de **Beth Anders-Beck, James Shore**, et **Pete Hodgson**.

Une fois que j'ai achevé une première ébauche suffisamment complète, je l'ai envoyée pour un examen plus approfondi, parce que je voulais avoir un regard neuf sur l'ensemble du projet. **William Chargin** et **Michael Hunger** ont tous deux fait des



commentaires incroyablement détaillés. J'ai également reçu de nombreux commentaires utiles de la part de **Bob Martin** et **Scott Davis**. **Bill Wake**, outre ses contributions sur la liste de diffusion, a réalisé une relecture complète de la première ébauche.

Mes collègues de chez ThoughtWorks sont une source constante d'idées et de commentaires sur mon manuscrit. Il y a d'innombrables questions, commentaires et observations qui ont alimenté la pensée et l'écriture de ce livre. Être employé chez ThoughtWorks est extraordinaire car ils me permettent de passer beaucoup de temps à écrire et j'apprécie particulièrement les conversations régulières et les idées que me soumet **Rebecca Parsons**, notre directrice technique.

NOTE DU TRADUCTEUR

La traduction des termes du vocabulaire informatique de l'anglais vers le français est très souvent problématique car le traducteur doit s'inspirer de plusieurs sources qui fournissent parfois des résultats fort différents. Même si le traducteur s'appuie principalement sur l'usage des informaticiens professionnels, il doit aussi prendre en compte le travail des commissions de terminologie (www.culture.fr/franceterme), la terminologie prônée par les éditeurs de logiciels (Microsoft, par exemple, publie des glossaires multilingues pour toutes ses applications), ou bien les choix opérés par les premiers traducteurs de certaines technologies et que l'usage a consacrés.

D'une manière générale, la communauté francophone du développement semble avoir privilégié la conservation de la terminologie anglophone et c'est la raison pour laquelle nous avons pris le parti de ne pas traduire les mots comme *refactoring*, *commit*, *immutable*, *feature envy*, ou bien encore *middle-man*. Quand nous proposons un équivalent français qui est susceptible de ne pas être compris immédiatement par la communauté des développeurs, nous indiquons entre parenthèses et en italique le terme original anglais. Des notes de bas de page explicitent également certains choix de traduction ou bien expliquent certaines références culturelles qui peuvent échapper à un lecteur francophone.

Martin Fowler emploie très souvent dans cet ouvrage le terme client pour désigner un programme qui en utilise un autre, comme dans l'expression client-serveur. Pour éviter la confusion entre les termes anglais *client* et *customer*, qui se traduisent en général tous les deux par client en français, nous avons systématiquement traduit *client* par client et *customer* par acheteur.

Certains termes, comme *value object*, que nous avons traduit par objet-valeur sont explicités par des références entre crochets à des articles de Martin Fowler et il est conseillé de lire ces articles pour mieux cerner les concepts s'ils ne vous sont pas familiers.

Nous espérons que nos choix terminologiques faciliteront la lecture de cet ouvrage.

Dominique Maniez



Refactoring : exemple introductif

Par où commencer quand on veut parler de refactoring¹ ? En général, on démarre par l'histoire de la discipline, les principes généraux, etc. Quand quelqu'un procède de la sorte lors d'une conférence, mon attention a tendance à se relâcher. Mon esprit se met à vagabonder, et le conférencier passe au second plan jusqu'à ce qu'il donne un exemple.

Les exemples me réveillent parce que je peux voir ce qui se passe. Avec les principes, il est trop facile de tomber dans des généralisations, et cela ne me permet pas de comprendre comment appliquer les concepts. Un exemple m'aide à clarifier les choses.

Je vais donc commencer ce livre par un exemple de refactoring en vous parlant de la façon dont cela fonctionne et en vous donnant une idée du processus. Je pourrai alors faire l'introduction habituelle sur les principes dans le chapitre suivant.

Quel que soit l'exemple introductif que je prenne, j'ai cependant un problème. Si je choisis un programme long, puis le décris et explique comment réaliser son refactoring, c'est trop compliqué pour le lecteur (j'ai tenté cela avec la première édition de ce livre et j'ai fini par mettre au panier deux exemples, qui étaient de taille modeste, mais dont la description prenait pour chacun d'entre eux une centaine de pages). D'un autre côté, si je choisis un programme qui est suffisamment petit pour être compréhensible, l'intérêt du refactoring ne semble pas évident.

Je suis donc dans la contrainte classique de toute personne qui veut décrire les techniques qui sont utiles pour des programmes du monde réel. Honnêtement, cela ne vaut pas le coup d'effectuer le refactoring dont je vais vous faire la démonstration

1. NdT : nous avons choisi de garder le terme anglais dans la mesure où il est largement utilisé dans la communauté des développeurs. Nous n'avons donc pas adopté l'équivalent québécois de réusinage (calqué sur la traduction de *factory*), mais nous nous autoriserons parfois dans cette traduction les équivalences remaniement, refonte, voire réingénierie.

sur le petit programme qui me sert d'exemple. Mais si le code que je vous montre fait partie d'un système plus large, alors le refactoring devient trop important. Je vous demande donc d'examiner mon exemple et de l'imaginer dans le contexte d'un système beaucoup plus grand.

— 1.1 PROGRAMME D'EXEMPLE

Dans la première édition de ce livre, mon programme d'exemple imprimait une facture d'un vidéo-club, mais aujourd'hui certains d'entre vous se demanderaient peut-être ce qu'est un vidéo-club... Plutôt que de répondre à cette question, j'ai modifié mon exemple en prenant un domaine qui est à la fois plus ancien, mais toujours d'actualité.

Imaginez une société regroupant des acteurs de théâtre qui participent à différents événements culturels en jouant des pièces. Typiquement, un acheteur² demande quelques pièces de théâtre et l'entreprise les facture en fonction de la taille de l'auditoire et du type de pièce qui est jouée. L'entreprise propose deux types de pièces : les tragédies et les comédies. En plus de fournir une facture pour la représentation, la société offre à ses acheteurs des « crédits de volume » qu'ils peuvent utiliser pour des rabais sur les futures représentations, ce qui constitue une sorte de mécanisme de fidélisation de la clientèle.

Les acteurs stockent des données sur leurs pièces dans un simple fichier JSON qui ressemble à ceci :

Plays.json...

```
{
  "hamlet": {"name": "Hamlet", "type": "tragedy"},
  "as-like": {"name": "As You Like It", "type": "comedy"},
  "othello": {"name": "Othello", "type": "tragedy"}
}
```

Les données de leurs factures se trouvent également dans un fichier JSON :

Invoices.json...

```
[
  {
    "customer": "BigCo",
    "performances": [
      {
        "playID": "hamlet",
        "audience": 55
      }
    ]
  }
]
```

2. NdT : l'auteur emploie très souvent dans cet ouvrage le terme client pour désigner un programme qui en utilise un autre, comme dans l'expression client-serveur. Pour éviter la confusion entre les termes anglais *client* et *customer*, qui se traduisent en général tous les deux par client en français, nous avons systématiquement traduit *client* par client et *customer* par acheteur.

```
    },
    {
      "playID": "as-like",
      "audience": 35
    },
    {
      "playID": "othello",
      "audience": 40
    }
  ]
}
```

Le code³ qui imprime la facture est constitué de cette fonction simple :

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
    { style: "currency", currency: "USD",
      minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
    case "tragedy":
      thisAmount = 40000;
      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audience - 30);
      }
      break;
    case "comedy":
      thisAmount = 30000;
      if (perf.audience > 20) {
        thisAmount += 10000 + 500 * (perf.audience - 20);
      }
      thisAmount += 300 * perf.audience;
      break;
    default:
```

3. NdT: nous n'avons pas traduit les identificateurs du code des exemples de cet ouvrage, mais il est peut-être utile de rappeler la signification de certains termes dans ce contexte ; *play* (pièce de théâtre), *performance* (représentation), *statement* (relevé de compte).

```
        throw new Error(`unknown type: ${play.type}`);
    }

    // ajoute des crédits de volume
    volumeCredits += Math.max(perf.audience - 30, 0);
    // ajoute un crédit par groupe de cinq spectateurs assistant à une comédie
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // imprime la ligne de cette commande
    result += ` ${play.name}: ${format(thisAmount/100)} (${perf.audience}
    seats)\n`;
    totalAmount += thisAmount;
}
result += `Amount owed is ${format(totalAmount/100)}\n`;
result += `You earned ${volumeCredits} credits\n`;
return result;
}
```

L'exécution de ce code sur les fichiers de données de test ci-dessus fournit la sortie suivante :

```
Statement for BigCo
  Hamlet: $650.00 (55 seats)
  As You Like It: $580.00 (35 seats)
  Othello: $500.00 (40 seats)
Amount owed is $1,730.00
You earned 47 credits
```

— 1.2 COMMENTAIRES SUR LE PROGRAMME D'EXEMPLE

Que pensez-vous de la conception de ce programme ? La première chose que je dirais, c'est qu'il est acceptable tel qu'il est, un programme aussi court ne nécessitant pas de structure profonde pour être compréhensible. Rappelez-vous cependant mon propos précédent : mes exemples doivent être courts. Imaginez ce programme à une plus grande échelle, par exemple plusieurs centaines de lignes de code. Avec un programme aussi long, une fonction unique serait difficile à comprendre.

Mais dans la mesure où le programme fonctionne, tout commentaire sur sa structure ne relève-t-il pas simplement du jugement esthétique et d'une aversion pour le code mal écrit ? Après tout, le compilateur ne se soucie pas de savoir si le code est bien ou mal écrit. Mais quand je veux changer le système, il y a un humain qui est impliqué, et les êtres humains se soucient de ces détails. Un système mal conçu est difficile à modifier, car il est difficile de comprendre ce qu'il faut changer et la manière dont ces changements vont avoir un impact sur le code existant pour obtenir

le comportement souhaité. Et s'il est difficile de comprendre ce qu'il faut changer, il y a un risque certain de commettre des erreurs et d'introduire des bugs.

Dans ces conditions, si je suis confronté à la modification d'un programme comportant des centaines de lignes de code, je préfère qu'il soit structuré en un ensemble de fonctions et d'autres éléments qui me permettent de comprendre plus facilement ce que le programme fait. Si le programme manque de structure, il est généralement plus facile d'ajouter d'abord de la structure au programme, pour ensuite faire les changements nécessaires.



Lorsque vous devez ajouter une fonctionnalité à un programme, mais que le code n'est pas structuré convenablement, modifiez d'abord le programme pour que l'ajout de la fonctionnalité soit plus facile, puis ajoutez la fonctionnalité.

Dans cet exemple, j'ai quelques changements que les utilisateurs aimeraient faire. Tout d'abord, ils veulent un relevé de compte affiché en HTML. Réfléchissez à l'impact de ce changement. Je suis confronté à l'ajout d'instructions conditionnelles autour de chaque instruction qui ajoute une chaîne de caractères au résultat. Cela va ajouter beaucoup de complexité à la fonction. Dans cette situation, la plupart des gens préfèrent copier la méthode et la modifier pour produire du code HTML. Faire une copie peut sembler une solution pas trop coûteuse en temps de développement, mais cela pose toutes sortes de problèmes pour les futures versions. Toute modification apportée à la logique de facturation me forcera à mettre à jour les deux méthodes, et à m'assurer qu'elles sont mises à jour de manière cohérente. Si j'écris un programme qui ne changera plus jamais, ce genre de copier-coller est très bien, mais s'il s'agit d'un programme qui est censé évoluer dans le temps, alors la duplication est une menace.

Cela m'amène à une deuxième modification. Les acteurs cherchent à diversifier leur répertoire et veulent ajouter d'autres types de pièces comme le drame historique, la pastorale, la comédie pastorale, la pastorale historique, la tragédie historique, la pastorale tragico-comico-historique, ainsi que les pièces sans divisions et les poèmes sans limites⁴. Ils n'ont pas encore décidé exactement ce qu'ils veulent faire et le moment où ils le feront. Cette modification affectera à la fois la façon dont leurs pièces sont facturées et la façon dont les crédits de volume sont calculés. En tant que développeur expérimenté, je peux être sûr que quelle que soit leur intention, ils vont en changer à nouveau dans les six mois. Après tout, quand les demandes de fonctionnalité sont exprimées, elles sont rarement isolées, mais arrivent en masse.

Encore une fois, cette méthode de relevé de compte (appelée `statement`) est l'endroit où les modifications doivent être apportées pour gérer les changements des règles de classification et de facturation. Mais si je copie cette méthode en une nouvelle méthode appelée `htmlStatement`, il faudra que je m'assure que toutes les modifications sont

4. NdT : cette liste hétéroclite est extraite d'une réplique de Polonius dans la scène 2 de l'acte 2 d'Hamlet. https://fr.wikisource.org/wiki/Hamlet/Traduction_Hugo,_1865/Le_Second_Hamlet

cohérentes. En outre, comme la complexité des règles augmente, il va être plus difficile de déterminer où faire les changements et de les faire sans commettre d'erreur.

Permettez-moi de souligner que ce sont ces changements qui conduisent à la nécessité d'effectuer le refactoring. Si le code fonctionne et n'a jamais besoin d'évoluer, il est parfaitement correct de ne pas y toucher. Il serait bon de l'améliorer mais, à moins que quelqu'un n'ait besoin de le comprendre, il n'y a aucun mal à le laisser en l'état. Pourtant, dès que quelqu'un a besoin de comprendre comment ce code fonctionne, et qu'il a du mal à en suivre la logique, alors vous devez intervenir.

— 1.3 LA PREMIÈRE ÉTAPE DU REFACTORING

Chaque fois que je fais du refactoring, la première étape est toujours la même. J'ai besoin de m'assurer que j'ai un ensemble solide de tests pour cette section de code. Les tests sont essentiels parce que même si j'effectue de façon structurée le refactoring afin d'éviter la plupart des risques d'introduction des bugs, je reste un être humain et peux donc faire des erreurs. Plus un programme est long, plus il est probable que mes changements provoqueront par inadvertance des erreurs ; à l'ère du numérique, le logiciel incarne la fragilité.

Étant donné que la méthode `statement` renvoie une chaîne, je me contente de créer quelques factures, j'attribue à chaque facture quelques représentations de différents types de pièces, et je génère les chaînes des relevés de compte. Je fais ensuite une comparaison de chaînes entre la nouvelle chaîne et quelques chaînes de référence que j'ai vérifiées à la main. J'ai mis en place tous ces tests à l'aide d'une infrastructure de test afin de pouvoir les exécuter en appuyant sur une seule touche dans mon environnement de développement. Les tests ne prennent que quelques secondes à s'exécuter, et comme vous le constaterez, je les lance souvent.

Une partie importante des tests concerne la façon dont ils signalent leurs résultats. Ils sont soit verts, ce qui signifie que toutes les chaînes sont identiques aux chaînes de référence, soit rouges, et ils affichent alors une liste d'erreurs, c'est-à-dire les lignes qui sont différentes. Les tests sont donc automatisés et c'est vital car si ce n'était pas le cas, je passerais du temps à vérifier les valeurs manuellement, ce qui me ralentirait. Les frameworks de tests modernes fournissent toutes les fonctionnalités nécessaires pour écrire et exécuter des tests automatisés.



Avant de commencer le refactoring, assurez-vous d'avoir une suite solide de tests. Ces tests doivent être automatisés.

Quand je fais du refactoring, je m'appuie sur les tests. Je me les représente comme un détecteur de bugs qui me protège contre mes propres erreurs. En écrivant ce que je veux deux fois, dans le code et dans le test, je dois faire la même erreur dans les deux endroits pour tromper le détecteur. En vérifiant deux fois mon travail, je réduis

les risques d'erreur. Bien que la création des tests prenne du temps, je finis par en économiser beaucoup car je passe moins de temps au débogage. C'est une partie tellement importante du refactoring que je consacre un chapitre entier à ce sujet (chapitre 4).

— 1.4 DÉCOMPOSITION DE LA FONCTION STATEMENT

Lors du refactoring d'une fonction longue comme celle-ci, j'essaie mentalement d'identifier les points qui séparent les différentes parties du comportement global. La première partie qui saute aux yeux est le bloc de l'instruction switch qui est en gras.

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
    { style: "currency", currency: "USD",
      minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
case "tragedy":
      thisAmount = 40000;
  if (perf.audience > 30) {
    thisAmount += 1000 * (perf.audience - 30);
  }
  break;
case "comedy":
      thisAmount = 30000;
  if (perf.audience > 20) {
    thisAmount += 10000 + 500 * (perf.audience - 20);
  }
  thisAmount += 300 * perf.audience;
  break;
default:
  throw new Error(`unknown type: ${play.type}`);
}

    // ajoute des crédits de volume
    volumeCredits += Math.max(perf.audience - 30, 0);
    // ajoute un crédit par groupe de cinq spectateurs assistant à
    une comédie
  }
}
```

```

    if ("comedy" === play.type) volumeCredits += Math.floor(perf.
      audience / 5);

    // imprime la ligne de cette commande
    result += ` ${play.name}: ${format(thisAmount/100)} (${perf.
      audience} seats)\n`;
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}

```

Quand je regarde cette partie, j'en conclus qu'il s'agit du calcul de la facturation d'une représentation. Cette conclusion représente une intuition à propos du code, mais comme le dit Ward Cunningham, cette connaissance se trouve dans ma tête, qui est une forme de stockage notoirement volatile... J'ai donc besoin de rendre cette information persistante en la déplaçant de ma tête dans le code lui-même. De cette façon, si j'examine plus tard mon code, il me dira ce qu'il fait et je n'aurai pas à l'analyser à nouveau.

Pour intégrer cette connaissance dans le code, il faut transformer cette partie en fonction, en lui donnant un nom significatif basé sur ce qu'elle fait, par exemple, `amountFor(aPerformance)`. Quand je veux transformer un morceau de code en une fonction comme celle-ci, j'ai une procédure pour ce faire qui minimise le risque de me tromper. J'ai écrit cette procédure et, afin de m'y référer plus facilement, je l'ai appelée *Extraire fonction (120)*.

Tout d'abord, j'ai besoin de regarder dans le fragment de code toutes les variables qui ne seront plus dans la portée une fois que j'aurai extrait le code dans sa propre fonction. Dans cet exemple, il s'agit des trois variables: `perf`, `play`, et `thisAmount`. Les deux premières sont utilisées par le code extrait, mais elles ne sont pas modifiées, si bien que je peux les passer en tant que paramètres. Les variables qui sont modifiées ont besoin de plus d'attention. Ici, il n'y en a qu'une si bien que la fonction peut la renvoyer. Je peux aussi l'initialiser à l'intérieur du code extrait. On obtient alors le résultat suivant:

Fonction statement...

```

function amountFor(perf, play) {
  let thisAmount = 0;
  switch (play.type) {
    case "tragedy":
      thisAmount = 40000;
      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audience - 30);
      }
  }
}

```

```
    break;
  case "comedy":
    thisAmount = 30000;
    if (perf.audience > 20) {
      thisAmount += 10000 + 500 * (perf.audience - 20);
    }
    thisAmount += 300 * perf.audience;
    break;
  default:
    throw new Error(`unknown type: ${play.type}`);
}
return thisAmount;
}
```

Quand j'utilise un en-tête comme « *Fonction quelqueChose...* » en italique pour certains codes, cela signifie que le code suivant se trouve dans la portée de la fonction, du fichier ou de la classe nommée dans l'en-tête. Il y a habituellement d'autres codes dans cette portée que je ne montre pas, car je ne les évoque pas à ce moment-là.

Le code initial de `statement` appelle maintenant cette fonction pour attribuer une valeur à `thisAmount`:

Niveau supérieur...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
    { style: "currency", currency: "USD",
      minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = amountFor(perf, play);

    // ajoute des crédits de volume
    volumeCredits += Math.max(perf.audience - 30, 0);
    // ajoute un crédit par groupe de cinq spectateurs assistant à
    // une comédie
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.
      audience / 5);

    // imprime la ligne de cette commande
    result += ` ${play.name}: ${format(thisAmount/100)} (${perf.
      audience} seats)\n`;
    totalAmount += thisAmount;
  }
}
```

```
}  
result += `Amount owed is ${format(totalAmount/100)}\n`;  
result += `You earned ${volumeCredits} credits\n`;  
return result;
```

Une fois que j'ai fait ce changement, je compile immédiatement et je teste pour voir si j'ai endommagé quoi que ce soit. C'est une habitude importante de tester après chaque refactoring, aussi simple soit-il. Comme il est facile de commettre des erreurs, effectuer un test après chaque modification signifie que lorsque je fais une erreur, je n'ai qu'un petit changement à prendre en compte afin de repérer l'erreur, ce qui permet de la trouver et de la corriger beaucoup plus facilement. C'est l'essence même du processus de refactoring : on effectue de petits changements et on teste après chaque modification. Si j'essaie d'en faire trop à la fois et si je commets une erreur, cela va me contraindre à une séance de débogage délicat qui peut prendre un certain temps. En réalisant de petits changements, j'ai un feedback immédiat, ce qui est la clé pour éviter le désordre.



J'utilise ici le verbe « compiler » pour signifier qu'il faut faire tout ce qui est nécessaire pour rendre le code JavaScript exécutable. Étant donné que JavaScript est directement exécutable, cela peut signifier que vous n'avez rien à faire, mais dans d'autres cas, cela peut impliquer le déplacement du code vers un répertoire de sortie et/ou l'utilisation d'un processeur tel que Babel [babel].



Le refactoring modifie les programmes par petites étapes, si bien que quand vous faites une erreur, il est facile de trouver où se situe le bug.

Comme ce code est en JavaScript, je peux extraire `amountFor` dans une fonction imbriquée de `statement`. C'est utile car cela signifie que je n'ai pas à passer les données qui sont à l'intérieur de la portée de la fonction appelante à la fonction nouvellement extraite. Cela ne fait pas de différence dans ce cas, mais c'est un problème de moins à traiter.

Dans cet exemple, les tests n'ont pas signalé d'erreur, et ma prochaine étape consiste donc à enregistrer la modification sur mon système local de contrôle de version. J'utilise un système de contrôle de version, tel que git ou mercurial, qui me permet de faire des commits privés. Je fais un commit après chaque refactoring réussi, afin de pouvoir facilement revenir à un état de fonctionnement stable si jamais je commets une erreur ultérieurement. Je regroupe ensuite les modifications en un unique commit plus important avant de pousser les modifications sur un référentiel partagé.

Extraire fonction (120) est un refactoring courant qu'il faut automatiser. Si je programmais en Java, j'aurais instinctivement appuyé sur la séquence de touches de mon IDE pour effectuer ce refactoring. Au moment où j'écris ces lignes, il n'y a pas

de prise en charge robuste pour ce type de refactoring dans les outils JavaScript, si bien que je dois le faire manuellement. Ce n'est pas difficile, mais il faut être prudent avec ces variables dont la portée est locale.

Une fois que j'ai utilisé *Extraire fonction (120)*, je jette un coup d'œil sur ce que j'ai extrait pour voir s'il y a des choses rapides et faciles que je peux faire pour améliorer la lisibilité de la fonction extraite. La première chose que je fais est de renommer certaines des variables pour les rendre plus claires, notamment en changeant `thisAmount` en `result`.

Fonction statement...

```
function amountFor(perf, play) {
  let result = 0;
  switch (play.type) {
    case "tragedy":
      result = 40000;
      if (perf.audience > 30) {
        result += 1000 * (perf.audience - 30);
      }
      break;
    case "comedy":
      result = 30000;
      if (perf.audience > 20) {
        result += 10000 + 500 * (perf.audience - 20);
      }
      result += 300 * perf.audience;
      break;
    default:
      throw new Error(`unknown type: ${play.type}`);
  }
  return result;
}
```

C'est mon habitude de toujours nommer la valeur de retour d'une fonction « `result` ». De cette façon, je connais toujours son rôle. De nouveau, je compile, teste et commite. Puis je passe au premier argument.

Fonction statement...

```
function amountFor(aPerformance, play) {
  let result = 0;
  switch (play.type) {
    case "tragedy":
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
  }
}
```

```
        break;
    case "comedy":
        result = 30000;
        if (aPerformance.audience > 20) {
            result += 10000 + 500 * (aPerformance.audience - 20);
        }
        result += 300 * aPerformance.audience;
        break;
    default:
        throw new Error(`unknown type: ${play.type}`);
    }
    return result;
}
```

Encore une fois, cela est conforme à mon style de codage. Avec un langage typé dynamiquement tel que JavaScript, il est utile de garder la trace des types, si bien que le nom par défaut d'un paramètre inclut le nom de son type. J'utilise un article indéfini (*a*) à moins que le nom recèle des informations spécifiques sur son rôle. C'est Kent Beck [Beck SBPP] qui m'a appris cette convention de nommage et je continue à la trouver utile.



N'importe quel imbécile peut écrire du code qu'un ordinateur comprend.
Les bons programmeurs écrivent du code que les humains peuvent comprendre.

Est-ce que ce renommage en vaut la peine ? Absolument ! Un bon programme doit clairement communiquer ce qu'il fait, et le nom des variables est un élément essentiel d'un code limpide. Ne rechignez jamais à changer un nom pour améliorer la clarté du code. Avec de bons outils de recherche et de remplacement, il n'est en général pas difficile de changer des noms ; des tests et un typage statique dans un langage qui prend en charge cette fonctionnalité mettront en surbrillance toutes les occurrences que vous avez manquées. De plus, avec les outils de refactoring automatisé, il est facile de renommer les fonctions qui sont très utilisées.

L'élément suivant dont nous allons étudier le renommage est le paramètre `play` (la pièce de théâtre), mais je lui réserve un sort différent.

— 1.5 SUPPRESSION DE LA VARIABLE PLAY

Quand j'examine les paramètres de la fonction `amountFor`, je cherche leur origine. `aPerformance` provient de la variable de la boucle, si bien qu'elle change à chaque itération. En revanche, la variable `play` est calculée à partir de la représentation, et il n'y a donc nul besoin de la passer en tant que paramètre puisque je peux simplement recalculer sa valeur au sein de la fonction. Quand je décompose une longue fonction,

j'aime bien me débarrasser des variables comme `play`, parce que les variables temporaires créent beaucoup de noms ayant une portée locale qui compliquent les extractions. Le refactoring que je vais utiliser ici est *Remplacer variable temporaire par requête* (190).

Je commence par extraire le côté droit de l'assignation dans une fonction.

Fonction statement...

```
function playFor(aPerformance) {
  return plays[aPerformance.playID];
}
```

Niveau supérieur...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
    { style: "currency", currency: "USD",
      minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    const play = playFor(perf);
    let thisAmount = amountFor(perf, play);

    // ajoute des crédits de volume
    volumeCredits += Math.max(perf.audience - 30, 0);
    // ajoute un crédit par groupe de cinq spectateurs assistant à une comédie
    if ("comedy" === play.type) volumeCredits += Math.floor(perf.audience / 5);

    // imprime la ligne de cette commande
    result += ` ${play.name}: ${format(thisAmount/100)} (${perf.audience}
    seats)\n`;
    totalAmount += thisAmount;
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

Je compile, teste et commite, puis utilise *Incorporer variable* (137).

Niveau supérieur...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
```

```

        { style: "currency", currency: "USD",
          minimumFractionDigits: 2 }).format;
for (let perf of invoice.performances) {
  const play = playFor(perf);
  let thisAmount = amountFor(perf, playFor(perf));

  // ajoute des crédits de volume
  volumeCredits += Math.max(perf.audience - 30, 0);
  // ajoute un crédit par groupe de cinq spectateurs assistant à
  // une comédie
  if ("comedy" === playFor(perf).type) volumeCredits += Math.
    floor(perf.audience / 5);

  // imprime la ligne de cette commande
  result += ` ${playFor(perf).name}: ${format(thisAmount/100)}
    (${perf.audience} seats)\n`;
  totalAmount += thisAmount;
}
result += `Amount owed is ${format(totalAmount/100)}\n`;
result += `You earned ${volumeCredits} credits\n`;
return result;

```

Je compile, teste et commite. Avec cette variable incorporée (*inline*), je peux ensuite appliquer *Modifier déclaration de fonction (138)* à `amountFor` pour supprimer le paramètre `play`. Je le fais en deux étapes. Tout d'abord, j'utilise la nouvelle fonction à l'intérieur de `amountFor`.

Fonction statement...

```

function amountFor(aPerformance, play) {
  let result = 0;
  switch (playFor(aPerformance).type) {
    case "tragedy":
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
      break;
    case "comedy":
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
      result += 300 * aPerformance.audience;
      break;
    default:

```

```

        throw new Error(`unknown type: ${playFor(aPerformance).type}`);
    }
    return result;
}

```

Je compile, teste et commite, puis je supprime le paramètre.

Niveau supérieur...

```

function statement (invoice, plays) {
    let totalAmount = 0;
    let volumeCredits = 0;
    let result = `Statement for ${invoice.customer}\n`;
    const format = new Intl.NumberFormat("en-US",
        { style: "currency", currency: "USD",
          minimumFractionDigits: 2 }).format;
    for (let perf of invoice.performances) {
        let thisAmount = amountFor(perf, playFor(perf));

        // ajoute des crédits de volume
        volumeCredits += Math.max(perf.audience - 30, 0);
        // ajoute un crédit par groupe de cinq spectateurs assistant à
        // une comédie
        if ("comedy" === playFor(perf).type) volumeCredits += Math.
            floor(perf.audience / 5);

        // imprime la ligne de cette commande
        result += ` ${playFor(perf).name}: ${format(thisAmount/100)}
            (${perf.audience} seats)\n`;
        totalAmount += thisAmount;
    }
    result += `Amount owed is ${format(totalAmount/100)}\n`;
    result += `You earned ${volumeCredits} credits\n`;
    return result;
}

```

Fonction statement...

```

function amountFor(aPerformance, play) {
    let result = 0;
    switch (playFor(aPerformance).type) {
        case "tragedy":
            result = 40000;
            if (aPerformance.audience > 30) {
                result += 1000 * (aPerformance.audience - 30);
            }
            break;
        case "comedy":

```

```

    result = 30000;
    if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
default:
    throw new Error(`unknown type: ${playFor(aPerformance).type}`);
}
return result;
}

```

Et je compile, teste et commite à nouveau.

Ce type de refactoring inquiète certains programmeurs. Auparavant, le code pour rechercher la pièce était exécuté une fois à chaque itération de la boucle; maintenant, il est exécuté trois fois. J'évoquerai plus tard la relation entre le refactoring et la performance, mais pour le moment j'observerai juste que ce changement est peu susceptible d'affecter de manière significative les performances, et même si c'était le cas, il est beaucoup plus facile d'améliorer les performances d'un code dont les bases sont saines.

Le grand avantage de la suppression des variables locales est qu'il est beaucoup plus facile de faire des extractions, car il faut gérer moins de portée locale. En effet, je vais en général supprimer les variables locales avant de faire des extractions.

Maintenant que j'en ai fini avec les arguments de `amountFor`, je reviens à l'endroit où la fonction est appelée. Elle est utilisée pour définir une variable temporaire qui, une nouvelle fois, n'est pas mise à jour, si bien que j'applique la technique *Incorporer variable* (137).

Niveau supérieur...

```

function statement (invoice, plays) {
    let totalAmount = 0;
    let volumeCredits = 0;
    let result = `Statement for ${invoice.customer}\n`;
    const format = new Intl.NumberFormat("en-US",
        { style: "currency", currency: "USD",
          minimumFractionDigits: 2 }).format;
    for (let perf of invoice.performances) {

        // ajoute des crédits de volume
        volumeCredits += Math.max(perf.audience - 30, 0);
        // ajoute un crédit par groupe de cinq spectateurs assistant à
        // une comédie
        if ("comedy" === playFor(perf).type) volumeCredits += Math.
            floor(perf.audience / 5);
    }
}

```

```

// imprime la ligne de cette commande
result += ` ${playFor(perf).name}:
${format(amountFor(perf)/100)} (${perf.audience} seats)\n`;
totalAmount += amountFor(perf);
}
result += `Amount owed is ${format(totalAmount/100)}\n`;
result += `You earned ${volumeCredits} credits\n`;
return result;

```

— 1.6 EXTRACTION DE CRÉDITS DE VOLUME

Voici l'état actuel du corps de la fonction `statement` :

Niveau supérieur...

```

function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
    { style: "currency", currency: "USD",
      minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {

    // ajoute des crédits de volume
    volumeCredits += Math.max(perf.audience - 30, 0);
    // ajoute un crédit par groupe de cinq spectateurs assistant à
    // une comédie
    if ("comedy" === playFor(perf).type) volumeCredits += Math.
      floor(perf.audience / 5);

    // imprime la ligne de cette commande
    result += ` ${playFor(perf).name}:
    ${format(amountFor(perf)/100)} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}

```

À présent, j'obtiens le bénéfice de la suppression de la variable `play` car cela facilite l'extraction du calcul des crédits de volume en supprimant l'une des variables ayant une portée locale.

Je dois encore gérer les deux autres variables. Encore une fois, `perf` est facile à traiter, mais `volumeCredits` est un peu plus délicat car il s'agit d'un accumulateur

mis à jour à chaque itération de la boucle. La meilleure option est donc d'initialiser une variable *shadow*⁵ de celle-ci à l'intérieur de la fonction extraite et de la renvoyer.

Fonction statement...

```
function volumeCreditsFor(perf) {
  let volumeCredits = 0;
  volumeCredits += Math.max(perf.audience - 30, 0);
  if ("comedy" === playFor(perf).type) volumeCredits += Math.
    floor(perf.audience / 5);
  return volumeCredits;
}
```

Niveau supérieur...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
    { style: "currency", currency: "USD",
      minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);

    // imprime la ligne de cette commande
    result += ` ${playFor(perf).name}:
    ${format(amountFor(perf)/100)} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

Je retire le commentaire inutile (qui, en l'occurrence, est carrément trompeur).

Je compile, teste et commite, puis je renomme les variables à l'intérieur de la nouvelle fonction.

Fonction statement...

```
function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);
  if ("comedy" === playFor(aPerformance).type) result += Math.
    floor(aPerformance.audience / 5);
  return result;
}
```

5. NdT: dans certains langages, quand une variable déclarée dans une portée (bloc décisionnel, méthode, ou classe) a le même nom qu'une variable déclarée dans une portée externe, on parle alors de variable *shadow*.

Je vous l'ai montré en une seule étape, mais comme je l'ai fait auparavant, j'ai renommé une variable à la fois, avec une compilation, un test et un commit après chaque changement de nom.

— 1.7 SUPPRESSION DE LA VARIABLE FORMAT

Examinons à nouveau la méthode principale `statement` :

Niveau supérieur...

```
function statement (invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
    { style: "currency", currency: "USD",
      minimumFractionDigits: 2 }).format;
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);

    // imprime la ligne de cette commande
    result += ` ${playFor(perf).name}:
    ${format(amountFor(perf)/100)} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

Comme je l'ai déjà suggéré, les variables temporaires peuvent être un problème. Elles ne sont utiles qu'au sein de leur propre routine, et par conséquent elles encouragent les routines longues et complexes. Ma prochaine étape consiste donc à remplacer certaines d'entre elles, la plus facile étant la variable `format`. C'est un cas d'assignation d'une fonction à une variable temporaire, que je préfère remplacer par une fonction déclarée.

Fonction `statement`...

```
function format(aNumber) {
  return new Intl.NumberFormat("en-US",
    { style: "currency", currency: "USD",
      minimumFractionDigits: 2
    }).format(aNumber);
}
```