Référence

CODER proprement

Robert C. Martin

Réseaux et télécom

Programmation

Génie logiciel

Sécurité

Système d'exploitation

PEARSON

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.

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00
www.pearson.fr

Titre original :
Clean Code: a handbook of agile software craftsmanship

Traduit de l'américain par Hervé Soulard

Mise en pages : Hervé Soulard

Relecture technique : Éric Hébert

ISBN : 978-2-7440-4104-4

ISBN original : 978-0-13-235088-2

Table des matières

Préface

Les pastilles Ga-Jol sont parmi les sucreries préférées des Danois. Celles à la réglisse forte font un parfait pendant à notre climat humide et souvent froid. Le charme de ces pastilles réside notamment dans les dictons sages ou spirituels imprimés sur le rabat de chaque boîte. Ce matin, j'ai acheté un paquet de ces friandises sur lequel il était inscrit cet ancien adage danois :

> *Ærlighed i små ting er ikke nogen lille ting.*

"L'honnêteté dans les petites choses n'est pas une petite chose." Il présageait tout à fait ce que je souhaitais exprimer ici. Les petites choses ont une grande importance. Ce livre traite de sujets modestes dont la valeur est très loin d'être insignifiante.

Dieu est dans les détails, a dit l'architecte Ludwig Mies van der Rohe. Cette déclaration rappelle des arguments contemporains sur le rôle de l'architecture dans le développement de logiciels, plus particulièrement dans le monde agile. Avec Bob (Robert C. Martin), nous avons parfois conversé de manière passionnée sur ce sujet. Mies van der Rohe était attentif à l'utilité et aux aspects intemporels de ce qui sous-tend une bonne architecture. Néanmoins, il a choisi personnellement chaque poignée des portes des maisons qu'il a conçues. Pourquoi ? Tout simplement parce que les détails comptent.

Lors de nos "débats" permanents sur le développement piloté par les tests (TDD, *Test Driven Development*), nous avons découvert, Bob et moi-même, que nous étions d'accord sur le fait que l'architecture logicielle avait une place importante dans le développement, même si notre vision personnelle sur sa signification réelle pouvait être différente. Quoi qu'il en soit, ce pinaillage est relativement peu important car nous convenons que les professionnels responsables passent du temps à réfléchir sur l'objectif d'un projet et à le planifier. La notion de conception pilotée uniquement par les tests et le code, telle qu'elle était envisagée à la fin des années 1990, est désormais obsolète. L'attention portée aux détails est aujourd'hui une preuve de professionnalisme plus que n'importe quelle autre grande vision. Premièrement, c'est par leur participation aux petits projets que les professionnels acquièrent la compétence et la confiance nécessaires aux grands projets. Deuxièmement, les petits riens d'une construction négligée, comme une porte qui ferme mal ou un carreau légèrement ébréché, voire le bureau désordonné, annulent totalement le charme de l'ensemble. C'est toute l'idée du code propre.

L'architecture n'est qu'une métaphore pour le développement de logiciels, plus particulièrement en ce qui concerne la livraison du produit initial, comme un architecte qui livre un bâtiment impeccable. Aujourd'hui, avec Scrum et les méthodes agiles, l'objectif recherché est la mise sur le marché rapide d'un produit. Nous voulons que l'usine tourne à plein régime pour produire du logiciel. Voici les usines humaines : réfléchir, en pensant aux programmeurs qui travaillent à partir d'un produit existant ou d'un scénario utilisateur pour créer un produit. La métaphore de la fabrication apparaît encore plus fortement dans une telle approche. Les questions de production dans les usines japonaises, sur une ligne d'assemblage, ont inspiré une bonne part de Scrum.

Dans l'industrie automobile, le gros du travail réside non pas dans la fabrication, mais dans la maintenance, ou plutôt comment faire pour l'éviter. Dans le monde du logiciel, au moins 80 % de notre travail est bizarrement appelé "maintenance" : une opération de réparation. Au lieu d'adopter l'approche occidentale typique qui consiste à produire un bon logiciel, nous devons réfléchir comme des réparateurs ou des mécaniciens automobiles. Quel est l'avis du management japonais à ce sujet ?

En 1951, une approche qualité nommée maintenance productive totale (TPM, *Total Productive Maintenance*) est arrivée au Japon. Elle se focalise sur la maintenance, non sur la production. Elle s'appuie principalement sur cinq principes appelés les 5S. Il s'agit d'un ensemble de disciplines, le terme "discipline" n'étant pas choisi au hasard. Ces principes constituent en réalité les fondements du *Lean*, un autre mot à la mode sur la scène occidentale et de plus en plus présent dans le monde du logiciel, et ils ne sont pas facultatifs. Comme le relate l'Oncle Bob, les bonnes pratiques logiciels requièrent de telles disciplines : concentration, présence d'esprit et réflexion. Il ne s'agit pas toujours simplement de faire, de pousser les outils de fabrication à produire à la vitesse optimale. La philosophie des 5S comprend les concepts suivants :

- *Seiri*, ou organisation (pensez à "s'organiser" en français). Savoir où se trouvent les choses, par exemple en choisissant des noms appropriés, est essentiel. Si vous pensez que le nommage des identifiants n'est pas important, lisez les chapitres suivants.
- *Seiton*, ou ordre (pensez à "situer" en français). Vous connaissez certainement le dicton *une place pour chaque chose et chaque chose à sa place*. Un morceau de code doit se trouver là où l'on s'attend à le trouver. Si ce n'est pas le cas, un remaniement est nécessaire pour le remettre à sa place.
- *Seiso*, ou nettoyage (pensez à "scintiller" en français). L'espace de travail doit être dégagé de tout fil pendouillant, de graisse, de chutes et de déchets. Que pensent les auteurs de ce livre de la pollution du code par des commentaires et des lignes de

code mises en commentaires qui retracent l'historique ou prédisent le futur ? Il faut supprimer tout cela.

- *Seiketsu*, ou propre (pensez à "standardiser" en français). L'équipe est d'accord sur la manière de conserver un espace de travail propre. Pensez-vous que ce livre traite du style de codage et de pratiques cohérentes au sein de l'équipe ? D'où proviennent ces standards ? Poursuivez votre lecture.
- *Shutsuke*, ou éducation (pensez à "suivi" en français). Autrement dit, il faut s'efforcer de suivre les pratiques et de réfléchir au travail des autres, tout en étant prêt à évoluer.

Si vous relevez le défi – oui, le défi – de lire et d'appliquer les conseils donnés dans cet ouvrage, vous finirez par comprendre et apprécier le dernier point. Les auteurs nous guident vers un professionnalisme responsable, dans un métier où le cycle de vie des produits compte. Lorsque la maintenance des automobiles et des autres machines se fait sous la TPM, la maintenance de niveau dépannage – l'attente de l'arrivée des bogues – constitue l'exception. Au lieu de dépanner, nous prenons les devants : les machines sont inspectées quotidiennement et les éléments usagés sont remplacés avant qu'ils ne cassent ; autrement dit, nous effectuons l'équivalent de la vidange des 10 000 km afin d'anticiper l'usure. Dans le code, le remaniement doit se faire de manière implacable. Il est toujours possible d'apporter une amélioration, comme a innové la TPM il y a une cinquantaine d'années : construire dès le départ des machines dont la maintenance est plus facile. Rendre le code lisible est aussi important que le rendre exécutable. La pratique ultime, ajoutée dans la TPM vers 1960, consiste à renouveler totalement le parc machine ou à remplacer les plus anciennes. Comme le conseille Fred Brooks, nous devons reprendre à zéro des parties importantes du logiciel tous les sept ans afin de retirer tous les éléments obsolètes qui traînent. Il faut sans doute revoir les constantes de temps de Brooks et remplacer les années par des semaines, des jours ou des heures. C'est là que résident les détails.

Les détails renferment une grande puissance, encore qu'il y ait quelque chose d'humble et de profond dans cette vision de la vie, comme nous pourrions, de manière stéréotypée, le penser d'une approche qui revendique des racines japonaises. Mais il ne s'agit pas seulement d'une vision orientale de la vie ; la sagesse occidentale est elle aussi pleine de réprimandes. La citation *Seiton* précédente a été reprise par un ministre de l'Ohio qui a littéralement vu la régularité "comme un remède à chaque degré du mal". *Quid* de *Seiso* ? *La propreté est proche de la sainteté*. Une maison a beau être belle, un bureau en désordre lui retire sa splendeur. Qu'en est-il de *Shutsuke* sur ces petites questions ? *Qui est fidèle en très peu de chose est fidèle aussi en beaucoup*. Pourquoi ne pas s'empresser de remanier le code au moment responsable, en renforçant sa position pour les prochaines "grandes" décisions, au lieu de remettre cette intervention à plus

tard ? *Un point à temps en vaut cent. Le monde appartient à celui qui se lève tôt. Ne pas remettre au lendemain ce qu'on peut faire le jour même*. (C'était là le sens original de la phrase "le dernier moment responsable" dans Lean, jusqu'à ce qu'elle tombe dans les mains des consultants logiciels.) Pourquoi ne pas calibrer la place des petits efforts individuels dans un grand tout ? *Petit poisson deviendra grand*. Ou pourquoi ne pas intégrer un petit travail préventif dans la vie de tous les jours ? *Mieux vaut prévenir que guérir. Une pomme chaque matin éloigne le médecin*. Un code propre honore les racines profondes de la sagesse sous notre culture plus vaste, ou notre culture telle qu'elle a pu être, ou devait être, et peut être en prenant soin des détails.

Dans la littérature sur l'architecture, nous trouvons également des adages qui reviennent sur ces détails supposés. Nous retrouvons le concept *Seiri* dans les poignées de portes de Mies van der Rohe. Il s'agit d'être attentif à chaque nom de variable. Vous devez nommer une variable avec la même attention que vous portez au choix du nom de votre premier-né.

Comme le savent les propriétaires de maisons, l'entretien et les finitions n'ont pas de fin. L'architecte Christopher Alexander, le père des motifs et des *Pattern Languages*, voit chaque acte de conception lui-même comme un petit acte local de réparation. Il voit également l'art des structures fines comme le seul horizon de l'architecte ; les formes plus grandes peuvent être confiées aux plans et leur application aux habitants. La conception a non seulement lieu lors de l'ajout d'une nouvelle pièce à une maison, mais également lorsque nous changeons le papier peint, remplaçons la moquette usée ou modernisons l'évier de la cuisine. La plupart des arts reprennent des opinions analogues. Nous avons recherché d'autres personnes qui placent la maison de Dieu dans les détails. Nous nous somme retrouvés en bonne compagnie avec l'auteur français du XIX^{e} siècle Gustave Flaubert. Le poète Paul Valéry nous informe qu'un poème n'est jamais terminé et qu'il nécessite un travail perpétuel ; s'il n'est plus révisé, c'est qu'il est abandonné. Cette préoccupation du détail est commune à tous ceux qui recherchent l'excellence. Il n'y a donc sans doute rien de nouveau ici, mais en lisant ce livre vous serez mis au défi de reprendre une bonne discipline que vous aviez abandonnée au profit de la passivité ou d'un désir de spontanéité et pour juste "répondre au changement".

Malheureusement, de telles questions ne sont habituellement pas vues comme des fondements de l'art de la programmation. Nous abandonnons notre code très tôt, non pas qu'il soit terminé, mais parce que notre système de valeur se focalise plus sur les résultats apparents que sur la substance de ce que nous livrons. Ce manque d'assiduité finit par être coûteux : *un assassin revient toujours sur les lieux du crime*. La recherche, qu'elle soit industrielle ou académique, ne s'abaisse pas à garder un code propre. Lors de mes débuts chez Bell Labs Software Production Research (notez le mot Production dans le nom de la société !), nous avions des estimations qui suggéraient qu'une inden-

tation cohérente du code constituait l'un des indicateurs statistiquement significatifs d'une faible densité de bogues. Nous, nous voulions que l'architecture, le langage de programmation ou toute autre notion de haut niveau représentent la qualité. En tant que personnes dont le professionnalisme supposé était dû à la maîtrise d'outils et de méthodes de conception nobles, nous nous sommes sentis insultés par la valeur que ces machines industrielles, les codeurs, ajoutaient par la simple application cohérente d'un style d'indentation. Pour citer mon propre ouvrage écrit il y a dix-sept ans, c'est par un tel style que l'on différencie l'excellence de la simple compétence. Les Japonais comprennent la valeur essentielle de ce travail quotidien et, plus encore, des systèmes de développement qui sont redevables aux actions quotidiennes simples des travailleurs. La qualité est le résultat d'un million d'actes d'attention désintéressée, pas seulement d'une formidable méthode tombée du ciel. Que ces actes soient simples ne signifie pas qu'ils soient simplistes, et en aucun cas faciles. Ils n'en sont pas moins le tissu de la grandeur et, tout autant, de la beauté dans toute tentative humaine. Les ignorer, c'est alors ne pas encore être totalement humain.

Bien entendu, je suis toujours partisan d'une réflexion de plus grande ampleur, et particulièrement de la valeur des approches architecturales ancrées dans une profonde connaissance du domaine et des possibilités du logiciel. Il ne s'agit pas du propos de ce livre, ou, tout au moins, pas de celui affiché. Le message de cet ouvrage est plus subtil et sa profondeur ne doit pas être sous-estimée. Il s'accorde parfaitement à l'adage actuel des personnes orientées code, comme Peter Sommerlad, Kevlin Henney et Giovanni Asproni. "Le code représente la conception" et "du code simple" sont leurs mantras. S'il ne faut pas oublier que l'interface est le programme et que ses structures ont un rapport étroit avec notre structure de programme, il est essentiel de maintenir que la conception vit dans le code. Par ailleurs, alors que, dans la métaphore de la fabrication, le remaniement conduit à des coûts, le remaniement de la conception produit une valeur. Nous devons voir notre code comme la belle articulation de nobles efforts de conception – la conception en tant que processus, non comme une fin statique. C'est dans le code que les métriques architecturales de couplage et de cohésion s'évaluent. Si vous écoutez Larry Constantine décrire le couplage et la cohésion, vous l'entendrez parler de code, non de concepts très abstraits que l'on peut rencontrer en UML. Dans son article "Abstraction Descant", Richard Gabriel nous apprend que l'abstraction est le mal. Le code lutte contre le mal, et un code propre est sans doute divin.

Pour revenir à mon petit paquet de Ga-Jol, je pense qu'il est important de noter que la sagesse danoise conseille non seulement de faire attention aux petites choses, mais également d'être sincère dans les petites choses. Cela signifie être sincère envers le code, envers nos collègues quant à l'état de notre code et, plus important, envers nous-mêmes quant à notre code. Avons-nous fait de notre mieux pour "laisser l'endroit plus propre que nous l'avons trouvé" ? Avons-nous remanié notre code avant de l'archiver ?

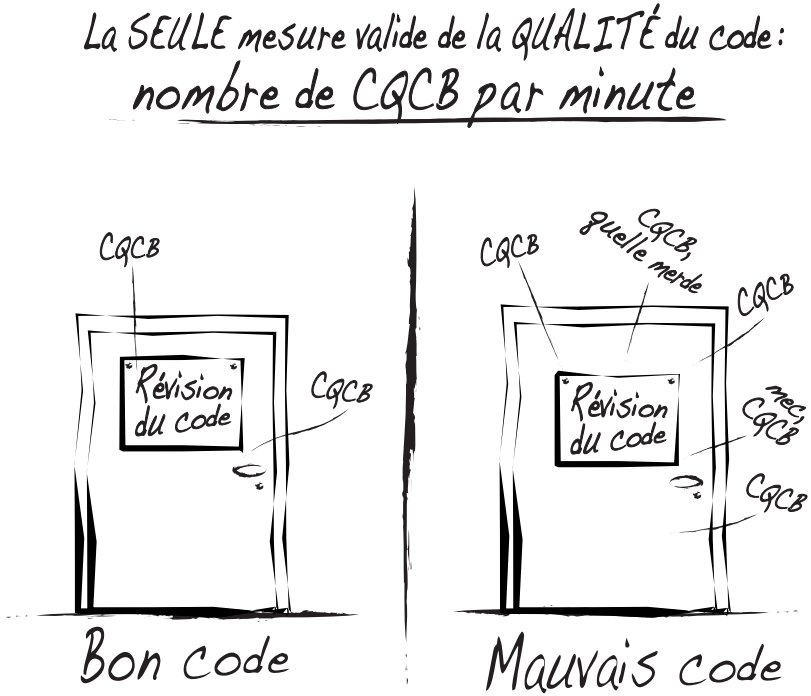
Il s'agit non pas de points secondaires, mais de questions au centre des valeurs agiles. Scrum recommande de placer le remaniement au cœur du concept "Terminé". Ni l'architecture ni un code propre n'insistent sur la perfection, uniquement sur l'honnêteté et faire de son mieux. *L'erreur est humaine, le pardon, divin*. Dans Scrum, tout est visible. Nous affichons notre linge sale. Nous sommes sincères quant à l'état de notre code, car le code n'est jamais parfait. Nous devenons plus humains, plus dignes du divin et plus proches de cette grandeur dans les détails.

Dans notre métier, nous avons désespérément besoin de toute l'aide que nous pouvons obtenir. Si le sol propre d'un magasin diminue le nombre d'accidents et si une bonne organisation des outils augmente la productivité, alors, j'y adhère. Ce livre est la meilleure application pragmatique des principes de Lean aux logiciels que j'ai jamais lue. Je n'attendais pas moins de ce petit groupe d'individus réfléchis qui s'efforcent ensemble depuis des années non seulement de s'améliorer, mais également de faire cadeau de leurs connaissances à l'industrie dans des travaux tels que l'ouvrage qui se trouve entre vos mains. Le monde est laissé dans un état un peu meilleur qu'avant que je ne reçoive le manuscrit de l'Oncle Bob.

Ces nobles idées étant étalées, je peux à présent aller ranger mon bureau.

James O. Coplien
Mørdrup, Danemark

Introduction



CQCB : C'est quoi ce bordel

Reproduction et adaptation avec l'aimable autorisation de Thom Holwerda (**http://www.osnews.com/story/19266/WTFs_m**). © 2008 Focus Shift

Quelle porte ouvre sur votre code ? Quelle porte ouvre sur votre équipe ou votre entreprise ? Pourquoi êtes-vous dans cette pièce ? S'agit-il simplement d'une révision normale du code ou avez-vous découvert tout un ensemble de problèmes désastreux peu après le lancement ? Procédez-vous à un débogage en urgence, en plongeant dans du code que vous pensiez opérationnel ? Les clients partent-ils en masse et les managers vous surveillent-ils ? Comment pouvez-vous être sûr que vous serez derrière la *bonne* porte lorsque les choses iront mal ? Les réponses tiennent en une seule : l'*art du métier.*

La maîtrise de l'art du métier englobe deux parties : connaissances et travail. Vous devez acquérir les connaissances concernant les principes, les motifs, les pratiques et les heuristiques connus de l'artisan, et vous devez également polir ces connaissances avec vos doigts, vos yeux et vos tripes, en travaillant dur et en pratiquant.

Je peux vous enseigner la physique qui permet de rouler à vélo. Il est vrai que les mathématiques classiques sont relativement simples. Gravité, frottements, moment angulaire, centre d'inertie, etc. peuvent être expliqués en moins d'une page d'équations. Grâce à cette formule, je peux prouver qu'il vous est possible de rouler à vélo et vous apporter toutes les connaissances dont vous avez besoin pour ce faire. Néanmoins, vous tomberez immanquablement la première fois que vous grimperez sur la bicyclette.

Écrire du code n'est pas si différent. Nous pourrions rédiger tous les bons principes d'écriture d'un code propre et vous faire confiance pour réaliser le travail (autrement dit, vous laisser tomber lorsque vous monterez sur le vélo), mais quelle sorte de professeurs serions-nous alors et quelle sorte d'étudiant seriez-vous ?

Ce n'est pas l'orientation que nous donnons à ce livre.

Apprendre à écrire du code propre est un *travail difficile*. Cela ne se limite pas à connaître des principes et des motifs. Vous devez *transpirer.* Vous devez pratiquer et constater vos échecs. Vous devez regarder d'autres personnes pratiquer et échouer. Vous devez les voir hésiter et revenir sur leurs pas. Vous devez les voir se tourmenter sur des décisions et payer le prix de leurs mauvais choix.

Vous devez être prêt à travailler dur au cours de la lecture de cet ouvrage. Il ne s'agit pas d'un livre que vous pourrez lire dans un avion et terminer avant d'atterrir. Il vous imposera de *travailler, dur.* Qu'est-ce qui vous attend ? Vous allez lire du code, beaucoup de code. Vous devrez réfléchir aux points positifs et aux points négatifs de ce code. Il vous sera demandé de nous suivre pendant que nous découpons des modules, pour ensuite les réunir à nouveau. Cela demandera du temps et des efforts, mais nous pensons que cela en vaut la peine.

Nous avons décomposé ce livre en trois parties. Les premiers chapitres décrivent les principes, les motifs et les pratiques employés dans l'écriture d'un code propre. Ils contiennent beaucoup de code et ne seront pas faciles à lire. Ils vous prépareront à la deuxième partie. Si vous refermez le livre après avoir lu la première partie, nous vous souhaitons bonne chance !

C'est dans la deuxième partie que se trouve le travail le plus difficile. Elle est constituée de plusieurs études de cas dont la complexité va croissant. Chaque étude de cas est un exercice de nettoyage : une base de code qui présente certains problèmes doit être transformée en une version soulagée de quelques problèmes. Dans cette section, le niveau de

détail est élevé. Vous devrez aller et venir entre le texte et les listings de code. Vous devrez analyser et comprendre le code sur lequel nous travaillons et suivre notre raisonnement lors de chaque modification effectuée. Trouvez du temps, car *cela demandera plusieurs jours*.

La troisième partie sera votre récompense. Son unique chapitre contient une liste d'heuristiques et d'indicateurs collectés lors de la création des études de cas. Pendant l'examen et le nettoyage du code dans les études de cas, nous avons documenté chaque raison de nos actions en tant qu'heuristique ou indicateurs. Nous avons essayé de comprendre nos propres réactions envers le code que nous lisions et modifiions. Nous avons fait tout notre possible pour consigner l'origine de nos sensations et de nos actes. Le résultat est une base de connaissances qui décrit notre manière de penser pendant que nous écrivons, lisons et nettoyons du code.

Cette base de connaissance restera d'un intérêt limité si vous ne faites pas l'effort d'examiner attentivement les études de cas de la deuxième partie de cet ouvrage. Dans ces études de cas, nous avons annoté consciencieusement chaque modification apportée, en ajoutant également des références vers les heuristiques. Ces références apparaissent entre crochets, par exemple [H22]. Cela vous permet de savoir dans quel *contexte* ces heuristiques ont été appliquées et écrites ! C'est non pas tant les heuristiques en soi qui ont de la valeur, mais *le lien entre ces heuristiques et chaque décision que nous avons prise pendant le nettoyage du code*.

Pour faciliter l'emploi de ces liens, nous avons ajouté à la fin du livre des références croisées qui indiquent le numéro de page de chaque référence d'heuristique. Vous pouvez les utiliser pour rechercher chaque contexte d'application d'une heuristique.

Si vous lisez la première et la troisième partie, en sautant les études de cas, vous n'aurez parcouru qu'un livre de plus sur la bonne écriture des logiciels. En revanche, si vous prenez le temps de travailler sur les études de cas, en suivant chaque petite étape, chaque décision, autrement dit en vous mettant à notre place et en vous forçant à réfléchir à notre manière, alors, vous comprendrez beaucoup mieux ces principes, motifs, pratiques et heuristiques. Ils ne seront plus alors des connaissances de "confort". Ils seront ancrés dans vos tripes, vos doigts et votre cœur. Ils feront partie de vous, de la même manière qu'un vélo devient une extension de votre volonté lorsque vous savez comment le conduire.

Remerciements

Illustrations

Je voudrais remercier mes deux artistes, Jeniffer Kohnke et Angela Brooks. Jeniffer est l'auteur des illustrations créatives et sensationnelles qui débutent chaque chapitre, ainsi que des portraits de Kent Beck, Ward Cunningham, Bjarne Stroustrup, Ron Jeffries, Grady Booch, Dave Thomas, Michael Feathers et moi-même.

Angela s'est chargée des illustrations qui agrémentent le contenu de chaque chapitre. Elle a déjà réalisé de nombreuses figures pour moi, notamment pour l'ouvrage *Agile Software Development: Principles, Patterns, and Practices*. Elle est également ma fille aînée, qui me comble de bonheur.

Sur la couverture

L'image de la couverture représente M104, également appelée galaxie du sombrero. M104 est située dans la constellation de la Vierge et se trouve à moins de trente millions d'années-lumière de la Terre. Son noyau est constitué d'un trou noir supermassif dont la masse équivaut à un milliard de fois celle du Soleil.

Cette image vous rappelle-t-elle l'explosion de la lune minière klingonne *Praxis* ? Je vous recommande vivement la scène de *Star Trek VI* qui montre un anneau équatorial de débris qui naît de cette explosion. Depuis cette scène, l'anneau équatorial est un artefact classique des explosions dans les films de science-fiction. Il a même été ajouté à l'explosion d'Alderaan dans les derniers épisodes de *La Guerre des étoiles*.

Quelle est l'origine de cet anneau qui s'est formé autour de M104 ? Pourquoi possède-t-elle un bulbe central si important et un noyau aussi brillant et petit ? J'ai l'impression que le trou noir central a un jour perdu son sang-froid et a créé un trou de trente mille années-lumière au milieu de la galaxie. Un grand malheur est arrivé aux civilisations qui pouvaient se trouver sur le chemin de ce bouleversement cosmique.

Les trous noirs supermassifs avalent des quantités d'étoiles au petit déjeuner, en convertissant une part assez importante de leur masse en énergie. $E = MC^2$ a suffisamment d'influence, mais lorsque M est une masse stellaire, attention ! Combien d'étoiles se sont précipitées tête baissée dans ce piège avant que le monstre ne soit rassasié ? Les dimensions du vide central seraient-elles une indication ?

L'image de la couverture combine la célèbre photographie en lumière visible prise par Hubble (en haut) et l'image infrarouge récente du télescope spatial Spitzer (en bas). La nature annulaire de la galaxie est révélée par l'image infrarouge. Dans la lumière visible, nous voyons juste le bord frontal de l'anneau, dont le restant est masqué par le bulbe central.

Image : © télescope spatial Spitzer

En revanche, dans les infrarouges, les particules chaudes de l'anneau apparaissent au travers du bulbe central. La combinaison des deux images produit une toute nouvelle vue de la galaxie et indique qu'elle faisait, il y a très longtemps, l'objet d'une activité intense.

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.

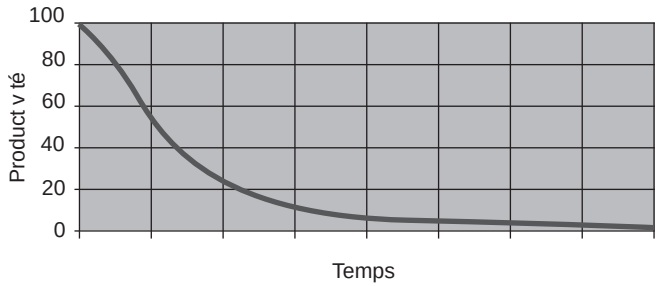
Plus le désordre augmente, plus la productivité de l'équipe décroît, de manière asymptotique en s'approchant de zéro. Lorsque la productivité diminue, la direction fait la seule chose qu'elle sache faire : affecter un plus grand nombre de personnes au projet en espérant augmenter la productivité. Mais 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 !"

2

Noms significatifs

Par Tim Ottinger

Les noms abondent dans les logiciels. Nous nommons les variables, les fonctions, les arguments, les classes et les paquetages. Nous nommons les fichiers sources et les répertoires qui les contiennent. Nous nommons les fichiers jar, les fichiers war et les fichiers ear. Puisque les noms sont omniprésents, il est préférable de bien les choisir. Les sections suivantes établissent des règles simples pour la création de bons noms.

Choisir des noms révélateurs des intentions

Il est facile de dire que les noms doivent révéler les intentions. Cependant, nous voulons que vous compreniez bien que ce point est essentiel. Choisir de bons noms prend du temps, mais permet d'en gagner plus encore. Vous devez donc faire attention à vos noms et les changer dès que vous en trouvez de meilleurs. Quiconque lit votre code, y compris vous, vous en sera reconnaissant.

Le nom d'une variable, d'une fonction ou d'une classe doit répondre à certaines grandes questions : la raison de son existence, son rôle et son utilisation. Si un nom exige un commentaire, c'est qu'il ne répond pas à ces questions.

```
int d; // Temps écoulé en jours.[1]
```

Le nom `d` ne révèle rien. Il n'évoque pas une durée écoulée, pas même des jours (*days*). Nous devons choisir un nom qui précise ce qui est mesuré et l'unité de mesure :

```
int elapsedTimeInDays;
int daysSinceCreation;
int daysSinceModification;
int fileAgeInDays;
```

En choisissant des noms révélateurs des intentions, il sera certainement beaucoup plus facile de comprendre et de modifier le code. Pouvez-vous comprendre l'objectif du code suivant ?

```
public List<int[]> getThem() {
  List<int[]> list1 = new ArrayList<int[]>();

  for (int[] x : theList)
    if (x[0] == 4)
      list1.add(x);
  return list1;
}
```

Pourquoi est-ce si compliqué d'en comprendre les intentions ? Il ne contient aucune expression complexe. Son utilisation des espaces et de l'indentation est correcte. Il ne fait référence qu'à trois variables et deux constantes. Il est même dépourvu de classes bizarres ou de méthodes polymorphes ; il n'y a qu'une liste de tableaux (semble-t-il).

---

1. N.d.T. : Afin de conserver la cohérence de l'ouvrage et des exemples, nous avons choisi de laisser le code en anglais, mais de traduire les commentaires. Toutes les remarques qui concernent les noms anglais s'appliquent également aux noms français.
   Bien que francophone, vous pourrez être amené à utiliser l'anglais comme langue de base pour vos développements, par exemple si vous travaillez dans une équipe regroupant plusieurs nationalités ou si votre logiciel est diffusé publiquement dans le monde entier. Pour que des développeurs d'origine allemande, chinoise, russe ou autre puissent lire votre code, à défaut des commentaires, il est préférable de l'écrire en anglais.

Le problème vient non pas de la simplicité du code mais de son *implicite* : le niveau à partir duquel le contexte n'est plus explicite dans le code lui-même. Le code exige implicitement que nous connaissions les réponses aux questions suivantes :

1. Quelles sortes de choses trouve-t-on dans `theList` ?
2. Quelle est la signification de l'indice zéro sur un élément de `theList` ?
3. Quelle est la signification de la valeur `4` ?
4. Comment dois-je utiliser la liste retournée ?

Les réponses à ces questions ne sont pas données dans l'exemple de code, *mais elles pourraient l'être*. Supposons que nous travaillions sur un jeu du type démineur. Nous déterminons que le plateau de jeu est une liste de cellules nommée `theList`. Nous pouvons alors changer son nom en `gameBoard`.

Chaque cellule du plateau est représentée par un simple tableau. Nous déterminons que l'indice zéro correspond à l'emplacement d'une valeur d'état et que la valeur d'état `4` signifie "marquée". En donnant simplement des noms à ces concepts, nous pouvons considérablement améliorer le code :

```
public List<int[]> getFlaggedCells() {
  List<int[]> flaggedCells = new ArrayList<int[]>();
  for (int[] cell : gameBoard)
    if (cell[STATUS_VALUE] == FLAGGED)
      flaggedCells.add(cell);
  return flaggedCells;
}
```

La simplicité du code n'a pas changé. Il contient toujours exactement le même nombre d'opérateurs et de constantes, et les niveaux d'imbrication sont identiques. En revanche, il est devenu beaucoup plus explicite.

Nous pouvons aller plus loin et écrire une simple classe pour les cellules au lieu d'utiliser un tableau entier. Elle peut offrir une fonction révélatrice des intentions, nommée `isFlagged`, pour cacher les constantes magiques. Voici la nouvelle version du code :

```
public List<Cell> getFlaggedCells() {
  List<Cell> flaggedCells = new ArrayList<Cell>();
  for (Cell cell : gameBoard)
    if (cell.isFlagged())
      flaggedCells.add(cell);
  return flaggedCells;
}
```

Grâce à ces simples changements de noms, il est facile de comprendre ce qui se passe. Voilà toute la puissance du choix de noms appropriés.

Éviter la désinformation

Les programmeurs ne doivent pas laisser de faux indices qui cachent la signification du code. Il faut éviter les mots dont le sens établi varie du sens voulu. Par exemple, `hp`, `aix` et `sco` ne sont pas des noms de variables appropriés car il s'agit de noms de plates-formes ou de systèmes Unix. Si vous voulez représenter une hypoténuse et que `hp` vous semble une abréviation pertinente, elle peut en réalité conduire à une désinformation.

Pour faire référence à des groupes de comptes, vous pouvez choisir `accountList` uniquement s'il s'agit bien d'une liste (`List`). Le mot liste possède un sens précis pour les programmeurs. Si le conteneur qui stocke les comptes n'est pas un `List`, le nom pourrait conduire à de fausses conclusions[2]. Ainsi, `accountGroup`, `bunchOfAccounts` ou simplement `accounts` sont mieux adaptés.

Vous devez également faire attention à ne pas choisir des noms trop proches. Il faut ainsi beaucoup de temps pour remarquer la subtile différence entre `XYZControllerForEfficientHandlingOfStrings` dans un module et, un peu plus loin, `XYZControllerForEfficientStorageOfStrings`. Ces noms se ressemblent trop.

En écrivant des concepts similaires de manière similaire, vous apportez une *information*. En employant des écritures incohérentes, vous faites de la *désinformation*. Les environnements Java modernes disposent d'un mécanisme de complétion automatique du code très apprécié. Il suffit d'écrire quelques caractères d'un nom et d'appuyer sur une combinaison de touches pour obtenir une liste des complétions possibles pour ce nom. Cet outil est encore plus utile si les noms de choses connexes sont regroupés par ordre alphabétique et si les différences sont évidentes. En effet, le développeur sélectionnera certainement un objet par son nom, sans consulter vos généreux commentaires ou même la liste des méthodes fournies par la classe correspondante.

L'utilisation des lettres `l` minuscule ou `O` majuscule pour les noms de variables est un parfait exemple de ce qu'il ne faut pas faire, en particulier lorsqu'elles sont combinées. En effet, elles ressemblent trop aux constantes un et zéro.

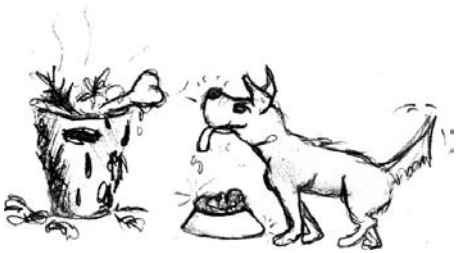
```
int a = l;
if ( O == l )
 a = O1;
else
 l = 01;
```

Vous pourriez penser que cet exemple est un tantinet arrangé, mais nous avons déjà rencontré du code dans lequel ce type de mauvais choix pullulait. Dans un cas, l'auteur du code avait suggéré d'employer une autre police afin que les différences soient plus

2. Même si le conteneur est bien un `List`, nous verrons plus loin qu'il est préférable de ne pas coder le type d'une variable dans le nom.

évidentes, mais c'est une solution qui doit être transmise à tous les futurs développeurs sous forme de tradition orale ou de document écrit. Le problème est résolu définitivement et sans effort en changeant simplement les noms.

Faire des distinctions significatives



Les programmeurs se créent des problèmes lorsqu'ils écrivent du code uniquement pour satisfaire le compilateur ou l'interpréteur. Par exemple, puisqu'il est impossible d'employer le même nom pour faire référence à deux choses différentes dans la même portée, vous pourriez être tenté de remplacer l'un des noms de manière arbitraire. Pour cela, certains introduisent parfois une faute d'orthographe dans le nom, mais cela conduit à une situation surprenante : si les erreurs d'orthographe sont corrigées, la compilation n'est plus possible[3].

Il ne suffit pas d'ajouter des numéros ou des mots parasites, bien que cela puisse satisfaire le compilateur. Si des noms doivent être différents, alors, ils doivent également représenter des choses différentes.

L'emploi des séries de numéros (`a1`, `a2`...`aN`) va à l'encontre d'un nommage intentionnel. S'ils n'entraînent pas une désinformation, ces noms ne sont pas informatifs. Ils ne donnent aucun indice quant aux intentions de l'auteur. Prenons l'exemple suivant :

```
public static void copyChars(char a1[], char a2[]) {
  for (int i = 0; i < a1.length; i++) {
    a2[i] = a1[i];
  }
}
```

Cette fonction serait beaucoup plus lisible si les arguments se nommaient `source` et `destination`.

Les mots parasites représentent une autre distinction dépourvue de sens. Imaginons qu'il existe une classe `Product`. Si vous en créez une autre nommée `ProductInfo` ou `ProductData`, vous avez choisi des noms différents sans qu'ils représentent quelque chose de différent. `Info` et `Data` sont des mots parasites vagues, tout comme les articles `a`, `an` et `the`.

Sachez cependant qu'il n'est pas insensé d'employer des conventions de préfixe comme `a` et `the`, mais à condition qu'elles créent une distinction significative. Par exemple,

3. C'est par exemple le cas de la pratique véritablement horrible qui consiste à créer une variable nommée `klass` simplement parce que le nom `class` est employé pour autre chose.

vous pourriez employer `a` pour toutes les variables locales et `the` pour tous les arguments de fonction[4]. Le problème se pose lorsque vous décidez d'appeler une variable `theZork` parce qu'une autre variable nommée `zork` existe déjà.

Les mots parasites sont redondants. Le mot `variable` ne doit jamais apparaître dans le nom d'une variable. Le mot `table` ne doit jamais apparaître dans le nom d'un tableau. En quoi `NameString` est-il meilleur que `Name` ? Est-ce que `Name` pourrait être un nombre en virgule flottante ? Dans l'affirmative, cela contredirait la règle précédente concernant la désinformation. Imaginons qu'il existe une classe nommée `Customer` et une autre nommée `CustomerObject`. Comment devez-vous interpréter cette distinction ? Laquelle représentera le meilleur accès vers l'historique de paiement d'un client ?

Nous connaissons une application dans laquelle ce point est illustré. Nous avons changé les noms afin de protéger le coupable, mais voici la forme exacte de l'erreur :

```
getActiveAccount();
getActiveAccounts();
getActiveAccountInfo();
```

Comment les programmeurs impliqués dans ce projet peuvent-ils savoir laquelle de ces fonctions invoquer ?

En l'absence de convention précise, la variable `moneyAmount` ne peut pas être différenciée de la variable `money`, `customerInfo` équivaut à `customer`, `accountData` est identique à `account`, et `theMessage` est indiscernable de `message`. Les noms doivent être distinctifs afin que le lecteur comprenne les différences qu'ils représentent.

Choisir des noms prononçables

Les humains manipulent les mots avec aisance. Une part importante de notre cerveau est dédiée au concept de mots. Par définition, les mots sont prononçables. Il serait dommage de ne pas exploiter cette vaste partie de notre cerveau qui a évolué de manière à traiter le langage parlé. Par conséquent, les noms doivent être prononçables.

Si nous ne pouvons pas les prononcer, nous ne pouvons pas en discuter sans paraître idiots. "Très bien, sur le bé cé erre trois cé enne té il y a pé esse zedde cu int, tu vois ?" La programmation étant une activité sociale, cet aspect est important.

Je connais une entreprise qui utilise `genymdhms` pour date de génération (`gen`), année (`y`), mois (`m`), jour (`d`), heure (`h`), minute (`m`) et seconde (`s`). Ils ont pris l'habitude de prononcer, en anglais, "gen why emm dee aich emm ess". Pour ma part, j'ai l'habitude de prononcer comme c'est écrit. Pour ce nom, cela donne "gen-yah-mudda-hims". Finale-

4. Oncle Bob avait l'habitude de procéder ainsi en C++, mais il a renoncé à cette pratique car les IDE modernes la rendent inutile.

ment, il a été appelé "çà" par différents concepteurs et analystes. Nous n'en avions pas moins l'air stupides. Mais, puisqu'il faisait l'objet d'une blague, cela restait marrant. Que ce soit amusant ou non, nous acceptions les noms médiocres. Les nouveaux développeurs devaient se faire expliquer les variables et ils les prononçaient ensuite en employant des termes étranges à la place de mots anglais corrects. Comparons

```
class DtaRcrd102 {
  private Date genymdhms;
  private Date modymdhms;
  private final String pszqint = "102";
  /* ... */
};
```

à

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;
  private final String recordId = "102";
  /* ... */
};
```

Nous pouvons avoir à présent une conversation intelligible : "Hé, Mikey, regarde cet enregistrement ! La date de génération est fixée à demain ! Comment est-ce possible ?"

Choisir des noms compatibles avec une recherche

Les noms d'une seule lettre et les constantes numériques présentent un problème particulier en cela qu'ils sont des éléments difficiles à localiser dans le corps d'un texte.

S'il est facile de rechercher `MAX CLASSES PER STUDENT`, il n'en va pas de même pour le chiffre 7. Les recherches peuvent s'arrêter sur ce chiffre à l'intérieur de noms de fichiers, d'autres définitions de constantes et diverses expressions, où cette valeur est employée avec un objectif différent. Pire encore, si une constante est un grand nombre et si quelqu'un a inversé certains chiffres, nous obtenons alors un bogue qui échappe aux recherches du programmeur.

De même, le nom `e` n'est pas vraiment un bon choix pour une variable que le programmeur peut avoir besoin de rechercher. En effet, puisqu'il s'agit de la lettre la plus courante en anglais (et en français), elle risque d'apparaître sur chaque ligne de code d'un programme. De ce point de vue, les noms longs ont plus d'atouts que les noms courts, et tout nom facilitant les recherches sont préférables à une constante numérique dans le code.

Pour moi, les noms d'une seule lettre ne doivent être utilisés que pour les variables locales à l'intérieur de méthodes courtes. *La longueur d'un nom doit correspondre à la taille de sa portée* [N5]. Si une variable ou une constante peut être visible ou utilisée en

plusieurs endroits du code, il est impératif de lui donner un nom compatible avec les recherches. À nouveau, comparons

```
for (int j=0; j<34; j++) {
  s += (t[j]*4)/5;
}
```

à

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
  int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
  int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
  sum += realTaskWeeks;
}
```

Vous conviendrez que `sum`, à défaut d'être un nom particulièrement utile, est au moins facile à rechercher. Le code qui utilise des noms intentionnels conduit à une fonction longue. Cependant, il faut prendre en compte la facilité de recherche du mot `WORK DAYS PER WEEK` par rapport à la recherche de tous les endroits où `5` a été utilisé et au filtrage des résultats afin de déterminer les occurrences qui présentent le sens voulu.

Éviter la codification

Nous croulons déjà sous les codes, alors, il est inutile de charger la barque encore plus. Le codage des informations de type ou de portée dans les noms ne fait qu'augmenter le travail de décodage. Il ne semble pas raisonnable d'obliger chaque nouvel employé à apprendre une autre "norme" de codification, en plus d'avoir à apprendre le corpus (souvent considérable) du code sur lequel il va travailler. Cela représente une pression mentale inutile lorsqu'on tente de résoudre un problème. Les noms codifiés sont rarement prononçables et sont sujets aux erreurs de saisie.

Notation hongroise

Autrefois, lorsque nous utilisions des langages où la longueur des noms était importante, nous transgressions cette règle par obligation, et regret. Fortran impose une codification dans laquelle la première lettre représente un code du type. Les premières versions de Basic acceptaient une lettre plus un chiffre. La notation hongroise a même élevé ce principe à un tout autre niveau.

La notation hongroise était plutôt importante dans l'API C de Windows, où tout était entier, pointeur sur un entier long, pointeur `void` ou l'une des nombreuses implémentations d'une chaîne de caractères (avec différents usages et attributs). En ce temps-là, le compilateur ne vérifiait pas les types. Les programmeurs avaient donc besoin d'un mécanisme pour mémoriser les types.

Les langages modernes disposent d'un système de types plus élaboré et les compilateurs mémorisent et font respecter le typage. Qui plus est, la tendance s'oriente vers des classes plus petites et des fonctions plus courtes. Ainsi, les programmeurs peuvent généralement voir la déclaration des variables qu'ils utilisent.

Les programmeurs Java n'ont pas besoin de codifier les types. Les objets sont fortement typés et les environnements de développement sont suffisamment élaborés pour détecter une erreur de type bien avant que la compilation ne soit lancée ! Aujourd'hui, la notation hongroise et les autres formes de codification ne constituent que des obstacles. Elles complexifient la modification du nom ou du type d'une variable, d'une fonction ou d'une classe. Elles rendent le code plus difficile à lire. Elles peuvent même conduire le système de codification à tromper le lecteur :

```
PhoneNumber phoneString; // Le nom n'a pas changé suite au changement de type !
```

Préfixes des membres

Aujourd'hui, il est également inutile de préfixer les variables membres par `m_`. Les classes et les fonctions doivent être suffisamment courtes pour rendre cette méthode obsolète. Il faut employer un environnement de développement qui surligne ou colore les membres afin qu'ils apparaissent clairement.

```
public class Part {
  private String m_dsc; // La description textuelle.
  void setName(String name) {
    m_dsc = name;
  }
}
```

```
-----------------------------------------------
```

```
public class Part {
  String description;
  void setDescription(String description) {
    this.description = description;
 }
}
```

Les programmeurs ont par ailleurs rapidement appris à ignorer le préfixe (ou le suffixe) pour ne voir que la partie significative du nom. Plus nous lisons du code, moins nous remarquons les préfixes. Ces derniers finissent par devenir un fouillis inutile et signalent un code ancien.

Interfaces et implémentations

Certains cas demandent parfois l'emploi d'une codification. Par exemple, supposons que nous construisions une FABRIQUE ABSTRAITE pour créer des formes. Cette fabrique sera une interface et sera implémentée par une classe concrète. Quels noms devons-nous choisir ? `IShapeFactory` et `ShapeFactory` ? Je préfère ne pas signaler les interfa-

ces. Le préfixe `I`, si classique dans les listings anciens, représente, au mieux, une distraction et, au pire, trop d'informations. Je ne souhaite pas que les utilisateurs sachent qu'ils passent par une interface. Je veux simplement qu'ils sachent qu'il existe un `ShapeFactory`. Par conséquent, si je dois codifier l'interface ou l'implémentation, j'opte pour l'implémentation. Il est préférable de la nommer `ShapeFactoryImp`, voire l'horrible `CShapeFactory`, que de codifier l'interface.

Éviter les associations mentales

Le lecteur ne doit pas avoir à convertir mentalement vos noms en noms qu'il connaît déjà. Ce problème survient généralement lorsque les termes choisis ne font pas partie du domaine du problème ou de la solution.

Les noms de variables sur une seule lettre introduisent ce problème. Un compteur de boucle peut évidemment être nommé `i`, `j` ou `k` (mais jamais `l` !) si sa portée est très réduite et si aucun autre nom n'entre en conflit. En effet, ces noms sur une seule lettre sont un classique pour les compteurs de boucle. Cependant, dans la majorité des autres contextes, un nom sur une seule lettre constitue un choix médiocre ; il s'agit simplement d'un paramètre que le lecteur doit mentalement associer au concept réel. Il n'y a pas de pire raison d'utiliser le nom `c` que d'avoir déjà employé `a` et `b`.

En général, les programmeurs sont des personnes plutôt intelligentes. Parfois, les personnes intelligentes aiment exhiber leur intelligence en montrant leur capacité de jonglage mental. En effet, si vous pouvez mémoriser sans faillir le fait que `r` est la version en minuscule de l'URL dans laquelle l'hôte et le schéma ont été supprimés, alors, vous devez certainement être très intelligent.

Il existe toutefois une différence entre un programmeur intelligent et un programmeur professionnel : le programmeur professionnel comprend que *la clarté prime*. Les professionnels emploient leurs facultés à bon escient et écrivent du code que les autres sont en mesure de comprendre.

Noms des classes

Pour les classes et les objets, nous devons choisir des noms ou des groupes nominaux comme `Customer`, `WikiPage`, `Account` ou `AddressParser`. Il est préférable d'éviter les termes `Manager`, `Processor`, `Data` ou `Info` dans le nom d'une classe. Un nom de classe ne doit pas être un verbe.

Noms des méthodes

Pour les méthodes, nous devons choisir des verbes ou des groupes verbaux comme `postPayment`, `deletePage` ou `save`. Les accesseurs, les mutateurs et les prédicats doivent être nommés d'après leur valeur et préfixés par `get`, `set` ou `is` conformément au standard JavaBean[5].

```
string name = employee.getName();
customer.setName("mike");
if (paycheck.isPosted())...
```

Lorsque des constructeurs sont surchargés, nous devons utiliser des méthodes de fabrique statiques avec des noms qui décrivent les arguments. Par exemple,

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

est généralement préférable à

```
Complex fulcrumPoint = new Complex(23.0);
```

Pour imposer l'emploi de ces méthodes de fabrique, les constructeurs correspondants doivent être rendus privés.

Ne pas faire le malin

Si les noms sont trop astucieux, ils ne seront mémorisés que par les personnes qui possèdent le même sens de l'humour que l'auteur, et uniquement aussi longtemps qu'elles se souviendront de la blague. Sauront-elles déterminer le rôle de la fonction `HolyHandGrenade` (littéralement, `SainteGrenadeAMain`) ? Si, pour certains, cela peut sembler malin, il est sans doute préférable de retenir le nom `DeleteItems` dans ce cas. Mieux vaut privilégier la clarté à l'humour.

Dans le code, les jeux d'esprit prennent souvent la forme d'expression familière ou d'argot. Par exemple, nous déconseillons l'emploi du nom `whack()` (littéralement, `pan()`) à la place de `kill()`. De même, les blagues liées à une culture doivent être évitées, comme `eatMyShorts()` (en référence à Bart Simpson) pour `abort()`.

Dites ce que vous pensez. Pensez ce que vous dites.

5. **http://java.sun.com/products/javabeans/docs/spec.html**.

Choisir un mot par concept

Choisissez un mot pour un concept abstrait et restez-y fidèle. Par exemple, il est assez déroutant d'avoir `fetch`, `retrieve` et `get` pour représenter des méthodes équivalentes dans des classes différentes. Comment pouvez-vous vous souvenir du nom de la méthode qui correspond à une classe ? Malheureusement, vous devez souvent vous rappeler de l'entreprise, du groupe ou de l'individu qui a écrit une bibliothèque ou une classe pour connaître le terme employé. Dans le cas contraire, vous passerez beaucoup de temps à parcourir les en-têtes et les exemples de code précédents.

Les environnements de développement modernes, comme Eclipse et IntelliJ, fournissent des indices liés au contexte, comme la liste des méthodes qu'il est possible d'invoquer sur un objet donné. Cependant, cette liste ne présente généralement pas les commentaires ajoutés aux noms des fonctions et aux paramètres. Vous serez déjà content de voir s'afficher les noms des paramètres indiqués dans les déclarations des fonctions. Les noms de fonctions doivent être autonomes et cohérents afin que vous puissiez choisir la méthode adéquate sans autre exploration.

De même, il est assez déroutant de rencontrer `controller`, `manager` et `driver` dans la même base de code. Quelle est la différence fondamentale entre `DeviceManager` et `ProtocolController` ? Pourquoi les deux ne sont-ils pas des `Controller` ou des `Manager` ? Et pourquoi pas des `Driver` ? Le nom vous amène à supposer que vous allez avoir deux objets de type très différent et de classe différente.

Un lexique cohérent sera précieux aux programmeurs qui emploieront votre code.

Éviter les jeux de mots

Vous devez éviter d'employer le même mot dans deux sens différents. Si vous utilisez le même terme pour deux idées différentes, il s'agit d'un jeu de mots.

Si vous respectez la règle "un mot par concept", vous pouvez arriver à un grand nombre de classes qui offrent, par exemple, une méthode `add`. Tant que la liste des paramètres et la valeur de retour des différentes méthodes `add` sont sémantiquement équivalentes, tout va bien.

En revanche, quelqu'un pourrait décider d'employer le mot `add` pour des raisons de "cohérence" alors que la notion d'ajout n'est pas la même. Supposons que plusieurs classes emploient `add` pour créer une nouvelle valeur en ajoutant ou en concaténant deux valeurs existantes. Supposons à présent que nous écrivions une nouvelle classe dont une méthode ajoute son seul argument dans une collection. Devons-nous appeler cette méthode `add` ? Ce choix peut sembler cohérent car il existe bien d'autres méthodes `add`, mais, dans ce cas, leur sémantique est différente. Il vaut mieux choisir à la place un

nom comme `insert` ou `append`. Si nous appelions `add` la nouvelle méthode, nous créerions un jeu de mots.

Notre objectif, en tant qu'auteur, est de créer un code aussi facile à comprendre que possible. Nous voulons que notre code soit rapide à parcourir, sans nécessiter un examen approfondi. Nous voulons employer le modèle classique du livre de poche dans lequel c'est à l'auteur d'être clair, non le modèle académique où c'est à l'étudiant de déterrer le sens d'un article.

Choisir des noms dans le domaine de la solution

N'oubliez pas que les personnes qui liront votre code seront des programmeurs. Par conséquent, n'hésitez pas et employez des termes informatiques, des noms d'algorithmes, des noms de motifs, des termes mathématiques, etc. Il n'est pas judicieux de prendre uniquement des noms issus du domaine du problème car vos collègues devront se tourner vers le client pour lui demander le sens de chaque mot alors qu'ils connaissent déjà le concept sous un autre nom.

Le nom `AccountVisitor` signifie quelque chose à tout programmeur familier du motif VISITEUR. Quel programmeur ne sait pas ce qu'est un `JobQueue` ? Les programmeurs ont énormément de choses très techniques à réaliser. Donner des noms techniques à ces choses se révèle généralement la meilleure option.

Choisir des noms dans le domaine du problème

Lorsqu'il n'existe aucun terme informatique pour ce que vous faites, utilisez le nom issu du domaine du problème. Le programmeur qui maintient votre code pourra au moins demander à un expert du domaine ce qu'il signifie.

La séparation des concepts du domaine de la solution et du problème fait partie du travail du bon programmeur et du bon concepteur. Le code qui est fortement lié aux concepts du domaine du problème doit employer des noms tirés de ce domaine.

Ajouter un contexte significatif

Quelques noms sont en eux-mêmes significatifs, mais ce n'est pas le cas de la plupart. Vous devez redonner aux noms leur contexte en les englobant dans les classes ou des fonctions aux noms appropriés, ou dans des espaces de noms. En dernier ressort, il peut être nécessaire d'ajouter un préfixe aux noms.

Imaginez qu'il existe les variables `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state` et `zipcode`. En les considérant ensemble, il est facile de comprendre qu'elles

forment une adresse. En revanche, si vous rencontrez uniquement la variable `state` (état) dans une méthode, en déduisez-vous automatiquement qu'elle fait partie d'une adresse ?

Vous pouvez ajouter un contexte en employant des préfixes : `addrFirstName`, `addrLastName`, `addrState`, etc. Les lecteurs comprendront facilement que ces variables font partie d'une structure plus vaste. Bien entendu, une meilleure solution consiste à créer une classe nommée `Address`. Ainsi, même le compilateur sait que la variable appartient à un concept plus vaste.

Examinons la méthode donnée au Listing 2.1. Les variables ont-elles besoin d'un contexte plus précis ? Le nom de la fonction fournit uniquement une partie du contexte ; l'algorithme fournit la suite. Après avoir lu la fonction, vous comprenez que les trois variables `number`, `verb` et `pluralModifier` font partie du message "estimation des statistiques". Malheureusement, le contexte doit être déduit. Lorsque vous voyez la méthode pour la première fois, la signification des variables reste opaque.

Listing 2.1 : Variables dans un contexte flou

```
private void printGuessStatistics(char candidate, int count) {
    String number;
    String verb;
    String pluralModifier;
    if (count == 0) {
      number = "no";
      verb = "are";
      pluralModifier = "s";
    } else if (count == 1) {
      number = "1";
      verb = "is";
      pluralModifier = "";
    } else {
      number = Integer.toString(count);
      verb = "are";
      pluralModifier = "s";
    }
    String guessMessage = String.format(
      "There %s %s %s%s", verb, number, candidate, pluralModifier
    );
    print(guessMessage);
}
```

La fonction n'est pas vraiment courte et les variables sont employées tout au long. Pour décomposer la fonction en éléments plus petits, nous devons créer une classe `GuessStatisticsMessage` et transformer ces trois variables en champs de cette classe. De cette manière, nous leur attribuons un contexte clair. Elles font *définitivement* partie de `GuessStatisticsMessage`. L'amélioration du contexte permet également de rendre l'algorithme plus clair en le décomposant en fonctions plus petites (voir Listing 2.2).

Listing 2.2 : Variables avec un contexte

```
public class GuessStatisticsMessage {
  private String number;
  private String verb;
  private String pluralModifier;

  public String make(char candidate, int count) {
    createPluralDependentMessageParts(count);
    return String.format(
      "There %s %s %s%s",
      verb, number, candidate, pluralModifier );
  }

  private void createPluralDependentMessageParts(int count) {
    if (count == 0) {
      thereAreNoLetters();
    } else if (count == 1) {
      thereIsOneLetter();
    } else {
      thereAreManyLetters(count);
    }
  }

  private void thereAreManyLetters(int count) {
    number = Integer.toString(count);
    verb = "are";
    pluralModifier = "s";
  }

  private void thereIsOneLetter() {
    number = "1";
    verb = "is";
    pluralModifier = "";
  }

  private void thereAreNoLetters() {
    number = "no";
    verb = "are";
    pluralModifier = "s";
  }
}
```

Ne pas ajouter de contexte inutile

Dans une application fictive appelée "Gas Station Deluxe" (station-service de luxe), il est déconseillé de préfixer chaque classe par `GSD`. Honnêtement, cela va à l'encontre du bénéfice apporté par les outils. Vous saisissez `G` et appuyez ensuite sur la touche de complétion. En résultat, vous obtenez une très longue liste de toutes les classes du système. Est-ce bien judicieux ? Pourquoi ne pas laisser l'IDE vous faciliter la tâche ?

De la même manière, supposons que vous ayez inventé une classe `MailingAddress` dans un module de comptabilité de `GSD` et que vous l'ayez nommée `GSDAccount`

`Address`. Plus tard, vous avez besoin d'une adresse postale pour votre application de gestion des contacts clients. Utilisez-vous `GSDAccountAddress` ? Pensez-vous que ce nom soit approprié ? Dix des dix-sept caractères sont redondants ou hors de propos.

Tant qu'ils restent clairs, il est généralement préférable de choisir des noms plus courts que des noms longs. N'ajoutez pas de contexte inutile à un nom.

Les noms `accountAddress` et `customerAddress` sont parfaits pour des instances de la classe `Address`, mais ils font de piètres noms de classes. `Address` convient parfaitement à une classe. S'il faut différencier des adresses postales, des adresses MAC et des adresses web, pourquoi ne pas s'orienter vers `PostalAddress`, `MAC` et `URI`. Ces noms sont plus précis, ce qui constitue le premier objectif du nommage.

Mots de la fin

Le choix de bons noms est difficile car cela demande une aptitude à la description et une culture générale partagée. Il s'agit d'un problème d'enseignement, non d'un problème technique, métier ou de gestion. C'est pourquoi peu de personnes apprennent à le faire correctement.

Les gens ont également peur de renommer les choses, de crainte que d'autres développeurs soulèvent des objections. Nous ne partageons pas cette crainte et sommes plutôt reconnaissants lorsque les noms changent (en mieux). La plupart du temps, nous ne mémorisons pas réellement les noms des classes et des méthodes. Nous employons les outils modernes pour prendre en charge ces détails et pouvoir nous focaliser sur un code qui doit se lire aussi facilement que des paragraphes et des phrases, tout au moins comme des tableaux et des structures de données (une phrase n'est pas vraiment le meilleur moyen d'afficher des données). Vous surprendrez probablement quelqu'un si vous changez des noms, mais pas plus que par n'importe quelle autre amélioration du code. Ne le laissez pas vous empêcher de suivre votre voie.

Respectez certaines des règles précédentes et voyez si vous améliorez la lisibilité de votre code. Si vous assurez la maintenance d'un code tiers, servez-vous des outils de remaniement pour résoudre les problèmes. Cela se révélera payant sur le court terme, ainsi que sur le long terme.

3

Fonctions

Aux premiers jours de la programmation, nos systèmes étaient constitués de routines et de sous-routines. Ensuite, à l'ère de Fortran et de PL/1, nous les composions de programmes, sous-programmes et fonctions. De cette époque, seules les fonctions ont survécu. Tous les programmes s'organisent autour des fonctions. Dans ce chapitre, nous verrons comment les écrire bien.

Prenons le code du Listing 3.1. Il n'est pas très facile de trouver une longue fonction dans FitNesse[1], mais, suite à une petite recherche, j'ai pu trouver celle-ci. Non seulement elle est longue, mais elle contient également du code dupliqué, de nombreuses chaînes de caractères bizarres et beaucoup de types de données et d'API étranges et peu évidents. En trois minutes, que pouvez-vous en comprendre ?

Listing 3.1 : `HtmlUtil.java` **(FitNesse 20070619)**

```
public static String testableHtml(
  PageData pageData,
  boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
      if (suiteSetup != null) {
        WikiPagePath pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
              .append(pagePathName)
              .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
    if (setup != null) {
      WikiPagePath setupPath =
        wikiPage.getPageCrawler().getFullPath(setup);
      String setupPathName = PathParser.render(setupPath);
      buffer.append("!include -setup .")
            .append(setupPathName)
            .append("\n");
    }
  }
  buffer.append(pageData.getContent());
  if (pageData.hasAttribute("Test")) {
    WikiPage teardown =
      PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
      WikiPagePath tearDownPath =
        wikiPage.getPageCrawler().getFullPath(teardown);
      String tearDownPathName = PathParser.render(tearDownPath);
      buffer.append("\n")
            .append("!include -teardown .")
```

1. Un outil open-source de test (**http://www.fitnesse.org**).

```
            .append(tearDownPathName)
            .append("\n");
      }
      if (includeSuiteSetup) {
        WikiPage suiteTeardown =
          PageCrawlerImpl.getInheritedPage(
                  SuiteResponder.SUITE_TEARDOWN_NAME,
                  wikiPage
          );
        if (suiteTeardown != null) {
          WikiPagePath pagePath =
            suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
          String pagePathName = PathParser.render(pagePath);
          buffer.append("!include -teardown .")
                .append(pagePathName)
                .append("\n");
        }
      }
    }
    pageData.setContent(buffer.toString());
    return pageData.getHtml();
  }
```

Après trois minutes d'examen, avez-vous réussi à comprendre l'objectif de la fonction ? C'est peu probable. Elle réalise beaucoup d'opérations, à un trop grand nombre de niveaux d'abstraction. Elle contient des chaînes étranges et des appels de fonctions bizarres, à l'intérieur d'instructions `if` imbriquées sur deux niveaux et contrôlées par des indicateurs.

Toutefois, grâce à quelques simples extractions de méthodes, un peu de renommage et de restructuration, il est possible de présenter les objectifs de la fonction en neuf lignes (voir Listing 3.2). En trois minutes, pouvez-vous les comprendre ?

Listing 3.2 : `HtmlUtil.java` **(remanié)**

```
  public static String renderPageWithSetupsAndTeardowns(
    PageData pageData, boolean isSuite
  ) throws Exception {
    boolean isTestPage = pageData.hasAttribute("Test");
    if (isTestPage) {
      WikiPage testPage = pageData.getWikiPage();
      StringBuffer newPageContent = new StringBuffer();
      includeSetupPages(testPage, newPageContent, isSuite);
      newPageContent.append(pageData.getContent());
      includeTeardownPages(testPage, newPageContent, isSuite);
      pageData.setContent(newPageContent.toString());
    }

    return pageData.getHtml();
  }
```

Si vous ne travaillez pas sur FitNesse, vous pourriez avoir du mal à saisir tous les détails. En revanche, vous comprenez certainement que cette fonction inclut des pages

de montage[2] (*setup*) et de démontage[3] (*teardown*) dans une page de test, dont elle effectue ensuite le rendu HTML. Si vous connaissez JUnit[4], vous devinez sans doute que cette fonction appartient à une sorte de framework de test fondé sur le Web. Bien entendu, vous avez raison. À partir du Listing 3.2, la déduction de ces informations est relativement facile, alors qu'elles sont plutôt embrouillées dans le Listing 3.1.

Par conséquent, qu'est-ce qui facilite la lecture et la compréhension d'une fonction comme celle du Listing 3.2 ? Comment pouvons-nous faire en sorte qu'une fonction transmette ses objectifs ? Quels attributs pouvons-nous donner à nos fonctions pour que le lecteur lambda ait une idée du programme dans lequel elles résident ?

Faire court

La première règle est d'écrire des fonctions courtes. La deuxième règle est qu'*elles doivent être encore plus courtes que cela*. Je ne peux pas justifier cette déclaration. Je ne peux m'appuyer sur aucune recherche qui montre que les fonctions très courtes sont préférables. Je peux seulement préciser que, depuis quatre décennies, j'ai écrit des fonctions de toutes tailles. J'en ai même écrit plusieurs approchant 3 000 lignes ; une horreur ! J'ai écrit des montagnes de fonctions de 100 à 300 lignes. J'ai également écrit des fonctions de 20 à 30 lignes. Cette expérience, par essais et erreurs, m'a montré que les fonctions doivent être les plus courtes possible.

Dans les années 1980, nous avions pour habitude de dire qu'une fonction ne devait pas dépasser la longueur d'un écran. À l'époque, il s'agissait d'écrans VT100 avec 25 lignes de 80 colonnes, et les éditeurs employaient 4 lignes pour les commandes. Aujourd'hui, avec une police adaptée et un grand moniteur, nous pouvons placer 150 caractères sur 1 ligne et au moins 100 lignes par écran. Cela dit, les lignes ne doivent pas contenir 150 caractères ni les fonctions, 100 lignes. Les fonctions doivent rarement dépasser 20 lignes.

Quelle doit être la taille d'une fonction ? En 1999, je suis allé rendre visite à Kent Beck chez lui dans l'Oregon. Nous nous sommes assis pour programmer ensemble. À un moment donné, il m'a montré un petit programme sympa en Java/Swing appelé *Sparkle*. Il produisait sur l'écran un effet visuel très ressemblant à la baguette magique de la bonne fée dans le film *Cendrillon*. Lors du déplacement de la souris, des étincelles jaillissaient copieusement du pointeur et retombaient vers le bas de la fenêtre, attirées par un champ gravitationnel simulé. Lorsque Kent m'a montré le code, j'ai été frappé par la taille réduite de toutes les fonctions. J'avais l'habitude des très longues fonctions

2. N.d.T. : Pour la préparation ou l'initialisation de la page de test afin qu'elle soit en état de servir.
3. N.d.T. : Pour le nettoyage ou la remise en état après utilisation de la page de test.
4. Un outil open-source de test unitaire pour Java (**http://www.junit.org**).

employées dans les programmes Swing. Dans celui-ci, chaque fonction occupait entre deux et quatre lignes. Chacune était d'une évidence transparente. Chacune avait son scénario. Et chacune amenait à la suivante de manière irréfutable. Vos fonctions doivent avoir cette taille[5] !

De manière générale, les fonctions doivent être plus courtes que celle du Listing 3.2 ! Il faudra même les raccourcir pour arriver à celle du Listing 3.3.

Listing 3.3 : `HtmlUtil.java` **(re-remanié)**

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

Blocs et indentation

Pour parvenir à une telle taille, les blocs des instructions `if`, des instructions `else`, des instructions `while`, etc. ne doivent occuper qu'une seule ligne. Il est fort probable que cette ligne sera un appel de fonction. Cela permet non seulement de garder la fonction englobante petite, mais également d'ajouter une valeur documentaire car la fonction appelée à l'intérieur du bloc aura un nom parfaitement descriptif.

Il faut aussi que la taille de ces fonctions ne leur permette pas de contenir des structures imbriquées. Le niveau d'indentation d'une fonction ne doit donc pas être supérieur à un ou deux. Par ailleurs, les fonctions sont ainsi plus faciles à lire et à comprendre.

Faire une seule chose

Vous ne devez pas avoir trop de mal à voir que la fonction du Listing 3.1 fait plusieurs choses. Entre autres, elle crée des tampons, récupère des pages, recherche des pages héritées, détermine des chemins, ajoute des chaînes mystérieuses et génère du contenu HTML. Le Listing 3.1 est vraiment très occupé à réaliser différentes tâches. *A contrario*, le Listing 3.3 se charge d'une seule chose. Il inclut des pages de montage et de démontage dans des pages de test.

5. J'ai demandé à Kent s'il en avait encore une copie, mais il n'a pas été en mesure d'en trouver une. J'ai également recherché sur mes anciens ordinateurs, mais impossible de retrouver ce programme. Il n'existe plus que dans ma mémoire.

Le conseil suivant est prodigué depuis plus de trente ans sous une forme ou sous une autre.

> ***UNE FONCTION DOIT FAIRE UNE SEULE CHOSE. ELLE DOIT LA FAIRE BIEN ET NE FAIRE QU'ELLE.***

Toutefois, que signifie précisément "une chose" ? Le Listing 3.3 fait-il une chose ? Nous pouvons facilement estimer qu'il fait trois choses :

1. Déterminer si la page est une page de test.
2. Dans l'affirmative, inclure les pages de montage et de démontage.
3. Effectuer un rendu HTML de la page.

Dans ce cas, qu'en est-il ? La fonction réalise-t-elle une ou trois choses ? Vous remarquerez que les trois étapes de la fonction se trouvent à un niveau d'abstraction en dessous du nom choisi pour la fonction. Nous pouvons expliquer la fonction en la décrivant sous forme d'un bref paragraphe *POUR* (*TO*[6]) :

> *POUR générer une page qui comprend un montage et un démontage (TO RenderPageWithSetupsAndTeardowns), nous vérifions que la page est une page de test et, dans l'affirmative, nous incluons les pages de montage et de démontage. Dans tous les cas, nous présentons la page en HTML.*

Lorsqu'une fonction met en œuvre des étapes qui se trouvent à un seul niveau sous son nom, alors, la fonction réalise une seule chose. En effet, nous décrivons des fonctions pour décomposer un concept plus vaste (autrement dit, le nom de la fonction) en un ensemble d'étapes se trouvant au niveau d'abstraction inférieur suivant.

Il est assez évident que le Listing 3.1 contient des étapes à de nombreux niveaux d'abstraction différents ; il fait donc plus d'une chose. Même le Listing 3.2 possède deux niveaux d'abstraction, comme le prouve la réduction que nous avons pu effectuer. En revanche, il est très difficile de réduire le Listing 3.3 de manière sensée. Nous pourrions déplacer l'instruction `if` dans une fonction nommée `includeSetupsAndTear downsIfTestPage`, mais cela ne fait que reformuler le code sans changer le niveau d'abstraction.

Par conséquent, une autre manière de savoir si une fonction fait "plusieurs choses" consiste à déterminer s'il est possible d'en extraire une autre fonction dont le nom n'est pas simplement une reformulation de son implémentation [G34].

---

6. Le langage LOGO utilisait le mot-clé "TO" de la même manière que Ruby et Python emploient "def". Chaque fonction commence donc par le mot "TO". Cela avait un effet intéressant sur la manière de concevoir les fonctions.

Sections à l'intérieur des fonctions

Examinez le Listing 4.7 à la page 78. Vous remarquerez que la fonction `generatePrimes` est décomposée en sections comme *déclarations*, *initialisations* et *crible*. Il s'agit d'un symptôme manifeste d'une fonction qui réalise plusieurs choses. Une fonction qui ne fait qu'une seule chose ne peut pas être décomposée en sections.

Un niveau d'abstraction par fonction

Pour être certain que chaque fonction ne fait qu'une seule chose, nous devons vérifier que les instructions qu'elle contient se trouvent toutes au même niveau d'abstraction. Il est facile de voir que le Listing 3.1 ne respecte pas cette règle. Il contient des concepts de très haut niveau, comme `getHtml()`, tandis que d'autres se trouvent à un niveau intermédiaire, comme `String pagePathName = PathParser.render(pagePath)`, et d'autres à un niveau très bas, comme `.append("\n")`.

Mélanger les niveaux d'abstraction au sein d'une même fonction est toujours déroutant. Le lecteur ne sera pas toujours en mesure de déterminer si une expression est un concept essentiel ou un détail. Pire encore, comme dans la métaphore des vitres brisées, dès lors que des détails sont mélangés à des concepts essentiels, de plus en plus de détails ont tendance à surgir à l'intérieur de la fonction.

Lire le code de haut en bas : la règle de décroissance

Nous voulons que le code puisse se lire du début à la fin comme un récit [KP78, p. 37]. Nous voulons que chaque fonction soit suivie des fonctions de niveau d'abstraction inférieure, afin que nous puissions lire le programme en descendant d'un niveau d'abstraction à la fois alors que nous parcourons la liste des fonctions vers le bas. J'appelle cela *règle de décroissance* (*Stepdown Rule*).

Autrement dit, nous voulons pouvoir lire le programme comme s'il s'agissait d'un ensemble de paragraphes *POUR*, chacun décrivant le niveau d'abstraction actuel et faisant référence à des paragraphes *POUR* de niveau inférieur.

> *Pour inclure les pages de montage et de démontage, nous incluons le montage, puis le contenu de la page de test et enfin le démontage.*
>
> *Pour inclure le montage, nous incluons le montage d'une suite s'il s'agit d'une suite, puis nous incluons le montage normal.*
>
> *Pour inclure le montage d'une suite, nous recherchons la page "SuiteSetUp" dans la hiérarchie parente et nous incluons une instruction avec le chemin de cette page.*
>
> *Pour rechercher le parent...*

Les programmeurs ont énormément de mal à apprendre à respecter cette règle et à écrire des fonctions qui restent à un seul niveau d'abstraction. Cependant, cet apprentissage est très important. Il s'agit de la clé qui permet d'obtenir des fonctions courtes et d'être certain qu'elles ne font qu'une seule chose. Faire en sorte que le code se lise de haut en bas comme un ensemble de paragraphes *POUR* constitue une technique efficace pour conserver un niveau d'abstraction cohérent.

Examinez le Listing 3.7 à la page 56. Il présente l'intégralité de la fonction `testableHtml` remaniée en suivant les principes décrits ici. Remarquez comme chaque fonction introduit la suivante et comment chacune reste à un niveau d'abstraction cohérent.

Instruction *switch*

Il est difficile d'écrire des instructions `switch` courtes[7]. Même une instruction `switch` qui ne comprend que deux cas est plus longue que la taille adéquate d'un bloc ou d'une fonction. Il est également difficile d'obtenir une instruction `switch` qui ne fait qu'une seule chose. Par essence, une instruction `switch` effectue *N* choses. S'il n'est pas toujours possible d'éviter les instructions `switch`, nous pouvons faire en sorte que chaque instruction `switch` soit enfouie dans une classe de bas niveau et qu'elle ne soit jamais répétée. Pour ce faire, nous employons évidemment le polymorphisme.

Prenons le Listing 3.4. Il montre l'une des opérations qui dépend du type de l'employé.

Listing 3.4 : `Payroll.java`

```
public Money calculatePay(Employee e)
throws InvalidEmployeeType {
    switch (e.type) {
      case COMMISSIONED:
        return calculateCommissionedPay(e);
      case HOURLY:
        return calculateHourlyPay(e);
      case SALARIED:
        return calculateSalariedPay(e);
      default:
        throw new InvalidEmployeeType(e.type);
    }
  }
```

Cette fonction présente plusieurs problèmes. Premièrement, elle est longue et ne fera que grandir lors de l'ajout de nouveaux types d'employés. Deuxièmement, elle prend manifestement en charge plusieurs choses. Troisièmement, elle ne respecte pas le principe de responsabilité unique (SRP, *Single Responsibility Principle*[8]) car il existe

7. Bien entendu, j'y inclus les chaînes `if/else`.
8. a. **http://en.wikipedia.org/wiki/Single_responsibility_principle**
   b. **http://www.objectmentor.com/resources/articles/srp.pdf**

plusieurs raisons de la modifier. Quatrièmement, elle ne respecte pas le principe ouvert/fermé (OCP, *Open Closed Principle*[9]) car elle doit être modifiée dès l'ajout de nouveaux types. Mais son principal problème est sans doute qu'un nombre illimité d'autres fonctions auront la même structure. Par exemple, nous pourrions avoir

```
isPayday(Employee e, Date date)
```

ou

```
deliverPay(Employee e, Money pay)
```

et bien d'autres encore. Toutes ces fonctions auront la même structure pernicieuse.

La solution à ce problème (voir Listing 3.5) consiste à enfouir l'instruction `switch` au plus profond d'une FABRIQUE ABSTRAITE [GOF] et de ne jamais la montrer à qui que ce soit. La fabrique utilise l'instruction `switch` pour créer les instances adéquates des classes dérivées de `Employee` et les différentes fonctions, comme `calculatePay`, `isPayday` et `deliverPay`, seront distribuées par polymorphisme depuis l'interface `Employee`.

Listing 3.5 : `Employee` **et sa fabrique**

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
-----------------
public interface EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType;
}
-----------------
public class EmployeeFactoryImpl implements EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws InvalidEmployeeType {
    switch (r.type) {
      case COMMISSIONED:
        return new CommissionedEmployee(r) ;
      case HOURLY:
        return new HourlyEmployee(r);
      case SALARIED:
        return new SalariedEmploye(r);
      default:
        throw new InvalidEmployeeType(r.type);
    }
  }
}
```

Voici ma règle générale concernant les instructions `switch` : elles peuvent être tolérées uniquement lorsqu'elles apparaissent une seule fois, sont employées pour créer des

9. a. **http://fr.wikipedia.org/wiki/Principe_ouvert/fermé**
   b. **http://www.objectmentor.com/resources/articles/ocp.pdf**

objets polymorphes et sont cachées derrière une relation d'héritage, de manière que le reste du système ne puisse pas les voir [G23]. Bien entendu, chaque cas est unique et il m'arrive parfois de ne pas respecter certaines parties de cette règle.

Choisir des noms descriptifs

Dans le Listing 3.7, j'ai modifié le nom de notre fonction d'exemple ; de `testableHtml` elle est devenue `SetupTeardownIncluder.render`. Ce nom est beaucoup mieux adapté car il décrit plus précisément l'objectif de la fonction. J'ai également donné à chaque méthode privée un nom descriptif, comme `isTestable` ou `includeSetupAndTear downPages`. Il est difficile de surestimer la valeur des noms bien choisis. Rappelez-vous le principe de Ward : "*Vous savez que vous travaillez avec du code propre lorsque chaque fonction que vous lisez correspond presque parfaitement à ce que vous attendiez*." La moitié du chemin qui mène à ce principe consiste à choisir de bons noms pour de courtes fonctions qui ne font qu'une seule chose. Plus une fonction est courte et ciblée, plus il est facile de choisir un nom descriptif.

N'ayez pas peur de créer un nom long. Un nom long descriptif vaut mieux qu'un nom court énigmatique complété d'un long commentaire descriptif. Servez-vous d'une convention de nommage qui facilite la lecture des multiples mots composant les noms de fonctions et qui permet à tous ces mots de créer des noms qui décrivent le rôle des fonctions.

N'ayez pas peur de passer du temps à choisir les noms. Vous devez essayer plusieurs noms différents et lire le code correspondant à chacun. Avec les IDE modernes, comme Eclipse ou IntelliJ, il est très simple de changer les noms. Servez-vous de l'un de ces IDE et testez les différents noms jusqu'à trouver celui qui soit le plus descriptif.

En choisissant des noms descriptifs, vous clarifiez la conception du module dans votre esprit et il sera ainsi plus facile à améliorer. Il arrive très souvent que la recherche d'un bon nom conduise à une restructuration bénéfique du code.

Restez cohérent dans le choix des noms. Employez les mêmes phrases, groupes nominaux et verbes dans les noms de fonctions de vos modules. Prenons, par exemple, les noms `includeSetupAndTeardownPages`, `includeSetupPages`, `includeSuiteSetupPage` et `includeSetupPage`. Leur phraséologie semblable leur permet globalement de raconter une histoire. Si je n'ai montré que cette séquence, c'est pour que vous vous demandiez ce qui est arrivé à `includeTeardownPages`, `includeSuiteTeardownPage` et `include TeardownPage`. Je vous le donne en mille, "... *pratiquement ce que vous attendiez*".

Arguments d'une fonction

Idéalement, le nombre d'arguments d'une fonction devrait être égal à zéro (niladique). Ensuite viennent les fonctions à un argument (monadique, ou unaire), puis à deux arguments (diadique). Les fonctions à trois arguments (triadique) doivent être évitées autant que possible. Les fonctions qui prennent plus de trois arguments (polyadique) exige une très bonne raison ou ne doivent jamais être employées.

Les arguments sont complexes. Ils possèdent une puissance conceptuelle importante. C'est pourquoi je les évite autant que possible, hormis à titre d'exemple. Prenons le cas du `StringBuffer` dans l'exemple. Nous aurions pu le passer en argument au lieu d'en faire une variable d'instance, mais le lecteur aurait alors dû l'interpréter chaque fois qu'il l'aurait rencontré. Lorsque nous lisons l'histoire racontée par le module, il est plus facile de comprendre `includeSetupPage()` que `includeSetupPageInto(newPageContent)`. L'argument se trouve à un niveau d'abstraction différent du nom de la fonction et nous oblige à connaître un détail (c'est-à-dire `StringBuffer`) qui n'est pas particulièrement important à ce stade.

Les arguments sont même encore plus pénibles du point de vue des tests. Imaginez la difficulté que représente l'écriture de tous les cas de test qui permettent de vérifier que les différentes combinaisons des arguments fonctionnent correctement. Lorsqu'il n'y a aucun argument, cette opération est simple. Lorsqu'il y a un argument, elle n'est pas trop difficile. Avec deux arguments, le problème devient un peu plus complexe. Avec plus de deux arguments, le test de chaque combinaison des valeurs peut être décourageant.

Les arguments de sortie sont plus difficiles à comprendre que les arguments d'entrée. Lorsque nous lisons une fonction, nous sommes habitués à l'idée d'informations qui entrent dans la fonction au travers des arguments et en sortent par l'intermédiaire de la valeur de retour. Généralement, nous ne nous attendons pas à ce que des informations ressortent au travers des arguments. Par conséquent, les arguments de sortie nous demandent souvent d'y regarder à deux fois.

Dans l'ordre de préférence, après l'absence totale d'argument vient la présence d'un argument. `SetupTeardownIncluder.render(pageData)` est relativement facile à comprendre. Il est assez clair que nous allons effectuer le *rendu* (`render`) des données contenues dans l'objet `pageData`.

Formes unaires classiques

Il existe deux raisons très classiques de passer un seul argument à une fonction. Dans le premier cas, vous posez une question à propos de cet argument, comme dans `boolean fileExists("MonFichier")`. Dans le deuxième cas, l'argument est manipulé, pour le transformer en autre chose et *le retourner*. Par exemple, `InputStream fileOpen ("MonFichier")` transforme un nom de fichier passé comme un `String` en une valeur de retour de type `InputStream`. Ces deux usages sont ceux à quoi s'attend le lecteur lorsqu'il voit une fonction. Il faut choisir des noms qui permettent de distinguer clairement ces deux cas et toujours employer les deux formes dans un contexte cohérent (voir la section "Séparer commandes et demandes", page 51).

L'*événement* est une forme moins fréquente, mais toutefois très utile, de fonction à un seul argument. Il existe dans ce cas un argument d'entrée et aucun argument de sortie. Le programme global est conçu de manière à interpréter l'appel de fonction comme un événement et à utiliser l'argument pour modifier l'état du système, par exemple `void passwordAttemptFailedNtimes(int attempts)`. Cette forme doit être employée avec prudence. Le lecteur doit voir très clairement qu'il s'agit d'un événement. Choisissez soigneusement les noms et les contextes.

Vous devez essayer d'éviter les fonctions unaires qui ne correspondent pas à ces formes, par exemple `void includeSetupPageInto(StringBuffer pageText)`. Pour une transformation, l'emploi d'un argument de sortie à la place d'une valeur de retour est déroutant. Si une fonction transforme son argument d'entrée, cette conversion doit transparaître dans la valeur de retour. `StringBuffer transform(StringBuffer in)` doit être préféré à `void transform(StringBuffer out)`, même si l'implémentation dans le premier cas retourne simplement l'argument d'entrée. Il a au moins l'intérêt de respecter la forme d'une transformation.

Arguments indicateurs

Les arguments indicateurs sont laids. Passer une valeur booléenne à une fonction est véritablement une pratique épouvantable. Cela complique immédiatement la signature de la méthode, en proclamant que cette fonction fait plusieurs choses. Elle réalise une chose lorsque l'indicateur vaut `true` et une autre lorsqu'il vaut `false` !

Dans le Listing 3.7, nous n'avons pas eu le choix car les appelants passaient déjà cet indicateur et nous voulions limiter l'étendue du remaniement à la fonction et à ce qui se trouvait derrière. L'appel de méthode `render(true)` n'en reste pas moins déroutant pour le lecteur. Placer la souris sur l'appel et voir s'afficher `render(boolean isSuite)` apporte une petite aide, mais c'est bien peu. Il aurait fallu décomposer la fonction en deux : `renderForSuite()` et `renderForSingleTest()`.

Fonctions diadiques

Une fonction à deux arguments est plus difficile à comprendre qu'une fonction unaire. Par exemple, `writeField(name)` est plus simple à comprendre que `writeField (outputStream, name)`[10]. Même si le sens de ces deux fonctions est clair, la première n'accroche pas l'œil car sa signification transparaît immédiatement, tandis que la seconde nécessite une petite pause afin d'ignorer le premier paramètre. Bien entendu, *cela* finit par amener des problèmes car il ne faut jamais ignorer une partie du code. C'est dans ces parties ignorées que se dissimuleront les bogues.

Il existe cependant des cas où les deux arguments sont appropriés. Par exemple, `Point p = new Point(0,0)` est parfaitement raisonnable. Les coordonnées cartésiennes prennent naturellement deux arguments. Nous serions plutôt surpris de voir `new Point(0)`. Toutefois, les deux arguments sont, dans ce cas, des *éléments ordonnés d'une même valeur* ! Ce n'était pas le cas de `outputStream` et de `name`, qui n'avaient aucun ordre naturel ou cohérence.

Même les fonctions diadiques évidentes, comme `assertEquals(expected, actual)` sont en réalité problématiques. Combien de fois avez-vous déjà inversé les arguments `actual` et `expected` ? Les deux arguments n'ont aucun ordre naturel. Le choix `expec ted, actual` n'est qu'une convention qu'il faut apprendre par la pratique.

Les diades ne sont pas diaboliques et vous devrez certainement en écrire. Cependant, vous devez savoir qu'elles ont un prix et que vous devez exploiter tous les mécanismes disponibles pour les convertir en fonctions unaires. Par exemple, vous pouvez faire de la méthode `writeField` un membre de `outputStream` afin de pouvoir écrire `output Stream.writeField(name)`. Vous pouvez également faire de `outputStream` un membre de la classe courante afin de ne pas avoir à le passer en argument. Ou bien vous pouvez créer une nouvelle classe, comme `FieldWriter`, qui prend le `outputStream` dans son constructeur et fournit une méthode `write`.

Fonctions triadiques

Les fonctions qui prennent trois arguments sont encore plus complexes à comprendre que les fonctions diadiques. Les problèmes d'ordre, de pause et d'ignorance sont multipliés. Je vous conseille d'y réfléchir à deux fois avant de créer une fonction triadique.

Prenons par exemple la surcharge classique à trois arguments de `assertEquals` : `assertEquals(message, expected, actual)`. Combien de fois avez-vous déjà lu

10. Je viens de terminer le remaniement d'un module qui utilisait la forme diadique. J'ai pu mettre le `outputStream` dans un champ de la classe et donner à tous les appels `writeField` la forme unaire. Le résultat est beaucoup plus propre.

`message` et pensé qu'il s'agissait de `expected` ? J'ai hésité et me suis arrêté de nombreuses fois sur ce cas précis. En réalité, *chaque fois que je le rencontre*, j'y regarde à deux fois pour ignorer le message.

En revanche, il existe une fonction triadique moins sournoise : `assertEquals(1.0, amount, .001)`. Même s'il faut l'examiner à deux fois, cela en vaut la peine. Il est toujours bon de se rappeler que l'égalité de valeurs en virgule flottante est une chose relative.

Objets en argument

Lorsqu'une fonction semble avoir besoin de plus de deux ou trois arguments, il est probable que certains d'entre eux feraient mieux d'être enveloppés dans leur propre classe. Par exemple, examinons la différence entre les deux déclarations suivantes :

```
Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);
```

Vous pourriez croire que la réduction du nombre d'arguments en créant des objets est une forme de tromperie, mais ce n'est pas le cas. Lorsque des groupes de variables sont passés ensemble, à l'instar de `x` et `y` dans l'exemple précédent, il est fort probable qu'ils fassent partie d'un concept qui mérite son propre nom.

Listes d'arguments

Il arrive parfois que nous devions passer un nombre variable d'arguments à une fonction. C'est par exemple le cas de la méthode `String.format` suivante :

```
String.format("%s worked %.2f hours.", name, hours);
```

Si les arguments variables sont tous traités de manière identique, comme dans ce cas, ils sont alors équivalents à un seul argument de type `List`. Grâce à ce raisonnement, `String.format` est en réalité une fonction diadique, comme le prouve sa déclaration suivante :

```
public String format(String format, Object... args)
```

Les mêmes règles s'appliquent donc. Les fonctions qui prennent un nombre variable d'arguments peuvent être unaires, diadiques ou même triadiques. Cependant, ce serait une erreur de leur donner un plus grand nombre d'arguments.

```
void monad(Integer... args);
void dyad(String name, Integer... args);
void triad(String name, int count, Integer... args);
```

Verbes et mots-clés

Le choix de noms appropriés pour les fonctions permet de faire un long chemin vers l'explication des objectifs de la fonction et vers l'ordre et les objectifs des arguments. Dans le cas d'une forme unaire, la fonction et l'argument doivent représenter un couple verbe/nom parfaitement associé. Par exemple, `write(name)` est très évocateur. Quelle que soit cette chose "nommée", elle est "écrite". Un nom encore meilleur pourrait être `writeField(name)`, qui indique que la chose "nommée" est un "champ".

Ce dernier cas est un exemple de nom de fonction de type *mot-clé*. Dans cette forme, nous encodons les noms des arguments dans le nom de la fonction. Par exemple, il serait préférable d'écrire `assertEquals` sous la forme `assertExpectedEquals Actual(expected, actual)`. Le problème de mémorisation de l'ordre des arguments s'en trouverait ainsi allégé.

Éviter les effets secondaires

Les effets secondaires sont des mensonges. Votre fonction promet de faire une chose, alors qu'elle fait également d'autres choses *cachées*. Parfois, elle apportera des modifications inattendues aux variables de sa propre classe. Parfois, elle les apportera aux paramètres passés à la fonction ou aux variables globales du système. Ces deux cas sont retors et des mensonges préjudiciables qui conduisent souvent à des couplages temporels étranges et à des dépendances d'ordre.

Prenons par exemple la fonction *a priori* inoffensive du Listing 3.6. Elle utilise un algorithme classique pour mettre en correspondance un nom d'utilisateur (`userName`) et un mot de passe (`password`). Elle retourne `true` s'ils correspondent, `false` en cas de problème. Elle dissimule cependant un effet secondaire. Pouvez-vous le trouver ?

Listing 3.6 : `UserValidator.java`

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
        return true;
      }
    }
    return false;
  }
}
```

L'effet secondaire se trouve dans l'appel `Session.initialize()`. Le nom de la fonction `checkPassword` indique qu'elle vérifie le mot de passe. Ce nom n'implique absolument pas l'initialisation de la session. Par conséquent, si l'appelant fait confiance au nom de la fonction, il risque d'écraser les données de session existantes lorsqu'il décide de vérifier la validité de l'utilisateur.

Cet effet secondaire crée un couplage temporel. Autrement dit, `checkPassword` ne peut être invoquée qu'à certains moments, lorsque la session peut être initialisée en toute sécurité. Si elle n'est pas invoquée au bon moment, les données de session peuvent être perdues accidentellement. Les couplages temporels sont déroutants, en particulier lorsqu'ils sont la conséquence d'un effet secondaire. Si vous devez créer un couplage temporel, indiquez-le clairement dans le nom de la fonction. Dans ce cas, nous pouvons renommer la fonction `checkPasswordAndInitializeSession`, mais cela ne respecte pas la règle "Faire une seule chose".

Arguments de sortie

Les arguments sont naturellement interprétés comme les *entrées* d'une fonction. Si vous programmez depuis plusieurs années, je suis sûr que vous y avez regardé à deux fois lorsqu'un argument était en réalité une *sortie* à la place d'une entrée. Prenons l'exemple suivant :

```
appendFooter(s);
```

Cette fonction ajoute-t-elle `s` à la fin de quelque chose, ou ajoute-t-elle quelque chose à la fin de `s` ? L'argument `s` est-il une entrée ou une sortie ? Il ne faut pas longtemps pour examiner la signature de la fonction :

```
public void appendFooter(StringBuffer report)
```

Elle clarifie la question, mais il faut prendre la peine de vérifier la déclaration de la fonction. Tout ce qui vous oblige à vérifier la signature de la fonction équivaut à une hésitation. Il s'agit d'une coupure cognitive qui doit être évitée.

Avant la programmation orientée objet, les arguments de sortie étaient indispensables. Ce besoin a quasiment disparu dans les langages orientés objet car `this` est *conçu* pour jouer le rôle d'un argument de sortie. Autrement dit, il serait mieux d'invoquer `appendFooter` de la manière suivante :

```
report.appendFooter();
```

En général, les arguments de sortie sont à proscrire. Si votre fonction doit modifier l'état de quelque chose, faites en sorte que ce soit l'état de l'objet auquel elle appartient.

Séparer commandes et demandes

Les fonctions doivent soit faire quelque chose, soit répondre à quelque chose, non les deux. Une fonction doit modifier l'état d'un objet ou retourner des informations concernant cet objet. En faisant les deux, elle amène une confusion. Prenons l'exemple de la fonction suivante :

```
public boolean set(String attribute, String value);
```

Elle fixe la valeur d'un attribut nommé et retourne `true` en cas de succès et `false` si l'attribut n'existe pas. Cela conduit à des instructions étranges :

```
if (set("nomUtilisateur", "OncleBob"))...
```

Mettez-vous à la place du lecteur. Que signifie cette instruction ? Demande-t-elle si l'attribut `"nomUtilisateur"` était précédemment fixée à `"OncleBob"` ou si cet attribut a pu être fixé à `"OncleBob"` ? Il est difficile de déduire l'objectif à partir de l'appel car le terme `"set"` peut être un verbe ou un adjectif.

L'auteur a employé `set` comme un verbe, mais, dans le contexte de l'instruction `if`, il ressemble plus à un adjectif. L'instruction se traduit donc "si l'attribut `nomUtilisateur` était précédemment fixé à `OncleBob`", non "fixer l'attribut `username` à `OncleBob` et si cela a réussi alors...". Nous pouvons essayer de résoudre la question en renommant la fonction `set` en `setAndCheckIfExists`, mais cela n'aide en rien la lisibilité de l'instruction `if`. La solution réelle consiste à séparer la commande de la demande afin d'écarter toute ambiguïté.

```
if (attributeExists("nomUtilisateur")) {
  setAttribute("nomUtilisateur", "OncleBob");
  ...
}
```

Préférer les exceptions au retour de codes d'erreur

Le retour de codes d'erreur à partir de fonctions de commande est une violation subtile de la séparation des commandes et des demandes. Elle incite à employer des commandes comme des expressions dans les prédicats des instructions `if`.

```
if (deletePage(page) == E_OK)
```

Cet exemple ne souffre pas de la confusion verbe/adjectif, mais conduit à des structures profondément imbriquées. Lorsqu'un code d'erreur est retourné, nous obligeons l'appelant à traiter immédiatement l'erreur.

```
if (deletePage(page) == E_OK) {
  if (registry.deleteReference(page.name) == E_OK) {
    if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
      logger.log("page deleted");
```

```
      } else {
        logger.log("configKey not deleted");
      }
    } else {
      logger.log("deleteReference from registry failed");
    }
  } else {
    logger.log("delete failed");
    return E_ERROR;
  }
```

En revanche, si nous utilisons des exceptions au lieu de retourner des codes d'erreur, le traitement de l'erreur peut alors être séparé du code principal et simplifié :

```
try {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
  logger.log(e.getMessage());
}
```

Extraire les blocs *try/catch*

Les blocs `try`/`catch` sont intrinsèquement laids. Ils perturbent la structure du code et mélangent le traitement des erreurs au code normal. Il est donc préférable d'extraire le corps des blocs `try` et `catch` pour les placer dans leurs propres fonctions.

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
}

private void deletePageAndAllReferences(Page page) throws Exception {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}

private void logError(Exception e) {
  logger.log(e.getMessage());
}
```

Dans cet exemple, la fonction `delete` s'occupe du traitement des erreurs. Elle est facile à comprendre, puis à ignorer. La fonction `deletePageAndAllReferences` concerne exclusivement la suppression d'une page. Le traitement des erreurs peut être ignoré. Nous obtenons ainsi une séparation élégante qui facilite la compréhension et la modification du code.

Traiter les erreurs est une chose

Les fonctions doivent faire une chose. Le traitement des erreurs est une chose. Par conséquent, une fonction qui traite les erreurs ne doit rien faire d'autre. Cela implique que, comme dans l'exemple précédent, si le mot-clé `try` est présent dans une fonction, il doit être le tout premier mot de cette fonction et il ne doit rien n'y avoir après les blocs `catch`/`finally`.

L'aimant à dépendances *Error.java*

Lorsque des codes d'erreur sont retournés, cela implique généralement qu'une classe ou une énumération définit tous les codes reconnus.

```
public enum Error {
  OK,
  INVALID,
  NO_SUCH,
  LOCKED,
  OUT_OF_RESOURCES,
  WAITING_FOR_EVENT;
}
```

Ces classes sont de véritables *aimants à dépendances* ; de nombreuses autres classes doivent les importer et les employer. Ainsi, dès que l'énumération `Error` change, toutes ces autres classes doivent être recompilées et redéployées[11]. Il existe donc une pression négative sur la classe `Error`. Les programmeurs ne veulent pas ajouter de nouvelles erreurs car ils doivent alors tout recompiler et tout redéployer. Par conséquent, ils réutilisent d'anciens codes d'erreur au lieu d'en ajouter de nouveaux.

En utilisant des exceptions à la place des codes d'erreur, les nouvelles exceptions *dérivent* de la classe `Exception`. Elles peuvent donc être ajoutées sans passer par une recompilation ou un redéploiement[12].

11. Certains ont pensé pouvoir éviter la recompilation et le redéploiement. Ils ont été identifiés et traités.
12. Il s'agit d'un exemple du principe ouvert/fermé [PPP].

Ne vous répétez pas[13]

En examinant attentivement le Listing 3.1, vous noterez qu'un algorithme est répété quatre fois, dans chacun des cas `SetUp`, `SuiteSetUp`, `TearDown` et `SuiteTearDown`. Cette redondance n'est pas facile à repérer car les quatre occurrences sont mélangées au code restant de manière non uniforme. La redondance pose problème car elle fait grossir le code et exige des modifications en quatre endroits si l'algorithme vient à évoluer. Elle représente également quatre sources d'erreur par omission.

Pour corriger cette redondance, nous avons employé la méthode `include` dans le Listing 3.7. Lisez à nouveau ce code et voyez combien la lisibilité de l'intégralité du module s'est améliorée en réduisant le code dupliqué.

Dans un logiciel, la redondance peut constituer la source du mal. Plusieurs principes et pratiques ont été imaginés dans le seul but de la contrôler ou de l'éliminer. Par exemple, les formes normales de Codd pour les bases de données ont pour objectif de supprimer la redondance dans les données. De même, la programmation orientée objet permet de concentrer dans des classes de base un code qui serait sinon redondant. La programmation structurée, la programmation orientée aspect ou la programmation orientée composant sont toutes des stratégies destinées en partie à éliminer la redondance. Depuis l'invention des sous-routines, les innovations en développement logiciel se sont attachées à supprimer la redondance dans le code source.

Programmation structurée

Certains programmeurs suivent les règles de programmation structurée d'Edsger Dijkstra [SP72]. Dijkstra stipule que chaque fonction, et chaque bloc dans une fonction, doit posséder une entrée et une sortie. Pour respecter ces règles, il ne doit y avoir qu'une seule instruction `return` dans une fonction, aucune instruction `break` ou `continue` dans une boucle et jamais, au grand jamais, d'instruction `goto`.

Même si nous sommes bien disposés envers les objectifs et les disciplines de la programmation structurée, ces règles ont peu d'intérêt lorsque les fonctions sont très courtes. Elles ne valent que dans les fonctions très longues.

13. Le principe DRY (*Don't Repeat Yourself*) [PRAG].

Par conséquent, si vos fonctions restent courtes, les multiples instructions `return`, `break` ou `continue` occasionnelles ne seront pas vraiment un problème. Elles peuvent même parfois être plus expressives que la règle une seule entrée et une seule sortie. En revanche, les instructions `goto` ne présentent un intérêt que dans les longues fonctions et doivent donc être évitées.

Écrire les fonctions de la sorte

L'écriture d'un logiciel ressemble à n'importe quelle autre sorte d'écriture. Lorsque nous écrivons un article, nous commençons par coucher nos idées, puis nous les triturons jusqu'à obtenir une lecture fluide. Le premier bouillon est souvent maladroit et mal organisé. Par conséquent, nous travaillons le texte, le restructurons et le remanions jusqu'à ce qu'il se lise comme nous le souhaitons.

Lorsque j'écris des fonctions, elles sont initialement longues et complexes. Elles contiennent de nombreuses instructions et boucles imbriquées. Leur liste d'arguments est longue, leurs noms sont arbitraires et elles contiennent du code redondant. Mais je dispose également d'une suite de tests unitaires qui couvrent chacune de ces lignes de code grossières.

Ensuite, je triture et remanie ce code, en décomposant les fonctions, en modifiant des noms et en éliminant la redondance. Je réduis les méthodes et les réordonne. Parfois, j'éclate même des classes entières, tout en faisant en sorte que les tests réussissent.

Au final, j'obtiens des fonctions qui respectent les règles établies dans ce chapitre. Je ne les écris pas directement de la sorte. Je ne pense pas que quiconque puisse y parvenir.

Conclusion

Tous les systèmes sont construits à partir d'un langage propre à un domaine et conçu par le programmeur pour décrire ce système. Les fonctions représentent les verbes de ce langage, les classes en sont les groupes nominaux. Cela ne constitue en rien un retour à l'ancienne notion hideuse que les groupes nominaux et les verbes dans un document d'exigence représentent la première estimation des classes et des fonctions d'un système. À la place, il s'agit d'une vérité beaucoup plus ancienne. L'art de la programmation est, et a toujours été, l'art de la conception du langage.

Les programmeurs experts voient les systèmes comme des histoires à raconter, non comme des programmes à écrire. Ils se servent des possibilités du langage de programmation choisi pour construire un langage plus riche et plus expressif, qui peut être employé pour raconter cette histoire. Une partie de ce langage spécifique est composée par la hiérarchie de fonctions qui décrivent toutes les actions ayant lieu au sein du système. En

une opération astucieuse de récursion, ces actions sont écrites de manière à utiliser le langage très spécifique qu'elles définissent pour raconter leur propre partie de l'histoire.

Ce chapitre s'est focalisé sur la bonne écriture des fonctions. Si vous suivez les règles établies, vos fonctions seront courtes, bien nommées et parfaitement organisées. Mais n'oubliez jamais que le véritable objectif est de raconter l'histoire du système et que les fonctions écrites doivent s'unir proprement en un langage clair et précis pour vous aider dans votre propos.

SetupTeardownIncluder

Listing 3.7 : `SetupTeardownIncluder.java`

```
package fitnesse.html;

import fitnesse.responders.run.SuiteResponder;
import fitnesse.wiki.*;

public class SetupTeardownIncluder {
  private PageData pageData;
  private boolean isSuite;
  private WikiPage testPage;
  private StringBuffer newPageContent;
  private PageCrawler pageCrawler;


  public static String render(PageData pageData) throws Exception {
    return render(pageData, false);
  }

  public static String render(PageData pageData, boolean isSuite)
    throws Exception {
    return new SetupTeardownIncluder(pageData).render(isSuite);
  }

  private SetupTeardownIncluder(PageData pageData) {
    this.pageData = pageData;
    testPage = pageData.getWikiPage();
    pageCrawler = testPage.getPageCrawler();
    newPageContent = new StringBuffer();
  }

  private String render(boolean isSuite) throws Exception {
    this.isSuite = isSuite;
    if (isTestPage())
      includeSetupAndTeardownPages();
    return pageData.getHtml();
  }

  private boolean isTestPage() throws Exception {
    return pageData.hasAttribute("Test");
  }
```

```
private void includeSetupAndTeardownPages() throws Exception {
  includeSetupPages();
  includePageContent();
  includeTeardownPages();
  updatePageContent();
}

private void includeSetupPages() throws Exception {
  if (isSuite)
    includeSuiteSetupPage();
  includeSetupPage();
}

private void includeSuiteSetupPage() throws Exception {
  include(SuiteResponder.SUITE_SETUP_NAME, "-setup");
}

private void includeSetupPage() throws Exception {
  include("SetUp", "-setup");
}

private void includePageContent() throws Exception {
  newPageContent.append(pageData.getContent());
}

private void includeTeardownPages() throws Exception {
  includeTeardownPage();
  if (isSuite)
    includeSuiteTeardownPage();
}

private void includeTeardownPage() throws Exception {
  include("TearDown", "-teardown");
}

private void includeSuiteTeardownPage() throws Exception {
  include(SuiteResponder.SUITE_TEARDOWN_NAME, "-teardown");
}

private void updatePageContent() throws Exception {
  pageData.setContent(newPageContent.toString());
}

private void include(String pageName, String arg) throws Exception {
  WikiPage inheritedPage = findInheritedPage(pageName);
  if (inheritedPage != null) {
    String pagePathName = getPathNameForPage(inheritedPage);
    buildIncludeDirective(pagePathName, arg);
  }
}

private WikiPage findInheritedPage(String pageName) throws Exception {
  return PageCrawlerImpl.getInheritedPage(pageName, testPage);
}
```

```
  private String getPathNameForPage(WikiPage page) throws Exception {
    WikiPagePath pagePath = pageCrawler.getFullPath(page);
    return PathParser.render(pagePath);
  }

  private void buildIncludeDirective(String pagePathName, String arg) {
    newPageContent
      .append("\n!include ")
      .append(arg)
      .append(" .")
      .append(pagePathName)
      .append("\n");
  }
}
```

4

Commentaires

Ne commentez pas le mauvais code, récrivez-le.

— Brian W. Kernighan et P. J. Plaugher [KP78, p. 144]

Rien ne peut être plus utile qu'un commentaire bien placé. Rien ne peut encombrer un module autant que des commentaires dogmatiques sans importance. Rien ne peut être plus préjudiciable qu'un ancien commentaire obsolète qui propage mensonges et désinformation.

Les commentaires ne sont pas comme la liste de Schindler. Ils ne représentent pas le "bien parfait". Les commentaires sont, au mieux, un mal nécessaire. Si nos langages de programmation étaient suffisamment expressifs ou si nous avions suffisamment de talent pour les manier de manière à exprimer nos intentions, nous n'aurions pas souvent besoin des commentaires, voire jamais.

Les commentaires sont bien employés lorsqu'ils pallient notre incapacité à exprimer nos intentions par le code. Notez que j'emploie le mot *incapacité*, car les commentaires masquent toujours nos échecs. Nous en avons besoin car nous ne parvenons pas toujours à nous exprimer sans eux, mais nous ne devons pas être fiers de les utiliser.

Par conséquent, lorsque vous éprouvez le besoin d'écrire un commentaire, réfléchissez et essayez de trouver une solution pour exprimer vos intentions avec du code. Chaque fois que le code parvient à transmettre vos intentions, vous pouvez vous féliciter. Chaque fois que vous écrivez un commentaire, faites la grimace et ressentez le poids de l'échec.

Pourquoi un tel refus des commentaires ? Simplement parce qu'ils mentent. Pas toujours, pas intentionnellement, mais beaucoup trop souvent. Plus un commentaire est ancien et placé loin du code qu'il décrit, plus il risque d'être totalement faux. La raison en est simple. En toute bonne foi, les programmeurs ne peuvent pas les maintenir.

Le code change et évolue. Des pans entiers peuvent être déplacés d'un endroit à un autre. Ces morceaux se séparent, se reproduisent et se réunissent à nouveau pour former des chimères. Malheureusement, les commentaires ne suivent pas toujours ces modifications, ou ne peuvent tout simplement pas. Trop souvent, ils sont séparés du code qu'ils décrivent et se transforment en un texte orphelin dont la justesse s'étiole au fur et à mesure. Voyez, par exemple, ce qui s'est passé avec le commentaire suivant et la ligne qu'il est censé décrire :

```
MockRequest request;
private final String HTTP_DATE_REGEXP =
  "[SMTWF][a-z]{2}\\,\\s[0-9]{2}\\s[JFMASOND][a-z]{2}\\s"+
  "[0-9]{4}\\s[0-9]{2}\\:[0-9]{2}\\:[0-9]{2}\\sGMT";
private Response response;
private FitNesseContext context;
private FileResponder responder;
private Locale saveLocale;
// Exemple : "Tue, 02 Apr 2003 22:18:49 GMT"
```

Des variables d'instance ont probablement été ajoutées à un moment donné et se sont intercalées entre la constante `HTTP DATE REGEXP` et son commentaire.

Je suis d'accord, les programmeurs devraient être suffisamment disciplinés pour garder des commentaires à jour, pertinents et justes. Cependant, cette énergie est mieux

employée lorsqu'elle permet d'obtenir un code tellement clair et explicite que les commentaires en deviennent inutiles.

Les commentaires inexacts font beaucoup plus de mal que l'absence de commentaires. Ils trompent et induisent en erreur. Ils fixent des attentes qui ne seront jamais satisfaites. Ils établissent d'anciennes règles qui n'ont plus besoin d'être suivies ou qui ne doivent pas l'être.

La vérité doit se trouver uniquement dans le code. Seul le code peut indiquer réellement ce qu'il fait. Il représente la seule source d'informations absolument justes. Par conséquent, bien que les commentaires puissent parfois être nécessaires, il est indispensable d'œuvrer à leur extinction.

Ne pas compenser le mauvais code par des commentaires

Très souvent, le mauvais code semble justifier les commentaires. Nous écrivons un module et nous savons qu'il n'est pas très clair et mal organisé. Nous savons qu'il est désordonné. Alors, nous nous disons : "Oh, je ferais mieux de commenter ce code !" Non ! Il est préférable de le nettoyer !

Un code clair et expressif avec peu de commentaires est bien supérieur à un code encombré et complexe avec de nombreux commentaires. Au lieu de passer du temps à écrire les commentaires qui expliquent le désordre créé, il est préférable de le passer à nettoyer ce fouillis.

S'expliquer dans le code

Il arrive que le code ne parvienne pas à véhiculer parfaitement les explications. Malheureusement, de nombreux programmeurs s'en sont fait une religion et considèrent que le code est rarement, voire jamais, un bon moyen d'explication. C'est ouvertement faux. Quel code préférez-vous voir ? Celui-ci ?

```
// Vérifier si l'employé peut bénéficier de tous les avantages.
if ((employee.flags & HOURLY_FLAG) &&
    (employee.age > 65))
```

Ou celui-ci ?

```
if (employee.isEligibleForFullBenefits())
```

En général, il ne faut pas plus de quelques secondes de réflexion pour expliquer la majorité des intentions dans le code. Dans la plupart des cas, il suffit simplement de créer une fonction qui exprime la même chose que le commentaire imaginé.

Bons commentaires

Certains commentaires sont indispensables ou bénéfiques. Nous allons en examiner quelques-uns qui valent pleinement les quelques octets qu'ils occupent. Cependant, n'oubliez pas que le meilleur commentaire est celui que vous pouvez éviter d'écrire.

Commentaires légaux

Les conventions de codage de notre entreprise nous obligent parfois à écrire certains commentaires pour des raisons légales. Par exemple, les déclarations de copyright et de propriété sont des éléments nécessaires et que l'on peut raisonnablement placer dans un commentaire au début de chaque fichier source.

Voici par exemple un extrait du commentaire que je place au début de chaque fichier source dans FitNesse. Par ailleurs, mon IDE masque automatiquement ce commentaire en le réduisant, ce qui lui évite d'encombrer le code.

```
// Copyright (C) 2003,2004,2005 par Object Mentor, Inc. Tous droits réservés.
// Publié sous les termes de la Licence Publique Générale de GNU version 2
// ou ultérieure.
```

Ces commentaires ne doivent pas être des contrats ou des dispositions juridiques. Lorsque c'est possible, faites référence à une licence standard ou à un autre document externe au lieu de placer les termes et les conditions dans le commentaire.

Commentaires informatifs

Il est parfois utile de fournir des informations de base à l'aide d'un commentaire. Par exemple, prenons le commentaire suivant, qui explique la valeur de retour d'une méthode abstraite :

```
// Retourne une instance du Responder en cours de test.
protected abstract Responder responderInstance();
```

Si ce type de commentaires se révèle parfois utile, il est préférable, si possible, d'employer le nom de la fonction pour transmettre des informations. Par exemple, dans le cas précédent, le commentaire pourrait devenir redondant si la fonction était renommée `responderBeingTested`.

Voici un commentaire un peu plus intéressant :

```
// Format reconnu kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile(
  "\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

Dans ce cas, le commentaire nous informe que l'expression régulière est utilisée pour reconnaître une heure et une date qui ont été mises en forme avec la fonction `Simple DateFormat.format` en utilisant la chaîne de format indiquée. Néanmoins, il aurait été

préférable, et plus clair, de déplacer ce code dans une classe particulière qui sert à convertir les formats de date et d'heure. Le commentaire serait alors devenu superflu.

Expliquer les intentions

Un commentaire ne se limite pas toujours à donner des informations utiles concernant l'implémentation, mais explique les raisons d'une décision. Le cas suivant montre un choix intéressant expliqué par un commentaire. Lors de la comparaison de deux objets, l'auteur a décidé que le tri des objets devait placer ceux de sa classe devant ceux des autres classes.

```
public int compareTo(Object o)
{
  if(o instanceof WikiPagePath)
  {
    WikiPagePath p = (WikiPagePath) o;
    String compressedName = StringUtil.join(names, "");
    String compressedArgumentName = StringUtil.join(p.names, "");
    return compressedName.compareTo(compressedArgumentName);
  }
  return 1; // Nous sommes supérieurs car nous sommes du bon type.
}
```

Voici un meilleur exemple. Vous ne serez peut-être pas d'accord avec la solution retenue par le programmeur, mais vous savez au moins ce qu'il tente de réaliser.

```
public void testConcurrentAddWidgets() throws Exception {
  WidgetBuilder widgetBuilder =
    new WidgetBuilder(new Class[]{BoldWidget.class});
  String text = "'''bold text'''";
  ParentWidget parent =
    new BoldWidget(new MockWidgetRoot(), "'''bold text'''");
  AtomicBoolean failFlag = new AtomicBoolean();
  failFlag.set(false);

  // Voici notre meilleure solution pour obtenir une condition de concurrence.
  // Nous créons un grand nombre de threads.
  for (int i = 0; i < 25000; i++) {
    WidgetBuilderThread widgetBuilderThread =
      new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);
    Thread thread = new Thread(widgetBuilderThread);
    thread.start();
  }
  assertEquals(false, failFlag.get());
}
```

Clarifier

Il est souvent utile de convertir la signification obscure d'un argument ou d'une valeur de retour en quelque chose de parfaitement lisible. En général, il est préférable de trouver une solution pour qu'un argument ou une valeur de retour clarifie de lui-même sa

signification. Cependant, lorsqu'ils font partie de la bibliothèque standard ou d'un code que nous ne pouvons pas modifier, il faut alors se tourner vers un commentaire de clarification.

```
public void testCompareTo() throws Exception
{
  WikiPagePath a = PathParser.parse("PageA");
  WikiPagePath ab = PathParser.parse("PageA.PageB");
  WikiPagePath b = PathParser.parse("PageB");
  WikiPagePath aa = PathParser.parse("PageA.PageA");
  WikiPagePath bb = PathParser.parse("PageB.PageB");
  WikiPagePath ba = PathParser.parse("PageB.PageA");

  assertTrue(a.compareTo(a) == 0);    // a == a
  assertTrue(a.compareTo(b) != 0);    // a != b
  assertTrue(ab.compareTo(ab) == 0);  // ab == ab
  assertTrue(a.compareTo(b) == -1);   // a < b
  assertTrue(aa.compareTo(ab) == -1); // aa < ab
  assertTrue(ba.compareTo(bb) == -1); // ba < bb
  assertTrue(b.compareTo(a) == 1);    // b > a
  assertTrue(ab.compareTo(aa) == 1);  // ab > aa
  assertTrue(bb.compareTo(ba) == 1);  // bb > ba
}
```

Il existe, bien entendu, un risque non négligeable qu'un commentaire de clarification soit erroné. Si vous examinez l'exemple précédent, vous constatez qu'il est difficile de vérifier que tous les commentaires sont justes. Il montre pourquoi la clarification est nécessaire et pourquoi elle est risquée. Par conséquent, avant d'écrire ce genre de commentaires, recherchez une meilleure solution et vérifiez bien qu'ils sont corrects.

Avertir des conséquences

Dans certains cas, il ne sera pas inutile d'avertir les autres programmeurs quant aux conséquences possibles. Par exemple, voici un commentaire qui explique pourquoi un cas de test a été désactivé :

```
// Ne pas exécuter, à moins que
// vous ayez du temps à tuer.
public void _testWithReallyBigFile()
{
  writeLinesToFile(10000000);

  response.setBody(testFile);
  response.readyToSend(this);
  String responseString = output.toString();
  assertSubString("Content-Length: 1000000000", responseString);
  assertTrue(bytesSent > 1000000000);
}
```

Aujourd'hui, nous désactiverions le cas de test à l'aide de l'attribut `@Ignore` et d'une chaîne d'explication appropriée, comme `@Ignore("Exécution trop longue")`. Mais, avant l'arrivée de JUnit 4, la convention courante était de placer un caractère souligné ( ) en début de nom de la méthode. Le commentaire, quoique désinvolte, souligne parfaitement le fait.

Voici un autre exemple :

```
public static SimpleDateFormat makeStandardHttpDateFormat()
{
  // SimpleDateFormat n'étant pas sûr vis-à-vis des threads,
  // nous devons créer chaque instance indépendamment.
  SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM  yyyy HH:mm:ss z");
  df.setTimeZone(TimeZone.getTimeZone("GMT"));
  return df;
}
```

Vous pourriez faire remarquer qu'il existe de meilleures solutions pour résoudre ce problème. C'est également mon avis. Cependant, le commentaire ajouté est parfaitement raisonnable. Il empêchera le programmeur trop empressé d'utiliser un initialiseur statique pour tenter d'améliorer l'efficacité.

Commentaires *TODO*

Les commentaires `//TODO`, qui permettent de laisser des notes du type "À faire", sont parfois pleinement justifiés. Dans le cas suivant, le commentaire `TODO` explique pourquoi l'implémentation de la fonction n'est pas terminée et ce qu'elle sera ultérieurement.

```
//TODO-MdM ce n'est pas nécessaire.
// Elle devrait disparaître lorsque nous implémenterons le modèle d'extraction.
protected VersionInfo makeVersion() throws Exception
{
  return null;
}
```

Ces commentaires correspondent à des travaux qui, selon le programmeur, devront être réalisés, mais qu'il est impossible de faire actuellement pour une raison ou pour une autre. Ils peuvent rappeler au programmeur de ne pas oublier de supprimer une fonctionnalité obsolète ou à une autre personne d'examiner un problème. Ils peuvent demander à quelqu'un de réfléchir à un meilleur nom ou rappeler qu'une modification dépendante d'un événement planifié devra être effectuée. Quelle que soit la raison d'être d'un commentaire `TODO`, elle ne doit pas justifier la présence d'un mauvais code dans le système.

La plupart des bons IDE actuels disposent d'une fonction qui permet de localiser tous les commentaires `TODO`. Par conséquent, il est peu probable qu'ils soient perdus. Néanmoins, votre code ne doit pas être jonché de tels commentaires, et vous devez les parcourir régulièrement pour les éliminer au fur et à mesure.

Amplifier

Un commentaire peut servir à amplifier l'importance d'un point qui, sinon, pourrait paraître sans conséquence.

```
String listItemContent = match.group(3).trim();
// L'appel à trim est très important. Il retire les espaces
// de tête qui pourraient faire croire que l'élément est
// une autre liste.
new ListItemWidget(this, listItemContent, this.level + 1);
return buildList(text.substring(match.end()));
```

Documentation Javadoc dans les API publiques

Rien n'est plus utile et satisfaisant que des API publiques parfaitement décrites. La documentation Javadoc dans la bibliothèque Java standard est un cas d'espèce. Sans elle, il serait difficile, voire impossible, d'écrire des programmes Java.

Si vous développez une API publique, vous allez certainement rédiger la documentation Javadoc correspondante. Cependant, n'oubliez pas les autres conseils de ce chapitre. Comme n'importe quel autre commentaire, cette documentation peut être mensongère, mal placée et malhonnête.

Mauvais commentaires

La majorité des commentaires entre dans cette catégorie. Ils servent généralement de béquilles ou d'excuses à du mauvais code, ou bien ils justifient des choix insuffisants, comme si le programmeur se parlait à lui-même.

Marmonner

Si vous ajoutez un commentaire simplement parce que vous pensez que c'est nécessaire ou que le processus l'exige, il s'agit ni plus ni moins d'une bidouille. Si vous décidez d'écrire un commentaire, vous devez passer suffisamment de temps pour vous assurer qu'il soit parfaitement rédigé.

Voici, par exemple, un cas extrait de FitNesse où un commentaire aurait pu être utile, mais l'auteur devait être pressé ou peu attentif. Son marmonnement tient beaucoup de l'énigme :

```
public void loadProperties()
{
  try
  {
    String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
    FileInputStream propertiesStream = new FileInputStream(propertiesPath);
    loadedProperties.load(propertiesStream);
  }
```

```
    catch(IOException e)
    {
      // L'absence de fichiers de propriétés signifie que toutes
      // les valeurs par défaut sont chargées.
    }
  }
```

Que signifie ce commentaire dans le bloc `catch` ? Il a manifestement un sens pour l'auteur, mais sa signification ne transparaît pas vraiment. Apparemment, la réception d'une exception `IOException` signifie qu'il n'y a aucun fichier de propriétés ; dans ce cas, toutes les valeurs par défaut sont chargées. Cependant, qui s'occupe de charger toutes les valeurs par défaut ? Cette opération se fait-elle avant l'appel à `loadedProperties.load` ? Ou bien l'invocation de `loadedProperties.load` intercepte-t-elle l'exception, charge les propriétés par défaut, puis nous transmet l'exception ? Ou bien la méthode `loadedProperties.load` charge-t-elle toutes les valeurs par défaut avant de tenter de charger le fichier ? L'auteur était-il embêté de laisser un bloc `catch` vide ? Ou bien, et cela fait peur, l'auteur voulait-il indiquer qu'il lui faudrait revenir plus tard sur ce code et écrire le chargement des valeurs par défaut ?

Notre seul recours est d'examiner le code dans les autres parties du système afin de déterminer ce qui se passe. Un commentaire qui nous oblige à consulter d'autres modules pour comprendre sa signification est un commentaire qui ne parvient pas à communiquer et qui ne vaut pas les quelques octets qu'il consomme.

Commentaires redondants

Le Listing 4.1 présente une fonction simple dont le commentaire d'en-tête est totalement redondant. Il faut presque plus de temps pour lire le commentaire que le code lui-même.

Listing 4.1 : `waitForClose`

```
// Méthode utilitaire qui se termine lorsque this.closed vaut true. Elle lance
// une exception lorsque la temporisation est écoulée.
public synchronized void waitForClose(final long timeoutMillis)
throws Exception
{
  if(!closed)
  {
    wait(timeoutMillis);
    if(!closed)
      throw new Exception("MockResponseSender could not be closed");
  }
}
```

Quel est l'objectif de ce commentaire ? Il ne donne pas plus d'informations que le code. Il ne justifie pas le code ni ne fournit d'intentions ou de raisons. Il n'est pas plus facile à lire que le code. Il est moins précis que le code et tente de faire accepter au lecteur ce

manque de précision, à défaut d'une bonne compréhension. Nous pourrions le comparer à un vendeur de voitures d'occasion enthousiaste qui vous assure qu'il est inutile de regarder sous le capot.

Examinons à présent la documentation Javadoc abondante et redondante extraite de Tomcat (voir Listing 4.2). Ces commentaires ne font qu'encombrer et embrouiller le code. Ils n'ont aucun rôle documentaire. Pire encore, je n'ai montré que les premiers ; le module en contient beaucoup d'autres.

Listing 4.2 : `ContainerBase.java` **(Tomcat)**

```
public abstract class ContainerBase
  implements Container, Lifecycle, Pipeline,
  MBeanRegistration, Serializable {

  /**
   * Délai de traitement de ce composant.
   */
  protected int backgroundProcessorDelay = -1;


  /**
   * Prise en charge d'un événement du cycle de vie de ce composant.
   */
  protected LifecycleSupport lifecycle =
    new LifecycleSupport(this);


  /**
   * Les auditeurs d'événements de conteneur pour ce Container.
   */
  protected ArrayList listeners = new ArrayList();


  /**
   * L'implémentation de Loader à laquelle ce Container est associé.
   */
  protected Loader loader = null;


  /**
   * L'implémentation de Logger à laquelle ce Container est associé.
   */
  protected Log logger = null;


  /**
   * Nom du journal associé.
   */
  protected String logName = null;
```

```
/**
 * L'implémentation de Manager à laquelle ce Container est associé.
 */
protected Manager manager = null;


/**
 * Le cluster auquel ce Container est associé.
 */
protected Cluster cluster = null;


/**
 * Nom de ce Container.
 */
protected String name = null;


/**
 * Le Container parent dont ce Container est un enfant.
 */
protected Container parent = null;


/**
 * Le chargeur de classes parent qui doit être configuré lorsque
 * nous installons un Loader.
 */
protected ClassLoader parentClassLoader = null;


/**
 * L'objet Pipeline auquel ce Container est associé.
 */
protected Pipeline pipeline = new StandardPipeline(this);


/**
 * Le Realm auquel ce Container est associé.
 */
protected Realm realm = null;


/**
 * L'objet de ressources DirContext auquel ce Container est associé.
 */
protected DirContext resources = null;
```

Commentaires trompeurs

Malgré toutes ses bonnes intentions, un programmeur peut parfois inclure dans ses commentaires une déclaration qui n'est pas suffisamment précise pour être juste. Revenons sur le commentaire redondant mais également trompeur du Listing 4.1.

Voyez-vous en quoi ce commentaire est trompeur ? La méthode ne se termine pas *lorsque* `this.closed` devient `true`, mais *si* `this.closed` est égal à `true` ; sinon elle attend pendant un certain temps, puis lance une exception *si* `this.closed` n'est toujours pas égal à `true`.

Cette petite désinformation, donnée dans un commentaire plus difficile à lire que le corps du code, peut conduire un autre programmeur à appeler cette fonction en pensant qu'elle retournera dès que `this.closed` devient `true`. Ce pauvre programmeur devra passer par une session de débogage pour tenter de comprendre l'origine de la lenteur de son code.

Commentaires obligés

La règle stipulant que chaque fonction doit disposer d'une documentation Javadoc est totalement stupide, tout comme celle obligeant à commenter chaque variable. Les commentaires de ce type ne font qu'encombrer le code, propager des mensonges et conduire à une confusion et à une désorganisation générales.

Par exemple, exiger une documentation Javadoc pour chaque fonction mène à des aberrations telles que celle illustrée par le Listing 4.3. Cet encombrement n'apporte rien, ne fait qu'obscurcir le code et crée une source de mensonges et de distraction.

Listing 4.3

```
/**
 *
 * @param title Le titre du CD.
 * @param author L'auteur du CD.
 * @param tracks Le nombre de morceaux sur le CD.
 * @param durationInMinutes La durée du CD en minutes.
 */
public void addCD(String title, String author,
                  int tracks, int durationInMinutes) {
  CD cd = new CD();
  cd.title = title;
  cd.author = author;
  cd.tracks = tracks;
  cd.duration = duration;
  cdList.add(cd);
}
```

Commentaires de journalisation

Les développeurs ajoutent parfois un commentaire au début de chaque module lorsqu'ils le modifient. Ces commentaires s'accumulent pour former une sorte de journal de toutes les modifications apportées. J'ai déjà rencontré certains modules contenant des dizaines de pages de ces journaux de modifications.

```
* Modifications (depuis le 11-Oct-2001)
* ----------------------------------
* 11-Oct-2001 : Réorganisation de la classe et son déplacement dans le nouveau
*               paquetage com.jrefinery.date (DG).
* 05-Nov-2001 : Ajout de la méthode getDescription() et suppression de la
*               classe NotableDate (DG).
* 12-Nov-2001 : IBD a besoin d'une méthode setDescription(), à présent que
*               la classe NotableDate a disparu (DG). Modification de
*               getPreviousDayOfWeek(), getFollowingDayOfWeek() et
*               getNearestDayOfWeek() pour corriger les bogues (DG).
* 05-Déc-2001 : Correction du bogue dans la classe SpreadsheetDate (DG).
* 29-Mai-2002 : Déplacement des constantes de mois dans une interface séparée
*               (MonthConstants) (DG).
* 27-Aou-2002 : Correction du bogue dans la méthode addMonths(), merci à
*               Nalevka Petr (DG).
* 03-Oct-2002 : Corrections des erreurs signalées par Checkstyle (DG).
* 13-Mar-2003 : Implémentation de Serializable (DG).
* 29-Mai-2003 : Correction du bogue dans la méthode addMonths (DG).
* 04-Sep-2003 : Implémentation de Comparable. Actualisation de la documentation
*               Javadoc de isInRange (DG).
* 05-Jan-2005 : Correction du bogue dans la méthode addYears() (1096282) (DG).
```

Il y a bien longtemps, nous avions une bonne raison de créer et de maintenir ces journaux au début de chaque module. Les systèmes de contrôle du code source n'existaient pas pour le faire à notre place. Aujourd'hui, ces longs journaux ne font qu'encombrer et obscurcir les modules. Ils doivent être totalement supprimés.

Commentaires parasites

Certains commentaires ne sont rien d'autre que du bruit. Ils répètent simplement l'évident et n'apportent aucune nouvelle information.

```
/**
 * Constructeur par défaut.
 */
protected AnnualDateRule() {
}
```

Non, *c'est vrai* ? Et celui-ci :

```
/** Le jour du mois. */
private int dayOfMonth;
```

Voici sans doute le summum de la redondance :

```
/**
 * Retourne le jour du mois.
 *
 * @return le jour du mois.
 */
public int getDayOfMonth() {
  return dayOfMonth;
}
```

Ces commentaires sont tellement parasites que nous apprenons à les ignorer. Lorsque nous lisons le code, nos yeux sautent simplement par-dessus. À un moment donné, les commentaires commencent à mentir car le code associé évolue.

Le premier commentaire du Listing 4.4 semble approprié[1]. Il explique pourquoi le bloc `catch` est ignoré. En revanche, le second n'est que du bruit. Apparemment, le programmeur n'était pas content d'avoir à écrire des blocs `try`/`catch` dans cette fonction et il a éprouvé le besoin d'exprimer son mal-être.

Listing 4.4 : `startSending`

```
private void startSending()
{
  try
  {
    doSending();
  }
  catch(SocketException e)
  {
    // Normal. Quelqu'un a stoppé la requête.
  }
  catch(Exception e)
  {
    try
    {
      response.add(ErrorResponder.makeExceptionString(e));
      response.closeAll();
    }
    catch(Exception e1)
    {
      // Accordez-moi une pause !
    }
  }
}
```

Au lieu de s'épancher dans un commentaire inutile et tapageur, le programmeur aurait mieux fait de reconnaître que sa frustration pouvait être soulagée en améliorant la structure de son code. Il aurait dû consacrer son énergie sur l'extraction du dernier bloc `try`/`catch` dans une fonction séparée (voir Listing 4.5).

Listing 4.5 : `startSending` **(remanié)**

```
private void startSending()
{
  try
  {
    doSending();
  }
```

1. Les IDE actuels ont tendance à vérifier l'orthographe dans les commentaires, ce qui constitue un baume pour ceux d'entre nous qui lisent beaucoup de code.

```
    catch(SocketException e)
    {
      // Normal. Quelqu'un a stoppé la requête
    }
    catch(Exception e)
    {
      addExceptionAndCloseResponse(e);
    }
  }

  private void addExceptionAndCloseResponse(Exception e)
  {
    try
    {
      response.add(ErrorResponder.makeExceptionString(e));
      response.closeAll();
    }
    catch(Exception e1)
    {
    }
  }
```

Vous devez remplacer la tentation de créer du bruit par la détermination à nettoyer votre code. Vous serez ainsi un meilleur programmeur, plus heureux.

Bruit effrayant

La documentation Javadoc peut également représenter du bruit. Quel est l'objectif des commentaires suivants (extraits d'une bibliothèque open-source très connue) ? Il s'agit simplement de commentaires parasites redondants écrits par la seule volonté déplacée de fournir une documentation.

```
/** Le nom. */
private String name;

/** La version. */
private String version;

/** Le licenceName. */
private String licenceName;

/** La version. */
private String info;
```

Relisez attentivement ces commentaires encore une fois. Avez-vous repéré l'erreur de copier-coller ? Si les auteurs ne font pas attention aux commentaires qu'ils écrivent (ou recopient), pourquoi le lecteur s'attendrait-il à en tirer profit ?

Ne pas remplacer une fonction ou une variable par un commentaire

Prenons l'extrait de code suivant :

```
// Le module issu de la liste globale <mod> dépend-il du sous-système
// dont nous faisons partie ?
if (smodule.getDependSubsystems().contains(subSysMod.getSubSystem()))
```

Il est possible de reformuler ce code sans introduire le commentaire :

```
ArrayList moduleDependees = smodule.getDependSubsystems();
String ourSubSystem = subSysMod.getSubSystem();
if (moduleDependees.contains(ourSubSystem))
```

L'auteur du code d'origine a pu tout d'abord écrire le commentaire (peu probable), puis le code qui correspond au commentaire. Cependant, il aurait dû ensuite remanier le code, comme je l'ai fait, afin de pouvoir supprimer le commentaire.

Marqueurs de position

Certains programmeurs ont l'habitude de marquer les emplacements précis dans un fichier source. Par exemple, j'ai récemment rencontré la ligne suivante dans un programme que j'étudiais :

```
// Actions ////////////////////////////////
```

Certaines fois, il peut être sensé de réunir certaines fonctions sous ce type de bannière. Cela dit, elles encombrent généralement inutilement le code et doivent être éliminées, en particulier toute la séquence de barres obliques.

Réfléchissez. Les bannières sont repérables et évidentes lorsqu'elles se font rares. Vous devez donc les employer en très petite quantité et uniquement lorsque leur intérêt est significatif. Lorsqu'elles sont omniprésentes, elles tombent dans la catégorie du bruit ambiant et sont ignorées.

Commentaires d'accolade fermante

Les programmeurs ajoutent parfois des commentaires spéciaux après les accolades fermantes (voir Listing 4.6). Si cela peut avoir un sens dans les longues fonctions contenant des structures profondément imbriquées, ces commentaires ne font que participer au désordre général dans le cas des petites fonctions encapsulées que nous recommandons. Si vous constatez que des accolades fermantes doivent être marquées, essayez plutôt de raccourcir vos fonctions.

Listing 4.6 : `wc.java`

```
public class wc {
  public static void main(String[] args) {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    String line;
    int lineCount = 0;
    int charCount = 0;
    int wordCount = 0;
    try {
      while ((line = in.readLine()) != null) {
        lineCount++;
        charCount += line.length();
```

```
        String words[] = line.split("\\W");
        wordCount += words.length;
      } // while
      System.out.println("wordCount = " + wordCount);
      System.out.println("lineCount = " + lineCount);
      System.out.println("charCount = " + charCount);
    } // try
    catch (IOException e) {
      System.err.println("Error:" + e.getMessage());
    } // catch
  } // main
}
```

Attributions et signatures

```
/* Ajouté par Rick. */
```

Les systèmes de gestion du code source parviennent très bien à mémoriser qui a ajouté quoi et quand. Il est inutile de polluer le code avec ce type de légende. Vous pourriez penser que de tels commentaires peuvent servir aux autres programmeurs afin qu'ils sachent qui contacter pour discuter du code. En réalité, ils ont tendance à s'installer dans le code pendant des années, en devenant de moins en moins justes et pertinents.

Une fois encore, le système de gestion du code source est beaucoup mieux adapté à la prise en charge de ce type d'information.

Mettre du code en commentaire

Peu de pratiques sont aussi détestables que la mise en commentaire d'un bout de code. Ne le faites jamais !

```
    InputStreamResponse response = new InputStreamResponse();
    response.setBody(formatter.getResultStream(), formatter.getByteCount());
//  InputStream resultsStream = formatter.getResultStream();
//  StreamReader reader = new StreamReader(resultsStream);
//  response.setContent(reader.read(formatter.getByteCount()));
```

Ceux qui rencontrent du code mis en commentaire n'ont pas le courage de le supprimer. Ils pensent qu'il existe une raison à sa présence et qu'il est trop important pour être supprimé. Le code en commentaire finit par s'accumuler, comme du dépôt au fond d'une bouteille de mauvais vin.

Prenons le code suivant :

```
this.bytePos = writeBytes(pngIdBytes, 0);
//hdrPos = bytePos;
writeHeader();
writeResolution();
//dataPos = bytePos;
if (writeImageData()) {
  writeEnd();
  this.pngBytes = resizeByteArray(this.pngBytes, this.maxPos);
}
```

```
else {
  this.pngBytes = null;
}
return this.pngBytes;
```

Pourquoi ces deux lignes de code sont-elles en commentaire ? Sont-elles importantes ? Ont-elles été laissées afin de ne pas oublier une modification imminente ? Correspondent-elles simplement à du code obsolète mis en commentaire il y a très longtemps et depuis oublié ?

Dans les années 1960, le code mis en commentaire pouvait être utile. Mais nous disposons depuis très longtemps de systèmes de gestion du code source. Ces systèmes mémorisent le code à notre place. Il est désormais inutile de le placer en commentaire. Supprimez simplement le code car il ne sera pas perdu. C'est promis.

Commentaires HTML

Le contenu HTML dans les commentaires est une abomination, comme vous pouvez le constater en lisant le code ci-après. Les commentaires sont plus difficiles à lire depuis le seul endroit où ils devraient être faciles à lire – l'éditeur/IDE. Si les commentaires doivent être extraits par un outil, comme Javadoc, pour être placés sur une page web, alors, c'est à cet outil, non au programmeur, de les décorer avec le HTML approprié.

```
/**
 * T&acirc;che pour ex&eacute;cuter des tests d'ad&eacute;quation.
 * Cette t&acirc;che ex&eacute;cute des tests fitnesse et publie les
 * r&eacute;sultats.
 * <p/>
 * <pre>
 * Utilisation :
 * &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
 *     classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot;
 *     classpathref=&quot;classpath&quot; /&gt;
 * OU
 * &lt;taskdef classpathref=&quot;classpath&quot;
 *              resource=&quot;tasks.properties&quot; /&gt;
 * <p/>
 * &lt;execute-fitnesse-tests
 *     suitepage=&quot;FitNesse.SuiteAcceptanceTests&quot;
 *     fitnesseport=&quot;8082&quot;
 *     resultsdir=&quot;${results.dir}&quot;
 *     resultshtmlpage=&quot;fit-results.html&quot;
 *     classpathref=&quot;classpath&quot; /&gt;
 * </pre>
 */
```

Information non locale

Si vous devez écrire un commentaire, assurez-vous qu'il concerne du code proche. Ne donnez pas des informations globales dans un commentaire local. Prenons, par exemple, le commentaire Javadoc suivant. Hormis sa redondance, il fournit également des

informations sur le port par défaut, alors que la fonction n'a absolument aucune prise sur cette valeur. Le commentaire ne décrit pas la fonction mais une autre partie plus éloignée du système. Bien entendu, rien ne garantit que ce commentaire sera modifié si le code qui gère la valeur par défaut est modifié.

```
/**
 * Port sur lequel fitnesse va s'exécuter (par défaut <b>8082</b>).
 *
 * @param fitnessePort
 */
public void setFitnessePort(int fitnessePort)
{
  this.fitnessePort = fitnessePort;
}
```

Trop d'informations

Il ne faut pas placer des discussions historiques intéressantes ou des détails non pertinents dans les commentaires. L'exemple ci-après est tiré d'un module qui teste si une fonction peut coder et décoder des données au format base64. Hormis le numéro de la RFC, celui qui lit ce code n'a aucunement besoin des informations détaillées contenues dans le commentaire.

```
/*
   RFC 2045 - Multipurpose Internet Mail Extensions (MIME)
   Première Partie : Format des corps de message internet
   Section 6.8. Encodage de transfert Base64 du contenu
   Le traitement d'encodage représente les groupes de 24 bits entrants en une
   chaîne sortante encodée de 4 caractères. En traitant de la gauche vers la
   droite, un groupe entrant de 24 bits est formé en concaténant 3 groupes de
   8 bits entrants. Ces 24 bits sont alors traités comme 4 groupes de 6 bits
   concaténés, chacun de ces groupes est traduit sur un simple chiffre dans
   l'alphabet base64. Quand on encode un flux de bit par l'encodage base64, le
   flux de bit est supposé être ordonné avec le bit le plus significatif
   d'abord. Autrement dit, le premier bit dans le flux sera le bit de poids
   fort dans le premier mot de 8 bits, et le huitième bit sera le bit de poids
   faible dans le mot de 8 bits, et ainsi de suite.
*/
```

Lien non évident

Le lien entre un commentaire et le code qu'il décrit doit être évident. Si vous vous donnez du mal à écrire un commentaire, vous aimeriez au moins que le lecteur soit en mesure de le consulter avec le code afin de comprendre son propos.

Prenons, par exemple, le commentaire suivant tiré d'Apache Commons :

```
/*
 * Débuter avec un tableau suffisamment grand pour contenir tous les pixels
 * (plus des octets de filtrage) et 200 octets supplémentaires pour les
 * informations d'en-tête.
 */
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

Qu'est-ce qu'un octet de filtrage ? Cela a-t-il un rapport avec `+1` ? Ou avec `*3` ? Avec les deux ? Un pixel équivaut-il à un octet ? Pourquoi 200 ? Un commentaire a pour objectif d'expliquer un code qui n'est pas facile à comprendre par lui-même. Il est dommage qu'un commentaire ait besoin de sa propre explication.

En-têtes de fonctions

Les fonctions courtes n'ont pas besoin d'une longue description. Il est généralement préférable de bien choisir le nom d'une fonction courte qui fait une seule chose que d'ajouter un commentaire en en-tête.

Documentation Javadoc dans du code non public

Si la documentation Javadoc est très utile pour des API publiques, elle est à condamner pour le code privé. La génération des pages Javadoc pour les classes et des fonctions internes au système se révèle généralement inutile et la solennité supplémentaire des commentaires Javadoc n'apporte rien d'autre que du superflu et une distraction.

Exemple

J'ai écrit le module présenté au Listing 4.7 pour le premier cours *XP Immersion*. Il devait servir d'exemple de mauvais code et de mauvaise utilisation des commentaires. Kent Beck l'a ensuite remanié devant des dizaines d'étudiants enthousiastes afin d'obtenir une version beaucoup plus plaisante. Par la suite, j'ai adapté cet exemple pour mon livre *Agile Software Development, Principles, Patterns, and Practices* et le premier de mes articles *Craftsman* publiés dans le magazine *Software Development*.

Ce module a ceci de fascinant que nombre d'entre nous l'auraient considéré, à une certaine époque, comme "bien documenté". À présent, il sert de contre-exemple. Combien de problèmes liés aux commentaires pouvez-vous identifier ?

Listing 4.7 : `GeneratePrimes.java`

```
/**
 * Cette classe génère des nombres premiers jusqu'au maximum indiqué par
 * utilisateur. L'algorithme employé est le crible d'Eratosthène.
 * <p>
 * Eratosthène est né à Cyrène (Lybié) en 284 avant Jésus-Christ. Il est
 * décédé à Alexandrie en 192 avant Jésus-Christ. Il a été le premier homme
 * à calculer la circonférence de la Terre. Il est également connu pour son
 * travail sur les calendriers contenant des années bissextiles et sa fonction
 * de directeur de la bibliothèque d'Alexandrie.
 * <p>
 * L'algorithme est relativement simple. Étant donné un tableau d'entiers
 * débutant à 2, rayer tous les multiples de 2. Prendre l'entier non rayé
 * suivant, puis rayer tous les multiples de celui-ci. Répéter l'opération
 * jusqu'à ce que la racine carrée de la valeur maximale soit atteinte.
 *
```

```
 * @author Alphonse
 * @version 13 Fév 2002
 */
import java.util.*;

public class GeneratePrimes
{
  /**
   * @param maxValue indique la limite de génération.
   */
  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue >= 2) // Le seul cas valide.
    {
      // Déclarations.
      int s = maxValue + 1; // Taille du tableau.
      boolean[] f = new boolean[s];
      int i;

      // Initialiser le tableau à true.
      for (i = 0; i < s; i++)
        f[i] = true;

      // Enlever les nombres non premiers connus.
      f[0] = f[1] = false;

      // Le crible.
      int j;
      for (i = 2; i < Math.sqrt(s) + 1; i++)
      {
        if (f[i]) // Si i n'est pas payé, rayer ses multiples.
        {
          for (j = 2 * i; j < s; j += i)
            f[j] = false; // Le multiple n'est pas premier.
        }
      }

      // Combien de nombres premiers avons-nous ?
      int count = 0;
      for (i = 0; i < s; i++)
      {
        if (f[i])
          count++; // Incrémenter le compteur.
      }

      int[] primes = new int[count];

      // Déplacer les nombres premiers dans le résultat.
      for (i = 0, j = 0; i < s; i++)
      {
        if (f[i])             // Si premier.
          primes[j++] = i;
      }

      return primes;  // Retourner les nombres premiers.
    }
```

```
    else // maxValue < 2
      return new int[0]; // Retourner un tableau vide en cas d'entrée invalide.
  }
}
```

Le Listing 4.8 propose une version remaniée du même module. Les commentaires sont beaucoup moins nombreux. Les deux seuls commentaires du nouveau module sont par nature évidents.

Listing 4.8 : `PrimeGenerator.java` **(remanié)**

```
/**
 * Cette classe génère des nombres premiers jusqu'au maximum indiqué par
 * l'utilisateur. L'algorithme employé est le crible d'Eratosthène.
 * Étant donné un tableau d'entiers débutant à 2, trouver le premier entier
 * non rayé et rayer tous ses multiples. Répéter l'opération jusqu'à ce que
 * le tableau ne contienne plus aucun multiple.
 */

public class PrimeGenerator
{
  private static boolean[] crossedOut;
  private static int[] result;

  public static int[] generatePrimes(int maxValue)
  {
    if (maxValue < 2)
      return new int[0];
    else
    {
      uncrossIntegersUpTo(maxValue);
      crossOutMultiples();
      putUncrossedIntegersIntoResult();
      return result;
    }
  }

  private static void uncrossIntegersUpTo(int maxValue)
  {
    crossedOut = new boolean[maxValue + 1];
    for (int i = 2; i < crossedOut.length; i++)
      crossedOut[i] = false;
  }

  private static void crossOutMultiples()
  {
    int limit = determineIterationLimit();
    for (int i = 2; i <= limit; i++)
      if (notCrossed(i))
        crossOutMultiplesOf(i);
  }

  private static int determineIterationLimit()
  {
    // Chaque multiple dans le tableau possède un facteur premier qui est
    // inférieur ou égal à la racine carrée de la taille du tableau.
```

```
    // Nous n'avons donc pas besoin de rayer les multiples des nombres
    // supérieurs à cette racine.
    double iterationLimit = Math.sqrt(crossedOut.length);
    return (int) iterationLimit;
  }

  private static void crossOutMultiplesOf(int i)
  {
    for (int multiple = 2*i;
         multiple < crossedOut.length;
         multiple += i)
      crossedOut[multiple] = true;
  }

  private static boolean notCrossed(int i)
  {
    return crossedOut[i] == false;
  }

  private static void putUncrossedIntegersIntoResult()
  {
    result = new int[numberOfUncrossedIntegers()];
    for (int j = 0, i = 2; i < crossedOut.length; i++)
      if (notCrossed(i))
        result[j++] = i;
  }

  private static int numberOfUncrossedIntegers()
  {
    int count = 0;
    for (int i = 2; i < crossedOut.length; i++)
      if (notCrossed(i))
        count++;
    return count;
  }
}
```

Nous pouvons sans mal prétendre que le premier commentaire est redondant car il se lit de manière très comparable à la fonction `generatePrimes` elle-même. Toutefois, je pense qu'il permet au lecteur d'entrer plus facilement dans l'algorithme et qu'il est donc préférable de le conserver.

Le second commentaire est certainement indispensable. Il explique la raison de l'utilisation de la racine carrée comme la limite de la boucle. Je ne peux pas trouver un nom variable simple ni une structure de codage différente, qui éclaircit ce point. D'un autre côté, l'utilisation de la racine carrée pourrait n'être qu'une marque de vanité. Est-ce que je gagne vraiment beaucoup de temps en limitant l'itération à la racine carrée ? Le calcul de la racine carrée ne prend-il pas plus de temps que celui économisé ?

Il est bon d'y réfléchir. L'utilisation de la racine carrée comme limite de l'itération satisfait l'ancien hacker C et assembleur qui sommeille en moi, mais je ne suis pas convaincu qu'elle vaille le temps et les efforts demandés aux autres lecteurs pour la comprendre.

5

Mise en forme

Si quelqu'un soulève le voile, nous voulons qu'il soit impressionné par l'élégance, la cohérence et l'attention portée aux détails dans ce qu'il voit. Nous voulons qu'il soit frappé par l'ordre. Nous voulons qu'il s'étonne lorsqu'il parcourt les modules. Nous voulons qu'il perçoive le travail de professionnels. S'il voit à la place une quantité de

code embrouillé qui semble avoir été écrit par une bande de marins soûls, il en conclura certainement que le même je-m'en-foutisme sous-tend chaque autre aspect du projet.

Vous devez faire attention à ce que votre code soit parfaitement présenté. Vous devez choisir un ensemble de règles simples qui guident cette mise en forme et les appliquer systématiquement. Si vous travaillez au sein d'une équipe, tous les membres doivent se mettre d'accord sur un ensemble de règles de mise en forme et s'y conformer. Un outil automatique qui applique ces règles de mise en forme à votre place pourra vous y aider.

Objectif de la mise en forme

Soyons extrêmement clairs. Le formatage du code est *important*. Il est trop important pour être ignoré et trop important pour être traité religieusement. La mise en forme du code se place au niveau de la communication et la communication est le premier commandement du développeur professionnel.

Vous pensiez peut-être que le premier commandement du développeur professionnel était "faire en sorte que cela fonctionne". J'espère cependant que, arrivé à ce stade du livre, vous avez changé d'avis. La fonctionnalité que vous allez créer aujourd'hui a de fortes chances d'évoluer dans la prochaine version, tandis que la lisibilité du code aura un effet profond sur toutes les modifications qui pourront être apportées. Le style de codage et la lisibilité établissent un précédent qui continue à affecter la facilité de maintenance et d'extension du code bien après que la version d'origine a évolué de manière méconnaissable. Votre style et votre discipline survivent, même si ce n'est pas le cas de votre code.

Quels sont donc les aspects de la mise en forme qui nous aident à mieux communiquer ?

Mise en forme verticale

Commençons par la mise en forme verticale. Quelle doit être la taille d'un fichier source ? En Java, la taille d'un fichier est étroitement liée à la taille d'une classe. Nous reviendrons sur la taille d'une classe au Chapitre 10. Pour le moment, focalisons-nous sur la taille d'un fichier.

En Java, quelle est la taille de la plupart des fichiers sources ? En réalité, il existe une grande variété de tailles et certaines différences de style remarquables. La Figure 5.1 en montre quelques-unes.

Sept projets différents sont illustrés : JUnit, FitNesse, testNG, Time and Money, JDepend, Ant et Tomcat. Les lignes qui traversent les boîtes indiquent les longueurs minimale et maximale des fichiers dans chaque projet. La boîte représente approximati-

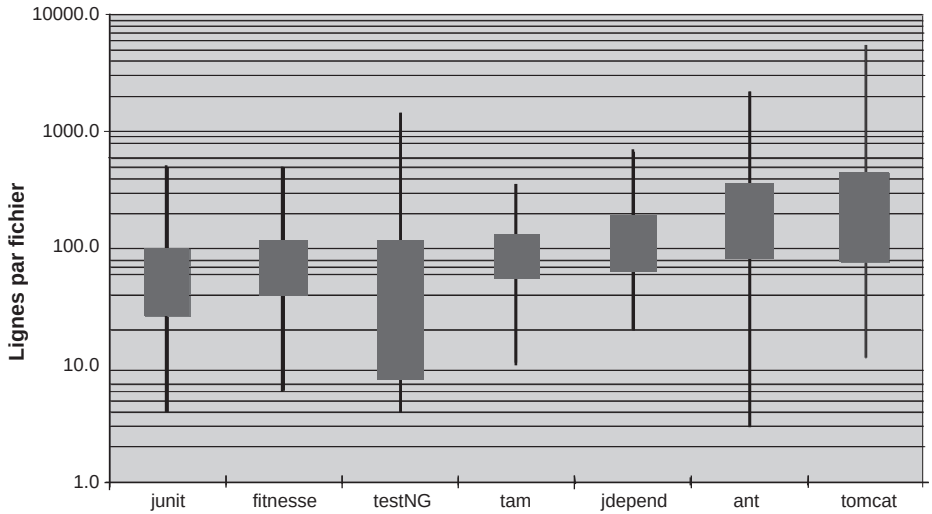


Figure 5.1

Distribution des tailles de fichiers dans une échelle logarithmique (la hauteur d'une boîte correspond à l'écart-type).

vement un tiers des fichiers (un écart-type[1]). Le milieu de la boîte correspond à la moyenne. Par conséquent, la taille moyenne d'un fichier dans le projet FitNesse est de 65 lignes, et environ un tiers des fichiers contiennent entre 40 et plus de 100 lignes. Dans FitNesse, le plus long fichier contient 400 lignes, le plus petit 6 lignes. Puisque nous employons une échelle logarithmique, une petite différence en ordonnée implique une très grande différence en taille absolue.

JUnit, FitNesse et Time and Money sont constitués de fichiers relativement petits. Aucun ne dépasse 500 lignes et la plupart ont une taille inférieure à 200 lignes. En revanche, Tomcat et Ant possèdent quelques fichiers de plusieurs milliers de lignes et près de la moitié dépasse 200 lignes.

Comment pouvons-nous interpréter ces chiffres ? Tout d'abord, il est possible de construire des systèmes importants (environ 50 000 lignes pour FitNesse) avec des fichiers contenant généralement 200 lignes, et une taille maximale de 500 lignes. Même si cela ne doit pas constituer une règle absolue, cette approche est très souhaitable. Les fichiers courts sont généralement plus faciles à comprendre que les fichiers longs.

1. La boîte montre l'écart-type/2 au-dessus et en dessous de la moyenne. Je sais pertinemment que la distribution des longueurs de fichiers n'est pas normale et que l'écart-type n'est donc pas mathématiquement précis. Cela dit, nous ne recherchons pas la précision ici. Nous souhaitons juste avoir une idée des tailles.

Métaphore du journal

Pensez à un article de journal bien écrit. Vous le lisez de haut en bas. Au début, vous attendez un titre qui indique le propos de l'article et vous permet de décider si vous allez poursuivre sa lecture. Le premier paragraphe est un synopsis du contenu global, dégagé de tous les détails, mais avec les concepts généraux. Plus vous progressez vers le bas de l'article, plus vous recevez de détails, jusqu'à obtenir toutes les dates, noms, citations, affirmations et autres petits détails.

Nous voudrions qu'un fichier source ressemble à un article de journal. Le nom doit être simple, mais explicatif. Ce nom doit nous permettre de déterminer si nous examinons le bon module. Les parties initiales du fichier source doivent fournir les concepts de haut niveau et les algorithmes. Le niveau de détail doit augmenter au fur et à mesure que nous descendons vers le bas du fichier source, pour arriver à la fin où se trouvent les fonctions et les détails de plus bas niveau.

Un journal est constitué de nombreux articles. La plupart sont très petits, certains sont assez longs. Très peu occupent une page entière. C'est pour cela que le journal est *fonctionnel*. S'il était constitué d'un seul long article contenant un ensemble désorganisé de faits, de dates et de noms, il serait tout simplement impossible à lire.

Espacement vertical des concepts

Le code se lit essentiellement de gauche à droite et de haut en bas. Chaque ligne représente une expression ou une clause, et chaque groupe de lignes représente une idée. Ces idées doivent être séparées les unes des autres par des lignes vides.

Étudions le Listing 5.1. Des lignes vides séparent la déclaration du paquetage, l'importation et chaque fonction. Cette règle extrêmement simple a un effet majeur sur l'organisation visuelle du code. Chaque ligne vide est un indice visuel qui identifie un nouveau concept distinct. Lorsque nous parcourons le listing vers le bas, nos yeux sont attirés par la première ligne qui suit une ligne vide.

Listing 5.1 : `BoldWidget.java`

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );
```

```
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }

  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

Si nous retirons ces lignes vides, comme dans le Listing 5.2, nous remettons considérablement en question la lisibilité du code.

Listing 5.2 : `BoldWidget.java`

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL);
  public BoldWidget(ParentWidget parent, String text) throws Exception {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));}
  public String render() throws Exception {
    StringBuffer html = new StringBuffer("<b>");
    html.append(childHtml()).append("</b>");
    return html.toString();
  }
}
```

L'effet est encore plus prononcé si nous ne concentrons pas notre attention. Dans le premier exemple, les différents groupes de lignes sautent aux yeux, tandis que le deuxième exemple s'approche plus d'un grand désordre. Pourtant, la seule différence entre ces deux listings réside dans un petit espacement vertical.

Concentration verticale

Si un espacement sépare des concepts, une concentration verticale implique une association étroite. Par conséquent, les lignes de code étroitement liées doivent apparaître verticalement concentrées. Dans le Listing 5.3, remarquez combien les commentaires inutiles rompent l'association étroite entre les deux variables d'instance.

Listing 5.3

```
public class ReporterConfig {

  /**
   * Le nom de classe de l'auditeur rapporteur.
   */
  private String m_className;

  /**
   * Les propriétés de l'auditeur rapporteur.
   */
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Le Listing 5.4 est beaucoup plus facile à lire. Il est, tout au moins pour moi, un "régal pour les yeux". Je peux le regarder et voir qu'il s'agit d'une classe avec deux variables et une méthode, sans avoir à bouger énormément ma tête ou mes yeux. Le listing précédent exigeait de moi des mouvements d'yeux et de tête pour obtenir le même niveau de compréhension.

Listing 5.4

```
public class ReporterConfig {
  private String m_className;
  private List<Property> m_properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    m_properties.add(property);
  }
```

Distance verticale

Avez-vous déjà tourné en rond dans une classe, en passant d'une fonction à la suivante, en faisant défiler le fichier source vers le haut et vers le bas, en tentant de deviner le lien entre les fonctions, pour finir totalement perdu dans un nid de confusion ? Avez-vous déjà remonté la chaîne d'héritage de la définition d'une fonction ou d'une variable ? Cette expérience est très frustrante car vous essayez de comprendre ce que fait le système, alors que vous passez votre temps et votre énergie à tenter de localiser les différents morceaux et à mémoriser leur emplacement.

Les concepts étroitement liés doivent être verticalement proches les uns des autres [G10]. Bien évidemment, cette règle ne concerne pas les concepts qui se trouvent dans des fichiers séparés. Toutefois, les concepts étroitement liés ne devraient pas se trouver dans des fichiers différents, à moins qu'il n'existe une très bonne raison à cela. C'est l'une des raisons pour lesquelles les variables protégées doivent être évitées.

Lorsque des concepts étroitement liés appartiennent au même fichier source, leur séparation verticale doit indiquer l'importance de chacun dans la compréhension de l'autre. Nous voulons éviter que le lecteur ne saute de part en part dans nos fichiers sources et nos classes.

Déclarations de variables

Les variables doivent être déclarées au plus près de leur utilisation. Puisque nos fonctions sont très courtes, les variables locales doivent apparaître au début de chaque fonction, comme dans la fonction plutôt longue suivante extraite de JUnit 4.3.1 :

```
private static void readPreferences() {
  InputStream is= null;
  try {
    is= new FileInputStream(getPreferencesFile());
    setPreferences(new Properties(getPreferences()));
    getPreferences().load(is);
  } catch (IOException e) {
    try {
      if (is != null)
        is.close();
    } catch (IOException e1) {
    }
  }
}
```

Les variables de contrôle des boucles doivent généralement être déclarées à l'intérieur de l'instruction de boucle, comme dans la petite fonction suivante provenant de la même source :

```
public int countTestCases() {
  int count= 0;
  for (Test each : tests)
    count += each.countTestCases();
  return count;
}
```

En de rares cas, lorsqu'une fonction est plutôt longue, une variable peut être déclarée au début d'un bloc ou juste avant une boucle. Vous pouvez rencontrer une telle variable au beau milieu d'une très longue fonction de TestNG :

```
...
for (XmlTest test : m_suite.getTests()) {
      TestRunner tr = m_runnerFactory.newTestRunner(this, test);
      tr.addListener(m_textReporter);
      m_testRunners.add(tr);

      invoker = tr.getInvoker();

      for (ITestNGMethod m : tr.getBeforeSuiteMethods()) {
        beforeSuiteMethods.put(m.getMethod(), m);
      }
```

```
        for (ITestNGMethod m : tr.getAfterSuiteMethods()) {
          afterSuiteMethods.put(m.getMethod(), m);
        }
      }
...
```

Variables d'instance

A contrario, les variables d'instance doivent être déclarées au début de la classe. Cela ne doit pas augmenter la distance verticale de ces variables, car, dans une classe bien conçue, elles sont employées dans plusieurs voire dans toutes les méthodes de la classe.

Les discussions à propos de l'emplacement des variables d'instance ont été nombreuses. En C++, nous avions l'habitude d'employer la bien nommée *règle des ciseaux*, qui place toutes les variables d'instance à la fin. En Java, la convention veut qu'elles soient toutes placées au début de la classe. Je ne vois aucune raison de suivre une autre convention. Le point important est que les variables d'instance soient déclarées en un endroit parfaitement connu. Tout le monde doit savoir où se rendre pour consulter les déclarations.

Étudions, par exemple, le cas étrange de la classe `TestSuite` dans JUnit 4.3.1. J'ai énormément réduit cette classe afin d'aller à l'essentiel. Vers la moitié du listing, deux variables d'instance sont déclarées. Il est difficile de trouver un endroit mieux caché. Pour les découvrir, celui qui lit ce code devra tomber par hasard sur ces déclarations (comme cela m'est arrivé).

```
public class TestSuite implements Test {
  static public Test createTest(Class<? extends TestCase> theClass,
                                String name) {
    ...
  }

  public static Constructor<? extends TestCase>
  getTestConstructor(Class<? extends TestCase> theClass)
  throws NoSuchMethodException {
    ...
  }

  public static Test warning(final String message) {
    ...
  }

  private static String exceptionToString(Throwable t) {
    ...
  }

  private String fName;

  private Vector<Test> fTests= new Vector<Test>(10);
```

```
  public TestSuite() {
  }

  public TestSuite(final Class<? extends TestCase> theClass) {
     ...
  }

  public TestSuite(Class<? extends TestCase> theClass, String name) {
    ...
  }
  ... ... ... ... ...
}
```

Fonctions dépendantes

Lorsqu'une fonction en appelle une autre, les deux doivent être verticalement proches et l'appelant doit se trouver au-dessus de l'appelé, si possible. De cette manière, le flux du programme est naturel. Si cette convention est fidèlement suivie, le lecteur saura que les définitions des fonctions se trouvent peu après leur utilisation. Prenons, par exemple, l'extrait de code de FitNesse donné au Listing 5.5. Vous remarquerez que la première fonction appelle celles qui se trouvent ensuite et que celles-ci appellent à leur tour des fonctions qui se trouvent plus bas. Il est ainsi plus facile de trouver les fonctions appelées et la lisibilité du module s'en trouve fortement améliorée.

Listing 5.5 : `WikiPageResponder.java`

```
public class WikiPageResponder implements SecureResponder {
  protected WikiPage page;
  protected PageData pageData;
  protected String pageTitle;
  protected Request request;
  protected PageCrawler crawler;

  public Response makeResponse(FitNesseContext context, Request request)
    throws Exception {
    String pageName = getPageNameOrDefault(request, "FrontPage");
    loadPage(pageName, context);
    if (page == null)
      return notFoundResponse(context, request);
    else
      return makePageResponse(context);
  }

  private String getPageNameOrDefault(Request request, String defaultPageName)
  {
    String pageName = request.getResource();
    if (StringUtil.isBlank(pageName))
      pageName = defaultPageName;

    return pageName;
  }
```

```
  protected void loadPage(String resource, FitNesseContext context)
    throws Exception {
    WikiPagePath path = PathParser.parse(resource);
    crawler = context.root.getPageCrawler();
    crawler.setDeadEndStrategy(new VirtualEnabledPageCrawler());
    page = crawler.getPage(context.root, path);
    if (page != null)
      pageData = page.getData();
  }

  private Response notFoundResponse(FitNesseContext context, Request request)
    throws Exception {
    return new NotFoundResponder().makeResponse(context, request);
  }

  private SimpleResponse makePageResponse(FitNesseContext context)
    throws Exception {
    pageTitle = PathParser.render(crawler.getFullPath(page));
    String html = makeHtml(context);

    SimpleResponse response = new SimpleResponse();
    response.setMaxAge(0);
    response.setContent(html);
    return response;
  }
...
```

Par ailleurs, cet extrait de code est un bon exemple de placement des constantes au niveau approprié [G35]. La constante `"FrontPage"` aurait pu être enfouie dans la fonction `getPageNameOrDefault`, mais cette solution aurait caché une constante connue et attendue dans une fonction de bas niveau inappropriée. Il était préférable de passer la constante depuis l'endroit où la connaître a un sens vers l'endroit où elle est employée réellement.

Affinité conceptuelle

Certaines parties du code *veulent* se trouver à côté de certaines autres. Elles présentent une affinité conceptuelle. Plus cette affinité est grande, moins la distance verticale qui les sépare est importante.

Nous l'avons vu, cette affinité peut se fonder sur une dépendance directe, comme l'appel d'une fonction par une autre, ou une fonction qui utilise une variable. Mais il existe également d'autres causes d'affinité. Par exemple, elle peut être due à un groupe de fonctions qui réalisent une opération semblable. Prenons le code suivant extrait de JUnit 4.3.1 :

```
public class Assert {
  static public void assertTrue(String message, boolean condition) {
    if (!condition)
      fail(message);
  }

  static public void assertTrue(boolean condition) {
    assertTrue(null, condition);
  }

  static public void assertFalse(String message, boolean condition) {
    assertTrue(message, !condition);
  }

  static public void assertFalse(boolean condition) {
    assertFalse(null, condition);
  }
...
```

Ces fonctions présentent une affinité conceptuelle car elles partagent un même schéma de nommage et mettent en œuvre des variantes d'une même tâche de base. Le fait qu'elles s'invoquent l'une et l'autre est secondaire. Même si ce n'était pas le cas, elles voudraient toujours être proches les unes des autres.

Rangement vertical

En général, nous préférons que les dépendances d'appel de fonctions se fassent vers le bas. Autrement dit, une fonction appelée doit se trouver en dessous d'une fonction qui l'appelle[2]. Cela crée un agréable flux descendant dans le module du code source, en allant des fonctions de haut niveau vers les fonctions de bas niveau.

Comme dans les articles d'un journal, nous nous attendons à trouver tout d'abord les concepts les plus importants, exprimés avec le minimum de détails superflus possible. Les détails de bas niveau sont supposés arriver en dernier. Cela nous permet de passer rapidement sur les fichiers sources, en retenant l'essentiel des quelques premières fonctions, sans avoir à nous plonger dans les détails. Le Listing 5.5 est organisé de cette manière. Le Listing 15.5 à la page 281 et le Listing 3.7 à la page 56 en sont même de meilleurs exemples.

Mise en forme horizontale

Quelle doit être la largeur d'une ligne ? Pour répondre à cette question, examinons la taille des lignes dans sept projets différents. La Figure 5.2 présente la distribution des largeurs de lignes dans les sept projets. La régularité est impressionnante, en particulier

2. Il s'agit de l'exact opposé de langages tels que Pascal, C et C++, qui imposent aux fonctions d'être définies, tout au moins déclarées, *avant* d'être utilisées.

autour de 45 caractères. Chaque taille entre 20 à 60 caractères représente environ 1 % du nombre total de lignes, ce qui fait 40 % ! 30 % supplémentaires sont constitués de lignes de moins de 10 caractères. N'oubliez pas qu'il s'agit d'une échelle logarithmique et donc que la diminution amorcée au-dessus de 80 caractères est très significative. Les programmeurs préfèrent manifestement les lignes courtes.

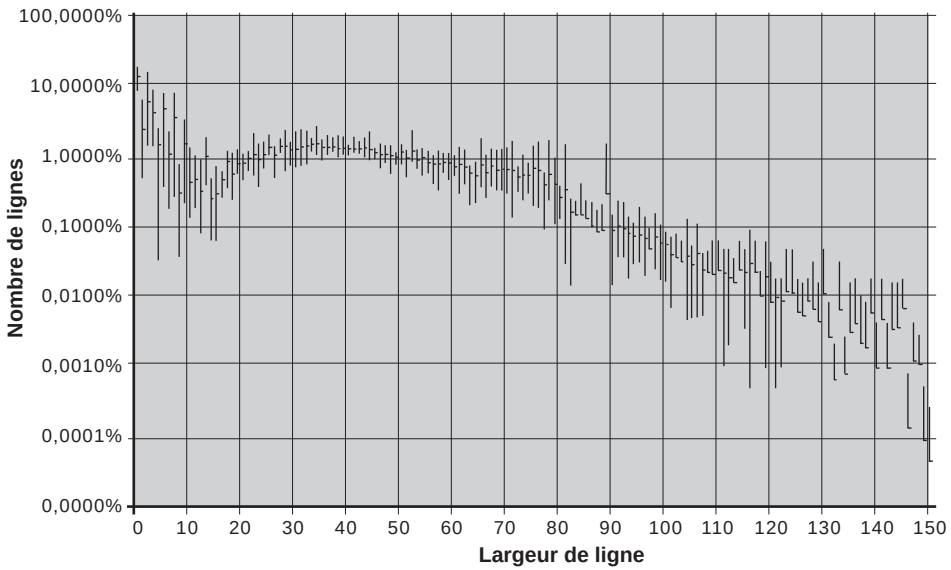


Figure 5.2

Distribution de la taille des lignes en Java.

En conclusion, nous devons nous efforcer à écrire des lignes courtes. L'ancienne limite de 80 caractères, fixée par Hollerith, est un tantinet arbitraire et je ne suis pas opposé aux lignes qui dépassent 100 caractères, voire 120. Les largeurs plus importantes indiquent probablement un manque de soin.

J'ai l'habitude de respecter la règle suivante : il ne faut jamais avoir à faire défiler les documents vers la droite. Cependant, les écrans actuels sont trop larges pour que cette règle garde tout son sens et les jeunes programmeurs peuvent réduire la taille de la police afin d'arriver à 200 caractères sur la largeur de l'écran. Ne le faites pas. Personnellement, je m'impose une limite supérieure égale à 120.

Espacement horizontal et densité

Nous employons un espacement horizontal pour associer des éléments étroitement liés et dissocier ceux qui le sont plus faiblement. Examinons la fonction suivante :

```
private void measureLine(String line) {
  lineCount++;
  int lineSize = line.length();
  totalChars += lineSize;
  lineWidthHistogram.addLine(lineSize, lineCount);
  recordWidestLine(lineSize);
}
```

Les opérateurs d'affectation sont entourés d'espaces horizontales pour les accentuer. Ces instructions sont constituées de deux éléments principaux et distincts : le côté gauche et le côté droit. Les espaces permettent de révéler cette séparation.

En revanche, je ne place aucune espace entre les noms de fonctions et la parenthèse ouvrante. En effet, la fonction et ses arguments sont étroitement liés et ne doivent pas paraître séparés. À l'intérieur des parenthèses d'appel de la fonction, je sépare les arguments afin d'accentuer la virgule et de montrer que les arguments sont distincts.

Voici un autre exemple où l'espacement accentue la précédence des opérateurs :

```
public class Quadratic {
  public static double root1(double a, double b, double c) {
    double determinant = determinant(a, b, c);
    return (-b + Math.sqrt(determinant)) / (2*a);
  }

  public static double root2(int a, int b, int c) {
    double determinant = determinant(a, b, c);
    return (-b - Math.sqrt(determinant)) / (2*a);
  }

  private static double determinant(double a, double b, double c) {
    return b*b - 4*a*c;
  }
}
```

La lecture des équations est très agréable. Les multiplications n'incluent pas d'espaces car leur précédence est élevée. Les différents termes sont séparés par des espaces, car une addition et une soustraction ont une précédence inférieure.

Malheureusement, la plupart des outils de mise en forme du code ne tiennent pas compte de la précédence des opérateurs et appliquent un espacement uniforme. L'usage subtil des espaces a tendance à disparaître si le code est reformaté.

Alignement horizontal

Lorsque je programmais en langage assembleur[3], j'employais l'alignement horizontal pour accentuer certaines structures. Lorsque j'ai commencé à coder en C, en C++, puis

3. Qui est-ce que j'essaie de convaincre ? Je suis toujours un programmeur en assembleur. Vous pouvez éloigner le garçon de la baie mais vous ne pouvez jamais éloigner la baie du garçon (proverbe de Terre-Neuve).

en Java, j'ai continué à aligner les noms de variables dans une suite de déclarations ou les valeurs de droite dans les affectations. Mon code ressemblait alors à celui-ci :

```
public class FitNesseExpediter implements ResponseSender
{
  private   Socket          socket;
  private   InputStream     input;
  private   OutputStream    output;
  private   Request         request;
  private   Response        response;
  private   FitNesseContext context;
  protected long            requestParsingTimeLimit;
  private   long            requestProgress;
  private   long            requestParsingDeadline;
  private   boolean         hasError;

  public FitNesseExpediter(Socket          s,
                           FitNesseContext context) throws Exception
  {
    this.context =            context;
    socket =                  s;
    input =                   s.getInputStream();
    output =                  s.getOutputStream();
    requestParsingTimeLimit = 10000;
  }
```

Cependant, j'ai fini par trouver cet alignement inutile. Il semble mettre l'accent sur les mauvais points et éloigne mon œil des véritables intentions. Par exemple, dans les déclarations précédentes, nous avons tendance à lire la liste des noms de variables vers le bas, sans examiner leur type. De la même manière, dans la liste des affectations, nous avons tendance à lire les valeurs de droite sans même voir l'opérateur d'affectation. Par ailleurs, les outils de mise en forme automatiques suppriment généralement ce type d'alignement.

Désormais, je n'applique plus ce genre de formatage. Je préfère les déclarations et les affectations non alignées, comme dans le code ci-après, car elles révèlent une faiblesse importante. Si je dois aligner de longues listes, *le problème se trouve dans la longueur des listes*, non dans l'absence d'alignement. La longueur de la liste des déclarations dans `FitNesseExpediter` indique que cette classe devrait être divisée.

```
public class FitNesseExpediter implements ResponseSender
{
  private Socket socket;
  private InputStream input;
  private OutputStream output;
  private Request request;
  private Response response;
  private FitNesseContext context;
  protected long requestParsingTimeLimit;
  private long requestProgress;
  private long requestParsingDeadline;
  private boolean hasError;
```

```
public FitNesseExpediter(Socket s, FitNesseContext context) throws Exception
{
  this.context = context;
  socket = s;
  input = s.getInputStream();
  output = s.getOutputStream();
  requestParsingTimeLimit = 10000;
}
```

Indentation

Un fichier source est plus une hiérarchie qu'un plan. Il contient des informations qui concernent l'ensemble du fichier, les classes individuelles dans le fichier, les méthodes à l'intérieur des classes, les blocs au sein des méthodes et, récursivement, les blocs dans les blocs. Chaque niveau de cette hiérarchie constitue une portée dans laquelle des noms peuvent être déclarés et des déclarations et des instructions exécutables sont interprétées.

Pour que cette hiérarchie de portées soit visualisable, nous indentons les lignes de code source proportionnellement à leur emplacement dans la hiérarchie. Les instructions qui se trouvent au niveau du fichier, comme la plupart des déclarations de classe, ne sont pas indentées. Les méthodes d'une classe sont indentées d'un niveau vers la droite par rapport à la classe. Les implémentations de ces méthodes sont indentées d'un niveau vers la droite par rapport aux déclarations des méthodes. Les blocs sont indentés d'un niveau vers la droite par rapport à leur bloc conteneur. Et ainsi de suite.

Les programmeurs s'appuient énormément sur ce modèle d'indentation. Ils alignent visuellement les lignes sur la gauche pour connaître leur portée. Cela leur permet de passer rapidement sur des portions, comme les implémentations des instructions `if` ou `while`, qui n'ont pas de rapport avec leur intérêt du moment. Ils examinent la partie gauche pour repérer les déclarations de nouvelles méthodes, de nouvelles variables et même de nouvelles classes. Sans l'indentation, les programmes seraient quasiment illisibles pour les humains.

Prenons les deux programmes suivants totalement identiques d'un point de vue syntaxique et sémantique :

```
public class FitNesseServer implements SocketServer { private FitNesseContext
context; public FitNesseServer(FitNesseContext context) { this.context =
context; } public void serve(Socket s) { serve(s, 10000); } public void
serve(Socket s, long requestTimeout) { try { FitNesseExpediter sender = new
FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start(); }
catch(Exception e) { e.printStackTrace(); } } }

-----

public class FitNesseServer implements SocketServer {
  private FitNesseContext context;
```

```
  public FitNesseServer(FitNesseContext context) {
    this.context = context;
  }

  public void serve(Socket s) {
    serve(s, 10000);
  }

  public void serve(Socket s, long requestTimeout) {
    try {
      FitNesseExpediter sender = new FitNesseExpediter(s, context);
      sender.setRequestParsingTimeLimit(requestTimeout);
      sender.start();
    }
    catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Notre œil peut discerner rapidement la structure des fichiers indentés. Nous pouvons presque instantanément repérer les variables, les constructeurs, les accesseurs et les méthodes. Il ne faut que quelques secondes pour réaliser que ce programme est une sorte d'interface simple pour les sockets, avec une temporisation. En revanche, la version non indentée est incompréhensible sans une étude approfondie.

Rompre l'indentation

Il est parfois tentant de passer outre la règle d'indentation pour les instructions `if`, les boucles `while` ou les fonctions courtes. S'il m'est arrivé de succomber à cette tentation, j'ai pratiquement toujours fait marche arrière et remis en place l'indentation. J'évite de réduire les portées à une ligne, comme dans le cas suivant :

```
public class CommentWidget extends TextWidget
{
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

  public CommentWidget(ParentWidget parent, String text){super(parent, text);}
  public String render() throws Exception {return ""; }
}
```

Je préfère les développer et les indenter :

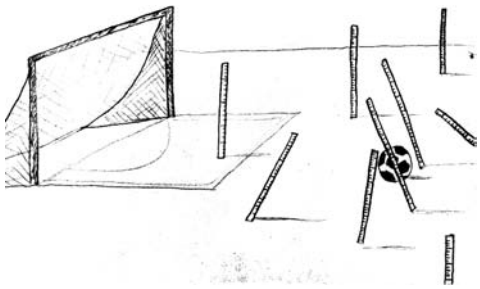
```
public class CommentWidget extends TextWidget {
  public static final String REGEXP = "^#[^\r\n]*(?:(?:\r\n)|\n|\r)?";

  public CommentWidget(ParentWidget parent, String text) {
    super(parent, text);
  }

  public String render() throws Exception {
    return "";
  }
}
```

Portées fictives

Parfois, le corps d'une instruction `while` ou `for` est fictif, comme dans le code ci-après. Je n'aime pas ce genre de structures et j'essaie de les éviter. Lorsque cela m'est impossible, je fais en sorte que le corps fictif soit correctement indenté et entouré d'accolades. Je ne compte plus le nombre de fois où j'ai été dupé par un point-virgule placé à la fin d'une boucle `while` sur la même ligne. Si ce point-virgule n'est pas indenté sur sa propre ligne, il n'est pas réellement visible.

```
while (dis.read(buf, 0, readBufferSize) != -1)
  ;
```

Règles d'une équipe



Chaque programmeur se fonde sur son propre jeu de règles de mise en forme préférées, mais, s'il travaille dans une équipe, alors, les règles de celle-ci s'appliquent.

Les développeurs de l'équipe doivent se mettre d'accord sur un même style de formatage, puis chaque membre doit le respecter. Un logiciel doit employer un style cohérent. Il ne faut pas que l'on ait l'impression qu'il a été écrit par un ensemble d'individus en désaccord.

Lorsque j'ai débuté le projet FitNesse en 2002, l'équipe s'est réunie afin de travailler sur un style de codage. Cela nous a demandé environ 10 minutes. Nous avons décidé de l'emplacement des accolades, de la taille de l'indentation, du nommage des classes, des variables et des méthodes, etc. Nous avons ensuite défini ces règles dans le système de mise en forme du code de notre IDE et les avons conservées depuis lors. Il s'agissait non pas de mes règles favorites, mais des règles établies par l'équipe. En tant que membre de cette équipe, je les ai respectées lors de l'écriture d'un code pour le projet FitNesse.

Vous ne devez pas oublier qu'un bon système logiciel est constitué d'un ensemble de documents dont la lecture est agréable. Ils doivent avoir un style cohérent et fluide. Le lecteur doit être certain que la mise en forme rencontrée dans un fichier source aura la même signification dans les autres. Il ne faut pas compliquer inutilement le code source en employant des styles individuels différents.

Règles de mise en forme de l'Oncle Bob

Les règles que j'emploie sont très simples et sont illustrées par le Listing 5.6. Vous pouvez considérer cet exemple comme un document de standardisation d'un bon codage.

Listing 5.6 : `CodeAnalyzer.java`

```
public class CodeAnalyzer implements JavaFileAnalysis {
  private int lineCount;
  private int maxLineWidth;
  private int widestLineNumber;
  private LineWidthHistogram lineWidthHistogram;
  private int totalChars;

  public CodeAnalyzer() {
    lineWidthHistogram = new LineWidthHistogram();
  }

  public static List<File> findJavaFiles(File parentDirectory) {
    List<File> files = new ArrayList<File>();
    findJavaFiles(parentDirectory, files);
    return files;
  }

  private static void findJavaFiles(File parentDirectory, List<File> files) {
    for (File file : parentDirectory.listFiles()) {
      if (file.getName().endsWith(".java"))
        files.add(file);
      else if (file.isDirectory())
        findJavaFiles(file, files);
    }
  }

  public void analyzeFile(File javaFile) throws Exception {
    BufferedReader br = new BufferedReader(new FileReader(javaFile));
    String line;
    while ((line = br.readLine()) != null)
      measureLine(line);
  }

  private void measureLine(String line) {
    lineCount++;
    int lineSize = line.length();
    totalChars += lineSize;
    lineWidthHistogram.addLine(lineSize, lineCount);
    recordWidestLine(lineSize);
  }

  private void recordWidestLine(int lineSize) {
    if (lineSize > maxLineWidth) {
      maxLineWidth = lineSize;
      widestLineNumber = lineCount;
    }
  }
```

```
  public int getLineCount() {
    return lineCount;
  }

  public int getMaxLineWidth() {
    return maxLineWidth;
  }

  public int getWidestLineNumber() {
    return widestLineNumber;
  }

  public LineWidthHistogram getLineWidthHistogram() {
    return lineWidthHistogram;
  }

  public double getMeanLineWidth() {
    return (double)totalChars/lineCount;
  }

  public int getMedianLineWidth() {
    Integer[] sortedWidths = getSortedWidths();
    int cumulativeLineCount = 0;
    for (int width : sortedWidths) {
      cumulativeLineCount += lineCountForWidth(width);
      if (cumulativeLineCount > lineCount/2)
        return width;
    }
    throw new Error("Cannot get here");
  }

  private int lineCountForWidth(int width) {
    return lineWidthHistogram.getLinesforWidth(width).size();
  }

  private Integer[] getSortedWidths() {
    Set<Integer> widths = lineWidthHistogram.getWidths();
    Integer[] sortedWidths = (widths.toArray(new Integer[0]));
    Arrays.sort(sortedWidths);
    return sortedWidths;
  }
}
```

6

Objets et structures de données

Si nous voulons que nos variables soient privées, c'est pour une bonne raison : nous voulons que personne d'autre en dépende. Nous voulons garder la liberté de modifier leur type ou leur implémentation sur un coup de tête ou une envie soudaine. Dans ce cas, pourquoi tant de programmeurs ajoutent automatiquement des accesseurs (méthodes `get` et `set`) à leurs objets, exposant ainsi leurs variables privées comme si elles étaient publiques ?

Abstraction de données

Examinons la différence entre le Listing 6.1 et le Listing 6.2. Ils représentent tous deux les données d'un point dans le plan cartésien, mais un seul expose son implémentation.

Listing 6.1 : Classe `Point` concrète

```
public class Point {
  public double x;
  public double y;
}
```

Listing 6.2 : Classe `Point` abstraite

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

Le Listing 6.2 a ceci d'intéressant qu'il n'existe aucun moyen de savoir si l'implémentation est fondée sur des coordonnées rectangulaires ou polaires, voire ni l'une ni l'autre. Malgré tout, l'interface représente incontestablement une structure de données.

Toutefois, elle ne représente pas uniquement une structure de données. Les méthodes imposent une politique d'accès. Nous pouvons lire indépendamment chaque coordonnée, mais nous devons les fixer ensemble par une opération atomique.

Le Listing 6.1, quant à lui, est très clairement mis en œuvre à l'aide de coordonnées rectangulaires et nous oblige à les manipuler de manière indépendante. Il expose l'implémentation. Bien évidemment, il continuerait d'exposer l'implémentation même si les variables étaient privées et si nous utilisions des accesseurs pour chacune.

Pour masquer l'implémentation, il ne suffit pas de placer une couche de fonctions devant les variables. Ce problème concerne les abstractions ! Une classe ne peut pas se contenter de présenter ses variables au travers d'accesseurs. À la place, elle doit exposer des interfaces abstraites qui permettent à ses utilisateurs de manipuler l'*essence* des données, sans avoir à en connaître l'implémentation.

Examinons le Listing 6.3 et le Listing 6.4. Le premier emploie des termes concrets pour indiquer le niveau de carburant dans un véhicule, tandis que le second utilise pour cela l'abstraction des pourcentages. Dans la classe concrète, nous pouvons être pratiquement certains qu'il s'agit de méthodes d'accès aux variables. Dans le cas abstrait, nous n'avons aucun indice sur la forme des données.

Listing 6.3 : Classe `Vehicle` concrète

```
public interface Vehicle {
  double getFuelTankCapacityInGallons();
  double getGallonsOfGasoline();
}
```

Listing 6.4 : Classe `Vehicle` abstraite

```
public interface Vehicle {
  double getPercentFuelRemaining();
}
```

De ces deux approches, il est préférable d'opter pour la seconde. Nous ne souhaitons pas exposer les détails de nos données. Nous voulons à la place exprimer nos données en termes abstraits. Cela ne passe pas simplement par l'emploi d'interfaces et/ou d'accesseurs. Il faut une réflexion sérieuse sur la meilleure manière de représenter les données contenues dans un objet. La pire solution consiste à ajouter sans discernement des méthodes d'accès.

Antisymétrie données/objet

Les deux exemples précédents révèlent la différence entre les objets et les structures de données. Les objets cachent leurs données derrière des abstractions et fournissent des fonctions qui manipulent ces données. Les structures de données exposent directement leurs données et ne fournissent aucune fonction significative. Ces deux concepts sont virtuellement opposés. La différence peut sembler évidente, mais ses implications sont profondes.

Prenons l'exemple des formes procédurales du Listing 6.5. La classe `Geometry` opère sur les trois classes de formes, qui sont de simples structures de données sans aucun comportement. Tout le comportement est défini dans la classe `Geometry`.

Listing 6.5 : Formes procédurales

```
public class Square {
  public Point topLeft;
  public double side;
}

public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}

public class Circle {
  public Point center;
  public double radius;
}
```

```
public class Geometry {
  public final double PI = 3.141592653589793;

  public double area(Object shape) throws NoSuchShapeException
  {
    if (shape instanceof Square) {
      Square s = (Square)shape;
      return s.side * s.side;
    }
    else if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

Les programmeurs orientés objet pourraient faire la grimace devant ce code et critiquer son approche procédurale — ils auraient raison. Cependant, le mépris n'est pas justifié. Voyons ce qui se passe si une fonction `perimeter()` est ajoutée à `Geometry`. Les classes de formes ne sont pas affectées ! Aucune autre classe qui dépend des formes n'est affectée ! En revanche, si nous ajoutons une nouvelle forme, nous devons modifier toutes les fonctions de `Geometry` pour en tenir compte. Relisez à nouveau ces derniers points. Vous pouvez remarquer que ces deux critères sont diamétralement opposés.

Examinons à présent le cas de la solution orientée objet du Listing 6.6, où la méthode `area()` est polymorphe. Aucune classe `Geometry` n'est requise. Si nous ajoutons une nouvelle forme, aucune des *fonctions* existantes n'est affectée mais, si nous ajoutons une nouvelle fonction, toutes les *formes* doivent être modifiées[1] !

Listing 6.6 : Formes polymorphes

```
public class Square implements Shape {
  private Point topLeft;
  private double side;

  public double area() {
    return side*side;
  }
}
```

1. Il existe des solutions pour contourner cet inconvénient et les concepteurs orientés objet expérimentés les connaissent parfaitement : le motif VISITEUR ou la double distribution, par exemple. Mais les coûts intrinsèques à ces techniques ne sont pas négligeables et elles renvoient généralement la structure vers une approche procédurale.

```
public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;

  public double area() {
    return height * width;
  }
}

public class Circle implements Shape {
  private Point center;
  private double radius;
  public final double PI = 3.141592653589793;

  public double area() {
    return PI * radius * radius;
  }
}
```

Nous constatons à nouveau la nature complémentaire de ces deux définitions ; elles sont virtuellement opposées ! Cela révèle la dichotomie fondamentale entre les objets et les structures de données :

> *Un code procédural (un code qui utilise des structures de données) facilite l'ajout de nouvelles fonctions sans modifier les structures de données existantes. Un code orienté objet facilite l'ajout de nouvelles classes sans modifier les fonctions existantes.*

L'inverse est également vrai :

> *Un code procédural complexifie l'ajout de nouvelles structures de données car toutes les fonctions doivent être modifiées. Un code orienté objet complexifie l'ajout de nouvelles fonctions car toutes les classes doivent être modifiées.*

Par conséquent, ce qui est difficile pour l'orienté objet est facile pour le procédural, tandis que ce qui est difficile pour le procédural est facile pour l'orienté objet !

Dans un système complexe, nous aurons parfois à ajouter de nouveaux types de données à la place de nouvelles fonctions. Dans ce cas, l'approche orientée objet est appropriée. En revanche, lorsque nous aurons à ajouter de nouvelles fonctions, non des types de données, un code procédural et des structures de données seront mieux adaptés.

Les programmeurs expérimentés savent très bien que l'idée du tout objet est un *mythe*. Parfois, nous voulons réellement de simples structures de données avec des procédures qui les manipulent.

Loi de Déméter

Il existe une heuristique très connue, appelée *loi de Déméter*[2], qui dit qu'un module ne doit pas connaître les détails internes des *objets* qu'il manipule. Nous l'avons vu à la section précédente, les objets cachent leurs données et exposent des opérations. Autrement dit, un objet ne doit pas présenter sa structure interne au travers d'accesseurs car, en procédant ainsi, il expose au lieu de cacher sa structure interne.

Plus précisément, la loi de Déméter stipule qu'une méthode *f* d'une classe *C* ne doit appeler que les méthodes des éléments suivants :

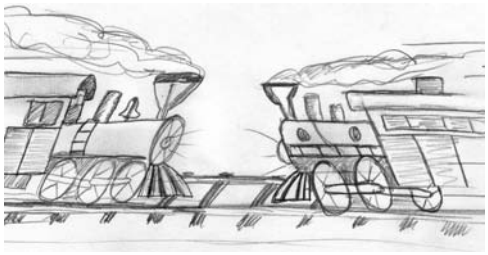
- *C* ;
- un objet créé par *f* ;
- un objet passé en argument à *f* ;
- un objet contenu dans une variable d'instance de *C*.

La méthode ne doit *pas* invoquer les méthodes sur les objets qui sont retournés par les fonctions autorisées. Autrement dit, il ne faut pas parler aux inconnus, uniquement aux amis.

Le code suivant[3] transgresse la loi de Déméter (entre autres choses) car il appelle la fonction `getScratchDir()` sur la valeur de retour de `getOptions()` et invoque ensuite `getAbsolutePath()` sur la valeur de retour de `getScratchDir()`.

```
final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Catastrophe ferroviaire



Le code précédent est souvent appelé une *catastrophe ferroviaire* car il ressemble à un ensemble de wagons accrochés les uns aux autres. Ces chaînes d'appel sont généralement considérées comme du code peu soigné et doivent être évitées [G36]. Il est préférable de les décomposer de la manière suivante :

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

2. **http://fr.wikipedia.org/wiki/Loi_de_Déméter**.
3. Tiré du code du framework Apache.

Ces deux extraits de code transgressent-ils la loi de Déméter ? Le module englobant sait pertinemment que l'objet `ctxt` contient des options, qui contiennent un répertoire temporaire, qui possède un chemin absolu. Cela fait beaucoup de connaissances pour une fonction. La fonction appelante sait comment naviguer au travers de nombreux objets différents.

Selon que `ctxt`, `Options` et `ScratchDir` sont des objets ou des structures de données, il s'agira ou non d'une transgression de la loi de Déméter. Si ce sont des objets, alors leur structure interne doit être masquée, non exposée, et la connaissance des détails internes est une violation manifeste de la loi de Déméter. En revanche, si `ctxt`, `Options` et `ScratchDir` sont simplement des structures de données sans comportement, elles exposent alors naturellement leur structure interne et la loi de Déméter ne s'applique donc pas.

L'emploi des accesseurs perturbe la question. Si le code avait été écrit comme suit, nous ne nous serions sans doute pas interrogés à propos d'une quelconque transgression.

```
final String outputDir = ctxt.options.scratchDir.absolutePath;
```

Ce problème serait beaucoup moins déroutant si les structures de données avaient simplement des variables publiques et aucune fonction publique, et si les objets avaient des variables privées et des fonctions publiques. Cependant, il existe des frameworks et des standards (comme les "beans") qui imposent aux structures de données, même simples, d'avoir des accesseurs et des mutateurs.

Hybrides

Cette confusion conduit parfois à des structures hybrides malheureuses qui sont pour moitié des objets et pour moitié des structures de données. Elles fournissent des fonctions qui réalisent des actions significatives et possèdent également des variables publiques ou des accesseurs et des mutateurs publics qui, en fait, rendent publiques les variables privées. Les autres fonctions externes sont alors tentées d'utiliser ces variables à la manière dont un programme procédural utiliserait une structure de données[4].

Avec de tels hybrides, il est difficile d'ajouter non seulement de nouvelles fonctions, mais également de nouvelles structures de données. Ils représentent le pire des deux mondes. Vous devez absolument éviter d'en créer. Ils sont signes d'une conception confuse dans laquelle les auteurs ne sont pas certains ou, pire, ne savent pas s'ils doivent protéger les fonctions ou les types.

4. Cette situation est parfois appelée "envie de fonctionnalité" dans [Refactoring].

Cacher la structure

Et si `ctxt`, `options` et `scratchDir` étaient des objets avec un véritable comportement ? Dans ce cas, puisque les objets sont supposés cacher leur structure interne, nous ne pourrions pas les parcourir. Mais, alors, comment pourrions-nous obtenir le chemin absolu du répertoire temporaire ?

```
ctxt.getAbsolutePathOfScratchDirectoryOption();
```

ou

```
ctx.getScratchDirectoryOption().getAbsolutePath();
```

La première solution pourrait conduire à une explosion du nombre de méthodes dans l'objet `ctxt`. La seconde suppose que `getScratchDirectoryOption()` retourne une structure de données, non un objet. Aucune des deux n'est satisfaisante.

Si `ctxt` est un objet, nous devons lui demander de *faire quelque chose* ; nous ne devons pas lui demander quelque chose concernant ses détails internes. Pourquoi avons-nous besoin du chemin absolu du répertoire temporaire ? À quoi va-t-il nous servir ? Examinons le code suivant, qui provient du même module (beaucoup plus loin) :

```
String outFile = outputDir + "/" + className.replace('.', '/') + ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

Le mélange de différents niveaux de détails [G6] [G34] est un tantinet troublant. Des points, des barres obliques, des extensions de fichiers et des objets `File` ne doivent pas être aussi négligemment mélangés et mêlés au code englobant. En faisant fi de cet aspect, nous pouvons déterminer que le chemin absolu du répertoire par défaut nous sert à créer un fichier temporaire d'un nom donné.

Par conséquent, il est préférable de demander à l'objet `ctxt` d'implémenter cette fonctionnalité :

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

Il ne semble pas déraisonnable de demander un tel comportement à un objet ! Cela permet à `ctxt` de cacher ses détails internes et d'empêcher la fonction courante de transgresser la loi de Déméter en parcourant les objets qu'elle n'est pas censée connaître.

Objets de transfert de données

La forme la plus pure d'une structure de données est une classe avec des variables publiques et aucune fonction. Nous les appelons parfois objets de transfert de données (DTO, *Data Transfer Object*). Les DTO sont des structures très utiles, en particulier pour les communications avec les bases de données ou les messages provenant des

sockets. Ils se trouvent souvent au début d'une séquence d'étapes de conversion qui permettent de transformer des données brutes issues d'une base de données en des objets du code applicatif.

Le "bean", illustré au Listing 6.7, en est une forme plus répandue. Les beans possèdent des variables privées manipulées par des accesseurs. La quasi-encapsulation des beans permet aux puristes de l'orienté objet de se sentir mieux, mais n'apporte généralement aucun autre avantage.

Listing 6.7 : `Address.java`

```
public class Address {
  private String street;
  private String streetExtra;
  private String city;
  private String state;
  private String zip;

  public Address(String street, String streetExtra,
                 String city, String state, String zip) {
    this.street = street;
    this.streetExtra = streetExtra;
    this.city = city;
    this.state = state;
    this.zip = zip;
  }

  public String getStreet() {
    return street;
  }

  public String getStreetExtra() {
    return streetExtra;
  }

  public String getCity() {
    return city;
  }

  public String getState() {
    return state;
  }

  public String getZip() {
    return zip;
  }
}
```

Enregistrement actif

Les enregistrements actifs (*Active Record*) sont des formes particulières de DTO. Il s'agit de structures de données avec des variables publiques (ou avec un accès comme

un bean), mais qui offrent généralement des fonctions de navigation telles que `save` et `find`. Ces enregistrements actifs sont habituellement des transformations directes des tables de bases de données ou d'autres sources de données.

Malheureusement, de nombreux programmeurs s'acharnent à traiter ces structures de données comme s'il s'agissait d'objets, en leur ajoutant des méthodes métiers. C'est tout à fait maladroit car ils créent alors un hybride entre une structure de données et un objet.

La solution consiste évidemment à manipuler un enregistrement actif comme une structure de données et à créer des objets séparés qui contiennent les règles métiers et cachent leurs données internes (qui sont probablement des instances de l'enregistrement actif).

Conclusion

Les objets exposent un comportement et masquent les données. Il est ainsi facile d'ajouter de nouvelles sortes d'objets sans changer les comportements existants. Mais il est également plus difficile d'ajouter de nouveaux comportements à des objets existants. Les structures de données exposent des données et n'ont pas de comportement significatif. Il est ainsi facile d'ajouter de nouveaux comportements à des structures de données existantes, mais difficile d'ajouter de nouvelles structures de données à des fonctions existantes.

Dans un système, nous avons parfois besoin d'une souplesse d'ajout de nouveaux types de données et choisissons alors une implémentation fondée sur des objets. Si nous voulons disposer d'une souplesse d'ajout de nouveaux comportements, alors, nous optons pour les types de données et les procédures. Les bons développeurs de logiciels abordent ces questions sans préjugé et retiennent l'approche adaptée à leurs besoins.

7

Gestion des erreurs

Par Michael Feathers

La présence d’un chapitre sur le traitement des erreurs dans un livre qui explique comment coder proprement pourrait sembler déplacée. Toutefois, la gestion des erreurs fait partie des tâches indispensables dans un programme. L’entrée peut être anormale et les périphériques peuvent échouer. Bref, les choses peuvent mal se passer et, lorsque c’est le cas, nous sommes, en tant que programmeurs, responsables du bon comportement de notre code.

Le lien avec le code propre doit être évident. De nombreuses bases de code sont totalement dominées par le traitement des erreurs. Attention, je ne sous-entends pas qu’elles se focalisent sur la gestion des erreurs, mais je dis qu’il est quasiment impossible de

comprendre ce que le code réalise car il est encombré d'instructions de gestion des erreurs. Le traitement des erreurs est important, *mais s'il masque la logique il est mauvais*.

Dans ce chapitre, nous présentons des techniques et des conseils qui conduisent à du code clair et robuste, c'est-à-dire du code qui gère les erreurs avec élégance et style.

Utiliser des exceptions à la place des codes de retour

Revenons à l'époque où de nombreux langages ne connaissaient pas les exceptions. Les techniques de gestion et de notification des erreurs étaient alors limitées. Nous positionnions un indicateur d'erreur ou retournions un code d'erreur que l'appelant pouvait consulter. Le Listing 7.1 illustre ces approches.

Listing 7.1 : `DeviceController.java`

```
public class DeviceController {
  ...
  public void sendShutDown() {
    DeviceHandle handle = getHandle(DEV1);
    // Vérifier l'état du périphérique.
    if (handle != DeviceHandle.INVALID) {
      // Placer l'état du périphérique dans l'enregistrement.
      retrieveDeviceRecord(handle);
      // S'il n'est pas suspendu, l'arrêter.
      if (record.getStatus() != DEVICE_SUSPENDED) {
        pauseDevice(handle);
        clearDeviceWorkQueue(handle);
        closeDevice(handle);
      } else {
        logger.log("Device suspended.  Unable to shut down");
      }
    } else {
      logger.log("Invalid handle for: " + DEV1.toString());
    }
  }
  ...
}
```

Ces solutions ont pour inconvénient d'encombrer le code appelant, qui doit vérifier la présence d'erreurs immédiatement après l'appel. Malheureusement, il est facile d'oublier de faire cette vérification. C'est pourquoi il est préférable de lancer une exception dès qu'une erreur est détectée. Le code appelant est plus propre puisque sa logique n'est pas masquée par le traitement des erreurs.

Le Listing 7.2 montre une version du code dans laquelle des exceptions sont lancées depuis les méthodes qui détectent les erreurs.

Listing 7.2 : `DeviceController.java` **(avec les exceptions)**

```
public class DeviceController {
  ...

  public void sendShutDown() {
    try {
      tryToShutDown();
    } catch (DeviceShutDownError e) {
      logger.log(e);
    }
  }

  private void tryToShutDown() throws DeviceShutDownError {
    DeviceHandle handle = getHandle(DEV1);
    DeviceRecord record = retrieveDeviceRecord(handle);

    pauseDevice(handle);
    clearDeviceWorkQueue(handle);
    closeDevice(handle);
  }

  private DeviceHandle getHandle(DeviceID id) {
    ...
    throw new DeviceShutDownError("Invalid handle for: " + id.toString());
    ...
  }

  ...
}
```

Vous pouvez constater combien il est plus clair. Il ne s'agit pas simplement d'une question d'esthétique. Le code est meilleur car les deux aspects qui étaient enchevêtrés, l'algorithme d'arrêt du périphérique et la gestion des erreurs, sont à présent séparés. Vous pouvez examiner chacun d'eux et les comprendre indépendamment.

Commencer par écrire l'instruction *try-catch-finally*

Les exceptions ont ceci d'intéressant qu'elles *définissent une portée* à l'intérieur du programme. Lorsque nous exécutons du code dans la partie `try` d'une instruction `try-catch-finally`, nous précisons que l'exécution peut s'interrompre à tout moment et reprendre dans le bloc `catch`.

D'une certaine manière, les blocs `try` sont des transactions. La partie `catch` doit laisser le programme dans un état cohérent, quel que soit ce qui s'est produit dans la partie `try`. C'est pourquoi, lorsque vous écrivez du code qui peut lancer des exceptions, nous vous conseillons de débuter avec une instruction `try-catch-finally`. Il est ainsi plus facile de définir ce à quoi l'utilisateur du code doit s'attendre, sans se préoccuper de ce qui pourrait mal se passer dans le bloc `try`.

Prenons pour exemple un code qui accède à un fichier et qui lit des objets sérialisés. Nous commençons par un test unitaire qui montre que nous recevrons une exception si le fichier n'existe pas :

```
@Test(expected = StorageException.class)
public void retrieveSectionShouldThrowOnInvalidFileName() {
  sectionStore.retrieveSection("invalid - file");
}
```

Ce test nous conduit à créer le bouchon (*stub*) suivant :

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  // Retour factice en attendant l'implémentation réelle.
  return new ArrayList<RecordedGrip>();
}
```

Notre test échoue car il ne lance pas d'exception. Nous modifions ensuite l'implémentation pour qu'elle tente d'accéder à un fichier invalide. Cette opération lance une exception :

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  try {
    FileInputStream stream = new FileInputStream(sectionName)
  } catch (Exception e) {
    throw new StorageException("retrieval error", e);
  }
  return new ArrayList<RecordedGrip>();
}
```

Le test réussit alors, car nous avons intercepté l'exception. À ce stade, nous pouvons remanier le code. Nous pouvons restreindre le type de l'exception interceptée afin qu'il corresponde au type de l'exception réellement lancée par le constructeur de `FileInputStream`, c'est-à-dire `FileNotFoundException` :

```
public List<RecordedGrip> retrieveSection(String sectionName) {
  try {
    FileInputStream stream = new FileInputStream(sectionName);
    stream.close();
  } catch (FileNotFoundException e) {
    throw new StorageException("retrieval error", e);
  }
  return new ArrayList<RecordedGrip>();
}
```

Puisque nous avons défini la portée avec une structure `try-catch`, nous pouvons employer le TDD pour mettre en place le reste de la logique. Elle sera ajoutée entre la création du `FileInputStream` et l'appel à `close`. Nous pouvons également faire comme si tout allait bien.

Vous devez essayer d'écrire des tests qui imposent des exceptions, puis ajouter un comportement au gestionnaire afin de satisfaire les tests. Ainsi, vous commencerez par définir la portée transactionnelle du bloc `try` et il vous sera plus facile de conserver la nature transactionnelle de cette portée.

Employer des exceptions non vérifiées

Le débat est terminé. Pendant des années, les programmeurs Java ont discuté des avantages et des inconvénients des exceptions vérifiées. Lorsque les exceptions vérifiées sont arrivées dans la première version de Java, elles semblaient une bonne idée. Une méthode indiquait par sa signature toutes les exceptions qu'elle pouvait passer à l'appelant. Par ailleurs, ces exceptions faisaient partie de son type. Il n'était pas possible de compiler le code si la signature ne correspondait pas aux utilisations du code.

À ce moment-là, les exceptions vérifiées semblaient être une bonne idée et elles avaient, il est vrai, certains avantages. Toutefois, il est à présent clair qu'elles ne sont pas indispensables au développement d'un logiciel robuste. C# ne dispose pas des exceptions vérifiées et, malgré quelques tentatives valeureuses, elles sont également absentes de C++, tout comme de Python ou de Ruby. Pourtant, il est tout à fait possible d'écrire des logiciels robustes avec tous ces langages. Puisque c'est le cas, nous devons décider si les exceptions vérifiées valent leur prix.

Quel est ce prix ? Les exceptions vérifiées constituent une violation du principe ouvert/fermé (OCP, *Open/Closed Principle*) [PPP]. Si nous lançons une exception vérifiée à partir d'une méthode du code et que la clause `catch` se trouve trois niveaux au-dessus, *nous devons déclarer cette exception dans la signature de chaque méthode qui se trouve entre notre code et la clause `catch`.* Autrement dit, une modification apportée à un bas niveau du code peut conduire à la modification d'une signature à de nombreux niveaux supérieurs. Les modules modifiés doivent être reconstruits et redéployés, même si le changement initial ne les concernait pas directement.

Examinons la hiérarchie des appels dans un grand système. Les fonctions au sommet de la hiérarchie invoquent des fonctions situées en dessous, qui appellent d'autres fonctions situées sous elles, à l'infini. Supposons à présent que l'une des fonctions du bas de la hiérarchie soit modifiée de manière telle qu'elle lance une exception. Si cette exception est vérifiée, nous devons alors ajouter une clause `throws` à la signature de la fonction. Cela signifie également que chaque fonction qui appelle notre fonction modifiée doit également être adaptée, soit pour intercepter la nouvelle exception, soit en ajoutant à sa signature la clause `throws` appropriée. À l'infini. Tout cela conduit à une cascade de modifications qui commence aux niveaux inférieurs du logiciel et remonte vers les niveaux les plus élevés ! L'encapsulation est remise en cause car toutes les fonctions qui se trouvent sur le chemin d'une exception lancée doivent connaître les détails de cette exception de bas niveau. Puisque les exceptions doivent nous permettre de traiter les erreurs à distance, il est dommage que les exceptions vérifiées rompent ainsi l'encapsulation.

Les exceptions vérifiées peuvent être utiles dans le cas d'une bibliothèque critique : il devient obligatoire de les intercepter. Mais, dans le développement d'applications générales, le coût de la dépendance dépasse les bénéfices potentiels.

Fournir un contexte avec les exceptions

Chaque exception lancée doit fournir un contexte suffisant pour déterminer l'origine et l'emplacement de l'erreur. En Java, nous pouvons obtenir une trace de la pile à partir de n'importe quelle exception, mais cette trace n'indique pas les intentions de l'opération qui a échoué.

Vous devez créer des messages d'erreur informatifs et les passer avec les exceptions. Mentionnez l'opération qui a échoué et le type de défaillance. Si vous maintenez un journal de fonctionnement de l'application, passez suffisamment d'informations pour consigner l'erreur dans la clause `catch`.

Définir les classes d'exceptions en fonction des besoins de l'appelant

Il existe plusieurs manières de classifier les erreurs, par exemple en fonction de leur origine (proviennent-elles d'un composant ou d'un autre ?) ou de leur type (s'agit-il de défaillances de périphériques, de défaillances réseau ou d'erreur de programmation ?). Néanmoins, lorsque nous définissons des classes d'exceptions dans une application, le point le plus important concerne *la manière de les intercepter.*

Prenons un exemple de classification médiocre. Voici une instruction `try-catch-finally` pour un appel à une bibliothèque tierce. Elle traite toutes les exceptions que l'appel peut provoquer :

```
ACMEPort port = new ACMEPort(12);

try {
  port.open();
} catch (DeviceResponseException e) {
  reportPortError(e);
  logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
  reportPortError(e);
  logger.log("Unlock exception", e);
} catch (GMXError e) {
  reportPortError(e);
  logger.log("Device response exception");
} finally {
  ...
}
```

Cette instruction contient beaucoup de redondance, ce qui ne devrait pas vous surprendre. Dans la plupart des traitements des exceptions, le travail effectué est relativement standard quoi qu'en soit la cause réelle. Nous devons enregistrer une erreur et nous assurer que le processus peut se poursuivre.

Dans ce cas, puisque nous savons que le travail à effectuer est quasiment le même quelle que soit l'exception, nous pouvons simplifier considérablement le code en enveloppant l'API invoquée et en retournant un type d'exception commun :

```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportError(e);
  logger.log(e.getMessage(), e);
} finally {
  ...
}
```

Notre classe `LocalPort` n'est qu'une simple enveloppe qui intercepte et traduit les exceptions lancées par la classe `ACMEPort` :

```
public class LocalPort {
  private ACMEPort innerPort;

  public LocalPort(int portNumber) {
    innerPort = new ACMEPort(portNumber);
  }

  public void open() {
    try {
      innerPort.open();
    } catch (DeviceResponseException e) {
      throw new PortDeviceFailure(e);
    } catch (ATM1212UnlockedException e) {
      throw new PortDeviceFailure(e);
    } catch (GMXError e) {
      throw new PortDeviceFailure(e);
    }
  }
  ...
}
```

Les enveloppes, telles que celle définie pour `ACMEPort`, peuvent se révéler très utiles. En réalité, envelopper des API tierces fait partie des bonnes pratiques. Lorsque nous enveloppons une API tierce, nous minimisons nos dépendances avec cette API : nous pouvons décider de changer de bibliothèque sans avoir à payer le prix fort. Par ailleurs, grâce à cette approche, il est également plus facile de simuler les invocations d'une API ou d'un logiciel tiers lorsque nous testons notre propre code.

Enfin, grâce aux enveloppes, nous ne sommes plus dépendants des choix de conception d'une API d'un fournisseur particulier. Nous pouvons définir l'API qui nous convient. Dans l'exemple précédent, nous avons défini un seul type d'exception pour la défaillance du périphérique `port` et cela nous a permis d'écrire un code plus propre.

Bien souvent, une seule classe d'exceptions suffit pour une section particulière d'un code. Les informations transmises avec l'exception permettent de distinguer les erreurs. Vous ne devez utiliser des classes différentes que si vous souhaitez intercepter une exception et en ignorer d'autres.

Définir le flux normal

Si vous suivez les conseils des sections précédentes, vous obtiendrez un haut niveau de séparation entre la logique métier et le traitement des erreurs. La majeure partie de votre code commencera à ressembler à un algorithme dépouillé propre. Cependant, cette approche a tendance à déplacer la détection des erreurs à la lisière du programme. Vous enveloppez les API externes afin de pouvoir lancer vos propres exceptions et vous définissez un gestionnaire au-dessus du code pour traiter les exécutions interrompues. En général, cette approche est la bonne, mais il arrive parfois que l'exécution ne doive pas être interrompue.

Prenons un exemple. Le code maladroit suivant additionne les dépenses dans une application de facturation :

```
try {
  MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
  m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
  m_total += getMealPerDiem();
}
```

Dans cette entreprise, si les repas sont des frais, ils sont inclus dans le total. Sinon l'employé reçoit des indemnités journalières. L'exception encombre la logique. Il serait préférable de ne pas avoir à traiter un cas particulier. Le code serait alors beaucoup plus simple :

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

Pour arriver à cette simplicité, nous pouvons modifier la classe `ExpenseReportDAO` afin qu'elle retourne toujours un objet `MealExpense`. En l'absence de frais de repas, elle retourne un objet `MealExpense` qui indique des indemnités journalières dans le total :

```
public class PerDiemMealExpenses implements MealExpenses {
  public int getTotal() {
    // Par défaut, retourner des indemnités journalières.
  }
}
```

Il s'agit du motif CAS PARTICULIER [Fowler]. Nous créons une classe ou configurons un objet de manière à traiter un cas particulier. Ainsi, le code du client n'a pas à s'occuper d'un comportement exceptionnel. Ce comportement est encapsulé dans l'objet dédié au cas particulier.

Ne pas retourner *null*

Tout propos sur le traitement des erreurs doit s'attarder sur certaines habitudes de codage qui ont tendance à attirer les erreurs. Retourner `null` arrive en tête de liste. Je ne compte plus le nombre de fois où j'ai rencontré des applications dont quasiment chaque ligne contenait un test avec `null`. En voici un exemple :

```
public void registerItem(Item item) {
  if (item != null) {
    ItemRegistry registry = peristentStore.getItemRegistry();
    if (registry != null) {
      Item existing = registry.segetItem(item.getID());
      if (existing.getBillingPeriod().hasRetailOwner()) {
        existing.register(item);
      }
    }
  }
}
```

Si vous travaillez avec du code de ce genre, vous ne le voyez sans doute pas, mais il est mauvais ! Lorsque nous retournons `null`, nous ne faisons qu'augmenter notre charge de travail et transmettre des problèmes aux appelants. Il suffit d'oublier une vérification de `null` pour perdre le contrôle de l'application.

Avez-vous remarqué que la deuxième ligne de l'instruction `if` imbriquée ne comportait aucun test de `null` ? Que va-t-il se passer à l'exécution si `persistentStore` est `null` ? Nous recevrons une exception `NullPointerException`, sans savoir si cette exception est interceptée à un niveau supérieur. Que ce soit le cas ou non, cette solution est *mauvaise*. Que devons-nous faire en réponse à une `NullPointerException` lancée depuis le plus profond de notre application ?

Nous pourrions facilement prétendre que le problème du code précédent vient du test de `null` manquant. En réalité, le problème vient du fait que ce code en contient beaucoup trop. Si vous êtes tenté d'écrire une méthode qui retourne `null`, vous devez à la place envisager de lancer une exception ou de retourner un objet CAS PARTICULIER. Si vous invoquez une méthode d'une API tierce qui retourne `null`, vous devez envisager d'envelopper cette méthode dans une méthode qui lance une exception ou retourne un objet de cas particulier.

En général, les objets CAS PARTICULIER sont un remède facile. Étudions le code suivant :

```
List<Employee> employees = getEmployees();
if (employees != null) {
  for(Employee e : employees) {
    totalPay += e.getPay();
  }
}
```

Pour le moment, `getEmployees` peut retourner `null`, mais est-ce bien nécessaire ? En modifiant `getEmployees` de manière à retourner une liste vide, nous pouvons nettoyer le code :

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
  totalPay += e.getPay();
}
```

Pour faciliter les choses, Java fournit `Collections.emptyList()`, qui retourne une liste immuable prédéfinie et qui convient parfaitement à notre objectif :

```
public List<Employee> getEmployees() {
  if( .. il n'y a pas d'employés .. )
    return Collections.emptyList();
}
```

En programmant de cette manière, vous réduirez les risques d'exception `NullPointer Exceptions` et votre code sera plus propre.

Ne pas passer *null*

Si écrire des méthodes qui retournent `null` est mauvais, passer `null` à des méthodes est encore pire. À moins de travailler avec une API qui vous oblige à passer `null`, vous devez tout faire pour éviter cette mauvaise pratique.

Voyons pourquoi à partir d'un exemple. Voici une méthode simple qui calcule une distance entre deux points :

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    return (p2.x – p1.x) * 1.5;
  }
  ...
}
```

Que se produit-il si l'appelant passe `null` en argument ?

```
calculator.xProjection(null, new Point(12, 13));
```

C'est évident, nous recevons une exception `NullPointerException`.

Comment pouvons-nous corriger ce problème ? Nous pourrions créer un nouveau type d'exception et le lancer :

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    if (p1 == null || p2 == null) {
      throw InvalidArgumentException(
        "Invalid argument for MetricsCalculator.xProjection");
    }
    return (p2.x - p1.x) * 1.5;
  }
}
```

Est-ce vraiment mieux ? Cette solution est sans doute meilleure qu'une exception de pointeur `null`, mais n'oubliez pas que nous avons défini un gestionnaire pour `InvalidArgumentException`. Que doit-il faire ? Dispose-t-il d'une bonne ligne de conduite ?

Il existe une autre solution. Nous pouvons employer des assertions :

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    assert p1 != null : "p1 should not be null";
    assert p2 != null : "p2 should not be null";
    return (p2.x - p1.x) * 1.5;
  }
}
```

Si cette méthode améliore la documentation, elle ne résout pas le problème. Si quelqu'un passe `null` en argument, nous recevons toujours une erreur d'exécution.

Dans la plupart des langages de programmation, il n'existe aucun bon moyen de gérer le passage accidentel d'un argument `null`. Puisque c'est ainsi, l'approche rationnelle consiste à interdire par défaut de passer `null`. En procédant ainsi, vous pouvez programmer en sachant que la présence de `null` dans une liste d'arguments signale un problème et obtenir un code qui contient bien moins d'erreurs d'étourderie.

Conclusion

Le code popre est lisible, mais il doit également être robuste. Ces deux objectifs ne sont pas contradictoires. Il est possible d'écrire du code propre robuste si la gestion des erreurs est traitée comme une question séparée, c'est-à-dire indépendamment de la logique principale. Selon le degré de séparation que nous sommes en mesure d'obtenir, nous pourrons réfléchir à cette question de manière indépendante et améliorer considérablement la facilité de maintenance du code.

8

Limites

Par James Grenning

Nous avons rarement la maîtrise de tous les logiciels présents dans nos systèmes. Nous achetons des paquetages tierce partie ou employons des logiciels open-source. Nous dépendons des équipes de notre propre société pour développer des composants ou des

sous-systèmes répondant à nos attentes. Nous devons parfois intégrer du code étranger dans le nôtre. Dans ce chapitre, nous examinons des pratiques et des techniques qui permettent de garder propres les frontières de nos logiciels.

Utiliser du code tiers

Il existe une tension naturelle entre le fournisseur d'une interface et son utilisateur. Les fournisseurs de paquetages et de frameworks tierce partie s'efforcent d'obtenir une applicabilité étendue afin de travailler dans de nombreux environnements et de répondre aux attentes d'un large public. *A contrario*, les utilisateurs attendent une interface qui se focalise sur leurs besoins précis. Cette tension peut amener des problèmes aux frontières de nos systèmes.

Prenons comme exemple la classe `java.util.Map`. Le Listing 8.1 montre que son interface est vaste, avec un grand nombre de fonctionnalités. Cette puissance et cette souplesse sont certes utiles, mais elles peuvent également constituer un handicap. Par exemple, notre application pourrait créer un `Map` et le faire circuler dans le code, mais en souhaitant qu'aucun des destinataires n'en supprime des éléments. La liste des méthodes des données au Listing 8.1 commence par `clear()`. Tout utilisateur du `Map` a donc la possibilité de le vider. Notre conception pourrait stipuler que seuls certains types d'objets doivent pouvoir être stockés dans le `Map`, mais cette classe ne fixe aucune contrainte sur les objets qu'un utilisateur peut y placer.

Listing 8.1 : Les méthodes de `Map`

```
clear() void - Map
containsKey(Object key) boolean - Map
containsValue(Object value) boolean - Map
entrySet() Set - Map
equals(Object o) boolean - Map
get(Object key) Object - Map
getClass() Class<? extends Object> - Object
hashCode() int - Map
isEmpty() boolean - Map
keySet() Set - Map
notify() void - Object
notifyAll() void - Object
put(Object key, Object value) Object - Map
putAll(Map t) void - Map
remove(Object key) Object - Map
size() int - Map
toString() String - Object
values() Collection - Map
wait() void - Object
wait(long timeout) void - Object
wait(long timeout, int nanos) void - Object
```

Si notre application a besoin d'un `Map` de `Sensor`, nous pouvons créer l'ensemble des sondes de la manière suivante :

```
Map sensors = new HashMap();
```

Lorsqu'une autre partie du code doit accéder à une sonde, elle utilise le code suivant :

```
Sensor s = (Sensor)sensors.get(sensorId);
```

Cette ligne risque d'apparaître en de nombreux endroits du code. Le client est donc responsable d'obtenir un `Object` à partir du `Map` et de le forcer dans le bon type. Cela fonctionne, mais ce code n'est pas propre. Par ailleurs, il ne raconte pas son histoire comme il le devrait. La lisibilité de ce code peut être énormément améliorée en employant les génériques :

```
Map<Sensor> sensors = new HashMap<Sensor>();
...
Sensor s = sensors.get(sensorId );
```

Cependant, cela ne résout en rien notre problème initial : `Map<Sensor>` offre plus de fonctionnalités que nécessaire ou que nous ne le souhaitons.

Si nous faisons circuler librement une instance de `Map<Sensor>` dans le système, il existe de nombreux endroits où nous devrons intervenir en cas de modification de l'interface de `Map`. Vous pourriez penser qu'un tel changement est peu probable, mais n'oubliez pas que cela s'est produit lorsque les génériques ont été ajoutés dans Java 5. Certains systèmes refusent de passer aux génériques en raison de l'ampleur colossale des modifications nécessaires à l'usage général de `Map`.

Voici une manière plus propre d'employer `Map`. Aucun utilisateur de `Sensors` n'est concerné par l'utilisation ou non des génériques. Ce choix est, et doit toujours rester, un détail d'implémentation.

```
public class Sensors {
  private Map sensors = new HashMap();

  public Sensor getById(String id) {
    return (Sensor) sensors.get(id);
  }

  //...
}
```

L'interface qui se situe aux limites (`Map`) est cachée. Elle peut évoluer avec très peu d'impact sur le reste de l'application. L'utilisation des génériques n'est plus un problème car le forçage de type et la gestion des types se font dans la classe `Sensors`.

L'interface est également adaptée et contrainte de manière à répondre aux besoins de l'application. Le code résultant est plus facile à comprendre et plus difficile à employer

à mauvais escient. La classe `Sensors` peut imposer des règles de conception et des règles métiers.

Nous ne suggérons aucunement que `Map` doive toujours être encapsulé ainsi. À la place, nous conseillons de ne pas passer des `Map` (ou toute autre interface située à une limite) librement dans le système. Si vous employez une interface limitrophe, comme `Map`, gardez-la à l'intérieur de la classe, ou de la famille de classes proche, dans laquelle elle est employée. Évitez de l'utiliser en valeur de retour ou de l'accepter en argument dans des API publiques.

Explorer et apprendre les limites

Le code tierce partie nous aide à proposer plus rapidement un plus grand nombre de fonctionnalités. Par où devons-nous commencer lorsque nous souhaitons employer un paquetage tiers ? Notre travail n'est pas de tester le code tiers, mais il peut être de notre intérêt d'écrire des tests pour celui que nous utilisons.

Supposons que la manière d'utiliser la bibliothèque tierce partie ne soit pas claire. Nous pouvons passer un jour ou deux, voire plus, à lire la documentation et à décider de la façon de l'employer. Ensuite, nous pouvons écrire notre code qui se fonde sur le code tiers et voir si nous obtenons ce que nous souhaitions. Il ne serait pas surprenant que nous devions passer par de longues sessions de débogage pour savoir si les bogues viennent de notre fait ou du leur.

Il est difficile de maîtriser le code tiers, tout comme il est difficile de l'intégrer. Il est doublement difficile de faire les deux à la fois. Pourquoi ne pas prendre une approche différente ? Au lieu d'expérimenter et d'essayer ces nouvelles choses dans notre code de production, nous pourrions écrire des tests afin de découvrir notre compréhension du code tiers. Jim Newkirk les appelle *tests d'apprentissage* [BeckTDD, p. 136–137].

Dans les tests d'apprentissage, nous invoquons l'API tierce partie comme nous pensons le faire dans notre application. Nous menons en réalité des expériences maîtrisées qui nous permettent de vérifier notre compréhension de cette API. Les tests se focalisent sur ce que nous voulons obtenir de l'API.

Apprendre *log4j*

Supposons que nous souhaitions employer le paquetage `log4j` d'Apache à la place de notre système de journalisation maison. Nous le téléchargeons et ouvrons la première page de la documentation. Très rapidement, nous pouvons écrire notre premier cas de test, qui doit afficher "hello" sur la console.

```
@Test
public void testLogCreate() {
  Logger logger = Logger.getLogger("MyLogger");
  logger.info("hello");
}
```

Lorsque nous l'exécutons, l'enregistreur produit une erreur qui signale que nous avons besoin de quelque chose appelé `Appender`. Nous continuons notre lecture et constatons qu'il existe un `ConsoleAppender`. Nous créons donc un `ConsoleAppender` et voyons si nous avons découvert le secret de la journalisation sur la console.

```
@Test
public void testLogAddAppender() {
  Logger logger = Logger.getLogger("MyLogger");
  ConsoleAppender appender = new ConsoleAppender();
  logger.addAppender(appender);
  logger.info("hello");
}
```

Cette fois-ci, nous apprenons que l'`Appender` ne dispose d'aucun flux de sortie. Bizarre, il semblerait pourtant logique d'en trouver un. Nous nous tournons donc vers Google pour rechercher de l'aide et essayons le code suivant :

```
@Test
public void testLogAddAppender() {
  Logger logger = Logger.getLogger("MyLogger");
  logger.removeAllAppenders();
  logger.addAppender(new ConsoleAppender(
      new PatternLayout("%p %t %m%n"),
      ConsoleAppender.SYSTEM_OUT));
  logger.info("hello");
}
```

Il fonctionne ! Un message de journalisation contenant "hello" s'affiche sur la console ! Il nous semble étrange d'avoir à indiquer au `ConsoleAppender` qu'il écrit sur la console.

Si nous retirons l'argument `ConsoleAppender.SYSTEM OUT`, nous constatons avec intérêt que "hello" est toujours affiché. En revanche, si nous retirons `PatternLayout`, l'exécution signale à nouveau qu'il manque un flux de sortie. Ce comportement est vraiment très étrange.

En examinant plus attentivement la documentation, nous découvrons que le constructeur par défaut de `ConsoleAppender` n'est pas "configuré", ce qui n'est pas très évident ou utile. Cela ressemble à un bogue, ou tout au moins à une incohérence, de `log4j`.

En approfondissant nos recherches sur Google, en poursuivant la lecture de la documentation et en procédant à d'autres tests, nous finissons par écrire le code du Listing 8.2. Nous avons découvert le fonctionnement de `log4j` et avons traduit ces connaissances en un petit jeu de tests unitaires simples.

Listing 8.2 : `LogTest.java`

```
public class LogTest {
    private Logger logger;

    @Before
    public void initialize() {
        logger = Logger.getLogger("logger");
        logger.removeAllAppenders();
        Logger.getRootLogger().removeAllAppenders();
    }
    @Test
    public void basicLogger() {
        BasicConfigurator.configure();
        logger.info("basicLogger");
    }

    @Test
    public void addAppenderWithStream() {
    logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p %t %m%n"),
        ConsoleAppender.SYSTEM_OUT));
        logger.info("addAppenderWithStream");
    }

    @Test
    public void addAppenderWithoutStream() {
    logger.addAppender(new ConsoleAppender(
        new PatternLayout("%p %t %m%n")));
        logger.info("addAppenderWithoutStream");
    }
}
```

Nous savons comment obtenir un mécanisme de journalisation simple sur la console et nous pouvons encapsuler ces connaissances dans notre propre classe de journalisation afin que le reste de l'application soit isolé de l'interface limitrophe de `log4j`.

Les tests d'apprentissage sont plus que gratuits

Les tests d'apprentissage ne coûtent finalement rien. Nous devions apprendre l'API, et l'écriture de ces tests nous a permis d'obtenir ces connaissances facilement et en toute indépendance. Les tests d'apprentissage étaient des expériences précises qui nous ont aidés à améliorer notre compréhension.

Les tests d'apprentissage ne sont pas seulement gratuits, ils ont un retour sur investissement positif. Si de nouvelles versions du paquetage tiers sont publiées, nous exécutons les tests d'apprentissage pour savoir s'il existe des différences de comportement.

Les tests d'apprentissage vérifient que les paquetages tiers employés fonctionnent comme nous le pensons. Une fois l'intégration terminée, rien ne garantit que le code tiers restera compatible avec nos besoins. Les auteurs peuvent modifier ce code pour

répondre à leurs propres besoins nouveaux. Ils corrigeront des bogues et ajouteront de nouvelles fonctionnalités. Chaque nouvelle version amène son nouveau jeu de risques. Si le paquetage tiers doit évoluer de manière incompatible avec nos tests, nous le saurons immédiatement.

Que vous ayez ou non besoin des connaissances apportées par les tests d'apprentissage, une limite précise doit être prise en charge par un ensemble de tests qui évaluent l'interface de la même manière que le code de production. Sans ces *tests aux limites* pour simplifier la migration, nous serions tentés de conserver l'ancienne version plus longtemps que conseillé.

Utiliser du code qui n'existe pas encore

Il existe un autre type de limite, qui sépare le connu de l'inconnu. Très souvent, nous trouvons dans le code des endroits où notre connaissance semble incomplète. Parfois, c'est l'autre côté de la limite qui est inconnu (tout au moins pour le moment). D'autres fois, nous décidons de ne pas regarder plus loin que la limite.

Il y a plusieurs années, je faisais partie d'une équipe de développement d'un logiciel pour un système de communications radio. Un sous-système, le transmetteur, nous était quasiment inconnu et les personnes en charge de ce sous-système n'étaient pas encore arrivées à la définition de son interface. Puisque nous ne voulions pas rester bloqués, nous avons commencé notre travail loin de la partie inconnue du code.

Nous avions une assez bonne idée de l'endroit où notre monde s'arrêtait et où le nouveau commençait. Au cours de notre travail, nous nous sommes parfois heurtés à cette frontière. Même si un épais brouillard d'ignorance obscurcissait notre vision après la limite, notre travail nous a permis de savoir ce que nous attendions de l'interface limitrophe. Nous voulions pouvoir exprimer la chose suivante :

> *Régler le transmetteur sur la fréquence fournie et émettre une représentation analogique des données qui arrivent sur ce flux.*

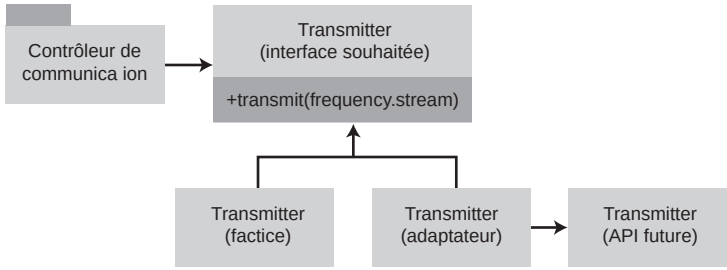
Nous n'avions aucune idée de la forme que cela prendrait car l'API n'était pas encore conçue. Nous avons donc choisi de laisser les détails pour plus tard.

Pour éviter d'être bloqués, nous avons défini notre propre interface, en choisissant un nom particulièrement accrocheur, comme `Transmitter`. Nous lui avons donné une méthode nommée `transmit` qui prend en arguments une fréquence et un flux de données. Il s'agissait de l'interface que nous *souhaitions* avoir.

Écrire l'interface que nous souhaitons avoir a pour avantage de la laisser sous notre contrôle. Nous pouvons ainsi garder un code client plus lisible et focalisé sur ce que nous tentons d'accomplir.

La Figure 8.1 montre que nous avons isolé les classes `CommunicationController` de l'API du transmetteur (qui était indéfinie et hors de notre contrôle). En employant notre propre interface, nous avons pu obtenir un code de `CommunicationController` propre et expressif. Dès que l'API du transmetteur a été définie, nous avons écrit `TransmitterAdapter` pour faire le lien. L'ADAPTATEUR [GOF] a encapsulé l'interaction avec l'API et a fourni un seul point de modification en cas d'évolution de cette API.

Figure 8.1
Prévoir le transmetteur.



Cette conception établit également un joint (*seam*) [WELC] très pratique dans le code pour les tests. En utilisant une classe `FakeTransmitter` adéquate, nous pouvons tester les classes `CommunicationsController`. Par ailleurs, nous pouvons créer des tests aux limites dès que nous disposons de la classe `TransmitterAPI` afin de vérifier que l'API est employée correctement.

Limites propres

Il se passe des choses intéressantes aux limites, notamment des modifications. Une bonne conception du logiciel s'accommode des modifications sans investissements et efforts importants. Lorsque nous employons du code qui n'est pas sous notre contrôle, il faut faire très attention à protéger notre investissement et à nous assurer que les évolutions ne seront pas trop onéreuses.

Le code aux limites a besoin d'une séparation claire et de tests qui définissent les attentes. Il faut éviter qu'une trop grande partie de notre code connaisse les détails du code tierce partie. Il est préférable de dépendre de quelque chose qui se trouve sous notre contrôle plutôt que de quelque chose que nous ne maîtrisons pas, de crainte que cette chose ne finisse par nous contrôler.

Pour gérer les éléments tierce partie, il faut réduire au maximum le nombre d'endroits du code qui y font référence. Nous pouvons les envelopper, comme cela a été le cas avec `Map`, ou utiliser un ADAPTATEUR pour faire le lien entre notre interface idéale et celle fournie. Dans tous les cas, notre code nous est parfaitement compréhensible, il incite à un usage interne cohérent pour traverser la limite et il affiche un nombre inférieur de points de maintenance en cas d'évolution du code tiers.

9

Tests unitaires

Notre métier a beaucoup évolué ces dix dernières années. En 1997, personne n’avait entendu parler du développement piloté par les tests (TDD, *Test Driven Development*). Pour la plupart d’entre nous, les tests unitaires représentaient de petits bouts de code jetables écrits uniquement pour vérifier que nos programmes "fonctionnaient". Nous écrivions consciencieusement nos classes et nos méthodes, puis nous concoctions du code *ad hoc* pour les tester. En général, cela impliquait une sorte de programme pilote simple qui nous permettait d’interagir manuellement avec le programme que nous avions développé.

Je me souviens avoir écrit, au milieu des années 1990, un programme C++ pour un système temps réel embarqué. Il s'agissait d'une simple minuterie dont la signature était la suivante :

```
void Timer::ScheduleCommand(Command* theCommand, int milliseconds)
```

L'idée était simple : la méthode `execute` de la classe `Command` était exécutée dans un nouveau thread après expiration de la temporisation indiquée en millisecondes. Le problème était comment tester cette fonctionnalité.

J'ai bâclé un petit programme pilote qui surveillait le clavier. Chaque fois qu'un caractère était saisi, il planifiait une commande qui reproduisait la saisie du même caractère cinq secondes plus tard. J'ai ensuite tapé une mélodie rythmée sur le clavier et attendu qu'elle soit rejouée à l'écran au bout de cinq secondes.

"I ... want-a-girl ... just ... like-the-girl-who-marr ... ied ... dear ... old ... dad."[1]

Je chantais cette mélodie pendant que je tapais sur la touche ".", puis je la chantais encore lorsque les points apparaissaient à l'écran.

C'était mon test ! Après l'avoir vu fonctionner et présenté à mes collègues, je m'en suis débarrassé.

Je l'ai déjà dit, notre profession a évolué. Aujourd'hui, j'écrirais un test qui vérifierait que tous les aspects de ce code fonctionnent comme je le souhaite. J'isolerais mon code du système d'exploitation au lieu d'appeler les fonctions de temporisation standard. Je simulerais ces fonctions afin de disposer d'un contrôle absolu sur le temps. Je planifierais des commandes qui positionnent des indicateurs booléens, puis je ferais avancer le temps, en surveillant ces indicateurs et en vérifiant qu'ils passent bien de `false` à `true` au moment opportun.

Dès qu'une suite de tests réussirait, je m'assurerais que ces tests pourraient être facilement exécutés par quiconque aurait besoin de travailler avec le code. Je m'assurerais que les tests et le code seraient sauvegardés ensemble dans le même paquetage source.

Si nous avons fait un long chemin, la route est encore longue. Les mouvements Agile et TDD ont encouragé de nombreux programmeurs à écrire des tests unitaires automatisés et beaucoup rejoignent les rangs quotidiennement. Mais, dans leur folle précipitation à ajouter les tests à leur discipline, de nombreux programmeurs ont oublié certains des points plus subtils, et importants, de l'écriture de bons tests.

1. N.d.T. : Extrait de la chanson *I Want a Girl*, de Harry Von Tilzer et William Dillon.

Les trois lois du TDD

À présent, chacun sait que le TDD nous impose de commencer par écrire les tests unitaires, avant le code de production. Toutefois, cette règle ne constitue que la partie visible de l'iceberg. Examinons les trois lois suivantes[2] :

Première loi. Vous ne devez pas écrire un code de production tant que vous n'avez pas écrit un test unitaire d'échec.

Deuxième loi. Vous devez uniquement écrire le test unitaire suffisant pour échouer ; l'impossibilité de compiler est un échec.

Troisième loi. Vous devez uniquement écrire le code de production suffisant pour réussir le test d'échec courant.

Ces trois lois nous enferment dans un cycle qui peut durer une trentaine de secondes. Les tests et le code de production sont écrits *ensemble*, en commençant par les tests.

Si nous travaillons de cette manière, nous écrivons des dizaines de tests par jour, des centaines de tests par mois et des milliers de tests par an. Si nous travaillons ainsi, ces tests couvrent virtuellement tout le code de production. L'ensemble de ces tests, dont la taille rivalise avec celle du code de production lui-même, peut présenter un problème de gestion.

Garder des tests propres

Il y a quelques années, on m'avait demandé de diriger une équipe qui avait expressément décidé que son code de test *ne devait pas* présenter le même niveau de qualité que son code de production. Elle acceptait que toutes ses règles de codage soient transgressées dans les tests unitaires. Le mot d'ordre était "vite fait mal fait". Il était inutile de bien nommer les variables, il était inutile d'écrire des fonctions de tests courtes et descriptives. Le code de test n'avait pas besoin d'être parfaitement conçu et consciencieusement cloisonné. Tant que le code de test fonctionnait et tant qu'il couvrait le code de production, il affichait une qualité suffisante.

En lisant cela, certains d'entre vous pourraient souscrire à ce choix. Peut-être, par le passé, avez-vous écrit des tests semblables à celui que j'ai écrit pour la classe `Timer`. Le pas est grand entre écrire des tests jetables et écrire un ensemble de tests unitaires automatisés. Par conséquent, comme l'équipe que j'encadrais, vous pourriez avoir décidé que les tests grossiers sont préférables à aucun test.

---

2. *Professionalism and Test-Driven Development*, Robert C. Martin, Object Mentor, IEEE Software, mai/juin 2007 (volume 24, n° 3), pages 32–36, **http://doi ieeecomputersociety.org/10.1109/MS.2007.85**.

Cependant, cette équipe n'avait pas compris que les tests grossiers étaient équivalents, sinon pires, à aucun test. En effet, les tests doivent évoluer avec le code de production. Plus les tests sont négligés, plus il est difficile de les modifier. Plus le code de test est embrouillé, plus le temps nécessaire à l'ajout de nouveaux tests dépassera celui nécessaire à écrire le nouveau code de production. Si vous modifiez le code de production, les anciens tests commencent à échouer et, en raison du code de test désordonné, il est plus difficile de les faire réussir à nouveau. Les tests deviennent alors un handicap de plus en plus important.

De version en version, le coût de maintenance de la suite de tests développée par mon équipe ne faisait qu'augmenter. Elle a fini par devenir l'unique sujet de grief chez les développeurs. Lorsque les managers ont demandé aux développeurs pourquoi leurs estimations étaient si grandes, ils ont accusé les tests. Finalement, ils ont été obligés d'abandonner l'intégralité de la suite de tests.

Cependant, sans une suite de tests, ils n'avaient plus la possibilité de vérifier que les modifications de la base de code ne remettaient pas en cause le fonctionnement. Sans les tests, ils ne pouvaient plus savoir si des modifications apportées à une partie du système ne créaient pas des dysfonctionnements dans d'autres parties. Par conséquent, leur taux d'échec a commencé à augmenter. En raison de cette recrudescence de défauts involontaires, ils ont commencé à craindre les modifications. Ils ont arrêté de nettoyer le code de production car ils avaient peur que les changements fassent plus de mal que de bien. Le code de production a commencé à se dégrader. Pour finir, ils n'avaient plus de tests, le code de production était embrouillé et criblé de bogues, les clients étaient mécontents et les développeurs avaient le sentiment que leurs efforts sur les tests étaient la raison de leur échec.

En un sens, ils n'avaient pas tort. Les tests ont bien été à l'origine de leur échec, mais uniquement pour avoir choisi d'accepter des tests négligés. S'ils avaient décidé de garder des tests propres, leur travail sur les tests ne les aurait pas conduits à l'échec. Je peux l'affirmer sans faillir car j'ai intégré et encadré de nombreuses équipes couronnées de succès qui employaient des tests unitaires *propres*.

La morale de cette histoire est simple : *le code de test est aussi important que le code de production.* Il ne représente pas un travail de seconde zone. Il exige réflexion, conception et soin. Il doit rester aussi propre que le code de production.

Les tests rendent possibles les "-ilities"[3]

Si vous ne maintenez pas vos tests, vous les perdrez. Sans les tests, vous perdez ce qui permet de conserver un code de production flexible. Vous avez peut-être du mal à le croire, mais c'est ainsi. Ce sont bien les *tests unitaires* qui permettent d'obtenir un code

flexible, maintenable et réutilisable. La raison en est simple. Si vous disposez de tests, vous n'avez pas peur de modifier le code. Sans les tests, chaque modification est un bogue potentiel. Quelle que soit la souplesse de votre architecture, quel que soit le cloisonnement de votre conception, sans les tests vous serez peu disposé à apporter des modifications de crainte d'introduire des bogues non détectés.

En revanche, *avec* des tests, cette crainte disparaît presque totalement. Plus la couverture des tests est vaste, plus les craintes s'amenuisent. Vous pouvez, quasiment en toute quiétude, modifier le code dont l'architecture et la conception ne sont pas embrouillées ou opaques. Bien évidemment, vous pouvez *améliorer* cette architecture et cette conception sans crainte !

Par conséquent, disposer d'une suite de tests automatisés qui couvrent l'ensemble du code de production est indispensable pour conserver une conception et une architecture aussi propres que possible. Les tests rendent possibles les "-ilities", car les tests permettent le *changement*.

Si vos tests sont négligés, votre capacité à modifier le code est entravée et vous commencez à perdre la possibilité d'améliorer sa structure. Plus les tests sont sales, plus votre code se salit. Pour finir, vous perdez les tests et votre code se dégrade.

Tests propres

Quelles sont les caractéristiques d'un test propre ? Elles sont au nombre de trois : lisibilité, lisibilité et lisibilité. La lisibilité est sans doute encore plus importante dans les tests unitaires qu'elle ne l'est dans le code de production. Comment peut-on obtenir les tests lisibles ? De la même manière que pour tout autre code : clarté, simplicité et densité d'une expression. Dans un test, vous devez dire beaucoup de choses avec aussi peu d'expressions que possible.

Prenons le code du Listing 9.1 extrait de FitNesse. Ces trois tests sont difficiles à comprendre et peuvent certainement être améliorés. Tout d'abord, ils contiennent énormément de code redondant [G5] dans les appels répétés à `addPage` et à `assertSubString`. Plus important encore, ce code est chargé de détails qui interfèrent avec l'expressivité du test.

3. N.d.T. : En ingénierie, les "-ilities" représentent les caractéristiques d'un projet exprimées sous forme d'adjectifs. Ce nom provient du suffixe partagé par ces mots en anglais, au pluriel, par exemple *adaptability, configurability* ou *maintainability.* Pour de plus amples informations, consultez l'article Wikipedia correspondant à l'adresse **http://en.wikipedia.org/wiki/Ilities**.

Listing 9.1 : `SerializedPageResponderTest.java`

```
public void testGetPageHieratchyAsXml() throws Exception
{
  crawler.addPage(root, PathParser.parse("PageOne"));
  crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.parse("PageTwo"));

  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
       new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
}

public void testGetPageHieratchyAsXmlDoesntContainSymbolicLinks()
throws Exception
{
  WikiPage pageOne = crawler.addPage(root, PathParser.parse("PageOne"));
  crawler.addPage(root, PathParser.parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.parse("PageTwo"));

  PageData data = pageOne.getData();
  WikiPageProperties properties = data.getProperties();
  WikiPageProperty symLinks = properties.set(SymbolicPage.PROPERTY_NAME);
  symLinks.set("SymPage", "PageTwo");
  pageOne.commit(data);

  request.setResource("root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response =
    (SimpleResponse) responder.makeResponse(
       new FitNesseContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubString("<name>PageOne</name>", xml);
  assertSubString("<name>PageTwo</name>", xml);
  assertSubString("<name>ChildOne</name>", xml);
  assertNotSubString("SymPage", xml);
}

public void testGetDataAsHtml() throws Exception
{
  crawler.addPage(root, PathParser.parse("TestPageOne"), "test page");

  request.setResource("TestPageOne");
  request.addInput("type", "data");
```

```
    Responder responder = new SerializedPageResponder();
    SimpleResponse response =
       (SimpleResponse) responder.makeResponse(
          new FitNesseContext(root), request);
    String xml = response.getContent();

    assertEquals("text/xml", response.getContentType());
    assertSubString("test page", xml);
    assertSubString("<Test", xml);
  }
```

Prenons, par exemple, les appels à `PathParser`. Ils convertissent des chaînes en instances de `PagePath` qui sont employées par les robots. Cette transformation n'a absolument aucun rapport avec le test courant et ne fait que masquer les intentions réelles. Les détails qui entourent la création de `responder` ainsi que l'obtention et le forçage de type `response` ne sont que du bruit. Ensuite, nous trouvons la manière maladroite de construire l'URL de requêtes à partir de `resource` et d'un argument. (Puisque j'ai participé à l'écriture de ce code, je peux me permettre ces critiques.)

En résumé, ce code n'a jamais été fait pour être lu. Le pauvre lecteur est submergé de détails qu'il doit comprendre avant que les tests prennent sens.

Examinons à présent les tests améliorés du Listing 9.2. Ils réalisent exactement la même chose, mais ils ont été remaniés afin d'être plus propres et plus explicatifs.

Listing 9.2 : `SerializedPageResponderTest.java` **(remanié)**

```
public void testGetPageHierarchyAsXml() throws Exception {
  makePages("PageOne", "PageOne.ChildOne", "PageTwo");

  submitRequest("root", "type:pages");

  assertResponseIsXML();
  assertResponseContains(
    "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
  );
}

public void testSymbolicLinksAreNotInXmlPageHierarchy() throws Exception {
  WikiPage page = makePage("PageOne");
  makePages("PageOne.ChildOne", "PageTwo");

  addLinkTo(page, "PageTwo", "SymPage");

  submitRequest("root", "type:pages");

  assertResponseIsXML();
  assertResponseContains(
    "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
  );
  assertResponseDoesNotContain("SymPage");
}
```

```
public void testGetDataAsXml() throws Exception {
  makePageWithContent("TestPageOne", "test page");

  submitRequest("TestPageOne", "type:data");

  assertResponseIsXML();
  assertResponseContains("test page", "<Test");
}
```

Le motif BUILD-OPERATE-CHECK[4] devient plus évident grâce à la structure de ces tests. Chacun est clairement divisé en trois parties. La première construit les données du test, la deuxième exploite ces données et la troisième vérifie que l'opération a produit les résultats escomptés.

Vous le constatez, la grande majorité des détails pénibles a été supprimée. Les tests vont directement au but et emploient uniquement les types de données et les fonctions dont ils ont réellement besoin. Quiconque lit ce code est capable de déterminer très rapidement son objectif, sans être trompé ou submergé par les détails.

Langage de test propre à un domaine

Les tests du Listing 9.2 illustrent la technique de construction d'un langage propre à un domaine pour vos tests. Au lieu de se fonder sur les API employées par les programmeurs pour manipuler le système, nous construisons un ensemble de fonctions et d'outils qui exploitent ces API et qui facilitent l'écriture et la lecture des tests. Cet ensemble devient une API spécialisée employée par les tests. Elle constitue un langage de test que les programmeurs utilisent pour simplifier l'écriture de leurs tests et aider les personnes qui les liront ultérieurement.

Cette API de test n'est pas conçue à l'avance. À la place, elle évolue à partir du remaniement continu du code de test, qui était trop encombré de détails. De la même manière que je suis passé du Listing 9.1 au Listing 9.2, les développeurs disciplinés remanieront leur code de test afin d'en obtenir une forme plus succincte et expressive.

Deux standards

En un sens, l'équipe que j'ai mentionnée au début de ce chapitre avait fait les choses comme il fallait. Le code de l'API de test employait bien des standards d'ingénierie différents de ceux du code de production. Cet ensemble de standards doit être simple, succinct et expressif, mais il ne doit pas nécessairement être aussi efficace que du code de production. En effet, il fonctionne dans un environnement de test, non un environnement de production, et ces deux environnements présentent des besoins très différents.

4. **http://fitnesse.org/FitNesse.AcceptanceTestPatterns**.

Prenons le test du Listing 9.3. Je l'ai écrit pour le prototypage d'un système de contrôle de l'environnement. Sans entrer dans les détails, vous pouvez deviner que ce test vérifie si l'alarme de température faible, le système de chauffage et le ventilateur sont en marche lorsque la température "baisse trop".

Listing 9.3 : `EnvironmentControllerTest.java`

```
@Test
  public void turnOnLoTempAlarmAtThreashold() throws Exception {
    hw.setTemp(WAY_TOO_COLD);
    controller.tic();
    assertTrue(hw.heaterState());
    assertTrue(hw.blowerState());
    assertFalse(hw.coolerState());
    assertFalse(hw.hiTempAlarm());
    assertTrue(hw.loTempAlarm());
  }
```

Bien entendu, ce code contient de nombreux détails. Par exemple, que fait donc cette fonction `tic` ? En réalité, j'aimerais mieux que vous ne vous en occupiez pas lors de la lecture de ce texte. J'aimerais simplement que vous soyez d'accord sur le fait que l'état final du système est cohérent avec la température qui "baisse trop".

Lorsque vous lisez le test, vous notez que vos yeux ont besoin d'aller et de venir entre le nom de l'état contrôlé et la signification de cet état. Vous voyez `heaterState` et votre regard glisse vers la gauche sur `assertTrue`. Vous voyez `coolerState` et vos yeux doivent aller vers la gauche sur `assertFalse`. Ce mouvement obligatoire des yeux est pénible et non fiable. Il complique la lecture du test.

J'ai énormément augmenté la facilité de lecture de ce test en le transformant de manière à obtenir le Listing 9.4.

Listing 9.4 : `EnvironmentControllerTest.java` **(remanié)**

```
@Test
  public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
  }
```

Bien entendu, je cache l'existence de la fonction `tic` en créant la fonction `wayTooCold`. Mais le point important est la chaîne étrange dans l'appel à `assertEquals`. Une lettre majuscule signifie "allumé", une lettre minuscule signifie "éteint", et les lettres sont toujours données dans l'ordre suivant : `{heater, blower, cooler, hi temp alarm, lo temp alarm}`.

Même si cette solution s'apparente à une transgression de la règle concernant les associations mentales[5], elle me semble appropriée dans ce cas. Dès lors que vous connaissez la signification des lettres, votre regard glisse sur la chaîne et vous interprétez rapidement les résultats. La lecture du texte en devient presque plaisante. Examinez le Listing 9.5 et voyez combien il est facile de comprendre ces tests.

Listing 9.5 : `EnvironmentControllerTest.java` **(extrait plus long)**

```
@Test
  public void turnOnCoolerAndBlowerIfTooHot() throws Exception {
    tooHot();
    assertEquals("hBChl", hw.getState());
  }

  @Test
  public void turnOnHeaterAndBlowerIfTooCold() throws Exception {
    tooCold();
    assertEquals("HBchl", hw.getState());
  }

  @Test
  public void turnOnHiTempAlarmAtThreshold() throws Exception {
    wayTooHot();
    assertEquals("hBCHl", hw.getState());
  }

  @Test
  public void turnOnLoTempAlarmAtThreshold() throws Exception {
    wayTooCold();
    assertEquals("HBchL", hw.getState());
  }
```

La fonction `getState` est présentée au Listing 9.6. Vous remarquerez que ce code n'est pas très efficace. J'aurais probablement dû utiliser un `StringBuffer`.

Listing 9.6 : `MockControlHardware.java`

```
public String getState() {
    String state = "";
    state += heater ? "H" : "h";
    state += blower ? "B" : "b";
    state += cooler ? "C" : "c";
    state += hiTempAlarm ? "H" : "h";
    state += loTempAlarm ? "L" : "l";
    return state;
  }
```

Les `StringBuffer` sont assez laids. Même dans un code de production, je les évite lorsque le coût est réduit ; vous pouvez soutenir que le coût du code du Listing 9.6 est très faible. Cependant, cette application se trouve dans un système embarqué temps réel et il

5. Section "Éviter les associations mentales" au Chapitre 2.

est probable que les ressources de calcul et les ressources mémoire soient très limitées. En revanche, l'environnement de test ne présente probablement pas ces contraintes.

Voilà l'essence des deux standards. Certaines choses ne doivent jamais être faites dans un environnement de production, alors qu'elles sont totalement pertinentes dans un environnement de test. En général, cela concerne des problèmes d'efficacité mémoire ou de processeur. En revanche, cela ne concerne jamais des questions de propreté.

Une assertion par test

Dans une certaine école de pensée[6], il est dit que chaque fonction de test dans un test JUnit ne doit inclure qu'une et une seule instruction d'assertion. Cette règle peut sembler draconienne, mais son intérêt est illustré au Listing 9.5. Ces tests conduisent à une seule conclusion qui se révèle rapide et facile à comprendre.

Qu'en est-il du Listing 9.2 ? Il ne semble pas facile de réunir raisonnablement l'assertion que la sortie est au format XML et qu'elle contient certaines sous-chaînes. Cependant, nous pouvons décomposer le test en deux tests séparés, chacun avec sa propre assertion (voir Listing 9.7).

Listing 9.7 : `SerializedPageResponderTest.java` **(une seule assertion)**

```
public void testGetPageHierarchyAsXml() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

    whenRequestIsIssued("root", "type:pages");

    thenResponseShouldBeXML();
  }

  public void testGetPageHierarchyHasRightTags() throws Exception {
    givenPages("PageOne", "PageOne.ChildOne", "PageTwo");

    whenRequestIsIssued("root", "type:pages");

    thenResponseShouldContain(
      "<name>PageOne</name>", "<name>PageTwo</name>", "<name>ChildOne</name>"
    );
  }
```

Vous le remarquez, j'ai changé les noms des fonctions afin d'employer la convention classique *given-when-then* [RSpec]. Les tests sont ainsi encore plus faciles à lire. Malheureusement, le découpage des tests conduit à du code redondant.

6. Voir le billet de Dave Astel à l'adresse **http://www.artima.com/weblogs/viewpost.jsp?thread=35578**.

Nous pouvons éliminer la redondance à l'aide du motif PATRON DE MÉTHODE [GOF] et en plaçant les parties *given/when* dans la classe de base et les parties *then* dans les différentes classes dérivées. Ou nous pourrions créer une classe de test totalement distincte et placer les parties *given* et *when* dans la fonction `@Before`, et les parties *when* dans chaque fonction `@Test`. Mais ce mécanisme semble disproportionné par rapport aux problèmes. En fin de compte, je préfère les multiples assertions du Listing 9.2.

Je pense que la règle d'assertion unique est un bon guide[7]. J'essaie habituellement de créer un langage de test spécifique au domaine qui la prend en charge, comme dans le Listing 9.5. Toutefois, je n'hésite pas à placer plusieurs assertions dans un test. En résumé, je pense qu'il est cependant préférable d'essayer de minimiser le nombre d'assertions dans un test.

Un seul concept par test

En réalité, la règle serait plutôt que nous souhaitons tester un seul concept dans chaque fonction de test. Nous ne voulons pas de longues fonctions qui testent diverses choses hétéroclites l'une après l'autre. Le Listing 9.8 présente un test de ce type. Il devrait être décomposé en trois tests indépendants, car il concerne trois aspects indépendants. Lorsqu'elles sont réunies dans la même fonction, le lecteur est obligé de déterminer pourquoi chaque section se trouve là et ce qu'elle teste.

Listing 9.8

```
/**
 * Divers tests pour la méthode addMonths().
 */
public void testAddMonths() {
    SerialDate d1 = SerialDate.createInstance(31, 5, 2004);

    SerialDate d2 = SerialDate.addMonths(1, d1);
    assertEquals(30, d2.getDayOfMonth());
    assertEquals(6, d2.getMonth());
    assertEquals(2004, d2.getYYYY());

    SerialDate d3 = SerialDate.addMonths(2, d1);
    assertEquals(31, d3.getDayOfMonth());
    assertEquals(7, d3.getMonth());
    assertEquals(2004, d3.getYYYY());

    SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
    assertEquals(30, d4.getDayOfMonth());
    assertEquals(7, d4.getMonth());
    assertEquals(2004, d4.getYYYY());
}
```

7. "Ne pas s'écarter du code !"

Voici les trois fonctions de tests que nous devrions créer :

- *Étant donné* le dernier jour d'un mois de 31 jours (comme mai) :
  1. *Lorsque* nous ajoutons un mois, que le dernier jour de ce mois est le 30 (comme juin), *alors*, la date doit être le 30 de ce mois, non le 31.
  2. *Lorsque* nous ajoutons deux mois à cette date, avec le dernier mois ayant 31 jours, *alors*, la date doit être le 31.
- *Étant donné* le dernier jour d'un mois de 30 jours (comme juin) :
  1. *Lorsque* nous ajoutons un mois, que le dernier jour de ce mois est le 31, *alors*, la date doit être le 30, non le 31.

Formulé de cette manière, vous pouvez constater qu'une règle générale se cache au milieu des tests hétéroclites. Lorsque nous incrémentons le mois, la date ne peut pas être supérieure au dernier jour du mois. Autrement dit, si l'incrémentation du mois se fait le 28 février, nous devons obtenir le 28 mars. Ce test est absent et il serait bon de l'ajouter.

Le problème vient donc non pas des multiples assertions dans chaque section du Listing 9.8, mais du fait que plusieurs concepts sont testés. Par conséquent, la meilleure règle est probablement de minimiser le nombre d'assertions par concept et de tester un seul concept par fonction de test.

F.I.R.S.T.[8]

Les tests propres suivent cinq autres règles qui forment l'acronyme précédent :

Fast (*rapide*). Les tests doivent être rapides. Lorsque des tests s'exécutent lentement, il est peu probable qu'ils soient lancés fréquemment. Lorsque les tests ne sont pas exécutés très souvent, il est impossible de trouver les problèmes suffisamment tôt pour les corriger facilement. Vous n'êtes pas aussi libre de nettoyer le code, et il finit par se dégrader.

Independent (*indépendant*). Les tests ne doivent pas dépendre les uns des autres. Un test ne doit pas établir les conditions d'exécution du test suivant. Vous devez être en mesure d'exécuter chaque test indépendamment et dans l'ordre que vous voulez. Lorsque des tests dépendent l'un de l'autre, le premier qui échoue provoque une cascade d'échecs en aval rendant difficile le diagnostic et masquant les défauts en aval.

8. Matériel pédagogique d'Object Mentor.

Repeatable (*reproductible*). Les tests doivent pouvoir être reproduits dans n'importe quel environnement. Vous devez être en mesure d'exécuter les tests dans l'environnement de production, dans l'environnement de contrôle de qualité et sur votre portable pendant que vous êtes dans le train sans connexion réseau. Si vos tests ne sont pas reproductibles dans n'importe quel environnement, vous aurez toujours une excuse pour leur échec. Vous serez également incapable de les exécuter lorsque l'environnement n'est pas disponible.

Self-Validating (*auto validant*). Les tests doivent avoir un résultat binaire : ils réussissent ou ils échouent. Vous ne devez pas avoir à consulter un fichier de journalisation ou comparer manuellement deux fichiers texte différents pour savoir si les tests ont réussi. Si les tests ne sont pas auto validants, l'échec peut alors devenir subjectif et l'exécution des tests peut exiger une longue évaluation manuelle.

Timely (*au moment opportun*). Les tests doivent être écrits au moment opportun. Les tests unitaires doivent être écrits *juste avant* le code de production qui permet de les réussir. Si vous écrivez les tests après le code de production, vous constaterez qu'il sera difficile de tester ce code. Vous pourriez décider qu'un code de production est trop complexe à tester. Vous pourriez ne pas concevoir le code de production pour qu'il puisse être testé.

Conclusion

Nous n'avons fait qu'aborder ce sujet. Très clairement, un livre pourrait être dédié aux *tests propres*. Les tests sont aussi importants pour la santé d'un projet que peut l'être le code de production. Ils sont peut-être même plus importants, car ils préservent et améliorent la souplesse, les possibilités de maintenance et l'utilisabilité du code de production. C'est pourquoi vous devez constamment garder vos tests en bon état. Vous devez faire l'effort de les rendre expressifs et succincts. Inventez des API de test qui jouent le rôle de langage spécifique au domaine afin de vous aider à écrire les tests.

Si vous laissez les tests se dégrader, votre code se dégradera également. Gardez vos tests propres.

10

Classes

Avec Jeff Langr

Jusqu’à présent, notre intérêt s’est tourné vers la bonne écriture de lignes et de blocs de code. Nous nous sommes plongés dans la formulation correcte des fonctions et de leurs relations. Toutefois, malgré toute l’attention portée à l’expressivité des instructions et des fonctions, nous ne pourrons pas obtenir un code propre tant que nous n’aurons pas pris soin des niveaux supérieurs de l’organisation du code. Examinons la propreté des classes.

Organiser une classe

Selon la convention standard en Java, une classe doit débuter par une liste des variables. Les constantes statiques publiques, si elles existent, viennent en premier. Suivent ensuite les variables statiques privées, puis les variables d'instance privées. Les bonnes raisons d'ajouter des variables publiques sont rares.

Les fonctions publiques doivent suivre la liste des variables. En général, nous plaçons les fonctions privées invoquées par une fonction publique immédiatement après celle-ci. Cette approche est conforme à la règle de décroissance et permet de lire le programme comme un article de journal.

Encapsulation

Nous préférons garder privées nos variables et nos fonctions utilitaires, mais nous savons faire preuve de souplesse. Parfois, nous devons définir une variable ou une fonction utilitaire protégée afin qu'un test puisse y accéder. Pour nous, les tests gouvernent. Si un test du même paquetage doit invoquer une fonction ou accéder à une variable, nous la déclarerons protégée ou la placerons dans la portée du paquetage. Cependant, nous rechercherons tout d'abord un moyen de la conserver privée. Nous ne relâchons l'encapsulation qu'en dernier ressort.

De petites classes

La première règle est que les classes doivent être petites. La seconde règle est qu'elles doivent être encore plus petites que cela. Ne vous inquiétez pas, nous n'allons pas répéter notre propos du Chapitre 3 sur les fonctions. Toutefois, comme pour les fonctions, la concision est la première règle qui doit guider la conception des classes. La première question qui se pose est donc "petite jusqu'à quel point ?".

Pour les fonctions, nous mesurions la taille en comptant le nombre de lignes physiques. Pour les classes, nous choisissions une mesure différente : les *responsabilités* [RDD].

Le Listing 10.1 présente la classe `SuperDashboard`, qui expose environ 70 méthodes publiques. La plupart des développeurs seront d'accord pour dire que sa taille est trop grande. Certains qualifieraient même `SuperDashboard` de "classe divine".

Listing 10.1 : Beaucoup trop de responsabilités

```
public class SuperDashboard extends JFrame implements MetaDataUser
   public String getCustomizerLanguagePath()
   public void setSystemConfigPath(String systemConfigPath)
   public String getSystemConfigDocument()
   public void setSystemConfigDocument(String systemConfigDocument)
   public boolean getGuruState()
```

```
public boolean getNoviceState()
public boolean getOpenSourceState()
public void showObject(MetaObject object)
public void showProgress(String s)
public boolean isMetadataDirty()
public void setIsMetadataDirty(boolean isMetadataDirty)
public Component getLastFocusedComponent()
public void setLastFocused(Component lastFocused)
public void setMouseSelectState(boolean isMouseSelected)
public boolean isMouseSelected()
public LanguageManager getLanguageManager()
public Project getProject()
public Project getFirstProject()
public Project getLastProject()
public String getNewProjectName()
public void setComponentSizes(Dimension dim)
public String getCurrentDir()
public void setCurrentDir(String newDir)
public void updateStatus(int dotPos, int markPos)
public Class[] getDataBaseClasses()
public MetadataFeeder getMetadataFeeder()
public void addProject(Project project)
public boolean setCurrentProject(Project project)
public boolean removeProject(Project project)
public MetaProjectHeader getProgramMetadata()
public void resetDashboard()
public Project loadProject(String fileName, String projectName)
public void setCanSaveMetadata(boolean canSave)
public MetaObject getSelectedObject()
public void deselectObjects()
public void setProject(Project project)
public void editorAction(String actionName, ActionEvent event)
public void setMode(int mode)
public FileManager getFileManager()
public void setFileManager(FileManager fileManager)
public ConfigManager getConfigManager()
public void setConfigManager(ConfigManager configManager)
public ClassLoader getClassLoader()
public void setClassLoader(ClassLoader classLoader)
public Properties getProps()
public String getUserHome()
public String getBaseDir()
public int getMajorVersionNumber()
public int getMinorVersionNumber()
public int getBuildNumber()
public MetaObject pasting(
   MetaObject target, MetaObject pasted, MetaProject project)
public void processMenuItems(MetaObject metaObject)
public void processMenuSeparators(MetaObject metaObject)
public void processTabPages(MetaObject metaObject)
public void processPlacement(MetaObject object)
public void processCreateLayout(MetaObject object)
public void updateDisplayLayer(MetaObject object, int layerIndex)
public void propertyEditedRepaint(MetaObject object)
public void processDeleteObject(MetaObject object)
public boolean getAttachedToDesigner()
```

```
    public void processProjectChangedState(boolean hasProjectChanged)
    public void processObjectNameChanged(MetaObject object)
    public void runProject()
    public void setAçowDragging(boolean allowDragging)
    public boolean allowDragging()
    public boolean isCustomizing()
    public void setTitle(String title)
    public IdeMenuBar getIdeMenuBar()
    public void showHelper(MetaObject metaObject, String propertyName)
    // ... plusieurs autres méthodes non publiques suivent ...
}
```

Et si `SuperDashboard` ne proposait que les méthodes indiquées au Listing 10.2 ?

Listing 10.2 : Est-ce suffisamment petit ?

```
public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

Cinq méthodes, cela ne semble pas exagéré. Toutefois, dans ce cas, c'est beaucoup de méthodes car `SuperDashboard` a trop de *responsabilités*.

Le nom d'une classe doit décrire ses responsabilités. En réalité, le nommage est sans doute la première manière de déterminer la taille d'une classe. S'il nous est impossible de trouver un nom concis, alors, la classe est sans doute trop grande. Plus le nom de la classe est ambigu, plus elle risque d'avoir trop de responsabilités. Par exemple, les noms de classes qui comprennent des mots ambigus, comme `Processor`, `Manager` ou `Super`, font souvent allusion à un cumul regrettable de responsabilités.

Nous devons également être en mesure de décrire brièvement la classe, en employant environ 25 mots, sans utiliser "si", "et", "ou" ou "mais". Essayons de décrire la classe `SuperDashboard` : "La classe `SuperDashboard` permet d'accéder au dernier composant qui possédait le focus *et* d'effectuer un suivi des numéros de version et de compilation." Le premier "et" signale que les responsabilités de `SuperDashboard` sont trop nombreuses.

Principe de responsabilité unique

Le principe de responsabilité unique (SRP, *Single Responsibility Principle*)[1] stipule qu'il ne doit exister qu'une et une seule raison de modifier une classe ou un module. Ce principe nous donne à la fois une définition de la responsabilité et des indications sur la taille d'une classe. Les classes doivent avoir une seule responsabilité, c'est-à-dire une seule raison d'être modifiée.

1. Vous trouverez de plus amples informations sur ce principe dans [PPP].

La petite, en apparence, classe `SuperDashboard` du Listing 10.2 a deux raisons d'être modifiée. Premièrement, elle effectue un suivi des informations de version qui devront être mises à jour chaque fois que le logiciel est livré. Deuxièmement, elle gère des composants Java Swing (elle dérive de `JFrame`, c'est-à-dire la représentation Swing d'une fenêtre graphique de premier niveau). S'il nous faudra modifier le numéro de version en cas de changement du code Swing, ce n'est pas nécessairement la seule raison : nous pourrions avoir à modifier les informations de version en fonction de changements apportés dans le code d'une autre partie du système.

Lorsque nous essayons d'identifier les responsabilités (c'est-à-dire les raisons d'être modifié), il nous est plus facile de reconnaître et de créer de meilleures abstractions dans notre code. Nous pouvons facilement extraire les trois méthodes de `SuperDashboard` qui concernent les informations de version et les placer dans une classe séparée nommée `Version` (voir Listing 10.3). Par ailleurs, cette nouvelle classe présente un potentiel élevé de réutilisation dans d'autres applications !

Listing 10.3 : Une classe avec une seule responsabilité

```
public class Version {
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

SRP fait partie des concepts les plus importants de la conception orientée objet. Il est également l'un des concepts les plus simples à comprendre et à respecter. Néanmoins, il est également le principe de conception des classes le plus malmené. Nous rencontrons régulièrement des classes qui font beaucoup trop de choses. Pourquoi ?

Faire en sorte que le logiciel fonctionne et faire en sorte que le logiciel soit propre sont deux activités très différentes. Bon nombre d'entre nous ont des capacités limitées et nous nous concentrons principalement sur l'obtention d'un code opérationnel plutôt que sur son organisation et sa propreté. Ce choix est tout à fait approprié. Séparer les préoccupations est tout aussi important dans nos actes de programmation que dans nos programmes.

Cependant, nous sommes trop nombreux à penser que le travail est terminé lorsque le programme fonctionne. Nous laissons tomber notre deuxième préoccupation, c'est-à-dire l'organisation et la propreté. Nous passons directement au problème suivant au lieu de revenir en arrière et de décomposer les classes surchargées en entités découplées possédant une seule responsabilité.

Par ailleurs, de nombreux développeurs craignent qu'un grand nombre de petites classes à objectif unique complexifient la compréhension de l'ensemble. Ils s'inquiètent

d'avoir à naviguer d'une classe à l'autre pour comprendre la mise en œuvre d'une opération plus importante.

Cependant, un système constitué de nombreuses petites classes n'a pas plus de partie en mouvement qu'un système avec quelques grandes classes. Il y a autant à apprendre dans le système qui utilise des grandes classes. Par conséquent, la question est de savoir si vous voulez que vos outils soient rangés dans des boîtes à outils avec de nombreux petits tiroirs contenant des éléments parfaitement définis et étiquetés, ou si vous voulez avoir quelques tiroirs dans lesquels vous mettez tous vos outils en vrac.

Chaque système de taille importante sera complexe et sa logique, compliquée. Pour gérer une telle complexité, il faut commencer par l'*organiser* afin qu'un développeur sache où trouver les choses et n'ait besoin, à tout moment, de comprendre que les points de complexité qui le concernent directement. À l'opposé, un système qui contient de longues classes réalisant plusieurs objectifs pose toujours des difficultés car il nous oblige à patauger parmi un grand nombre de choses dont nous n'avons pas forcément besoin sur le moment.

Insistons sur le point important : nous voulons que nos systèmes soient constitués de nombreuses petites classes, non de quelques grandes classes. Chaque petite classe encapsule une seule responsabilité, n'a qu'une seule raison d'être modifiée et collabore avec d'autres classes pour parvenir au fonctionnement souhaité.

Cohésion

Les classes doivent contenir un nombre réduit de variables d'instance. Chaque méthode d'une classe doit manipuler une ou plusieurs de ces variables. En général, plus le nombre de variables manipulées par une méthode est élevé, plus il existe de cohésion entre cette méthode et sa classe. Lorsque chaque variable d'une classe est employée par chacune de ses méthodes, la cohésion est maximale.

En général, il est ni recommandé ni possible de créer des classes ayant un tel degré de cohésion. Toutefois, nous souhaitons une cohésion élevée, car cela signifie que les méthodes et les variables de la classe sont interdépendantes et forment un tout logique.

Prenons l'implémentation de la classe `Stack` du Listing 10.4. Cette classe présente une cohésion élevée, car, des trois méthodes, seule `size()` n'utilise pas les deux variables.

Listing 10.4 : `Stack.java`, une classe cohésive

```
public class Stack {
  private int topOfStack = 0;
  List<Integer> elements = new LinkedList<Integer>();
```

```
    public int size() {
      return topOfStack;
    }

    public void push(int element) {
      topOfStack++;
      elements.add(element);
    }

    public int pop() throws PoppedWhenEmpty {
      if (topOfStack == 0)
        throw new PoppedWhenEmpty();
      int element = elements.get(--topOfStack);
      elements.remove(topOfStack);
      return element;
    }
  }
```

En s'efforçant de garder des fonctions et des listes de paramètres courtes, nous pouvons parfois arriver à une prolifération de variables d'instance utilisées par un sous-ensemble de méthodes. Lorsque cela se produit, cela signifie presque toujours qu'il existe au moins une autre classe qui essaie de sortir de la classe plus grande. Vous devez essayer de séparer les variables et les méthodes en deux classes ou plus afin que les nouvelles classes soient plus cohésives.

Maintenir la cohésion mène à de nombreuses petites classes

Le simple fait de décomposer de longues fonctions en fonctions plus courtes conduit à la prolifération des classes. Prenons une longue fonction qui déclare de nombreuses variables. Supposons que nous voulions en extraire une petite partie pour créer une fonction séparée. Cependant, le code que nous souhaitons extraire utilise quatre des variables déclarées dans la fonction. Devons-nous passer ces quatre variables en argument de la nouvelle fonction ?

Absolument pas ! Si nous transformons ces quatre variables en variables d'instance de la classe, nous pouvons extraire le code sans n'en passer *aucune*. Il est alors *facile* de décomposer la fonction en petites parties.

Malheureusement, cette approche signifie également que nos classes perdent en cohésion car elles accumulent de plus en plus de variables d'instance partagées uniquement par quelques fonctions. Mais, si quelques fonctions partagent certaines variables, pourquoi ne pas les réunir dans une classe ? C'est effectivement la bonne solution. Lorsque la cohésion des classes diminue, elles doivent être découpées !

Ainsi, le découpage d'une longue fonction en plusieurs petites fonctions nous donne souvent l'opportunité de créer plusieurs classes plus petites. Notre programme est alors mieux organisé et sa structure est plus transparente.

Pour démontrer mon propos, prenons un exemple tiré du merveilleux livre *Literate Programming* de Knuth [Knuth92]. Le Listing 10.5 présente une traduction en Java de son programme `PrintPrimes`. Pour être franc, il ne s'agit pas du programme tel qu'il l'a écrit, mais tel que l'a généré son outil WEB. Je l'utilise car il représente un bon point de départ pour illustrer la décomposition d'une longue fonction en de nombreuses petites fonctions et classes.

Listing 10.5 : `PrintPrimes.java`

```
package literatePrimes;

public class PrintPrimes {
  public static void main(String[] args) {
    final int M = 1000;
    final int RR = 50;
    final int CC = 4;
    final int WW = 10;
    final int ORDMAX = 30;
    int P[] = new int[M + 1];
    int PAGENUMBER;
    int PAGEOFFSET;
    int ROWOFFSET;
    int C;
    int J;
    int K;
    boolean JPRIME;
    int ORD;
    int SQUARE;
    int N;
    int MULT[] = new int[ORDMAX + 1];

    J = 1;
    K = 1;
    P[1] = 2;
    ORD = 2;
    SQUARE = 9;

    while (K < M) {
      do {
        J = J + 2;
        if (J == SQUARE) {
          ORD = ORD + 1;
          SQUARE = P[ORD] * P[ORD];
          MULT[ORD - 1] = J;
        }
        N = 2;
        JPRIME = true;
        while (N < ORD && JPRIME) {
          while (MULT[N] < J)
            MULT[N] = MULT[N] + P[N] + P[N];
          if (MULT[N] == J)
            JPRIME = false;
          N = N + 1;
        }
```

```
        } while (!JPRIME);
        K = K + 1;
        P[K] = J;
      }
      {
        PAGENUMBER = 1;
        PAGEOFFSET = 1;
        while (PAGEOFFSET <= M) {
          System.out.println("The First " + M +
                             " Prime Numbers --- Page " + PAGENUMBER);
          System.out.println("");
          for (ROWOFFSET = PAGEOFFSET; ROWOFFSET < PAGEOFFSET + RR; ROWOFFSET++){
            for (C = 0; C < CC;C++)
              if (ROWOFFSET + C * RR <= M)
                System.out.format("%10d", P[ROWOFFSET + C * RR]);
            System.out.println("");
          }
          System.out.println("\f");
          PAGENUMBER = PAGENUMBER + 1;
          PAGEOFFSET = PAGEOFFSET + RR * CC;
        }
      }
    }
  }
```

Écrit comme une seule fonction, ce programme est une catastrophe. Sa structure est profondément indentée et fortement couplée. Il contient pléthore de variables étranges. La longue fonction unique doit au moins être décomposée en quelques fonctions plus petites.

Les Listings 10.6 à 10.8 montrent le résultat du découpage du code du Listing 10.5 en petites classes et fonctions, ainsi que du choix de noms significatifs pour ces classes, fonctions et variables.

Listing 10.6 : `PrimePrinter.java` **(remanié)**

```
package literatePrimes;

public class PrimePrinter {
  public static void main(String[] args) {
    final int NUMBER_OF_PRIMES = 1000;
    int[] primes = PrimeGenerator.generate(NUMBER_OF_PRIMES);

    final int ROWS_PER_PAGE = 50;
    final int COLUMNS_PER_PAGE = 4;
    RowColumnPagePrinter tablePrinter =
      new RowColumnPagePrinter(ROWS_PER_PAGE,
                               COLUMNS_PER_PAGE,
                               "The First " + NUMBER_OF_PRIMES +
                               " Prime Numbers");

    tablePrinter.print(primes);
  }
}
```

Listing 10.7 : `RowColumnPagePrinter.java`

```
package literatePrimes;

import java.io.PrintStream;

public class RowColumnPagePrinter {
  private int rowsPerPage;
  private int columnsPerPage;
  private int numbersPerPage;
  private String pageHeader;
  private PrintStream printStream;

  public RowColumnPagePrinter(int rowsPerPage,
                              int columnsPerPage,
                              String pageHeader) {
    this.rowsPerPage = rowsPerPage;
    this.columnsPerPage = columnsPerPage;
    this.pageHeader = pageHeader;
    numbersPerPage = rowsPerPage * columnsPerPage;
    printStream = System.out;
  }

  public void print(int data[]) {
    int pageNumber = 1;
    for (int firstIndexOnPage = 0;
         firstIndexOnPage < data.length;
         firstIndexOnPage += numbersPerPage) {
      int lastIndexOnPage =
        Math.min(firstIndexOnPage + numbersPerPage - 1,
                 data.length - 1);
      printPageHeader(pageHeader, pageNumber);
      printPage(firstIndexOnPage, lastIndexOnPage, data);
      printStream.println("\f");
      pageNumber++;
    }
  }

  private void printPage(int firstIndexOnPage,
                         int lastIndexOnPage,
                         int[] data) {
    int firstIndexOfLastRowOnPage =
      firstIndexOnPage + rowsPerPage - 1;
    for (int firstIndexInRow = firstIndexOnPage;
         firstIndexInRow <= firstIndexOfLastRowOnPage;
         firstIndexInRow++) {
      printRow(firstIndexInRow, lastIndexOnPage, data);
      printStream.println("");
    }
  }

  private void printRow(int firstIndexInRow,
                        int lastIndexOnPage,
                        int[] data) {
    for (int column = 0; column < columnsPerPage; column++) {
      int index = firstIndexInRow + column * rowsPerPage;
```

```
      if (index <= lastIndexOnPage)
        printStream.format("%10d", data[index]);
    }
  }

  private void printPageHeader(String pageHeader,
                               int pageNumber) {
    printStream.println(pageHeader + " --- Page " + pageNumber);
    printStream.println("");
  }

  public void setOutput(PrintStream printStream) {
    this.printStream = printStream;
  }
}
```

Listing 10.8 : `PrimeGenerator.java`

```
package literatePrimes;

import java.util.ArrayList;

public class PrimeGenerator {
  private static int[] primes;
  private static ArrayList<Integer> multiplesOfPrimeFactors;

  protected static int[] generate(int n) {
    primes = new int[n];
    multiplesOfPrimeFactors = new ArrayList<Integer>();
    set2AsFirstPrime();
    checkOddNumbersForSubsequentPrimes();
    return primes;
  }

  private static void set2AsFirstPrime() {
    primes[0] = 2;
    multiplesOfPrimeFactors.add(2);
  }

  private static void checkOddNumbersForSubsequentPrimes() {
    int primeIndex = 1;
    for (int candidate = 3;
         primeIndex < primes.length;
         candidate += 2) {
      if (isPrime(candidate))
        primes[primeIndex++] = candidate;
    }
  }

  private static boolean isPrime(int candidate) {
    if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate)) {
      multiplesOfPrimeFactors.add(candidate);
      return false;
    }
    return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
  }
```

```
    private static boolean
    isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate) {
      int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.size()];
      int leastRelevantMultiple = nextLargerPrimeFactor * nextLargerPrimeFactor;
      return candidate == leastRelevantMultiple;
    }

    private static boolean
    isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
      for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
        if (isMultipleOfNthPrimeFactor(candidate, n))
          return false;
      }
      return true;
    }

    private static boolean
    isMultipleOfNthPrimeFactor(int candidate, int n) {
      return
        candidate == smallestOddNthMultipleNotLessThanCandidate(candidate, n);
    }

    private static int
    smallestOddNthMultipleNotLessThanCandidate(int candidate, int n) {
      int multiple = multiplesOfPrimeFactors.get(n);
      while (multiple < candidate)
        multiple += 2 * primes[n];
      multiplesOfPrimeFactors.set(n, multiple);
      return multiple;
    }
  }
```

Pour commencer, vous noterez que le programme s'est considérablement allongé. D'un peu plus d'une page, il est passé à pratiquement trois pages. Cela est dû à plusieurs raisons. Premièrement, dans le programme remanié, les noms de variables sont plus descriptifs, mais plus longs. Deuxièmement, il utilise des déclarations de fonctions et de classes pour documenter le code. Troisièmement, nous avons employé des techniques d'espacement et de mise en forme pour que le programme reste lisible.

Le programme initial a été décomposé en trois responsabilités. Le programme principal tient entièrement dans la classe `PrimePrinter`. Sa responsabilité est de prendre en charge l'environnement d'exécution. Il sera modifié si la méthode d'invocation change. Par exemple, si le programme était transformé en service SOAP, cela affecterait cette classe.

La classe `RowColumnPagePrinter` sait comment mettre en forme une liste de nombres dans une page constituée d'un certain nombre de lignes et de colonnes. Si la présentation de la sortie doit être modifiée, cela concerne cette classe.

La classe `PrimeGenerator` sait comment générer une liste de nombres premiers. Notez qu'elle n'est pas conçue pour être instanciée en tant qu'objet. La classe est simplement

une portée utile dans laquelle des variables sont déclarées et restent cachées. Cette classe évoluera si l'algorithme de calcul des nombres premiers est modifié.

Il ne s'agit pas d'une réécriture ! Nous ne sommes pas repartis de zéro pour récrire à nouveau le programme. Si vous examinez attentivement les deux programmes, vous constaterez qu'ils se fondent sur le même algorithme et la même logique.

Ce remaniement a été réalisé en écrivant une suite de test qui corroborait le comportement *exact* du premier programme. Ensuite, une myriade de minuscules modifications ont été apportées, une à la fois. Après chaque changement, le programme était exécuté afin de vérifier que son comportement n'avait pas changé. Petit à petit, le premier programme a été nettoyé et converti en le second.

Organiser en vue du changement

Pour la plupart des systèmes, le changement est perpétuel. Chaque nouvelle modification représente un risque que le système ne fonctionne plus comme supposé. Dans un système propre, nous organisons nos classes afin de réduire les risques liés aux changements.

La classe `Sql` du Listing 10.9 sert à générer des chaînes SQL correctement formées à partir des métadonnées adéquates. Il s'agit d'un travail en cours et, en tant que tel, toutes les fonctionnalités SQL, comme les instructions `update`, ne sont pas prises en charge. Lorsque sera venu le temps pour la classe `Sql` de reconnaître une instruction `update`, nous devrons "ouvrir" cette classe pour effectuer les modifications. Cependant, ouvrir une classe introduit un risque. Toute modification de la classe a la possibilité de casser une autre partie du code de la classe. Elle doit être intégralement testée à nouveau.

Listing 10.9 : Une classe qui doit être ouverte pour sa modification

```
public class Sql {
   public Sql(String table, Column[] columns)
   public String create()
   public String insert(Object[] fields)
   public String selectAll()
   public String findByKey(String keyColumn, String keyValue)
   public String select(Column column, String pattern)
   public String select(Criteria criteria)
   public String preparedInsert()
   private String columnList(Column[] columns)
   private String valuesList(Object[] fields, final Column[] columns)
   private String selectWithCriteria(String criteria)
   private String placeholderList(Column[] columns)
}
```

La classe `Sql` doit être modifiée lorsque nous ajoutons un nouveau type d'instruction. Elle doit également être modifiée si nous revoyons les détails d'implémentation d'un

seul type d'instruction, par exemple si nous modifions la prise en charge de `select` afin de reconnaître les sous-sélections. Puisqu'il existe deux raisons de la modifier, la classe `Sql` ne respecte pas le principe SRP.

Cette transgression du principe SRP est repérable en examinant simplement l'organisation. La liste des méthodes de `Sql` montre que certaines méthodes privées, comme `selectWithCriteria`, semblent liées uniquement aux instructions `select`.

Lorsque des méthodes privées s'appliquent uniquement à un petit sous-ensemble d'une classe, nous disposons d'un indicateur de parties pouvant faire l'objet d'une amélioration. Toutefois, la première incitation à agir doit venir d'un changement du système lui-même. Si la classe `Sql` est considérée comme logiquement terminée, nous ne devons pas craindre de séparer les responsabilités. Si la prise en charge des instructions `update` dans un futur proche n'est pas nécessaire, nous pouvons laisser `Sql` inchangée. Mais, dès que nous ouvrons une classe, nous devons envisager la correction de la conception.

Et si nous envisagions une solution telle que celle dépeinte au Listing 10.10 ? Chaque méthode de l'interface publique définie dans la classe `Sql` du Listing 10.9 est remaniée pour devenir sa propre classe dérivée de `Sql`. Les méthodes privées, comme `values List`, sont déplacées directement là où elles sont requises. Le comportement privé commun est isolé dans deux classes utilitaires, `Where` et `ColumnList`.

Listing 10.10 : Un ensemble de classes fermées

```
abstract public class Sql {
   public Sql(String table, Column[] columns)
   abstract public String generate();
}

public class CreateSql extends Sql {
   public CreateSql(String table, Column[] columns)
   @Override public String generate()
}

public class SelectSql extends Sql {
   public SelectSql(String table, Column[] columns)
   @Override public String generate()
}

public class InsertSql extends Sql {
   public InsertSql(String table, Column[] columns, Object[] fields)
   @Override public String generate()
   private String valuesList(Object[] fields, final Column[] columns)
}

public class SelectWithCriteriaSql extends Sql {
   public SelectWithCriteriaSql(
      String table, Column[] columns, Criteria criteria)
   @Override public String generate()
}
```

```
public class SelectWithMatchSql extends Sql {
   public SelectWithMatchSql(
      String table, Column[] columns, Column column, String pattern)
   @Override public String generate()
}

public class FindByKeySql extends Sql
   public FindByKeySql(
      String table, Column[] columns, String keyColumn, String keyValue)
   @Override public String generate()
}

public class PreparedInsertSql extends Sql {
   public PreparedInsertSql(String table, Column[] columns)
   @Override public String generate() {
   private String placeholderList(Column[] columns)
}

public class Where {
   public Where(String criteria)
   public String generate()
}

public class ColumnList {
   public ColumnList(Column[] columns)
   public String generate()
}
```

Le code de chaque classe devient atrocement simple. Le temps nécessaire à la compréhension d'une classe est pratiquement nul. Le risque qu'une fonction soit à l'origine d'un dysfonctionnement dans une autre est ridiculement faible. Du point de vue des tests, il devient plus facile de prouver la logique de cette solution, car les classes sont toutes isolées les unes des autres.

Tout aussi important, lorsque le moment d'ajouter les instructions `update` sera venu, aucune des classes existantes ne devra être modifiée ! Nous codons la logique de construction des instructions `update` dans une nouvelle sous-classe de `Sql` nommée `UpdateSql`. Aucun code du système ne sera affecté par ce changement.

La logique de notre classe `Sql` restructurée représente le meilleur des deux mondes. Elle prend en charge le principe SRP. Elle prend également en charge un second principe clé de la conception des classes appelé principe ouvert/fermé (OCP, *Open-Closed Principle*) [PPP] : les classes doivent être ouvertes à l'extension mais fermées à la modification. Notre classe `Sql` restructurée est ouverte afin qu'une nouvelle fonctionnalité soit ajoutée par dérivation, mais nous pouvons effectuer ce changement tout en gardant les autres classes fermées. Il suffit de déposer notre classe `UpdateSql` au bon endroit.

Nous voulons structurer nos systèmes de manière à réduire au maximum le nettoyage impliqué par l'ajout ou la modification de fonctionnalités. Dans un système idéal, nous incorporons de nouvelles fonctionnalités en étendant le système, non en modifiant le code existant.

Cloisonner le changement

Puisque les besoins évoluent, le code change. En cours de programmation orientée objet, nous avons appris qu'il existe des classes concrètes, qui contiennent les détails d'implémentation (du code), et des classes abstraites, qui représentent uniquement les concepts. Une classe cliente qui dépend de détails concrets présente un risque lorsque ces détails changent. Nous pouvons introduire des interfaces et des classes abstraites pour limiter l'influence de ces détails.

Lorsqu'il existe des dépendances avec des détails concrets, le test d'un système est plus complexe. Si nous construisons une classe `Portfolio` qui dépend de l'API externe `TokyoStockExchange` pour déterminer la valeur d'un portefeuille, nos cas de test sont affectés par l'instabilité d'une telle recherche. Il est difficile d'écrire un test lorsque nous recevons une réponse différente toutes les cinq minutes !

Au lieu de concevoir une classe `Portfolio` qui dépend directement de `TokyoStock Exchange`, nous créons une interface, `StockExchange`, qui déclare une seule méthode :

```
public interface StockExchange {
   Money currentPrice(String symbol);
}
```

`TokyoStockExchange` implémente cette interface. Nous définissons également le constructeur de `Portfolio` pour qu'il prenne en argument une référence à `StockExchange` :

```
public Portfolio {
   private StockExchange exchange;
   public Portfolio(StockExchange exchange) {
      this.exchange = exchange;
   }
   // ...
}
```

À présent, nous pouvons créer une implémentation testable de l'interface `Stock Exchange` qui simule `TokyoStockExchange`. Cette implémentation de test fixera la valeur courante du cours choisi. Si notre test vérifie l'achat de cinq actions Microsoft dans notre portefeuille, nous codons l'implémentation de test de manière à retourner systématiquement 100 dollars par action Microsoft. Notre implémentation de test pour l'interface `StockExchange` se résume à une simple recherche dans une table. Nous pouvons ensuite écrire un test qui vérifie que la valeur de notre portefeuille global est égale à 500 dollars.

```
public class PortfolioTest {
  private FixedStockExchangeStub exchange;
  private Portfolio portfolio;

  @Before
  protected void setUp() throws Exception {
    exchange = new FixedStockExchangeStub();
    exchange.fix("MSFT", 100);
    portfolio = new Portfolio(exchange);
  }
```

```
    @Test
    public void GivenFiveMSFTTotalShouldBe500() throws Exception {
      portfolio.add(5, "MSFT");
      Assert.assertEquals(500, portfolio.value());
    }
  }
```

Lorsqu'un système est suffisamment découplé pour être testé ainsi, il est également plus souple et favorise la réutilisation. L'absence de couplage signifie que les éléments du système sont mieux isolés les uns des autres et du changement. Cette isolation facilite la compréhension de chacun d'eux.

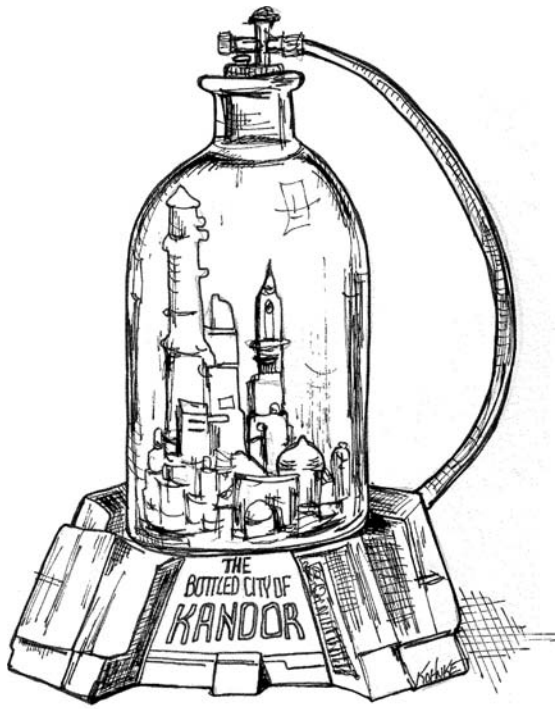
En réduisant le couplage, nos classes adhèrent à un autre principe de conception appelé principe d'inversion des dépendances (DIP, *Dependency Inversion Principle*) [PPP]. Essentiellement, ce principe stipule que nos classes doivent dépendre d'abstractions, non de détails concrets.

Au lieu de dépendre des détails d'implémentation de la classe `TokyoStockExchange`, notre classe `Portfolio` dépend désormais de l'interface `StockExchange`. Cette interface représente un concept abstrait : demander la valeur courante d'une action. Cette abstraction isole tous les détails spécifiques à l'obtention de cette valeur, y compris sa provenance.

11

Systèmes

Par Dr. Kevin Dean Wampler



"La complexité tue. Elle vide les développeurs de leur énergie, elle rend les produits difficiles à planifier, à construire et à tester."

— Ray Ozzie, directeur technique chez Microsoft.

Construire une ville

Si vous deviez construire une ville, pourriez-vous vous occuper de tous les détails vous-même ? Probablement pas. La gestion d'une ville existante est déjà une charge trop lourde pour une seule personne. Néanmoins, les villes fonctionnent (la plupart du temps). En effet, elles disposent d'équipes de plusieurs personnes qui prennent en charge chaque aspect, comme l'alimentation en eau, l'alimentation en électricité, le trafic, le respect de la loi, les règles d'urbanisme, etc. Certaines de ces personnes sont responsables de la vue d'ensemble, tandis que d'autres se focalisent sur les détails.

Les villes fonctionnent également car elles ont développé les niveaux d'abstraction et de modularité adéquats, qui permettent aux individus et aux "composants" dont ils s'occupent d'opérer efficacement, même sans comprendre la vue d'ensemble.

Bien que les équipes de développement soient souvent organisées de cette manière, les systèmes sur lesquels elles travaillent présentent rarement la même séparation des problèmes et des niveaux d'abstraction. Un code propre nous permet d'obtenir cette séparation aux niveaux d'abstraction inférieurs. Dans ce chapitre, nous allons voir comment rester propre aux niveaux d'abstraction supérieurs, le niveau *système*.

Séparer la construction d'un système de son utilisation

Tout d'abord, envisageons que la *construction* soit un processus très différent de l'*utilisation*. Alors que je rédige ce contenu, je peux voir un hôtel en construction devant ma fenêtre. Pour le moment, il s'agit d'une boîte de béton brut, avec une grue et un élévateur accroché à l'extérieur. Les ouvriers portent tous des casques et des vêtements de travail. Dans un an, l'hôtel devrait être terminé. La grue et l'élévateur auront disparu. Le bâtiment sera propre, bardé de murs vitrés et d'une jolie peinture. Les personnes qui y travailleront et y vivront auront également une apparence très différente.

> *Les systèmes logiciels doivent séparer le processus de démarrage, lorsque les objets de l'application sont construits et les dépendances sont "établies", de la logique d'exécution qui vient ensuite.*

Le démarrage est une *préoccupation* à laquelle toute application doit s'intéresser. Elle sera la première examinée dans ce chapitre. La *séparation des préoccupations* fait partie des plus anciennes et des plus importantes techniques de conception de notre métier.

Malheureusement, la plupart des applications ne séparent pas les préoccupations. Le processus de démarrage utilise un code idoine, mélangé à la logique d'exécution. En voici un exemple type :

```
public Service getService() {
  if (service == null)
    service = new MyServiceImpl(...);  // Service par défaut suffisant
  return service;                      // pour la plupart des cas ?
}
```

Il s'agit de l'idiome d'INITIALISATION/ÉVALUATION PARESSEUSE, dont les mérites sont nombreux. Nous ne voulons pas subir le coût d'une construction, à moins que nous n'utilisions réellement l'objet. Par ailleurs, le temps de démarrage peut alors être plus rapide. Nous nous assurons également que `null` n'est jamais retourné.

Cependant, nous avons à présent installé une dépendance avec `MyServiceImpl` et tout ce que requiert son constructeur (à la place des points de suspension). La compilation ne peut pas se faire sans résoudre ces dépendances, même si nous n'utilisons jamais un objet de ce type à l'exécution !

Les tests peuvent poser des difficultés. Si `MyServiceImpl` est un objet lourd, nous devons nous assurer qu'une DOUBLURE DE TEST (*test double*) [Mezzaros07] ou qu'un OBJET SIMULACRE (*mock object*) approprié est affecté au champ `service` avant que cette méthode ne soit invoquée au cours du test unitaire. Puisque la logique de construction est mélangée au processus d'exécution normal, nous devons tester tous les chemins d'exécution (par exemple, le test de `null` et de son bloc). Puisqu'elle a ces deux responsabilités, la méthode fait plus d'une chose et transgresse le principe de responsabilité unique.

Pire encore, nous ne savons pas si `MyServiceImpl` représente le bon objet dans tous les cas. Je l'ai laissé entendre dans le commentaire. Pourquoi la classe offrant cette méthode doit-elle connaître le contexte global ? Saurons-nous réellement un jour quel est le bon objet à employer ici ? Un même type peut-il être adapté à tous les contextes possibles ?

Bien entendu, l'existence d'une initialisation paresseuse n'est pas un problème trop sérieux. Cependant, il existe généralement de nombreuses instances de tels petits idiomes de configuration dans les applications. Par conséquent, la *stratégie* globale de configuration (si elle existe) est *disséminée* dans toute l'application, avec peu de modularité et souvent beaucoup de redondance.

Si nous nous appliquions à construire des systèmes bien formés et robustes, nous ne laisserions jamais de petits idiomes commodes remettre en cause la modularité. Le processus de démarrage de la construction et de liaison d'un objet ne fait pas exception. Nous devons modulariser ce processus séparément de la logique d'exécution normale et établir une stratégie globale cohérente pour résoudre nos dépendances majeures.

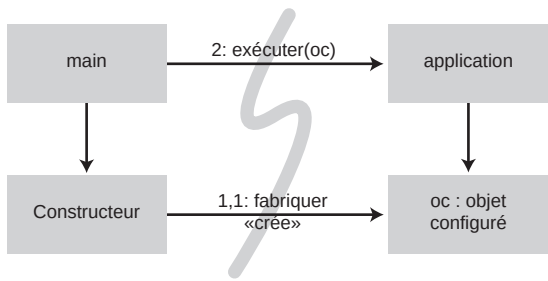
Construire dans la fonction *main*

Pour séparer la construction de l'utilisation, une solution consiste simplement à déplacer tous les aspects de la construction dans `main`, ou dans des modules appelés par `main`, et à concevoir le reste du système en supposant que tous les objets ont été construits et liés de manière appropriée (voir Figure 11.1).

Le flux de contrôle est facile à suivre. La fonction `main` crée les objets nécessaires au système, puis les passe à l'application, qui les utilise simplement. Notez le sens des

Figure 11.1

Séparer la construction dans `main()`.



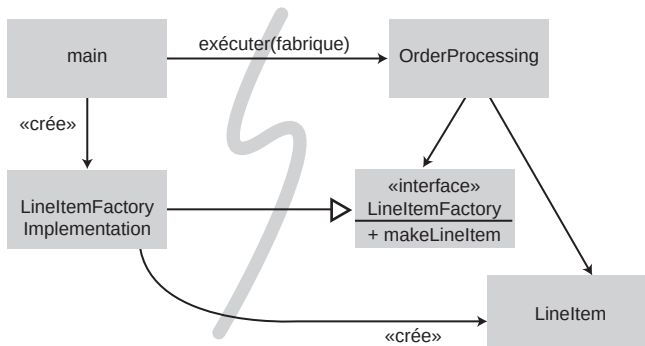
flèches de dépendance. Elles traversent la frontière entre `main` et l'application en suivant la même direction, en s'éloignant de `main`. Cela signifie que l'application n'a aucune connaissance de `main` ou du processus de construction. Elle s'attend simplement à ce que tout soit correctement construit.

Fabriques

Bien évidemment, il arrive que la responsabilité du moment de création d'un objet doive être laissée à l'application. Par exemple, dans un système de traitement des commandes, l'application doit créer des instances de `LineItem` qui sont ajoutées à un `Order`. Dans ce cas, nous pouvons employer le motif FABRIQUE ABSTRAITE [GOF] pour donner à l'application le contrôle du moment où les `LineItem` sont créés, mais garder les détails de cette construction séparés du code de l'application (voir Figure 11.2).

Figure 11.2

Séparer la construction avec une fabrique.



Vous remarquerez à nouveau que toutes les dépendances pointent de `main` vers l'application `OrderProcessing`. Cela signifie que l'application est découplée des détails de construction d'un `LineItem`. Cette responsabilité est laissée à `LineItemFactoryImplementation`, qui se trouve du côté de `main`. Néanmoins, l'application maîtrise totalement le moment de création des instances de `LineItem` et peut même fournir au constructeur des arguments propres à l'application.

Injection de dépendance

Pour séparer la construction de l'utilisation, nous pouvons employer un mécanisme puissant appelé *injection de dépendance* (DI, *Dependency Injection*), c'est-à-dire l'application de l'inversion de contrôle (IoC, *Inversion of Control*) à la gestion des dépendances (voir, par exemple, [Fowler]). L'inversion de contrôle déplace les responsabilités secondaires depuis un objet vers d'autres objets qui sont dédiés à chaque responsabilité et respecte ainsi le principe de responsabilité unique. Dans le contexte de la gestion des dépendances, un objet ne doit pas lui-même prendre en charge l'instanciation des dépendances. À la place, il doit laisser cette responsabilité à un autre mécanisme "faisant autorité", inversant ainsi le contrôle. Puisque la configuration est une préoccupation globale, ce mécanisme autoritaire sera généralement soit la méthode "principale", soit un *conteneur* à usage spécial.

Les recherches JNDI représentent une implémentation "partielle" de l'injection de dépendance, où un objet demande à un serveur d'annuaire de fournir un "service" correspondant à un nom précis :

```
MyService myService = (MyService)(jndiContext.lookup("NameOfMyService"));
```

L'objet invoquant n'a aucun contrôle sur le type d'objet réellement retourné (tant qu'il implémente l'interface appropriée), mais il résout néanmoins activement la dépendance.

L'injection de dépendance réelle va un peu plus loin. La classe ne fait rien directement pour résoudre ses dépendances ; elle est totalement passive. À la place, elle fournit des méthodes setter ou des arguments au constructeur (ou les deux) qui servent à *injecter* les dépendances. Au cours du processus de construction, le conteneur DI instancie les objets nécessaires, habituellement à la demande, et utilise les arguments du constructeur ou les méthodes setter pour câbler les dépendances. Les objets dépendants réellement employés sont indiqués par l'intermédiaire d'un fichier de configuration ou par programmation dans un module de construction à usage spécial.

Le framework Spring [Spring] propose le conteneur DI le plus connu dans le monde Java[1]. Vous pouvez définir les objets à câbler dans un fichier de configuration XML, puis vous demandez des objets précis en les nommant dans le code Java. Nous verrons un exemple plus loin.

Mais *quid* des vertus de l'initialisation paresseuse ? Cet idiome reste parfois utile avec l'injection de dépendance. Premièrement, la plupart des conteneurs DI ne construiront pas un objet tant qu'il n'est pas requis. Deuxièmement, ces conteneurs sont nombreux à

1. Le framework Spring.NET existe également.

fournir des mécanismes pour invoquer des fabriques ou construire des proxies qui peuvent être employés pour l'évaluation paresseuse et d'autres *optimisations* de ce type[2].

Grandir

Les villages deviennent des villes qui deviennent des mégapoles. Au début, les rues sont étroites et pratiquement inexistantes. Elles sont ensuite macadamisées, puis s'élargissent avec le temps. Les petits bâtiments et les parcelles vides deviennent peu à peu des bâtiments plus importants, qui finissent par être remplacés par des immeubles ou des tours.

Initialement, il n'existe aucun service, comme l'électricité, l'eau courante, le tout-à-l'égout et Internet (oups !). Ces services arrivent au fur et à mesure que la densité de population et de construction augmente.

Cette croissance ne se fait pas sans mal. Combien de fois avez-vous roulé, pare-chocs contre pare-chocs, sur une route en travaux d'agrandissement, en vous demandant "pourquoi ne l'ont-ils pas faite plus large dès le départ" ?

En réalité, ce processus de développement est normal. Qui peut justifier le coût d'une autoroute à six voies traversant le centre d'une petite ville qui veut anticiper sa croissance ? Qui voudrait avoir une telle route au milieu de sa ville ?

Il est impossible que le système soit parfait dès le départ. À la place, nous devons réaliser les *scénarios* du moment, puis remanier et étendre le système afin d'implémenter les nouveaux scénarios du lendemain. C'est là l'essence de l'agilité itérative et incrémentale. Le développement piloté par les tests, le remaniement et le code propre obtenu font que cela fonctionne au niveau du code.

En revanche, qu'en est-il au niveau du système ? Faut-il planifier à l'avance l'architecture du système ? Ne peut-il pas grandir progressivement, du simple vers le complexe ?

> *Les systèmes logiciels sont uniques si on les compare aux systèmes physiques. Leur architecture peut évoluer de manière incrémentale, **si** nous maintenons la séparation adéquate des préoccupations.*

Nous le verrons, c'est possible grâce à la nature éphémère des systèmes logiciels. Commençons par un contre-exemple, celui d'une architecture qui ne sépare pas correctement les préoccupations.

---

2. N'oubliez pas que l'instanciation et l'évaluation paresseuses constituent simplement des optimisations probablement prématurées !

Les architectures EJB1 et EJB2 initiales ne séparaient pas correctement les préoccupations et plaçaient par conséquent des barrières à la croissance organique. Examinons un *bean entité* pour une classe `Bank`. Un bean entité est une représentation en mémoire de données relationnelles, autrement dit une ligne d'une table.

Tout d'abord, nous devons définir une interface locale (dans le processus) ou distante (dans une autre JVM) employée par les clients. Le Listing 11.1 présente une interface locale possible.

Listing 11.1 : Une interface EJB2 locale pour un bean `Bank`

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public interface BankLocal extends java.ejb.EJBLocalObject {
  String getStreetAddr1() throws EJBException;
  String getStreetAddr2() throws EJBException;
  String getCity() throws EJBException;
  String getState() throws EJBException;
  String getZipCode() throws EJBException;
  void setStreetAddr1(String street1) throws EJBException;
  void setStreetAddr2(String street2) throws EJBException;
  void setCity(String city) throws EJBException;
  void setState(String state) throws EJBException;
  void setZipCode(String zip) throws EJBException;
  Collection getAccounts() throws EJBException;
  void setAccounts(Collection accounts) throws EJBException;
  void addAccount(AccountDTO accountDTO) throws EJBException;
}
```

Vous voyez plusieurs attributs pour l'adresse de la banque et une collection de comptes détenus par la banque, chacun ayant ses données gérées par un EJB `Account` séparé. Le Listing 11.2 montre la classe implémentant le bean `Bank`.

Listing 11.2 : Implémentation du bean entité

```
package com.example.banking;
import java.util.Collections;
import javax.ejb.*;

public abstract class Bank implements javax.ejb.EntityBean {
  // Logique métier...
  public abstract String getStreetAddr1();
  public abstract String getStreetAddr2();
  public abstract String getCity();
  public abstract String getState();
  public abstract String getZipCode();
  public abstract void setStreetAddr1(String street1);
  public abstract void setStreetAddr2(String street2);
  public abstract void setCity(String city);
```

```
    public abstract void setState(String state);
    public abstract void setZipCode(String zip);
    public abstract Collection getAccounts();
    public abstract void setAccounts(Collection accounts);
    public void addAccount(AccountDTO accountDTO) {
      InitialContext context = new InitialContext();
      AccountHomeLocal accountHome = context.lookup("AccountHomeLocal");
      AccountLocal account = accountHome.create(accountDTO);
      Collection accounts = getAccounts();
      accounts.add(account);
    }
    // Logique du conteneur EJB.
    public abstract void setId(Integer id);
    public abstract Integer getId();
    public Integer ejbCreate(Integer id) { ... }
    public void ejbPostCreate(Integer id) { ... }
    // La suite devait être implémentée mais restait généralement vide.
    public void setEntityContext(EntityContext ctx) {}
    public void unsetEntityContext() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void ejbRemove() {}
}
```

Je n'ai pas montré l'interface *LocalHome* correspondante, mais il s'agit essentiellement d'une fabrique qui crée des objets, avec les méthodes de recherche possibles de `Bank` que vous pourriez ajouter.

Enfin, il est nécessaire d'écrire un ou plusieurs descripteurs XML de déploiement qui précisent les détails de correspondance objet-relationnel pour le stockage persistant, le comportement transactionnel souhaité, les contraintes de sécurité, etc.

La logique métier est fortement couplée au "conteneur" d'application EJB2. Nous devons créer des sous-classes des types de conteneurs et fournir autant de méthodes du cycle de vie que requises par le conteneur.

En raison de ce couplage avec le conteneur lourd, des tests unitaires isolés sont difficiles. Il faut simuler le conteneur, ce qui est complexe, ou perdre beaucoup de temps à déployer des EJB et tester un serveur réel. Une réutilisation hors de l'architecture EJB2 est impossible, du fait de ce couplage étroit.

Pour finir, même la programmation orientée objet est ébranlée. Un bean ne peut pas hériter d'un autre bean. Remarquez la logique d'ajout d'un nouveau compte. Avec les beans EJB2, il est fréquent de définir des objets de transfert de données (DTO, *Data Transfer Object*) qui constituent essentiellement des structures sans comportement. Cela conduit généralement à des types redondants qui contiennent quasiment les mêmes données, et du code passe-partout sert à copier des données d'un objet vers un autre.

Préoccupations transversales

Sur certains plans, l'architecture EJB2 s'approche d'une véritable séparation des préoccupations. Par exemple, les comportements souhaités au niveau transactionnel, sécuritaire et, en partie, la persistance sont déclarés dans des descripteurs de déploiement, indépendamment du code source.

Notez que certaines *préoccupations*, comme la persistance, ont tendance à couper au travers des frontières naturelles des objets d'un domaine. Tous les objets doivent être rendus persistants en employant généralement la même stratégie, par exemple un système de gestion de bases de données à la place de fichiers à plat, en suivant une certaine convention de nommage pour les tables et les colonnes, en utilisant une sémantique transactionnelle cohérente, etc.

En principe, nous pouvons réfléchir à la stratégie de persistance de manière modulaire et encapsulée. En pratique, nous devons employer quasiment le même code qui implémente la stratégie de persistance dans de nombreux objets. Pour désigner ce genre de préoccupations, nous utilisons le terme *préoccupations transversales*. Une fois encore, le framework de persistance peut être modulaire et notre logique de domaine, isolément, peut être modulaire. Le problème vient de l'*intersection* fine entre ces domaines.

En réalité, la manière dont l'architecture EJB a pris en charge la persistance, la sécurité et les transactions "anticipe" la programmation orientée aspect (AOP, *Aspect-Oriented Programming*)[3], qui est une approche générale permettant de restaurer la modularité dans les préoccupations transversales.

En AOP, des constructions modulaires, appelées *aspects*, définissent les points du système qui voient leur comportement modifié de manière cohérente afin de prendre en charge une préoccupation précise. Cette spécification se fait en utilisant une déclaration succincte ou par programmation.

En prenant la persistance comme exemple, nous déclarons les objets et les attributs (ou leurs *motifs*) qui doivent persister, puis nous déléguons les tâches de persistance au framework. Les modifications du comportement sont apportées de manière *non invasive*[4] au code cible par le framework AOP. Examinons trois aspects, ou mécanismes de type aspect, en Java.

3. Pour des informations générales concernant les aspects, consultez [AOSD] ; pour des informations spécifiques à AspectJ, consultez [AspectJ] et [Colyer].
4. C'est-à-dire, sans modification manuelle du code source cible.

Proxies Java

Les proxies Java sont adaptés aux situations simples, comme envelopper des appels de méthodes dans des objets ou des classes individuelles. Toutefois, les proxies dynamiques fournis par le JDK ne fonctionnent qu'avec des interfaces. Pour obtenir des classes proxies, nous devons employer une bibliothèque de manipulation du byte-code, comme CGLIB [CGLIB], ASM [ASM] ou Javassist [Javassist].

Le Listing 11.3 correspond au squelette d'un proxy JDK qui apporte la persistance à l'application `Bank`. Il concerne uniquement les méthodes d'accès à la liste des comptes.

Listing 11.3 : Exemple de proxy du JDK

```
// Bank.java (sans les noms de paquetages...).
import java.utils.*;

// L'abstraction de banque.
public interface Bank {
  Collection<Account> getAccounts();
  void setAccounts(Collection<Account> accounts);
}

// BankImpl.java.
import java.utils.*;

// Le "bon vieil objet Java tout simple" (POJO, Plain Old Java Object)
// qui implémente l'abstraction.
public class BankImpl implements Bank {
  private List<Account> accounts;

  public Collection<Account> getAccounts() {
    return accounts;
  }
  public void setAccounts(Collection<Account> accounts) {
    this.accounts = new ArrayList<Account>();
    for (Account account: accounts) {
      this.accounts.add(account);
    }
  }
}
----------------

// BankProxyHandler.java
import java.lang.reflect.*;
import java.util.*;

// "InvocationHandler" requis par l'API d'un proxy.
public class BankProxyHandler implements InvocationHandler {
  private Bank bank;

  public BankHandler (Bank bank) {
    this.bank = bank;
  }
```

```
  // Méthode définie dans InvocationHandler.
  public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable {
    String methodName = method.getName();
    if (methodName.equals("getAccounts")) {
      bank.setAccounts(getAccountsFromDatabase());
      return bank.getAccounts();
    } else if (methodName.equals("setAccounts")) {
      bank.setAccounts((Collection<Account>) args[0]);
      setAccountsToDatabase(bank.getAccounts());
      return null;
    } else {
      ...
    }
  }

  // Nombreux détails ici.
  protected Collection<Account> getAccountsFromDatabase() { ... }
  protected void setAccountsToDatabase(Collection<Account> accounts) { ... }
}

----------------

// Quelque part, ailleurs...

Bank bank = (Bank) Proxy.newProxyInstance(
  Bank.class.getClassLoader(),
  new Class[] { Bank.class },
  new BankProxyHandler(new BankImpl()));
```

Nous avons défini une interface `Bank`, qui sera *enveloppée* par le proxy, et un bon vieil objet Java tout simple (POJO, *Plain-Old Java Object*), `BankImpl`, qui implémente la logique métier. Nous reviendrons sur les POJO plus loin.

L'API Proxy impose l'existence d'un objet `InvocationHandler` qui est invoqué pour implémenter les appels des méthodes de `Bank` effectués sur le proxy. Notre `BankProxyHandler` se sert de l'API de réflexion de Java pour associer les invocations de méthodes génériques et les méthodes correspondantes dans `BankImpl`.

Cet exemple, bien que représentant un cas simple, nécessite beaucoup de code et se révèle relativement complexe[5]. L'utilisation d'une bibliothèque de manipulation du byte-code est tout autant compliquée. Ce volume de code et cette complexité sont deux inconvénients des proxies. Ils ne facilitent pas la création d'un code propre ! Par ailleurs, les proxies ne fournissent aucun mécanisme pour préciser les points d'intérêt d'exécution de niveau système, ce qui est indispensable pour une véritable solution AOP[6].

5. Pour des exemples plus détaillés de l'API Proxy et son utilisation, consultez [Goetz].
6. On confond parfois l'AOP et les techniques employées pour la mettre en œuvre, comme l'interception de méthodes et l'emballage par des proxies. La véritable valeur d'un système AOP réside dans sa faculté à définir des comportements systémiques de manière concise et modulaire.

Frameworks AOP en Java pur

Heureusement, une grande partie du code passe-partout du proxy peut être pris en charge automatiquement par des outils. Les proxies sont employés en interne par différents frameworks Java, par exemple Spring AOP [Spring] et JBoss AOP [JBoss], pour implémenter les aspects en Java pur[7]. Avec Spring, la logique métier est écrite sous forme de bons vieux objets Java tout simples (POJO, *Plain-Old Java Object*). Les POJO sont ciblés dans leur domaine. Ils ne présentent aucune dépendance avec les frameworks d'entreprise (ou de n'importe quel autre domaine). Ainsi, ils sont conceptuellement plus simples et plus faciles à piloter par des tests. Grâce à cette relative simplicité, il est plus facile de garantir que les scénarios utilisateurs correspondants sont correctement implémentés et de maintenir et de faire évoluer le code pour les scénarios futurs.

L'infrastructure requise pour l'application, y compris les préoccupations transversales comme la persistance, les transactions, la sécurité, la mise en cache, le basculement, etc., est incluse en employant des fichiers de configuration déclaratifs ou des API. Dans de nombreux cas, nous définissons en réalité des aspects Spring ou JBoss, où le framework fournit à l'utilisateur le mécanisme d'utilisation transparente des proxies Java ou des bibliothèques de manipulation du byte-code. Ces déclarations guident le conteneur d'injection de dépendance, qui instancie les principaux objets et les lie ensemble.

Le Listing 11.4 présente un fragment type d'un fichier de configuration Spring V2.5, app.xml[8].

Listing 11.4 : Fichier de configuration Spring 2.X

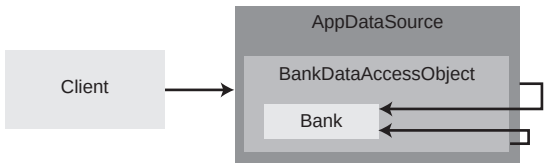
```
<beans>
  ...
  <bean id="appDataSource"
  class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close"
  p:driverClassName="com.mysql.jdbc.Driver"
  p:url="jdbc:mysql://localhost:3306/mydb"
  p:username="me"/>

  <bean id="bankDataAccessObject"
  class="com.example.banking.persistence.BankDataAccessObject"
  p:dataSource-ref="appDataSource"/>

  <bean id="bank"
  class="com.example.banking.model.Bank"
  p:dataAccessObject-ref="bankDataAccessObject"/>
  ...
</beans>
```

7. "Java pur" signifie sans employer AspectJ.
8. Adapté de **http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25**.

Chaque "bean" ressemble à un élément d'une "poupée russe", avec un objet de domaine pour un `Bank` enveloppé (proxy) par un objet d'accès aux données (DAO, *Data Accessor Object*), lui-même enveloppé par une source de données d'un pilote JDBC (voir Figure 11.3).

Figure 11.3

La "poupée russe" des décorateurs.



Le client pense qu'il invoque la méthode `getAccounts()` sur un objet `Bank`, alors qu'il converse en réalité avec l'élément externe d'un ensemble d'objets DÉCORATEUR [GOF] imbriqués qui étendent le comportement de base du POJO `Bank`. Nous pourrions ajouter d'autres décorateurs pour les transactions, la mise en cache, etc.

L'application nécessite quelques lignes pour demander au conteneur DI les objets de premier niveau du système, comme le précise le fichier XML.

```
XmlBeanFactory bf =
  new XmlBeanFactory(new ClassPathResource("app.xml", getClass()));
Bank bank = (Bank) bf.getBean("bank");
```

En raison de ces quelques lignes de code Java propre à Spring, *l'application est presque totalement découplée de Spring*, éliminant ainsi tous les problèmes de couplage étroit des systèmes comme EJB2.

Même si le format XML peut être verbeux et difficile à lire[9], la "stratégie" définie dans ces fichiers de configuration est plus simple que la logique compliquée de proxy et d'aspect qui est cachée et créée automatiquement. Ce type d'architecture est tellement intéressant que des frameworks comme Spring ont conduit à une complète révision du standard EJB dans sa version 3. EJB3 se fonde largement sur le modèle de Spring, qui prend en charge les préoccupations transversales au travers de déclarations dans des fichiers de configuration XML et/ou des annotations Java 5.

Le Listing 11.5 présente notre objet `Bank` récrit pour EJB3[10].

9. L'exemple peut être simplifié en utilisant des mécanismes qui s'appuient sur *une convention plutôt qu'une configuration* et des annotations Java 5 afin de réduire la quantité de logique de "lien" explicite requise.
10. Adapté de **http://www.onjava.com/pub/a/onjava/2006/05/17/standardizing-with-ejb3-java-persistence-api html**.

Listing 11.5 : L'EJB `Bank` en version EJB3

```
package com.example.banking.model;
import javax.persistence.*;
import java.util.ArrayList;
import java.util.Collection;

@Entity
@Table(name = "BANKS")
public class Bank implements java.io.Serializable {
   @Id @GeneratedValue(strategy=GenerationType.AUTO)
   private int id;

   @Embeddable // Un objet "incorporé" dans une ligne de la BDD de Bank.
   public class Address {
      protected String streetAddr1;
      protected String streetAddr2;
      protected String city;
      protected String state;
      protected String zipCode;
   }

   @Embedded
   private Address address;

   @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.EAGER,
              mappedBy="bank")
   private Collection<Account> accounts = new ArrayList<Account>();

   public int getId() {
      return id;
   }



   public void setId(int id) {
      this.id = id;
   }

   public void addAccount(Account account) {
      account.setBank(this);
      accounts.add(account);
   }

   public Collection<Account> getAccounts() {
      return accounts;
   }

   public void setAccounts(Collection<Account> accounts) {
      this.accounts = accounts;
   }
}
```

Ce code est beaucoup plus propre que le code EJB2 d'origine. Certains détails de l'entité sont encore présents, contenus dans les annotations. Cependant, puisque aucune

de ces informations ne se trouve en dehors des annotations, le code est propre, clair et facile à piloter par les tests, à maintenir, etc.

Certaines informations de persistance, voire toutes, présentes dans les annotations peuvent être déplacées dans des descripteurs XML de déploiement, si souhaité, afin d'obtenir un POJO réellement pur. Lorsque les détails de correspondance pour la persistance changent peu, de nombreuses équipes choisissent de conserver les annotations. Elles présentent beaucoup moins d'inconvénients que l'architecture EJB2 hautement invasive.

Aspects d'AspectJ

Le langage AspectJ [AspectJ] [Colyer] constitue l'outil le plus complet pour séparer les préoccupations à l'aide d'aspects. Cette extension Java fournit une excellente prise en charge des aspects en tant que construction pour la modularité. Les approches Java pur de Spring AOP et de JBoss AOP suffisent dans 80 à 90 % des cas où les aspects sont les plus utiles. Toutefois, AspectJ apporte un ensemble très riche d'outils puissants pour la séparation des préoccupations. Il a pour inconvénient d'imposer l'adoption de plusieurs nouveaux outils, ainsi que l'apprentissage de nouvelles constructions du langage et d'idiomes d'utilisation.

Les questions d'adoption ont été atténuées par une "forme d'annotations" récemment introduite dans AspectJ. Les annotations Java 5 sont employées pour définir des aspects à l'aide d'un code Java pur. Par ailleurs, le framework Spring dispose de plusieurs fonctionnalités qui facilitent énormément l'incorporation d'aspects basés sur les annotations. Elles se révèlent très utiles lorsque l'expérience AspectJ de l'équipe est limitée.

Une présentation complète d'AspectJ sort du cadre de ce livre. Pour de plus amples informations, consultez [AspectJ], [Colyer] et [Spring].

Piloter l'architecture du système par les tests

La puissance d'une séparation des préoccupations à l'aide d'approches de type aspect est indéniable. Si vous réussissez à écrire la logique de domaine de votre application en utilisant des POJO découplés de toute préoccupation architecturale au niveau du code, il est alors possible de véritablement *piloter par les tests* votre architecture. Vous pouvez la faire évoluer du plus simple au plus sophistiqué, en adoptant de nouvelles technologies à la demande. Il est inutile de procéder à une grande conception à l'avance (BDUF, *Big Design Up Front*[11]). En réalité, une BDUF peut même se révéler nocive car

11. BDUF, à ne pas confondre avec la bonne pratique de conception à l'avance, est une pratique qui consiste à *tout* concevoir à l'avance avant d'implémenter quoi que ce soit.

elle empêche de s'adapter aux changements, en raison d'une résistance psychologique à la mise au rebut d'un travail antérieur et de la manière dont les choix architecturaux influencent les réflexions ultérieures sur la conception.

Les architectes en bâtiment doivent procéder à une grande conception à l'avance car il est inconcevable d'apporter des modifications architecturales radicales à une grande structure physique lorsque sa construction est avancée[12]. Même si le logiciel possède sa propre *physique*[13], il est économiquement envisageable d'effectuer un changement radical *si* la structure du logiciel sépare réellement ses préoccupations.

Autrement dit, nous pourrons débuter un projet logiciel avec une architecture "naïvement simple", mais parfaitement découplée, en livrant rapidement des scénarios utilisateurs opérationnels, puis en étendant l'infrastructure. Certains des sites web les plus importants sont parvenus à une disponibilité et à des performances très élevées en employant des techniques sophistiquées de mise en cache des données, de sécurité, de virtualisation, etc., le tout de manière efficace et souple car les conceptions très faiblement couplées seront *simples* à chaque niveau d'abstraction et de portée.

Bien entendu, cela ne signifie pas que nous devions nous lancer dans un projet "sans gouvernail". Nous avons quelques idées générales sur la portée, les objectifs et la planification du projet, ainsi que sur la structure globale du système résultant. Toutefois, nous devons conserver la possibilité de varier de cap en réponse à l'évolution des circonstances.

L'architecture EJB initiale fait partie de ces API bien connues qui ont été trop travaillées et qui compromettent la séparation des préoccupations. Même des API bien conçues peuvent être disproportionnées lorsqu'elles ne sont pas réellement nécessaires. Une bonne API doit être pratiquement invisible la plupart du temps, afin que l'équipe dépense son énergie créatrice sur les scénarios utilisateurs à implémenter. Dans le cas contraire, les contraintes architecturales empêchent la livraison efficace d'une valeur optimale au client.

Pour résumer ce long propos :

> *Une architecture de système optimale est constituée de domaines de préoccupations modularisés, chacun étant implémenté à l'aide de bons vieux objets Java (ou autre) tout simples. Les différents domaines sont incorporés à l'aide d'aspects minimalement invasifs ou d'outils de type aspect. Cette architecture, tout comme le code, peut être pilotée par les tests.*

12. Il existe toujours une part non négligeable d'exploration itérative et de discussions sur les détails, même une fois la construction démarrée.
13. L'expression *physique du logiciel* a été employée la première fois par [Kolence].

Optimiser la prise de décision

La modularité et la séparation des préoccupations autorisent une décentralisation de la gestion et de la prise de décision. Dans un système suffisamment grand, que ce soit une mégapole ou un projet logiciel, personne ne peut prendre toutes les décisions.

Nous savons tous qu'il est préférable de donner des responsabilités aux personnes les plus qualifiées. Nous oublions souvent qu'il est également préférable de *reporter des décisions jusqu'au dernier moment possible*. Ce n'est en rien paresseux ou irresponsable. Cela nous permet d'effectuer des choix éclairés, à partir des meilleures informations disponibles. Une décision prématurée est une décision effectuée avec une connaissance moins qu'optimale. Lorsque les décisions sont prises trop tôt, cela se fait avec un retour des utilisateurs, une réflexion mentale sur le projet et une expérience des choix d'implémentation moindres.

> *L'agilité apportée par un système POJO avec des préoccupations modularisées permet de prendre des décisions optimales à temps, en fonction des connaissances les plus récentes. La complexité de ces décisions est également réduite.*

Utiliser les standards judicieusement, lorsqu'ils apportent une valeur démontrable

La construction des immeubles est très intéressante à étudier en raison du rythme auquel elle se fait (même au cœur de l'hiver) et des conceptions extraordinaires rendues possibles par la technologie actuelle. La construction est une industrie mature, avec des parties, des méthodes et des standards très optimisés qui ont évolué sous la pression des siècles.

De nombreuses équipes se sont habituées à l'architecture EJB2 car elle représentait un standard, même lorsque des conceptions plus légères et plus simples auraient suffi. J'ai rencontré des équipes tellement obsédées par divers standards ayant fait l'objet d'une publicité importante qu'elles en oubliaient de se focaliser sur la création d'une valeur pour leurs clients.

> *Les standards facilitent la réutilisation des idées et des composants, le recrutement de personnes ayant l'expérience appropriée, l'encapsulation de bonnes idées et la liaison des composants. Cependant, le processus de création des standards peut parfois prendre trop de temps pour que l'industrie puisse attendre, et certains ne parviennent pas à répondre aux besoins réels de leur cible.*

Les systèmes ont besoin de langages propres à un domaine

La construction des bâtiments, comme la plupart des domaines, a développé un langage riche, avec un vocabulaire, des idiomes et des motifs[14], qui convoie de manière claire et concise des informations essentielles. Le monde du logiciel connaît un regain d'intérêt récent pour la création de langages propres à un domaine (DSL, *Domain-Specific Language*)[15], qui sont de petits langages indépendants pour l'écriture de scripts ou des API développés dans des langages standard. Ils permettent d'écrire du code qui se lit comme une sorte de prose structurée que pourrait écrire un expert du domaine.

Un bon DSL réduit "l'espace de communication" qui existe entre un concept du domaine et le code qui l'implémente, comme les pratiques agiles optimisent les communications au sein d'une équipe et avec les parties prenantes du projet. Si vous implémentez la logique du domaine dans le langage employé par l'expert du domaine, vous réduisez les risques de mauvaise traduction du domaine dans l'implémentation.

Les DSL, lorsqu'ils sont bien employés, élèvent le niveau d'abstraction au-dessus des idiomes du code et des motifs de conception. Ils permettent au développeur de révéler les intentions du code au niveau d'abstraction approprié.

> *Les langages propres à un domaine permettent d'exprimer tous les niveaux d'abstraction et tous les domaines de l'application à l'aide de POJO, depuis les stratégies de haut niveau jusqu'aux détails de bas niveau.*

Conclusion

Les systèmes doivent également être propres. Une architecture invasive submerge la logique du domaine et a un impact sur l'agilité. Lorsque la logique du domaine est cachée, la qualité en souffre car les bogues se dissimulent plus facilement et les scénarios deviennent difficiles à implémenter. Si l'agilité est compromise, la productivité souffre et les avantages du développement piloté par les tests disparaissent.

À tous les niveaux d'abstraction, les intentions doivent être claires. Cela ne sera le cas que si vous écrivez des POJO et si vous employez des mécanismes de type aspect pour incorporer de manière non invasive d'autres préoccupations d'implémentation.

Que vous conceviez des systèmes ou des modules individuels, n'oubliez jamais *d'utiliser les choses les plus simples qui fonctionnent*.

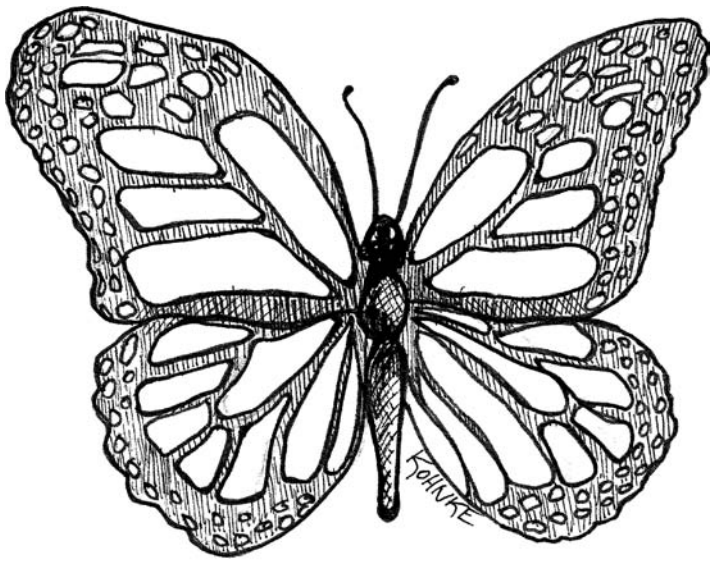
---

14. Le travail de [Alexander] a eu une influence importante sur la communauté logicielle.
15. Consultez, par exemple, [DSL]. [JMock] est un bon exemple d'API Java qui crée un DSL.

12

Émergences

Par Jeff Langr



Obtenir la propreté par une conception émergente

Et s'il existait quatre règles simples qui vous aideraient à créer de bonnes conceptions au fur et à mesure de l'avancement de votre travail ? Et si, en suivant ces règles, vous approfondissiez votre connaissance de la structure et de la conception du code, facilitant ainsi la mise en application des principes comme le SRP et le DIP ? Et si ces quatre règles facilitaient l'*émergence* de bonnes conceptions ?

Nous sommes nombreux à penser que les quatre règles de *conception simple* de Kent Beck [XPE] aident énormément à la création de logiciels bien conçus. Selon Kent, une conception est "simple" lorsqu'elle respecte les quatre règles suivantes, données par ordre d'importance :

- elle réussit tous les tests ;
- elle ne contient aucune redondance ;
- elle exprime les intentions du programmeur ;
- elle réduit le nombre de classes et de méthodes.

Règle de conception simple n° 1 : le code passe tous les tests

En premier lieu, une conception doit produire un système qui fonctionne comme prévu. Le système a beau être parfaitement conçu sur le papier, s'il n'existe aucune manière simple de vérifier qu'il fonctionne réellement comme prévu, tout ce travail sur papier est discutable.

Lorsqu'un système est intégralement testé et qu'il réussit tous ses tests en permanence, il s'agit d'un système testable. Cette déclaration est évidente, mais elle est importante. Les systèmes non testables ne sont pas vérifiables. Un système non vérifiable ne doit jamais être déployé.

En faisant en sorte que nos systèmes soient testables, nous nous orientons vers une conception dans laquelle les classes sont petites et ont des objectifs uniques. Il est simplement plus facile de tester des classes qui se conforment au principe SRP. Plus nous écrivons des tests, plus nous avançons vers des choses plus simples à tester. Par conséquent, en nous assurant que notre système est intégralement testable, nous contribuons à créer de meilleures conceptions.

Un couplage étroit complexifie l'écriture des tests. Ainsi, de manière similaire, plus nous écrivons des tests, plus nous employons des principes tels que DIP et des outils comme l'injection de dépendance, les interfaces et l'abstraction pour réduire le couplage. Nos conceptions s'en trouvent d'autant améliorées.

Il est remarquable de constater que le respect d'une règle simple et évidente, qui stipule que nous devons écrire des tests et les exécuter en permanence, a un impact sur l'adhésion de notre système aux principaux objectifs de la conception orientée objet, à savoir un couplage faible et une cohésion élevée. L'écriture des tests conduit à de meilleures conceptions.

Règles de conception simple n° 2 à 4 : remaniement

Une fois que les tests sont écrits, nous sommes habilités à garder notre code et nos classes propres. Pour ce faire, nous remanions progressivement le code. Pour chaque nouvelle ligne de code ajoutée, nous marquons une pause et réfléchissons à la nouvelle conception. L'avons-nous dégradée ? Dans l'affirmative, nous la nettoyons et exécutons nos tests afin de vérifier que nous n'avons rien cassé. *Grâce à l'existence de ces tests, nous ne craignons plus de casser le code en le nettoyant !*

Pendant la phase de remaniement, nous pouvons mettre en application toutes nos connaissances sur la bonne conception des logiciels. Nous pouvons augmenter la cohésion, réduire le couplage, séparer les préoccupations, modulariser les préoccupations du système, diminuer la taille de nos fonctions et nos classes, choisir de meilleurs noms, etc. C'est également à ce moment-là que nous appliquons les trois dernières règles de la conception simple : éliminer la redondance, assurer l'expressivité et minimiser le nombre de classes et de méthodes.

Pas de redondance

La redondance est le principal ennemi d'un système bien conçu. Elle représente un travail, un risque et une complexité inutile supplémentaires. La redondance se manifeste sous plusieurs formes. Les lignes de code très semblables constituent, bien évidemment, une redondance. En général, il est possible de triturer ces lignes de code pour qu'elles soient encore plus ressemblantes et ainsi plus faciles à remanier. La redondance existe également sous d'autres formes, comme la répétition d'une implémentation. Par exemple, une classe collection pourrait offrir les deux méthodes suivantes :

```
int size() {}
boolean isEmpty() {}
```

Nous pourrions les implémenter chacune séparément. La méthode `isEmpty` gérerait une valeur booléenne, tandis que `size` manipulerait un compteur. Ou bien nous pouvons supprimer cette redondance en liant `isEmpty` à la définition de `size` :

```
boolean isEmpty() {
   return 0 == size();
}
```

Pour créer un système propre, il faut avoir la volonté de supprimer la redondance, même lorsqu'elle ne concerne que quelques lignes de code. Par exemple, prenons le code suivant :

```
public void scaleToOneDimension(
     float desiredDimension, float imageDimension) {
  if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
     return;
```

```
    float scalingFactor = desiredDimension / imageDimension;
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);

    RenderedOp newImage = ImageUtilities.getScaledImage(
       image, scalingFactor, scalingFactor);
    image.dispose();
    System.gc();
    image = newImage;
  }
  public synchronized void rotate(int degrees) {
    RenderedOp newImage = ImageUtilities.getRotatedImage(
       image, degrees);
    image.dispose();
    System.gc();
    image = newImage;
  }
```

Pour le nettoyer, nous devons supprimer la petite redondance qui existe entre les méthodes `scaleToOneDimension` et `rotate` :

```
  public void scaleToOneDimension(
       float desiredDimension, float imageDimension) {
    if (Math.abs(desiredDimension - imageDimension) < errorThreshold)
       return;
    float scalingFactor = desiredDimension / imageDimension;
    scalingFactor = (float)(Math.floor(scalingFactor * 100) * 0.01f);

    replaceImage(ImageUtilities.getScaledImage(
       image, scalingFactor, scalingFactor));
  }

  public synchronized void rotate(int degrees) {
    replaceImage(ImageUtilities.getRotatedImage(image, degrees));
  }

  private void replaceImage(RenderedOp newImage) {
    image.dispose();
    System.gc();
    image = newImage;
  }
```

En extrayant les éléments communs à ce tout petit niveau, nous commençons à identifier des transgressions du principe SRP. Nous pouvons déplacer dans une autre classe une méthode que nous venons d'extraire, en améliorant ainsi sa visibilité. Un autre membre de l'équipe peut y voir une opportunité de rendre la nouvelle méthode plus abstraite et de la réutiliser dans un contexte différent. Cette "réutilisation à petite échelle" permet de réduire considérablement la complexité d'un système. Pour parvenir à une réutilisation à grande échelle, il est indispensable de comprendre comment l'obtenir à petite échelle.

Le motif PATRON DE MÉTHODE [GOF] est une technique classique pour supprimer la redondance aux niveaux supérieurs. Examinons, par exemple, le code suivant :

```
public class VacationPolicy {
   public void accrueUSDivisionVacation() {
      // Code de calcul des congés en fonction des heures travaillées.
      // ...
      // Code de vérification de la conformité des congés au droit US.
      // ...
      // Code d'application des congés aux fiches de paie.
      // ...
   }

   public void accrueEUDivisionVacation() {
      // Code de calcul des congés en fonction des heures travaillées.
      // ...
      // Code de vérification de la conformité des congés au droit EU.
      // ...
      // Code d'application des congés aux fiches de paie.
      // ...
   }
}
```

Le code des deux méthodes `accrueUSDivisionVacation` et `accrueEUDivision Vacation` est vraiment très proche, à l'exception de la vérification de la conformité des congés au minimum légal. Cette partie de l'algorithme change en fonction du statut de l'employé.

Nous pouvons supprimer la redondance évidente en appliquant le motif PATRON DE MÉTHODE.

```
abstract public class VacationPolicy {
   public void accrueVacation() {
      calculateBaseVacationHours();
      alterForLegalMinimums();
      applyToPayroll();
   }

   private void calculateBaseVacationHours() { /* ... */ };
   abstract protected void alterForLegalMinimums();
   private void applyToPayroll() { /* ... */ };
}

public class USVacationPolicy extends VacationPolicy {
   @Override protected void alterForLegalMinimums() {
      // Logique adaptée aux États-Unis.
   }
}

public class EUVacationPolicy extends VacationPolicy {
   @Override protected void alterForLegalMinimums() {
      // Logique adaptée à l'Europe.
   }
}
```

Les sous-classes comblent le "trou" dans l'algorithme `accrueVacation`, en fournissant les seules informations qui ne sont pas dupliquées.

Expressivité

La plupart d'entre nous avons déjà été confrontés à du code alambiqué. Nous avons nous-mêmes produit du code tarabiscoté. Il est facile d'écrire du code que *nous* comprenons, car, au moment où nous l'écrivons, nous sommes au cœur du problème que nous tentons de résoudre. Les autres personnes chargées de la maintenance du code n'en auront pas une compréhension aussi profonde.

Une grande partie du coût d'un projet logiciel se trouve dans la maintenance à long terme. Pour diminuer les défaillances potentielles liées aux modifications introduites, il est indispensable de comprendre le fonctionnement du système. Au fur et à mesure que le système se complexifie, il faut de plus en plus de temps au développeur pour le comprendre et les possibilités d'erreur de compréhension augmentent. Par conséquent, le code doit clairement exprimer les intentions de son auteur. Plus l'auteur saura écrire du code clair, moins les autres développeurs passeront du temps à le comprendre. Cela permettra de diminuer les défauts et le coût de la maintenance.

Nous pouvons nous exprimer en choisissant de bons noms. Nous voulons entendre le nom d'une classe ou d'une fonction sans être étonnés par ses responsabilités.

Nous pouvons nous exprimer en écrivant des fonctions et des classes de petite taille. En général, les classes et les fonctions courtes sont plus faciles à nommer, à écrire et à comprendre.

Nous pouvons nous exprimer en employant une nomenclature standard. Les motifs de conception, par exemple, réalisent un important travail de communication et d'expressivité. En utilisant des noms de motif standard, comme COMMAND ou VISITOR, dans les noms des classes qui implémentent ces motifs, nous pouvons décrire succinctement notre conception aux autres développeurs.

Les tests unitaires bien écrits participent également à l'expressivité. Les tests servent en premier lieu de documentation par l'exemple. La personne qui lit nos tests doit être capable de comprendre rapidement le rôle d'une classe.

Toutefois, la meilleure manière d'être expressif est d'*essayer* de l'être. Bien trop souvent, nous faisons en sorte que notre code fonctionne, puis nous passons au problème suivant sans prendre le temps de revenir sur le code afin de le rendre lisible pour les prochains développeurs. N'oubliez pas que la prochaine personne qui lira votre code sera probablement vous-même.

Vous devez donc soigner votre travail. Passez un peu de temps sur chaque fonction et chaque classe. Choisissez de meilleurs noms, décomposez les longues fonctions en fonctions plus courtes et, de manière générale, prenez soin de ce que vous créez. L'attention est une ressource précieuse.

Un minimum de classes et de méthodes

Même des concepts aussi fondamentaux que l'élimination de la redondance, l'expressivité du code et le principe SRP peuvent parfois nous conduire trop loin. En nous efforçant d'obtenir des classes et des méthodes courtes, nous risquons de créer beaucoup trop de toutes petites classes et méthodes. Par conséquent, la règle stipule que nous tentions également de maintenir le nombre de fonctions et de classes aussi faible que possible.

Lorsque le nombre de classes et de méthodes est élevé, cela provient parfois d'un dogmatisme absurde. Prenons, par exemple, un standard de codage qui insiste sur la création d'une interface pour chacune des classes. Cependant, certains développeurs s'obligent à séparer les champs et les comportements dans des classes de données et des classes de comportement. Nous vous laissons imaginer le résultat. Il faut éviter d'être aussi catégorique et adopter une approche plus pragmatique.

L'objectif est de conserver petit le système global, tout en écrivant des fonctions et des classes courtes. Cependant, n'oubliez pas que cette règle est la dernière des quatre règles de conception simple et qu'elle est donc la moins prioritaire. Par conséquent, même s'il est important de réduire au maximum le nombre de classes et de fonctions, il est plus important d'avoir des tests, de supprimer la redondance et d'améliorer l'expressivité.

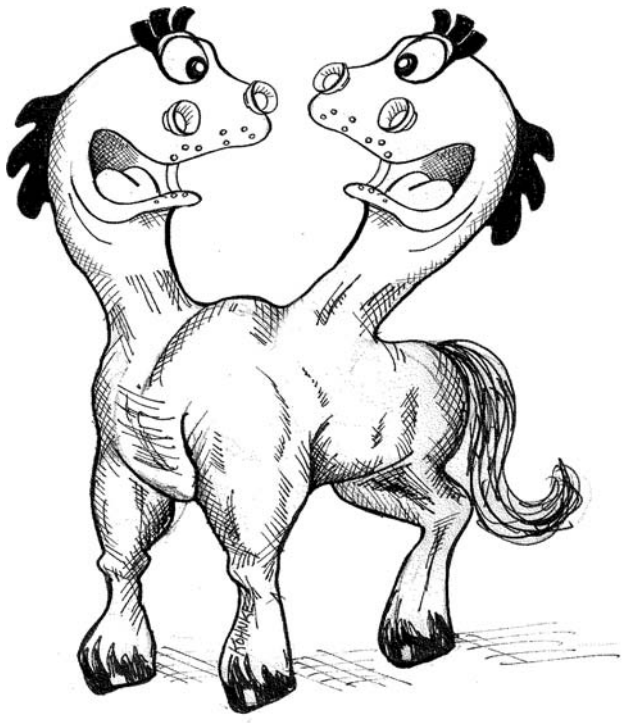
Conclusion

Existe-t-il un ensemble de pratiques simples qui peuvent remplacer l'expérience ? La réponse est clairement non. Toutefois, les pratiques décrites dans ce chapitre et dans ce livre proviennent de plusieurs dizaines d'années d'expérience des auteurs. En suivant les règles de conception simple, les développeurs peuvent adhérer immédiatement aux bons principes et motifs, alors qu'il leur faudrait sinon des années pour les découvrir.

13

Concurrence

Par Brett L. Schuchert



"Les objets sont des abstractions du traitement.
Les threads sont des abstractions de la planification."

— James O. Coplien (correspondance privée)

Il est difficile, même très difficile, d'écrire des programmes concurrents propres. Il est beaucoup plus facile d'écrire du code qui s'exécute dans un seul thread. Il est également facile d'écrire du code multithread qui, en apparence, semble parfait, mais qui soit défectueux à un niveau plus profond. Un tel code fonctionne parfaitement, jusqu'à ce que le système soit fortement sollicité.

Dans ce chapitre, nous traitons des besoins de programmation concurrente et des difficultés qu'elle représente. Nous donnons ensuite quelques conseils de prise en charge de ces difficultés et d'écriture d'un code concurrent propre. Nous concluons par les problèmes liés au test d'un code concurrent.

L'écriture d'un code concurrent propre est un sujet complexe qui mériterait un ouvrage en soi. Dans ce livre, notre objectif est de présenter une vue d'ensemble ; nous fournirons un didacticiel plus détaillé dans l'Annexe A. Si la concurrence n'est pour vous qu'un sujet de curiosité, ce chapitre vous suffira pour le moment. Si vous devez maîtriser plus profondément la concurrence, il vous faudra également lire le didacticiel.

Raisons de la concurrence

La concurrence représente une stratégie de découplage. Elle aide à découpler ce qui est réalisé (le *quoi*) du moment où cela se produit (le *quand*). Dans des applications monothreads, le lien entre le *quoi* et le *quand* est tellement fort que l'état global de l'application peut souvent être connu en examinant la trace de la pile. Le programmeur qui débogue un tel système peut placer un ou plusieurs points d'arrêt et connaître l'état du système en fonction des points d'arrêt atteints.

En découplant le *quoi* du *quand*, nous pouvons considérablement améliorer les capacités de traitement et la structure d'une application. D'un point de vue structurel, l'application ressemble à de nombreux petits ordinateurs coopérant, non à une grande boucle principale. Il est ainsi possible d'obtenir un système plus facile à comprendre et cela offre plusieurs solutions pour séparer les préoccupations.

Prenons, par exemple, le modèle standard des servlets pour les applications web. Ces systèmes s'exécutent sous le contrôle d'un conteneur web ou EJB qui s'occupe *partiellement* de la concurrence à notre place. Les servlets sont exécutées de manière asynchrone lorsque arrivent des requêtes web. Le programmeur d'une servlet n'a pas besoin de gérer toutes les requêtes entrantes. *En principe*, chaque servlet s'exécute dans son propre petit monde et est découplée des autres exécutions de servlets.

Cependant, si cela était aussi simple, ce chapitre serait superflu. En réalité, le découplage fourni par les conteneurs web est loin d'être parfait. Les programmeurs de servlets doivent faire très attention à la mise en œuvre de la concurrence dans leurs programmes. Néanmoins, les avantages structurels du modèle des servlets ne sont pas négligeables.

Toutefois, la structure n'est pas la seule motivation pour l'adoption de la concurrence. Certains systèmes présentent des contraintes de temps de réponse et de débit qui exigent des solutions concurrentes écrites à la main. Par exemple, prenons le cas d'un agrégateur monothread qui récupère ses informations à partir de différents sites web et les fusionne pour en faire un résumé quotidien. Puisque ce système est monothread, il consulte chaque site web un à un, en terminant toujours le précédent avant de passer au suivant. La tâche quotidienne doit s'exécuter en moins de 24 heures. Cependant, plus on ajoute de sites web, plus le temps de collecte des données augmente, jusqu'à dépasser 24 heures. Un processus monothread passe beaucoup de temps sur les sockets web, à attendre la fin des entrées/sorties. Nous pouvons améliorer ses performances en employant un algorithme multithread qui contacte plusieurs sites web à la fois.

Ou bien considérons un système qui répond à un utilisateur à la fois, en une seconde. Ce système est assez réactif pour quelques utilisateurs, mais lorsque leur nombre augmente son temps de réponse s'accroît. Aucun utilisateur ne souhaite se trouver derrière 150 autres ! Nous pouvons améliorer le temps de réponse de ce système en prenant en charge plusieurs utilisateurs de manière concurrente.

Ou bien encore considérons un système qui analyse de grands ensembles de données et qui donne une solution complète une fois tous ces jeux de données traités. Peut-être que chaque jeu de données pourrait être analysé sur un ordinateur différent afin qu'ils soient tous traités en parallèle.

Mythes et idées fausses

Il existe ainsi de bonnes raisons d'adopter la concurrence. Toutefois, nous l'avons déjà dit, la concurrence est *difficile*. Si vous n'êtes pas méticuleux, vous pouvez créer des situations très désagréables. Étudions les mythes et les idées fausses suivantes :

- *La concurrence améliore toujours les performances.*
  La concurrence améliore *parfois* les performances, uniquement lorsqu'il est possible de partager de longs temps d'attente entre plusieurs threads ou plusieurs processeurs. Aucun des cas n'est simple.
- *L'écriture de programmes concurrents n'a pas d'impact sur la conception.*
  En réalité, la conception d'un algorithme concurrent peut être très différente de celle d'un système monothread. Le découplage du *quoi* et du *quand* a généralement un impact très important sur la structure du système.
- *La compréhension des problèmes liés à la concurrence n'est pas importante lorsqu'on travaille avec un conteneur comme un conteneur web ou EJB.*
  En réalité, il est préférable de connaître le fonctionnement du conteneur et de savoir comment résoudre les problèmes de mises à jour concurrentes et d'interblocage décrits dans ce chapitre.

Voici quelques petites phrases qui concernent l'écriture d'un logiciel concurrent :

- *La concurrence implique un certain surcoût,* à la fois en terme de performances et d'écriture d'un code supplémentaire.
- *Une bonne mise en œuvre de la concurrence est complexe,* même dans le cas de problèmes simples.
- *Les bogues de concurrence ne sont généralement pas reproductibles,* et ils sont généralement considérés comme des événements exceptionnels[1] au lieu d'être traités comme les véritables défaillances qu'ils sont.
- *La concurrence implique souvent un changement fondamental dans la stratégie de conception*.

Défis

Pourquoi la programmation concurrente est-elle si difficile ? Examinons la classe simple suivante :

```
public class X {
   private int lastIdUsed;

   public int getNextId() {
        return ++lastIdUsed;
    }
}
```

Supposons que nous souhaitions créer une instance de `X`, fixer le champ `lastIdUsed` à 42 et partager ensuite l'instance entre deux threads. Supposons également que ces deux threads invoquent la méthode `getNextId()`. Trois résultats sont possibles :

- Le premier thread obtient la valeur 43, le second thread obtient la valeur 44, `lastIdUsed` vaut 44.
- Le premier thread obtient la valeur 44, le second thread obtient la valeur 43, `lastIdUsed` vaut 44.
- Le premier thread obtient la valeur 43, le second thread obtient la valeur 43, `lastIdUsed` vaut 43.

Le surprenant troisième résultat[2] est obtenu lorsque les deux threads se chevauchent. Cette situation existe car les deux threads peuvent emprunter différents chemins vers cette seule ligne de code Java et certains produisent des résultats incorrects. Combien existe-t-il de chemins différents dans ce cas ? Pour répondre à cette question, nous

1. Rayons cosmiques, microcoupures électriques, etc.
2. Voir la section "Examen plus approfondi" à l'Annexe A.

devons comprendre ce que le compilateur JIT (*Just-In-Time*) fait du byte-code généré et ce que le modèle mémoire de Java considère comme atomique.

Une réponse rapide, basée sur le byte-code généré, est qu'il existe 12 870 chemins d'exécution possibles[3] pour ces deux threads qui s'exécutent dans la méthode `getNextId`. Si `lastIdUsed` n'est plus de type `int` mais `long`, le nombre de chemins existants passe à 2 704 156. Bien entendu, la plupart de ces chemins conduisent aux bons résultats. Mais ce n'est pas le cas pour certains d'entre eux, et c'est là tout le problème.

Se prémunir des problèmes de concurrence

Nous allons donner, dans les sections suivantes, un ensemble de principes et de techniques qui permettent de protéger vos systèmes contre les problèmes liés au code concurrent.

Principe de responsabilité unique

Selon le principe SRP [PPP], il ne doit exister qu'une seule raison de modifier une méthode, une classe ou un composant. La conception concurrente est suffisamment complexe pour être une raison de changement en soi et mérite donc d'être séparée du reste du code. Malheureusement, il arrive trop fréquemment que les détails d'implémentation de la concurrence soient incorporés directement au reste du code de production. Voici quelques points à examiner :

- *Le code lié à la concurrence possède son propre cycle de développement*, de changement et de réglage.
- *Le code lié à la concurrence présente ses propres défis*, qui sont différents, et souvent plus complexes, de ceux du code non liés à la concurrence.
- Les causes de dysfonctionnement du code concurrent mal écrit suffisent amplement à compliquer son écriture, sans même prendre en compte la complexité du code de l'application elle-même.

> **Recommandation :** *gardez le code lié à la concurrence séparé de tout autre code*[4].

Corollaire : limiter la portée des données

Nous l'avons vu, deux threads qui modifient le même champ d'un objet partagé peuvent se perturber mutuellement, conduisant à un résultat inattendu. Une solution

3. Voir la section "Chemins d'exécution possibles" à l'Annexe A.
4. Voir la section "Exemple client/serveur" à l'Annexe A.

consiste à utiliser le mot-clé `synchronized` pour protéger une *section critique* du code qui utilise l'objet partagé. Il est important de limiter le nombre de ces sections critiques. Plus les endroits de modification des données partagées sont nombreux, plus les points suivants risquent de se vérifier :

- Vous oublierez de protéger un ou plusieurs de ces endroits, remettant en cause tout le code qui modifie les données partagées.
- Vous devrez redoubler d'efforts pour vous assurer que toutes les protections sont bien en place (transgression de DRY [PRAG]).
- Il sera difficile de déterminer les sources de dysfonctionnements, qui sont déjà suffisamment complexes à trouver.

> **Recommandation :** *prenez à cœur l'encapsulation des données ; limitez le plus possible les accès aux données qui sont partagées.*

Corollaire : utiliser des copies des données

Une bonne manière d'éviter les données partagées consiste simplement à éviter de les partager. Dans certains cas, il est possible de copier des objets et de les manipuler en lecture seule. Dans d'autres cas, il est possible de copier des objets, de collecter les résultats à partir de plusieurs threads sur ces copies, puis de fusionner les résultats dans un seul thread.

S'il existe une manière simple d'éviter le partage d'objets, le code résultant sera beaucoup moins enclin à provoquer des problèmes. Vous pourriez vous interroger sur le coût de création de tous ces objets supplémentaires. Dans ce cas, faites quelques essais pour savoir s'il s'agit réellement d'un problème. Mais n'oubliez pas que l'utilisation des copies d'objets permet d'éviter les points de synchronisation dans le code et que les économies ainsi réalisées au niveau des mécanismes de verrouillage compenseront probablement le surcoût dû à la création des copies et au ramasse-miettes.

Corollaire : les threads doivent être aussi indépendants que possible

Vous devez essayer d'écrire votre code de sorte que chaque thread existe dans son propre monde, en ne partageant aucune donnée avec un autre thread. Chaque thread traite une requête cliente, dont toutes les données nécessaires proviennent d'une source non partagée et sont stockées sous forme de variables locales. Ainsi, chacun de ces threads se comporte comme s'il était le seul thread au monde et la synchronisation est inutile.

Par exemple, les classes dérivées de `HttpServlet` reçoivent leurs informations sous forme de paramètres passés aux méthodes `doGet` et `doPost`. Ainsi, chaque servlet fonctionne comme si elle disposait de sa propre machine. Tant que le code de la servlet n'utilise que des variables locales, il n'y a aucun risque que la servlet soit à l'origine de

problèmes de synchronisation. Bien entendu, la plupart des applications qui utilisent des servlets finissent par utiliser des ressources partagées, comme une connexion à une base de données.

> **Recommandation :** *essayez de partitionner les données en sous-ensembles indépendants qui peuvent être manipulés par des threads indépendants, potentiellement sur des processeurs différents.*

Connaître la bibliothèque

Par rapport aux versions précédentes, Java 5 apporte de nombreuses améliorations pour l'écriture d'un code multithread. Lors d'un tel développement, voici les éléments à prendre en compte :

- Les collections sûres vis-à-vis des threads (*thread-safe*) doivent être employées.
- Le framework Executor doit être utilisé pour l'exécution de tâches indépendantes.
- Des solutions non bloquantes doivent être choisies lorsque c'est possible.
- Plusieurs classes bibliothèques ne sont pas sûres vis-à-vis des threads.

Collections sûres vis-à-vis des threads

Alors que Java était encore jeune, Doug Lea a écrit l'ouvrage de référence *Concurrent Programming in Java* [Lea99]. En parallèle au livre, il a également développé plusieurs collections sûres vis-à-vis des threads, qui ont plus tard intégré le paquetage `java.util.concurrent` du JDK. Les collections de ce paquetage peuvent être employées en toute sécurité dans un code multithread et leurs performances sont bonnes. En réalité, la mise en œuvre de `ConcurrentHashMap` est plus efficace que celle de `HashMap` dans pratiquement tous les cas. Elle accepte les lectures et les écritures concurrentes simultanées et fournit des méthodes prenant en charge les opérations composées classiques qui ne sont, sinon, pas sûres vis-à-vis des threads. Si Java 5 est votre environnement de déploiement, optez pour `ConcurrentHashMap`.

Plusieurs autres sortes de classes aident à obtenir une conception élaborée du code concurrent. En voici quelques exemples :

`ReentrantLock`	Un verrou qui peut être obtenu dans une méthode et libéré dans une autre.
`Semaphore`	Une implémentation du sémaphore classique, c'est-à-dire un verrou avec un compteur.
`CountDownLatch`	Un verrou qui attend un certain nombre d'événements avant de débloquer tous les threads qui sont en attente. Tous les threads ont ainsi une chance équitable de démarrer au même moment.

Recommandation : *examinez à nouveau les classes disponibles. Dans le cas de Java, familiarisez-vous avec les paquetages java.util.concurrent, java.util.concurrent.atomic et java.util.concurrent.locks.*

Connaître les modèles d'exécution

Dans une application concurrente, il existe plusieurs manières de partitionner le comportement. Avant de nous y intéresser, nous devons donner quelques définitions de base :

Ressources bornées	Ressources dont la taille ou le nombre est figé et qui sont employées dans un environnement concurrent. Il s'agit, par exemple, des connexions à une base de données ou des tampons de lecture/écriture de taille fixe.
Exclusion mutuelle	Un seul thread à la fois peut accéder à des données ou à des ressources partagées.
Famine	Un thread ou un groupe de threads est interdit de fonctionnement pendant une durée excessivement longue ou indéfiniment. Par exemple, si les threads d'exécution courte sont toujours privilégiés, les threads dont le traitement est plus long peuvent ne jamais être élus si les premiers ne se terminent pas.
Interblocage (*deadlock*)	Deux threads ou plus attendent chacun la fin de l'autre. Chaque thread dispose d'une ressource dont l'autre a besoin et aucun ne peut se terminer tant qu'il n'a pas obtenu la ressource de l'autre.
Interblocage actif (*livelock*)	Les threads sont synchrones, chacun tentant de faire son travail mais rencontrant toujours un autre thread "sur son chemin". Un phénomène de résonance se crée et les threads tentent d'avancer mais ces exécutions ne durent jamais très longtemps.

Ces définitions étant posées, nous pouvons à présent étudier les différents modèles d'exécution employés dans la programmation concurrente.

Producteur-consommateur[5]

Un ou plusieurs threads producteurs créent du travail et le placent dans un tampon ou une file. Un ou plusieurs threads consommateurs récupèrent ce travail à partir de la file et le mènent à terme. La file placée entre les producteurs et les consommateurs constitue une *ressource bornée*. Cela signifie que les producteurs doivent attendre que de l'espace

5. **http://en.wikipedia.org/wiki/Producer-consumer**.

se libère dans la file pour y ajouter un travail et que les consommateurs doivent attendre que la file contienne un travail pour le récupérer. La coordination entre les producteurs et les consommateurs au travers de la file les oblige à s'avertir l'un l'autre. Les producteurs écrivent dans la file et signalent qu'elle n'est plus vide. Les consommateurs lisent depuis la file et signalent qu'elle n'est plus pleine. Les deux acteurs peuvent attendre d'être avertis pour pouvoir continuer.

Lecteurs-rédacteurs[6]

Lorsque nous disposons d'une ressource partagée qui sert principalement de source d'information à des lecteurs mais qui est occasionnellement actualisée par des rédacteurs, le débit pose un problème. Accentuer le débit peut conduire à une situation de famine et à une accumulation d'informations obsolètes. Accepter les mises à jour peut avoir un impact sur le débit. Il est difficile de coordonner les lecteurs afin qu'ils ne lisent pas ce que des rédacteurs sont en train de mettre à jour, et que les rédacteurs ne mettent pas à jour ce que des lecteurs sont en train de lire. Les rédacteurs ont tendance à bloquer de nombreux lecteurs pendant une longue période de temps, conduisant à des problèmes de débit.

Le défi est d'équilibrer les besoins des lecteurs et des rédacteurs afin d'obtenir un fonctionnement correct dans lequel le débit est raisonnable et la famine, évitée. Une stratégie simple consiste à faire en sorte que les rédacteurs attendent qu'il n'y ait plus aucun lecteur pour faire leurs mises à jour. Cependant, si des lecteurs sont toujours présents, les rédacteurs entrent en famine. *A contrario*, si les rédacteurs sont nombreux et ont la priorité, le débit baissera. Tout le problème consiste à trouver le bon équilibre et à éviter les problèmes de mises à jour concurrentes.

Dîner des philosophes[7]

Imaginez un groupe de philosophes assis autour d'une table ronde. Une fourchette est placée à gauche de chacun d'eux. Un grand plat de spaghettis se trouve au centre de la table. Les philosophes passent leur temps à penser, sauf lorsqu'ils ont faim. À ce moment-là, ils prennent les deux fourchettes qui se trouvent à côté d'eux et mangent. Un philosophe ne peut pas manger s'il ne possède pas deux fourchettes. Si le philosophe qui se trouve à sa droite ou à sa gauche utilise déjà une fourchette dont il a besoin, il doit attendre jusqu'à ce que celui-ci termine de manger et replace les fourchettes sur la table. Lorsqu'un philosophe a fini de manger, il dépose ses fourchettes sur la table et attend d'avoir faim à nouveau.

---

6. **http://fr.wikipedia.org/wiki/Problème_des_lecteurs_et_des_rédacteurs**.
7. **http://en.wikipedia.org/wiki/Dining_philosophers_problem**.

En remplaçant les philosophes par des threads et les fourchettes par des ressources, nous obtenons un problème classique dans de nombreuses applications où des processus se disputent des ressources. S'ils ne sont pas parfaitement conçus, les systèmes ainsi en concurrence peuvent faire face à des interblocages, ainsi qu'à la dégradation du débit et de l'efficacité.

La plupart des problèmes de concurrence que vous rencontrerez seront des variantes de ces trois-là. Étudiez ces algorithmes et écrivez des solutions afin que vous soyez bien préparé lorsque vous serez confronté à des problèmes de concurrence.

> **Recommandation :** *apprenez ces algorithmes de base et comprenez leurs solutions.*

Attention aux dépendances entre des méthodes synchronisées

Les dépendances entre des méthodes synchronisées conduisent à des bogues subtils dans le code concurrent. Le langage Java fournit le mot-clé `synchronized` pour protéger une méthode individuelle. Cependant, si une même classe partagée possède plusieurs méthodes synchronisées, votre système risque alors d'être mal écrit[8].

> **Recommandation :** *évitez d'utiliser plusieurs méthodes sur un objet partagé.*

Il arrive parfois que vous deviez utiliser plusieurs méthodes sur un objet partagé. Si c'est le cas, voici trois manières d'obtenir un code correct :

- **Verrouillage basé sur le client.** Le client verrouille le serveur avant d'invoquer la première méthode et s'assure que le verrou s'étend jusqu'au code appelant la dernière méthode.
- **Verrouillage basé sur le serveur.** Au sein du serveur, créez une méthode qui verrouille le serveur, appelle toutes les méthodes, puis relâche le verrou. Le client appelle cette nouvelle méthode.
- **Serveur adapté.** Créez un intermédiaire qui se charge du verrouillage. Il s'agit d'un exemple de verrouillage basé sur le serveur lorsque le serveur d'origine ne peut pas être modifié.

Garder des sections synchronisées courtes

Le mot-clé `synchronized` introduit un verrou. Toutes les sections de code gardées par le même verrou sont assurées d'être exécutées par un seul thread à la fois. Les verrous coûtent cher car ils créent des retards et ajoutent une surcharge. Il ne faut donc pas

8. Voir la section "Impact des dépendances entre méthodes sur le code concurrent" à l'Annexe A.

joncher le code d'instructions `synchronized`. Toutefois, les sections critiques[9] doivent être protégées. Par conséquent, nous devons concevoir notre code de manière à réduire au maximum les sections critiques.

Certains programmeurs naïfs pensent y arriver en écrivant de très longues sections critiques. En réalité, lorsque la synchronisation s'étend au-delà de la section critique minimale, la contention augmente et les performances se dégradent[10].

> **Recommandation :** *conservez des sections synchronisées les plus courtes possible.*

Écrire du code d'arrêt est difficile

Le développement d'un système conçu pour fonctionner en permanence diffère du développement d'un logiciel qui fonctionne pendant un certain temps et s'arrête ensuite proprement.

Il peut être difficile d'obtenir un arrêt propre. L'interblocage[11] représente le problème classique : des threads attendent un signal pour poursuivre leur travail mais il n'arrive jamais.

Par exemple, imaginez un système dans lequel un thread parent crée plusieurs threads enfants et attend qu'ils se terminent pour libérer ses ressources et s'arrêter. Que se passe-t-il en cas d'interblocage dans l'un des threads enfants ? Le thread parent va attendre indéfiniment et le système ne s'arrêtera jamais.

Ou bien envisagez un système semblable auquel on a demandé de s'arrêter. Le thread parent demande à tous ses enfants d'abandonner leurs tâches en cours et de se terminer. Mais que se passe-t-il si deux des enfants fonctionnent selon le modèle producteur-consommateur ? Supposons que le producteur reçoive le signal du parent et se termine rapidement. Le consommateur pourrait être en attente d'un message de la part du producteur et donc se trouver bloqué dans un état où il ne peut plus recevoir le signal d'arrêt. S'il est en attente du producteur, il ne se terminera jamais, empêchant également le thread parent de se terminer.

Ces situations sont relativement fréquentes. Si vous devez écrire du code concurrent qui implique une phase d'arrêt propre, attendez-vous à passer du temps pour que cela se fasse correctement.

---

9. Une section critique est une section du code qui doit être protégée d'une utilisation simultanée pour que le programme fonctionne correctement.
10. Voir la section "Augmenter le débit" à l'Annexe A.
11. Voir la section "Interblocage" à l'Annexe A.

> **Recommandation :** *dès le départ, réfléchissez au problème de terminaison et à une solution opérationnelle. Cela vous demandera plus de temps que vous ne le pensez. Examinez les algorithmes existants car la solution est probablement plus complexe que vous ne le pensez.*

Tester du code multithread

Il est peu réaliste de vouloir prouver que du code est correct. Les tests ne garantissent pas la justesse. Toutefois, de bons tests peuvent réduire les risques. Si cela s'applique parfaitement à une solution monothread, dès que plusieurs threads utilisent le même code et manipulent des données partagées, la question devient beaucoup plus complexe.

> **Recommandation :** *écrivez des tests qui ont la possibilité d'exposer des problèmes et exécutez-les fréquemment, avec différentes configurations du programme, ainsi que différentes configurations et charges du système. Si des tests échouent, trouvez-en la cause. N'ignorez jamais un échec simplement parce que les tests ont réussi à un autre moment.*

Il y a beaucoup d'autres points à prendre en considération. Voici quelques recommandations plus précises :

- Considérez les faux dysfonctionnements comme des problèmes potentiellement liés au multithread.
- Commencez par rendre le code normal, non multithread, opérationnel.
- Faites en sorte que le code multithread soit enfichable.
- Faites en sorte que le code multithread soit réglable.
- Exécutez le code avec plus de threads que de processeurs.
- Exécutez le code sur différentes plates-formes.
- Instrumentez votre code pour essayer et forcer des échecs.

Considérer les faux dysfonctionnements comme des problèmes potentiellement liés au multithread

Le code multithread provoque des dysfonctionnements impossibles. La plupart des développeurs ne voient pas intuitivement comment le code multithread interagit avec le code normal (c'est le cas des auteurs). Les symptômes des bogues dans le code multithread peuvent apparaître une seule fois en plusieurs milliers ou millions d'exécutions. Les tentatives pour reproduire les bogues peuvent être pénibles. Les développeurs sont ainsi amenés à considérer que le dysfonctionnement est dû à des rayons cosmiques, à un pépin matériel ou à toute autre sorte d'événement exceptionnel. Il est préférable de

ne pas supposer l'existence d'événements exceptionnels. Plus vous pensez que le dysfonctionnement est lié à ce type d'événement, plus le code risque de se fonder sur une solution potentiellement défectueuse.

> **Recommandation :** *ne considérez pas les dysfonctionnements du système comme des cas uniques.*

Commencer par rendre le code normal opérationnel

Cela peut sembler évident, mais il n'est pas inutile de le rappeler. Faites en sorte que le code fonctionne en dehors de toute utilisation dans des threads. En général, cela signifie créer des POJO invoqués par des threads. Les POJO ne sont pas conscients des threads et peuvent donc être testés hors de l'environnement multithread. Plus vous pouvez placer de composants du système dans de tels POJO, mieux c'est.

> **Recommandation :** *ne tentez pas de pourchasser à la fois les bogues du code normal et ceux du code multithread. Assurez-vous que votre code fonctionne en dehors des threads.*

Faire en sorte que le code multithread soit enfichable

Vous devez écrire le code de prise en charge de la concurrence de manière à pouvoir l'exécuter dans différentes configurations :

- un ou plusieurs threads, en faisant varier leur nombre au cours de l'exécution ;
- interaction du code multithread avec un composant réel ou une doublure de test ;
- exécution avec des doublures de test qui s'exécutent rapidement, lentement ou de manière variable ;
- configuration des tests pour qu'ils puissent s'exécuter un certain nombre de fois.

> **Recommandation :** *faites en sorte que votre code basé sur les threads soit enfichable pour qu'il puisse s'exécuter dans différentes configurations.*

Faire en sorte que le code multithread soit réglable

Pour obtenir le bon équilibre au sein des threads, il faut généralement passer par des phases d'essais et d'erreurs. Dès le départ, trouvez des solutions pour mesurer les performances de votre système dans différentes configurations. Faites en sorte que le nombre de threads puisse être facilement paramétré. Envisagez même de pouvoir le modifier pendant l'exécution du système. Étudiez la faisabilité d'un réglage automatique basé sur le débit et l'utilisation du système.

Exécuter le code avec plus de threads que de processeurs

C'est lorsque le système bascule d'une tâche à l'autre que des choses se passent. Pour solliciter le changement de tâche, exécutez le code avec plus de threads qu'il n'y a de processeurs ou de cœurs dans l'ordinateur. Plus le changement de tâche est fréquent, plus vous risquez de rencontrer un code qui provoque un interblocage ou dans lequel il manque une section critique.

Exécuter le code sur différentes plates-formes

Vers la mi-2007, nous avons conçu un support de cours sur la programmation concurrente. Il s'appuyait principalement sur Mac OS X. Le cours était donné en utilisant Windows XP dans une machine virtuelle. Les tests écrits pour illustrer les conditions de dysfonctionnement échouaient moins fréquemment dans l'environnement XP que sous Mac OS X.

Nous savions que, dans tous les cas, le code testé n'était pas correct. Cela ne fait que confirmer le fait que les différentes stratégies de gestion du code concurrent par les systèmes d'exploitation ont chacune un impact différent sur l'exécution du code. Le code multithread se comporte différemment en fonction des environnements[12]. Vous devez donc exécuter vos tests dans tous les environnements de déploiement prévus.

> **Recommandation :** *exécutez au plus tôt et le plus souvent possible votre code multithread sur toutes les plates-formes cibles.*

Instrumenter le code pour essayer et forcer des échecs

Il est normal que les défauts du code concurrent se cachent. Bien souvent, les tests simples ne permettent pas de les débusquer. En général, ils restent cachés au cours du fonctionnement normal. Ils peuvent n'apparaître qu'une fois par heure, par jour ou par semaine !

Les bogues du code multithread sont peu fréquents, sporadiques et difficiles à reproduire car seul un très petit nombre de chemins qui passent par la section vulnérable, parmi les milliers existants, conduisent au dysfonctionnement. Par conséquent, la probabilité qu'un chemin d'échec soit pris est très faible. C'est pour cela que la détection et le débogage sont très difficiles.

Comment pouvez-vous augmenter vos chances de tomber sur ces situations rares ? Vous pouvez instrumenter votre code et l'obliger à s'exécuter d'une certaine manière en

12. Savez-vous que le modèle des threads dans Java ne garantit pas un fonctionnement préemptif ? Les systèmes d'exploitation modernes prennent en charge le multithread préemptif, dont vous disposez donc "gratuitement". Toutefois, il n'est pas garanti par la JVM.

ajoutant les appels à des méthodes comme `Object.wait()`, `Object.sleep()`, `Object.yield()` et `Object.priority()`.

Chacune de ces méthodes permet d'influencer l'exécution, augmentant ainsi les chances de détecter une faille. Il est préférable que le code incorrect échoue aussi tôt et aussi souvent que possible.

Il existe deux solutions pour l'instrumentation du code :

- instrumentation manuelle ;
- instrumentation automatisée.

Instrumentation manuelle

Vous pouvez insérer manuellement des appels à `wait()`, `sleep()`, `yield()` et `priority()` dans votre code. C'est peut-être la seule chose à faire lorsque vous testez un pan de code particulièrement épineux. En voici un exemple :

```
public synchronized String nextUrlOrNull() {
    if(hasNext()) {
        String url = urlGenerator.next();
        Thread.yield(); // Ajouté pour les tests.
        updateHasNext();
        return url;
    }
    return null;
}
```

L'appel à `yield()` modifie le chemin d'exécution emprunté par le code et peut le conduire à échouer là où ce n'était pas le cas auparavant. Si le code provoque un dysfonctionnement, cela vient non pas de l'ajout de l'appel à `yield()`[13], mais du fait que votre code était incorrect et que cet appel l'a simplement révélé.

Cependant, cette approche présente de nombreux problèmes :

- Vous devez déterminer manuellement les endroits appropriés pour insérer les appels.
- Comment savez-vous où insérer l'appel et lequel ?
- Lorsque ce code reste dans un environnement de production, il ralentit inutilement le système.

13. Ce n'est pas exactement le cas. Puisque la JVM ne garantit pas un multithread préemptif, un algorithme précis peut toujours fonctionner sur un système d'exploitation qui ne préempte pas les threads. L'inverse est également possible, mais pour des raisons différentes.

- Il s'agit d'une approche forcée. Il est possible que vous ne trouviez aucun défaut. La chance n'est peut-être pas de votre côté.

En réalité, nous avons besoin de mettre en place ce mécanisme pendant les tests, mais pas en production. Nous avons également besoin de varier les configurations entre différentes exécutions pour augmenter les chances de trouver des erreurs.

Si nous divisons notre système en POJO qui ne savent rien du multithread et des classes qui le contrôlent, il sera clairement plus facile de trouver des endroits appropriés pour l'instrumentation du code. Par ailleurs, nous pourrons créer plusieurs gabarits de tests différents qui invoquent les POJO sous différentes configurations d'appel à `sleep`, `yield` et autres.

Instrumentation automatisée

Vous pouvez employer des outils comme Aspect-Oriented Framework, CGLIB ou ASM pour instrumenter le code par programmation. Par exemple, vous pouvez utiliser une classe avec une seule méthode :

```
public class ThreadJigglePoint {
    public static void jiggle() {
    }
}
```

Vous pouvez ajouter des appels à cette méthode en différents endroits du code :

```
public synchronized String nextUrlOrNull() {
    if(hasNext()) {
        ThreadJiglePoint.jiggle();
        String url = urlGenerator.next();
        ThreadJiglePoint.jiggle();
        updateHasNext();
        ThreadJiglePoint.jiggle();
        return url;
    }
    return null;
}
```

Ensuite, il suffit d'employer un simple aspect qui choisit aléatoirement entre ne rien faire, s'endormir ou relâcher le flux de contrôle.

Nous pouvons également imaginer deux implémentations de la classe `ThreadJiggle Point`. La première implémente une version de `jiggle` qui ne fait rien et qui est utilisée en production. La seconde génère un nombre aléatoire pour choisir entre s'endormir, relâcher le flux de contrôle ou poursuivre. Si vous exécutez vos tests un millier de fois avec un choix aléatoire du comportement, vous pourrez débusquer des défauts. Si le test passe, vous pourrez au moins prétendre avoir travaillé consciencieusement. Quoiqu'un tantinet simpliste, cette solution peut se révéler tout à fait raisonnable à la place d'un outil plus sophistiqué.

IBM a développé un outil nommé ConTest[14] qui procède de manière semblable, mais en apportant une sophistication supplémentaire.

L'idée est de "secouer" le code afin que les threads s'exécutent dans un ordre différent à différents moments. L'association de tests bien écrits et du secouement peut énormément augmenter les chances de trouver des erreurs.

> **Recommandation :** *employez des stratégies d'agitation du code pour débusquer des erreurs.*

Conclusion

Le code concurrent correct est difficile à obtenir. Un code facile à suivre peut devenir cauchemardesque lorsque des threads et des données partagées entrent en scène. Si vous êtes confronté à l'écriture d'un code concurrent, vous devez écrire un code propre avec rigueur ou vous risquez de subir des dysfonctionnements subtils et sporadiques.

En premier lieu, conformez-vous au principe de responsabilité unique. Décomposez votre système en POJO qui séparent le code multithread du code normal. Lorsque vous testez le code multithread, assurez-vous que vous le testez lui et lui seul. Sous-entendu, ce code doit être petit et ciblé.

Vous devez connaître les sources potentielles de problèmes dus à la concurrence : multiples threads manipulant des données partagées ou utilisant un ensemble de ressources communes. Le fonctionnement aux limites, comme arrêter proprement le système ou terminer l'itération d'une boucle, peut devenir particulièrement épineux.

Maîtrisez vos bibliothèques et connaissez les algorithmes fondamentaux. Sachez comment les fonctionnalités apportées par la bibliothèque permettent de résoudre des problèmes semblables à ceux des algorithmes fondamentaux.

Apprenez à trouver les parties du code qui doivent être verrouillées et verrouillez-les. Ne verrouillez pas les pans de code qui n'ont pas besoin de l'être. Évitez d'imbriquer des appels aux sections verrouillées. Pour cela, vous devez savoir exactement si quelque chose est partagé ou non. Le nombre d'objets partagés et l'étendue du partage doivent être aussi réduits que possible. Modifiez la conception des objets contenant des données partagées afin de les adapter aux clients plutôt qu'obliger les clients à manipuler un état partagé.

Les problèmes se présenteront. Ceux qui ne se présentent pas tôt passent souvent pour des événements exceptionnels. Ils se produisent généralement sous forte charge ou à des moments apparemment aléatoires. Par conséquent, vous devez être en mesure

14. **http://www.alphaworks ibm.com/tech/contest**.

d'exécuter votre code multithread de manière répétée et continuelle, dans de nombreuses configurations sur de nombreuses plates-formes. La capacité d'être testé, qui vient naturellement lorsqu'on suit les trois lois du TDD, implique une certaine possibilité d'enfichage, qui fournit la prise en charge nécessaire à l'exécution du code dans une grande diversité de configurations.

Vous améliorerez énormément vos chances de trouver des erreurs dans le code si vous prenez le temps de l'instrumenter. Vous pouvez procéder manuellement ou employer une forme de technologie d'automatisation. Faites-le le plus tôt possible. Vous devez exécuter le code multithread aussi longtemps que possible, avant de le mettre en production.

Si vous adoptez une approche nette, vos chances de réussite augmentent énormément.

14

Améliorations successives

Étude de cas : analyseur syntaxique des arguments de la ligne de commande

Ce chapitre montre par l'exemple la mise en œuvre d'améliorations successives. Elles concernent un module bien démarré, mais qui ne pouvait pas évoluer. Vous verrez comment il a pu être remanié et nettoyé.

La grande majorité d'entre nous a déjà eu à analyser les arguments de la ligne de commande. Sans un utilitaire pratique, nous parcourons simplement le tableau des chaînes de caractères passées à la fonction `main`. Si plusieurs bons outils sont disponibles depuis différentes sources, aucun d'entre eux n'offre les fonctionnalités que je souhaite. Par conséquent, j'ai décidé d'écrire mon propre utilitaire et l'ai nommé `Args`.

`Args` est très simple d'usage. Il suffit de créer une instance de la classe `Args` en passant au constructeur les arguments d'entrée et une chaîne de format, puis de demander à cette instance la valeur des arguments. Examinons l'exemple suivant :

Listing 14.1 : Utilisation simple de `Args`

```
public static void main(String[] args) {
  try {
    Args arg = new Args("l,p#,d*", args);
    boolean logging = arg.getBoolean('l');
    int port = arg.getInt('p');
    String directory = arg.getString('d');
    executeApplication(logging, port, directory);
  } catch (ArgsException e) {
    System.out.printf("Argument error: %s\n", e.errorMessage());
  }
}
```

Vous le constatez, ce n'est pas très compliqué. Nous créons simplement une instance de la classe `Args` avec les deux paramètres attendus. Le premier représente la chaîne de format, ou *schéma* : `"l,p#,d*"`. Elle définit trois arguments de la ligne de commande. Le premier, `l`, est un argument booléen. Le deuxième, `p`, est un entier. Le troisième, `d`, est une chaîne de caractères. Le second argument du constructeur de `Args` n'est que le tableau des arguments de la ligne de commande passé à `main`.

Si le constructeur ne lance pas une exception `ArgsException`, cela signifie que la ligne de commande a été analysée et que l'instance de `Args` est prête à être interrogée. Les méthodes comme `getBoolean`, `getInteger` et `getString` permettent d'accéder aux valeurs des arguments d'après leur nom.

En cas de problèmes, que ce soit dans la chaîne de format ou dans les arguments de la ligne de commande eux-mêmes, une exception `ArgsException` est lancée. La méthode `errorMessage` de l'exception permet d'obtenir une description du problème.

Implémentation de *Args*

Le Listing 14.2 correspond à l'implémentation de la classe `Args`. Prenez le temps de le lire attentivement. J'ai fait beaucoup d'efforts sur le style et la structure, et j'espère que cela en valait la peine.

Listing 14.2 : `Args.java`

```
package com.objectmentor.utilities.args;

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;
import java.util.*;

public class Args {
  private Map<Character, ArgumentMarshaler> marshalers;
  private Set<Character> argsFound;
  private ListIterator<String> currentArgument;

  public Args(String schema, String[] args) throws ArgsException {
    marshalers = new HashMap<Character, ArgumentMarshaler>();
    argsFound = new HashSet<Character>();

    parseSchema(schema);
    parseArgumentStrings(Arrays.asList(args));
  }

  private void parseSchema(String schema) throws ArgsException {
    for (String element : schema.split(","))
      if (element.length() > 0)
        parseSchemaElement(element.trim());
  }

  private void parseSchemaElement(String element) throws ArgsException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
      marshalers.put(elementId, new DoubleArgumentMarshaler());
    else if (elementTail.equals("[*]"))
      marshalers.put(elementId, new StringArrayArgumentMarshaler());
    else
      throw new ArgsException(INVALID_ARGUMENT_FORMAT, elementId, elementTail);
  }

  private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId))
      throw new ArgsException(INVALID_ARGUMENT_NAME, elementId, null);
  }

  private void parseArgumentStrings(List<String> argsList) throws ArgsException
  {
    for (currentArgument = argsList.listIterator(); currentArgument.hasNext();)
    {
      String argString = currentArgument.next();
      if (argString.startsWith("-")) {
        parseArgumentCharacters(argString.substring(1));
```

```
        } else {
          currentArgument.previous();
          break;
        }
      }
    }

    private void parseArgumentCharacters(String argChars) throws ArgsException {
      for (int i = 0; i < argChars.length(); i++)
        parseArgumentCharacter(argChars.charAt(i));
    }

    private void parseArgumentCharacter(char argChar) throws ArgsException {
      ArgumentMarshaler m = marshalers.get(argChar);
      if (m == null) {
        throw new ArgsException(UNEXPECTED_ARGUMENT, argChar, null);
      } else {
        argsFound.add(argChar);
        try {
          m.set(currentArgument);
        } catch (ArgsException e) {
          e.setErrorArgumentId(argChar);
          throw e;
        }
      }
    }

    public boolean has(char arg) {
      return argsFound.contains(arg);
    }

    public int nextArgument() {
      return currentArgument.nextIndex();
    }

    public boolean getBoolean(char arg) {
      return BooleanArgumentMarshaler.getValue(marshalers.get(arg));
    }

    public String getString(char arg) {
      return StringArgumentMarshaler.getValue(marshalers.get(arg));
    }

    public int getInt(char arg) {
      return IntegerArgumentMarshaler.getValue(marshalers.get(arg));
    }

    public double getDouble(char arg) {
      return DoubleArgumentMarshaler.getValue(marshalers.get(arg));
    }

    public String[] getStringArray(char arg) {
      return StringArrayArgumentMarshaler.getValue(marshalers.get(arg));
    }
  }
```

Vous noterez que ce code se lit du début à la fin, sans sauter de part en part ni jeter un œil en avant. La seule chose que vous risquez de rechercher plus loin est la définition de `ArgumentMarshaler`, que j'ai volontairement exclue. Après avoir lu ce code, vous devez comprendre les intentions de l'interface `ArgumentMarshaler` et des classes qui en dérivent. Les Listings 14.3 à 14.6 en présentent quelques-unes.

Listing 14.3 : `ArgumentMarshaler.java`

```
public interface ArgumentMarshaler {
  void set(Iterator<String> currentArgument) throws ArgsException;
}
```

Listing 14.4 : `BooleanArgumentMarshaler.java`

```
public class BooleanArgumentMarshaler implements ArgumentMarshaler {
  private boolean booleanValue = false;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    booleanValue = true;
  }

  public static boolean getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof BooleanArgumentMarshaler)
      return ((BooleanArgumentMarshaler) am).booleanValue;
    else
      return false;
  }
}
```

Listing 14.5 : `StringArgumentMarshaler.java`

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class StringArgumentMarshaler implements ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
    try {
      stringValue = currentArgument.next();
    } catch (NoSuchElementException e) {
      throw new ArgsException(MISSING_STRING);
    }
  }

  public static String getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof StringArgumentMarshaler)
      return ((StringArgumentMarshaler) am).stringValue;
    else
      return "";
  }
}
```

Listing 14.6 : `IntegerArgumentMarshaler.java`

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class IntegerArgumentMarshaler implements ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      intValue = Integer.parseInt(parameter);
    } catch (NoSuchElementException e) {
      throw new ArgsException(MISSING_INTEGER);
    } catch (NumberFormatException e) {
      throw new ArgsException(INVALID_INTEGER, parameter);
    }
  }

  public static int getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof IntegerArgumentMarshaler)
      return ((IntegerArgumentMarshaler) am).intValue;
    else
      return 0;
  }
}
```

Les autres classes dérivées de `ArgumentMarshaler` reproduisent simplement ce motif pour les nombres en virgule flottante (`double`) et les tableaux de chaînes de caractères (`String`). Puisqu'elles ne feraient qu'encombrer ce chapitre, faites-les laisse en exercice.

Vous vous interrogez sans doute à propos de la définition des constantes des codes d'erreur. Elles se trouvent dans la classe `ArgsException` (voir Listing 14.7).

Listing 14.7 : `ArgsException.java`

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = null;
  private ErrorCode errorCode = OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public ArgsException(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public ArgsException(ErrorCode errorCode, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
  }
```

```
public ArgsException(ErrorCode errorCode,
                     char errorArgumentId, String errorParameter) {
  this.errorCode = errorCode;
  this.errorParameter = errorParameter;
  this.errorArgumentId = errorArgumentId;
}

public char getErrorArgumentId() {
  return errorArgumentId;
}

public void setErrorArgumentId(char errorArgumentId) {
  this.errorArgumentId = errorArgumentId;
}

public String getErrorParameter() {
  return errorParameter;
}

public void setErrorParameter(String errorParameter) {
  this.errorParameter = errorParameter;
}

public ErrorCode getErrorCode() {
  return errorCode;
}

public void setErrorCode(ErrorCode errorCode) {
  this.errorCode = errorCode;
}

public String errorMessage() {
  switch (errorCode) {
    case OK:
      return "TILT: Should not get here.";
    case UNEXPECTED_ARGUMENT:
      return String.format("Argument -%c unexpected.", errorArgumentId);
    case MISSING_STRING:
      return String.format("Could not find string parameter for -%c.",
                           errorArgumentId);
    case INVALID_INTEGER:
      return String.format("Argument -%c expects an integer but was '%s'.",
                           errorArgumentId, errorParameter);
    case MISSING_INTEGER:
      return String.format("Could not find integer parameter for -%c.",
                           errorArgumentId);
    case INVALID_DOUBLE:
      return String.format("Argument -%c expects a double but was '%s'.",
                           errorArgumentId, errorParameter);
    case MISSING_DOUBLE:
      return String.format("Could not find double parameter for -%c.",
                           errorArgumentId);
    case INVALID_ARGUMENT_NAME:
      return String.format("'%c' is not a valid argument name.",
                           errorArgumentId);
    case INVALID_ARGUMENT_FORMAT:
```

```
          return String.format("'%s' is not a valid argument format.",
                               errorParameter);
        }
        return "";
      }

      public enum ErrorCode {
        OK, INVALID_ARGUMENT_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
        MISSING_STRING,
        MISSING_INTEGER, INVALID_INTEGER,
        MISSING_DOUBLE, INVALID_DOUBLE}
    }
```

Il est étonnant de voir la quantité de code nécessaire à la mise en œuvre des détails de ce simple concept. L'une des raisons vient de notre utilisation d'un langage particulièrement verbeux. Java étant un langage typé statiquement, il exige un grand nombre de mots pour satisfaire le typage. Dans d'autres langages, comme Ruby, Python ou Smalltalk, ce programme serait beaucoup plus court[1].

Veuillez lire le code une nouvelle fois. Faites particulièrement attention aux noms des choses, à la taille des fonctions et à la mise en forme du code. Si vous êtes un programmeur expérimenté, certains points de style ou de structure pourraient vous faire grincer des dents. Toutefois, j'espère que vous serez d'accord pour dire que ce programme est globalement bien écrit et que sa structure est propre.

Par exemple, il doit être évident que l'ajout d'un nouveau type d'argument, comme une date ou un nombre complexe, ne nécessite qu'un petit effort. En bref, il suffit d'écrire une nouvelle classe dérivée de `ArgumentMarshaler`, une nouvelle fonction `getXXX` et une nouvelle instruction de cas dans la fonction `parseSchemaElement`. Il faudra également un nouveau `ArgsException.ErrorCode` et un nouveau message d'erreur.

Comment ai-je procédé ?

Que les choses soient claires. Je n'ai pas écrit ce programme du début à la fin sous sa forme actuelle. Plus important encore, je ne m'attends pas à ce que vous soyez capable d'écrire des programmes propres et élégants en une seule passe. Ces deux dernières décennies nous ont appris que la programmation est plus un art qu'une science. Pour écrire du code propre, vous devez commencer par écrire du code sale *et ensuite le nettoyer.*

Cela ne devrait pas trop vous surprendre. C'est au collège que nous avons appris cette manière de procéder, lorsque nos professeurs essayaient, généralement en vain, de nous faire rédiger des brouillons de nos rédactions. La procédure, nous disaient-ils, consiste

1. J'ai récemment récrit ce module en Ruby. Sa taille a été divisée par sept et sa structure était subtilement meilleure.

à écrire un brouillon sommaire, puis un deuxième brouillon, puis d'autres brouillons jusqu'à obtenir la version finale. L'écriture du texte parfait, tentaient-ils de nous apprendre, est une question d'améliorations successives.

La plupart des programmeurs novices, comme la plupart des élèves sortis du collège, ne suivent pas parfaitement ce conseil. Ils pensent que l'objectif premier est d'obtenir un programme opérationnel. Dès que c'est le cas, ils passent à la tâche suivante, en laissant le programme "opérationnel" dans l'état qui lui a permis de fonctionner. La plupart des programmeurs expérimentés savent que cela s'apparente à un suicide professionnel.

Args : le brouillon initial

Le Listing 14.8 présente une version initiale de la classe `Args`. Elle "fonctionne", mais elle est désordonnée.

Listing 14.8 : `Args.java` **(brouillon initial)**

```
import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private Map<Character, String> stringArgs = new HashMap<Character, String>();
  private Map<Character, Integer> intArgs = new HashMap<Character, Integer>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
```

```
    }
    return valid;
  }

  private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        String trimmedElement = element.trim();
        parseSchemaElement(trimmedElement);
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      parseBooleanSchemaElement(elementId);
    else if (isStringSchemaElement(elementTail))
      parseStringSchemaElement(elementId);
    else if (isIntegerSchemaElement(elementTail)) {
      parseIntegerSchemaElement(elementId);
    } else {
      throw new ParseException(
        String.format("Argument: %c has invalid format: %s.",
                      elementId, elementTail), 0);
    }
  }

  private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
      throw new ParseException(
        "Bad character:" + elementId + "in Args format: " + schema, 0);
    }
  }

  private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, false);
  }

  private void parseIntegerSchemaElement(char elementId) {
    intArgs.put(elementId, 0);
  }

  private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, "");
  }

  private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
  }

  private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
  }
```

```
private boolean isIntegerSchemaElement(String elementTail) {
  return elementTail.equals("#");
}

private boolean parseArguments() throws ArgsException {
  for (currentArgument = 0; currentArgument < args.length; currentArgument++)
  {
    String arg = args[currentArgument];
    parseArgument(arg);
  }
  return true;
}

private void parseArgument(String arg) throws ArgsException {
  if (arg.startsWith("-"))
    parseElements(arg);
}

private void parseElements(String arg) throws ArgsException {
  for (int i = 1; i < arg.length(); i++)
    parseElement(arg.charAt(i));
}

private void parseElement(char argChar) throws ArgsException {
  if (setArgument(argChar))
    argsFound.add(argChar);
  else {
    unexpectedArguments.add(argChar);
    errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
    valid = false;
  }
}

private boolean setArgument(char argChar) throws ArgsException {
  if (isBooleanArg(argChar))
    setBooleanArg(argChar, true);
  else if (isStringArg(argChar))
    setStringArg(argChar);
  else if (isIntArg(argChar))
    setIntArg(argChar);
  else
    return false;

  return true;
}

private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

private void setIntArg(char argChar) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    intArgs.put(argChar, new Integer(parameter));
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
```

```
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (NumberFormatException e) {
      valid = false;
      errorArgumentId = argChar;
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw new ArgsException();
    }
  }

  private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
      stringArgs.put(argChar, args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }

  private boolean isStringArg(char argChar) {
    return stringArgs.containsKey(argChar);
  }

  private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
  }

  private boolean isBooleanArg(char argChar) {
    return booleanArgs.containsKey(argChar);
  }

  public int cardinality() {
    return argsFound.size();
  }

  public String usage() {
    if (schema.length() > 0)
      return "-[" + schema + "]";
    else
      return "";
  }

  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Should not get here.");
      case UNEXPECTED_ARGUMENT:
        return unexpectedArgumentMessage();
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
                             errorArgumentId);
      case INVALID_INTEGER:
```

```
          return String.format("Argument -%c expects an integer but was '%s'.",
                               errorArgumentId, errorParameter);
        case MISSING_INTEGER:
          return String.format("Could not find integer parameter for -%c.",
                               errorArgumentId);
      }
      return "";
    }

    private String unexpectedArgumentMessage() {
      StringBuffer message = new StringBuffer("Argument(s) -");
      for (char c : unexpectedArguments) {
        message.append(c);
      }
      message.append(" unexpected.");
      return message.toString();
    }

    private boolean falseIfNull(Boolean b) {
      return b != null && b;
    }

    private int zeroIfNull(Integer i) {
      return i == null ? 0 : i;
    }

    private String blankIfNull(String s) {
      return s == null ? "" : s;
    }

    public String getString(char arg) {
      return blankIfNull(stringArgs.get(arg));
    }

    public int getInt(char arg) {
      return zeroIfNull(intArgs.get(arg));
    }

    public boolean getBoolean(char arg) {
      return falseIfNull(booleanArgs.get(arg));
    }

    public boolean has(char arg) {
      return argsFound.contains(arg);
    }

    public boolean isValid() {
      return valid;
    }

    private class ArgsException extends Exception {
    }
  }
```

J’espère que votre première réaction à cette quantité de code est "je suis heureux qu’il ne l’ait pas laissé ainsi". Si c’est bien le cas, n’oubliez pas que les autres programmeurs

pourraient avoir la même réaction face à votre code que vous laissez au stade de brouillon initial.

En réalité, qualifier de "brouillon sommaire" ce code est sans doute ce que vous pourriez dire de plus gentil le concernant. Il s'agit clairement d'un travail en cours. Le nombre de variables d'instance est effrayant. Les chaînes étranges, comme `"TILT"`, les `HashSet` et les `TreeSet`, ainsi que les blocs `try-catch-catch` ne font qu'ajouter au "tas d'immondices".

Je ne voulais pas écrire une telle chose. J'ai évidemment essayé d'organiser à peu près raisonnablement les éléments. Vous pouvez certainement le deviner à partir des noms choisis pour les fonctions et les variables, ainsi que de la structure sommaire du programme. Mais il est clair que j'ai laissé le problème m'échapper.

Le désordre s'installe progressivement. Les premières versions n'étaient pas aussi mauvaises. Par exemple, le Listing 14.9 correspond à une version initiale dans laquelle seuls les arguments booléens étaient pris en charge.

Listing 14.9 : `Args.java` **(arguments** `Boolean` **uniquement)**

```
package com.objectmentor.utilities.getopts;

import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private int numberOfArguments = 0;

  public Args(String schema, String[] args) {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  public boolean isValid() {
    return valid;
  }

  private boolean parse() {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    parseArguments();
    return unexpectedArguments.size() == 0;
  }
```

```
private boolean parseSchema() {
  for (String element : schema.split(",")) {
    parseSchemaElement(element);
  }
  return true;
}

private void parseSchemaElement(String element) {
  if (element.length() == 1) {
    parseBooleanSchemaElement(element);
  }
}

private void parseBooleanSchemaElement(String element) {
  char c = element.charAt(0);
  if (Character.isLetter(c)) {
    booleanArgs.put(c, false);
  }
}

private boolean parseArguments() {
  for (String arg : args)
    parseArgument(arg);
  return true;
}

private void parseArgument(String arg) {
  if (arg.startsWith("-"))
    parseElements(arg);
}

private void parseElements(String arg) {
  for (int i = 1; i < arg.length(); i++)
    parseElement(arg.charAt(i));
}

private void parseElement(char argChar) {
  if (isBoolean(argChar)) {
    numberOfArguments++;
    setBooleanArg(argChar, true);
  } else
    unexpectedArguments.add(argChar);
}

private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.put(argChar, value);
}

private boolean isBoolean(char argChar) {
  return booleanArgs.containsKey(argChar);
}

public int cardinality() {
  return numberOfArguments;
}
```

```
  public String usage() {
    if (schema.length() > 0)
       return "-["+schema+"]";
    else
      return "";
  }

  public String errorMessage() {
    if (unexpectedArguments.size() > 0) {
      return unexpectedArgumentMessage();
    } else
      return "";
  }

  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
      message.append(c);
    }
    message.append(" unexpected.");
    return message.toString();
  }

  public boolean getBoolean(char arg) {
    return booleanArgs.get(arg);
  }
}
```

Même si vous pouvez formuler de nombreuses critiques sur ce code, il n'est pas réellement si mauvais. Il est compact, simple et facile à comprendre. Toutefois, il est facile de voir dans le code comment il a pu finalement se transformer en un "tas d'immondices".

Vous remarquerez que le simple ajout de deux autres types d'arguments, `String` et `integer`, a eu un impact massivement négatif sur le code. Il est passé d'un programme qui aurait pu être raisonnablement maintenu à une chose qui sera probablement criblée de bogues et de défauts.

J'ai ajouté progressivement les deux types d'arguments. Tout d'abord, la prise en charge d'un argument `String` a donné le Listing 14.10.

Listing 14.10 : `Args.java` **(arguments** `Boolean` **et** `String`**)**

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
```

```
private Map<Character, String> stringArgs =
  new HashMap<Character, String>();
private Set<Character> argsFound = new HashSet<Character>();
private int currentArgument;
private char errorArgument = '\0';

enum ErrorCode {
  OK, MISSING_STRING}

private ErrorCode errorCode = ErrorCode.OK;

public Args(String schema, String[] args) throws ParseException {
  this.schema = schema;
  this.args = args;
  valid = parse();
}

private boolean parse() throws ParseException {
  if (schema.length() == 0 && args.length == 0)
    return true;
  parseSchema();
  parseArguments();
  return valid;
}

private boolean parseSchema() throws ParseException {
  for (String element : schema.split(",")) {
    if (element.length() > 0) {
      String trimmedElement = element.trim();
      parseSchemaElement(trimmedElement);
    }
  }
  return true;
}

private void parseSchemaElement(String element) throws ParseException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (isBooleanSchemaElement(elementTail))
    parseBooleanSchemaElement(elementId);
  else if (isStringSchemaElement(elementTail))
    parseStringSchemaElement(elementId);
}

private void validateSchemaElementId(char elementId) throws ParseException {
  if (!Character.isLetter(elementId)) {
    throw new ParseException(
      "Bad character:" + elementId + "in Args format: " + schema, 0);
  }
}

private void parseStringSchemaElement(char elementId) {
  stringArgs.put(elementId, "");
}
```

```
private boolean isStringSchemaElement(String elementTail) {
  return elementTail.equals("*");
}

private boolean isBooleanSchemaElement(String elementTail) {
  return elementTail.length() == 0;
}

private void parseBooleanSchemaElement(char elementId) {
  booleanArgs.put(elementId, false);
}

private boolean parseArguments() {
  for (currentArgument = 0; currentArgument < args.length; currentArgument++)
  {
    String arg = args[currentArgument];
    parseArgument(arg);
  }
  return true;
}

private void parseArgument(String arg) {
  if (arg.startsWith("-"))
    parseElements(arg);
}

private void parseElements(String arg) {
  for (int i = 1; i < arg.length(); i++)
    parseElement(arg.charAt(i));
}

private void parseElement(char argChar) {
  if (setArgument(argChar))
    argsFound.add(argChar);
  else {
    unexpectedArguments.add(argChar);
    valid = false;
  }
}

private boolean setArgument(char argChar) {
  boolean set = true;
  if (isBoolean(argChar))
    setBooleanArg(argChar, true);
  else if (isString(argChar))
    setStringArg(argChar, "");
  else
    set = false;

  return set;
}

private void setStringArg(char argChar, String s) {
  currentArgument++;
  try {
    stringArgs.put(argChar, args[currentArgument]);
```

```
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgument = argChar;
      errorCode = ErrorCode.MISSING_STRING;
    }
  }

  private boolean isString(char argChar) {
    return stringArgs.containsKey(argChar);
  }

  private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
  }

  private boolean isBoolean(char argChar) {
    return booleanArgs.containsKey(argChar);
  }

  public int cardinality() {
    return argsFound.size();
  }

  public String usage() {
    if (schema.length() > 0)
      return "-[" + schema + "]";
    else
      return "";
  }

  public String errorMessage() throws Exception {
    if (unexpectedArguments.size() > 0) {
      return unexpectedArgumentMessage();
    } else
      switch (errorCode) {
        case MISSING_STRING:
          return String.format("Could not find string parameter for -%c.",
                               errorArgument);
        case OK:
          throw new Exception("TILT: Should not get here.");
      }
    return "";
  }

  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
      message.append(c);
    }
    message.append(" unexpected.");
    return message.toString();
  }

  public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg));
  }
```

```
    private boolean falseIfNull(Boolean b) {
      return b == null ? false : b;
    }

    public String getString(char arg) {
      return blankIfNull(stringArgs.get(arg));
    }

    private String blankIfNull(String s) {
      return s == null ? "" : s;
    }

    public boolean has(char arg) {
      return argsFound.contains(arg);
    }

    public boolean isValid() {
      return valid;
    }
  }
```

Vous le constatez, le programme commençait à déraper. Il n'était pas trop horrible, mais le désordre s'installe sûrement. Il s'agissait d'un tas, mais pas encore d'immondices. Cependant, il me suffisait d'ajouter la prise en charge des arguments de type entier pour que tout s'écroule.

J'ai donc arrêté

J'avais encore au moins deux autres types d'arguments à ajouter, ce qui n'aurait fait qu'empirer les choses. En forçant, j'aurais sans doute réussi à obtenir un code opérationnel, mais j'aurais laissé un désordre trop important pour être réparé. Si la structure de ce code a été un jour maintenable, il est temps à présent de la corriger.

J'ai donc arrêté d'ajouter des fonctionnalités et commencé à remanier le code. Puisque je venais d'ajouter les arguments `String` et `integer`, je savais que chaque type d'argument nécessitait du nouveau code en trois principaux endroits. Tout d'abord, pour chaque type d'argument, il faut mettre en œuvre une analyse de son élément du schéma afin de sélectionner le `HashMap` adéquat. Ensuite, chaque type d'argument doit être analysé dans les chaînes de la ligne de commande et converti en son type réel. Enfin, chaque type d'argument a besoin d'une méthode `getXXX` afin qu'il puisse être retourné à l'appelant dans son type réel.

Plusieurs types différents, tous avec des méthodes semblables, cela ressemble fort à une classe. Le concept de `ArgumentMarshaler` était né.

De manière incrémentale

L'une des meilleures façons de détruire un programme consiste à modifier profondément sa structure sous prétexte de l'améliorer. Certains programmes ne se relèvent

jamais d'une telle "amélioration". En effet, il est très difficile d'obtenir un programme qui fonctionne comme avant l'"amélioration".

Pour éviter ce problème, j'emploie le développement piloté par les tests (TDD, *Test-Driven Development*). Parmi les idées sous-jacentes à cette approche, l'une est de garder en permanence un système opérationnel. Autrement dit, en utilisant le TDD, je m'interdis d'apporter au système une modification qui remet en cause son fonctionnement. Chaque changement effectué doit laisser le système dans l'état de fonctionnement où il se trouvait auparavant.

Pour y parvenir, j'ai besoin d'une suite de tests automatisés que je peux exécuter à souhait pour vérifier le bon comportement du système. Pour la classe `Args`, j'ai créé une suite de tests unitaires et de recette pendant que j'amoncelais les immondices. Les tests unitaires ont été écrits en Java sous le contrôle de JUnit. Les tests de recette ont été écrits sous forme de pages wiki dans FitNesse. Je suis en mesure d'exécuter ces tests à tout moment et, s'ils réussissent, je suis certain que le système fonctionne comme prévu.

J'ai donc continué à effectuer un grand nombre de très petites modifications. Chaque changement rapprochait la structure du système vers le concept de `ArgumentMarshaler`, sans modifier son comportement. Ma première modification a été d'ajouter le squelette de `ArgumentMarshaller` (voir Listing 14.11).

Listing 14.11 : `ArgumentMarshaller` **ajouté dans** `Args.java`

```
private class ArgumentMarshaler {
    private boolean booleanValue = false;

    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {return booleanValue;}
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  }

  private class StringArgumentMarshaler extends ArgumentMarshaler {
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  }
}
```

Il était clair que cet ajout n'allait rien casser. Par conséquent, j'ai ensuite apporté la modification la plus simple possible, celle qui avait le plus petit impact négatif. J'ai modifié le `HashMap` dédié aux arguments booléens afin qu'il prenne un `ArgumentMarshaler`.

```
private Map<Character, ArgumentMarshaler> booleanArgs =
  new HashMap<Character, ArgumentMarshaler>();
```

Cette opération a cassé quelques instructions, que j'ai rapidement corrigées.

```
...
private void parseBooleanSchemaElement(char elementId) {
  booleanArgs.put(elementId, new BooleanArgumentMarshaler());
}
...
private void setBooleanArg(char argChar, boolean value) {
  booleanArgs.get(argChar).setBoolean(value);
}
...
public boolean getBoolean(char arg) {
  return falseIfNull(booleanArgs.get(arg).getBoolean());
}
```

Vous remarquerez que ces modifications concernent précisément les endroits mentionnés précédemment : les méthodes `parse`, `set` et `get` pour le type d'argument. Malheureusement, aussi limitée qu'ait pu être cette modification, certains tests ont commencé à échouer. En examinant attentivement `getBoolean`, vous verrez que si elle est invoquée avec la valeur `'y,'` sans que la ligne de commande ne contienne d'argument `y`, alors, `booleanArgs.get('y')` retourne `null` et la fonction lance une exception `NullPointerException`. La fonction `falseIfNull` permet de se protéger contre ce cas, mais la modification apportée remet en cause sa pertinence.

L'approche incrémentale m'imposait de corriger cela rapidement avant de procéder à tout autre changement. Bien sûr, ce n'était pas trop difficile. J'ai simplement dû déplacer la vérification de `null`. Cette vérification devait concerner non plus le booléen, mais `ArgumentMarshaller`.

J'ai commencé par supprimer l'appel à `falseIfNull` dans la fonction `getBoolean`. Puisqu'elle était devenue inutile, j'ai également supprimé la fonction correspondante. Les tests échouaient toujours de la même manière, ce qui confirmait que je n'avais pas introduit de nouvelles erreurs.

```
public boolean getBoolean(char arg) {
  return booleanArgs.get(arg).getBoolean();
}
```

Ensuite, j'ai découpé la fonction en deux lignes et placé `ArgumentMarshaller` dans sa propre variable nommée `argumentMarshaller`. Je n'avais pas fait attention à la longueur de ce nom. Puisqu'il était redondant et encombrait la fonction, je l'ai raccourci à `am` [N5].

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am.getBoolean();
}
```

J'ai ensuite ajouté la logique de détection de `null`.

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am != null && am.getBoolean();
}
```

Arguments de type *String*

L'ajout des arguments de type `String` s'est fait de manière très comparable à celui des arguments `boolean`. J'ai modifié le `HashMap` et remis en état les fonctions `parse`, `set` et `get`. Le code suivant ne devrait pas présenter de surprises, excepté, peut-être, que toute l'implémentation de l'encodage va dans la classe de base `ArgumentMarshaller` au lieu de se répartir dans les classes dérivées.

```
private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
...
private void parseStringSchemaElement(char elementId) {
  stringArgs.put(elementId, new StringArgumentMarshaler());
}
...
private void setStringArg(char argChar) throws ArgsException {
  currentArgument++;
  try {
    stringArgs.get(argChar).setString(args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}
...
public String getString(char arg) {
  Args.ArgumentMarshaler am = stringArgs.get(arg);
  return am == null ? "" : am.getString();
}
...
private class ArgumentMarshaler {
  private boolean booleanValue = false;
  private String stringValue;

  public void setBoolean(boolean value) {
    booleanValue = value;
  }

  public boolean getBoolean() {
    return booleanValue;
  }

  public void setString(String s) {
    stringValue = s;
  }
```

```
    public String getString() {
      return stringValue == null ? "" : stringValue;
    }
  }
```

À nouveau, ces modifications ont été apportées une par une, en vérifiant que les tests sont exécutés, à défaut de réussir. Lorsqu'un test échoue, je fais en sorte qu'il réussisse à nouveau avant de passer à la modification suivante.

Vous devez à présent deviner mes intentions. Dès que toutes les procédures d'encodage existantes auront intégré la classe de base `ArgumentMarshaler`, je commencerai à les déplacer vers les classes dérivées. Je pourrai ainsi conserver un programme opérationnel tout en changeant progressivement sa structure.

L'étape suivante a été de déplacer la prise en charge des arguments entiers dans `ArgumentMarshaler`. Cela s'est fait sans difficultés particulières.

```
private Map<Character, ArgumentMarshaler> intArgs =
   new HashMap<Character, ArgumentMarshaler>();
...
private void parseIntegerSchemaElement(char elementId) {
  intArgs.put(elementId, new IntegerArgumentMarshaler());
}
...
private void setIntArg(char argChar) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    intArgs.get(argChar).setInteger(Integer.parseInt(parameter));
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
  } catch (NumberFormatException e) {
    valid = false;
    errorArgumentId = argChar;
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw new ArgsException();
  }
}
...
public int getInt(char arg) {
  Args.ArgumentMarshaler am = intArgs.get(arg);
  return am == null ? 0 : am.getInteger();
}
...
private class ArgumentMarshaler {
  private boolean booleanValue = false;
  private String stringValue;
  private int integerValue;
```

```
    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {
      return booleanValue;
    }

    public void setString(String s) {
      stringValue = s;
    }

    public String getString() {
      return stringValue == null ? "" : stringValue;
    }

    public void setInteger(int i) {
      integerValue = i;
    }

    public int getInteger() {
      return integerValue;
    }
  }
```

Puisque tout l'encodage se trouvait alors dans `ArgumentMarshaler`, j'ai commencé à transférer cette fonctionnalité dans les classes dérivées. La première étape a été de déplacer la fonction `setBoolean` dans `BooleanArgumentMarshaller`, en vérifiant qu'elle est appelée correctement. J'ai donc créé une méthode `set` abstraite.

```
  private abstract class ArgumentMarshaler {
    protected boolean booleanValue = false;
    private String stringValue;
    private int integerValue;

    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {
      return booleanValue;
    }

    public void setString(String s) {
      stringValue = s;
    }

    public String getString() {
      return stringValue == null ? "" : stringValue;
    }

    public void setInteger(int i) {
      integerValue = i;
    }

    public int getInteger() {
```

```
      return integerValue;
    }

    public abstract void set(String s);
  }
```

Puis j'ai implémenté la méthode `set` dans `BooleanArgumentMarshaller`.

```
  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    public void set(String s) {
      booleanValue = true;
    }
  }
```

Enfin, j'ai remplacé l'appel à `setBoolean` par une invocation de `set`.

```
  private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.get(argChar).set("true");
  }
```

Les tests réussissaient toujours. Puisque cette modification a déplacé `set` dans `BooleanArgumentMarshaler`, j'ai supprimé la méthode `setBoolean` de la classe de base `ArgumentMarshaler`.

Notez que la fonction `set` abstraite prend un argument de type `String`, mais que l'implémentation de `BooleanArgumentMarshaller` ne s'en sert pas. J'ai ajouté cet argument car je savais que `StringArgumentMarshaller` et `IntegerArgumentMarshaller` allaient l'utiliser.

Ensuite, j'ai voulu déplacer la méthode `get` dans `BooleanArgumentMarshaler`. Le traitement des fonctions `get` est toujours déplaisant car le type de retour doit être `Object` et, dans ce cas, il doit être forcé à `Boolean`.

```
  public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean)am.get();
  }
```

Pour que le code compile, j'ai ajouté la fonction `get` dans `ArgumentMarshaler`.

```
  private abstract class ArgumentMarshaler {
    ...

    public Object get() {
      return null;
    }
  }
```

Si la compilation était possible, les tests échouaient. Pour qu'ils réussissent à nouveau, il fallait simplement rendre `get` abstraite et l'implémenter dans `BooleanArgument Marshaler`.

```
  private abstract class ArgumentMarshaler {
    protected boolean booleanValue = false;
    ...
```

```
    public abstract Object get();
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    public void set(String s) {
      booleanValue = true;
    }

    public Object get() {
       return booleanValue;
    }
  }
```

Les tests réussissaient à nouveau. Les méthodes `get` et `set` avaient été transférées dans `BooleanArgumentMarshaler` ! Cela m'a permis de supprimer l'ancienne méthode `getBoolean` de `ArgumentMarshaler`, de déplacer la variable protégée `booleanValue` dans `BooleanArgumentMarshaler` et de la rendre privée.

J'ai appliqué les mêmes modifications pour les chaînes de caractères. J'ai déployé `set` et `get`, supprimé les fonctions inutiles et déplacé les variables.

```
private void setStringArg(char argChar) throws ArgsException {
  currentArgument++;
  try {
    stringArgs.get(argChar).set(args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}
...
public String getString(char arg) {
  Args.ArgumentMarshaler am = stringArgs.get(arg);
  return am == null ? "" : (String) am.get();
}
...
private abstract class ArgumentMarshaler {
  private int integerValue;

  public void setInteger(int i) {
    integerValue = i;
  }

  public int getInteger() {
    return integerValue;
  }

  public abstract void set(String s);

  public abstract Object get();
}
```

```
private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  private boolean booleanValue = false;

  public void set(String s) {
    booleanValue = true;
  }

  public Object get() {
    return booleanValue;
  }
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
  private String stringValue = "";

  public void set(String s) {
    stringValue = s;
  }

  public Object get() {
    return stringValue;
  }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {

  public void set(String s) {

  }

  public Object get() {
    return null;
  }
}
```

Pour finir, j'ai reproduit ces modifications pour les entiers. Elles étaient un tantinet plus complexes car les entiers doivent être analysés et l'opération `parse` peut lancer une exception. Néanmoins, le résultat obtenu était meilleur car tout ce qui concernait `NumberFormatException` s'est retrouvé enfoui dans `IntegerArgumentMarshaler`.

```
private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

private void setIntArg(char argChar) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    intArgs.get(argChar).set(parameter);
  } catch (ArrayIndexOutOfBoundsException e) {
    valid = false;
    errorArgumentId = argChar;
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
```

```
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }
  ...
  private void setBooleanArg(char argChar) {
    try {
      booleanArgs.get(argChar).set("true");
    } catch (ArgsException e) {
    }
  }
  ...
  public int getInt(char arg) {
    Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : (Integer) am.get();
  }
  ...
  private abstract class ArgumentMarshaler {
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
  }
  ...
  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(String s) throws ArgsException {
      try {
        intValue = Integer.parseInt(s);
      } catch (NumberFormatException e) {
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }
```

Les tests réussissaient encore. Je me suis ensuite débarrassé des trois `Map` au début de l'algorithme. Le système est alors devenu plus générique. Cependant, je ne pouvais pas simplement les supprimer, car cela aurait cassé le système. Au lieu de cela, j'ai ajouté un nouveau `Map` pour `ArgumentMarshaler`, puis j'ai modifié les méthodes une par une pour qu'elles l'utilisent à la place des trois `Map` initiaux.

```
public class Args {
...
  private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
```

```
...
  private void parseBooleanSchemaElement(char elementId) {
    ArgumentMarshaler m = new BooleanArgumentMarshaler();
    booleanArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }

  private void parseIntegerSchemaElement(char elementId) {
    ArgumentMarshaler m = new IntegerArgumentMarshaler();
    intArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }

  private void parseStringSchemaElement(char elementId) {
    ArgumentMarshaler m = new StringArgumentMarshaler();
    stringArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }
```

Les tests réussissaient toujours. Ensuite, j'ai transformé `isBooleanArg`, de :

```
private boolean isBooleanArg(char argChar) {
  return booleanArgs.containsKey(argChar);
}
```

en :

```
private boolean isBooleanArg(char argChar) {
  ArgumentMarshaler m = marshalers.get(argChar);
  return m instanceof BooleanArgumentMarshaler;
}
```

Les tests réussissaient toujours. J'ai donc procédé aux mêmes modifications pour `isIntArg` et `isStringArg`.

```
private boolean isIntArg(char argChar) {
  ArgumentMarshaler m = marshalers.get(argChar);
  return m instanceof IntegerArgumentMarshaler;
}

private boolean isStringArg(char argChar) {
  ArgumentMarshaler m = marshalers.get(argChar);
  return m instanceof StringArgumentMarshaler;
}
```

Les tests réussissaient toujours. J'ai donc pu éliminer tous les appels redondants à `marshalers.get` :

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (isBooleanArg(m))
    setBooleanArg(argChar);
  else if (isStringArg(m))
    setStringArg(argChar);
  else if (isIntArg(m))
    setIntArg(argChar);
  else
```

```
      return false;
    return true;
  }

  private boolean isIntArg(ArgumentMarshaler m) {
    return m instanceof IntegerArgumentMarshaler;
  }

  private boolean isStringArg(ArgumentMarshaler m) {
    return m instanceof StringArgumentMarshaler;
  }

  private boolean isBooleanArg(ArgumentMarshaler m) {
    return m instanceof BooleanArgumentMarshaler;
  }
```

Il n'y avait plus aucune bonne raison de conserver les méthodes `isxxxArg`. Je les ai donc incorporées :

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(argChar);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(argChar);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(argChar);
    else
      return false;
    return true;
  }
```

Ensuite, j'ai commencé à employer la carte `marshalers` dans les fonctions `set`, remettant en question l'utilisation des trois autres `Map`. J'ai débuté par les arguments booléens.

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(m);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(argChar);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(argChar);
    else
      return false;
    return true;
  }
  ...
  private void setBooleanArg(ArgumentMarshaler m) {
    try {
      m.set("true"); // Était à l'origine booleanArgs.get(argChar).set("true");
    } catch (ArgsException e) {
    }
  }
```

Les tests réussissaient toujours. J'ai donc appliqué la même procédure pour les `String` et les `Integer`. Cela m'a permis d'intégrer une partie du code de gestion des exceptions dans la fonction `setArgument`.

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  try {
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(m);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);
    else
      return false;
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}

private void setIntArg(ArgumentMarshaler m) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    m.set(parameter);
  } catch (ArrayIndexOutOfBoundsException e) {
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
  } catch (ArgsException e) {
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw e;
  }
}

private void setStringArg(ArgumentMarshaler m) throws ArgsException {
  currentArgument++;
  try {
    m.set(args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}
```

J'étais tout prêt de réussir à supprimer les trois anciens `Map`. Tout d'abord, je devais transformer la fonction `getBoolean`, de :

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am != null && (Boolean) am.get();
}
```

en :

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  boolean b = false;
  try {
    b = am != null && (Boolean) am.get();
  } catch (ClassCastException e) {
    b = false;
  }
  return b;
}
```

Cette dernière modification pourrait paraître surprenante. Pourquoi ai-je soudainement décidé d'utiliser l'exception `ClassCastException` ? La raison vient du fait que je disposais d'un ensemble séparé de tests unitaires et de tests de recette écrits dans FitNesse. Les tests FitNesse s'assuraient que `false` était retourné en cas d'invocation de `getBoolean` avec un argument non booléen. Ce n'était pas le cas des tests unitaires. Jusqu'à ce stade, je n'avais exécuté que les tests unitaires[2].

J'ai ensuite pu retirer une autre utilisation du `Map` pour les booléens :

```
private void parseBooleanSchemaElement(char elementId) {
  ArgumentMarshaler m = new BooleanArgumentMarshaler();
  booleanArgs.put(elementId, m);
  marshalers.put(elementId, m);
}
```

Et ainsi supprimer ce `Map`.

```
public class Args {
...
  private Map<Character, ArgumentMarshaler> booleanArgs =
  new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> stringArgs =
  new HashMap<Character, ArgumentMarshaler>();

  private Map<Character, ArgumentMarshaler> intArgs =
  new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =
  new HashMap<Character, ArgumentMarshaler>();
...
```

Ensuite, j'ai opéré de la même manière pour les arguments `String` et `Integer`, et nettoyé légèrement le code pour les booléens.

```
private void parseBooleanSchemaElement(char elementId) {
  marshalers.put(elementId, new BooleanArgumentMarshaler());
}
```

2. Pour éviter d'autres surprises de ce genre, j'ai ajouté un nouveau test unitaire qui invoquait tous les tests FitNesse.

```
private void parseIntegerSchemaElement(char elementId) {
  marshalers.put(elementId, new IntegerArgumentMarshaler());
}

private void parseStringSchemaElement(char elementId) {
  marshalers.put(elementId, new StringArgumentMarshaler());
}
...
public String getString(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  try {
    return am == null ? "" : (String) am.get();
  } catch (ClassCastException e) {
    return "";
  }
}

public int getInt(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  try {
    return am == null ? 0 : (Integer) am.get();
  } catch (Exception e) {
    return 0;
  }
}
...
public class Args {
...
- private Map<Character, ArgumentMarshaler> stringArgs =
- new HashMap<Character, ArgumentMarshaler>();
- private Map<Character, ArgumentMarshaler> intArgs =
- new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =
  new HashMap<Character, ArgumentMarshaler>();
...
```

Puis j'ai incorporé les trois méthodes `parse` dans la classe car elles n'apportaient rien de plus :

```
private void parseSchemaElement(String element) throws ParseException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (isBooleanSchemaElement(elementTail))
    marshalers.put(elementId, new BooleanArgumentMarshaler());
  else if (isStringSchemaElement(elementTail))
    marshalers.put(elementId, new StringArgumentMarshaler());
  else if (isIntegerSchemaElement(elementTail)) {
    marshalers.put(elementId, new IntegerArgumentMarshaler());
  } else {
    throw new ParseException(String.format(
    "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
  }
}
```

Très bien, examinons à présent le résultat de tous ces changements. Le Listing 14.12 présente la version courante de la classe `Args`.

Listing 14.12 : `Args.java` **(suite au premier remaniement)**

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, ArgumentMarshaler> marshalers =
  new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }

  private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        String trimmedElement = element.trim();
        parseSchemaElement(trimmedElement);
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
```

```
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (isStringSchemaElement(elementTail))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (isIntegerSchemaElement(elementTail)) {
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    } else {
      throw new ParseException(String.format(
      "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
    }
  }

  private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
      throw new ParseException(
      "Bad character:" + elementId + "in Args format: " + schema, 0);
    }
  }

  private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
  }

  private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
  }

  private boolean isIntegerSchemaElement(String elementTail) {
    return elementTail.equals("#");
  }

  private boolean parseArguments() throws ArgsException {
    for (currentArgument=0; currentArgument<args.length; currentArgument++) {
      String arg = args[currentArgument];
      parseArgument(arg);
    }
    return true;
  }

  private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      unexpectedArguments.add(argChar);
      errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
```

```
      valid = false;
    }
  }

  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    try {
      if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);
      else
        return false;
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }

  private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    try {
      m.set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }

  private void setBooleanArg(ArgumentMarshaler m) {
    try {
      m.set("true");
    } catch (ArgsException e) {
    }
  }
```

```
public int cardinality() {
  return argsFound.size();
}

public String usage() {
  if (schema.length() > 0)
    return "-[" + schema + "]";
  else
    return "";
}

public String errorMessage() throws Exception {
  switch (errorCode) {
    case OK:
      throw new Exception("TILT: Should not get here.");
    case UNEXPECTED_ARGUMENT:
      return unexpectedArgumentMessage();
    case MISSING_STRING:
      return String.format("Could not find string parameter for -%c.",
                           errorArgumentId);
    case INVALID_INTEGER:
      return String.format("Argument -%c expects an integer but was '%s'.",
                           errorArgumentId, errorParameter);
    case MISSING_INTEGER:
      return String.format("Could not find integer parameter for -%c.",
                           errorArgumentId);
  }
  return "";
}

private String unexpectedArgumentMessage() {
  StringBuffer message = new StringBuffer("Argument(s) -");
  for (char c : unexpectedArguments) {
    message.append(c);
  }
  message.append(" unexpected.");

  return message.toString();
}

public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  boolean b = false;
  try {
    b = am != null && (Boolean) am.get();
  } catch (ClassCastException e) {
    b = false;
  }
  return b;
}

public String getString(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  try {
    return am == null ? "" : (String) am.get();
  } catch (ClassCastException e) {
```

```
    return "";
  }
}

public int getInt(char arg) {
  Args.ArgumentMarshaler am = marshalers.get(arg);
  try {
    return am == null ? 0 : (Integer) am.get();
  } catch (Exception e) {
    return 0;
  }
}

public boolean has(char arg) {
  return argsFound.contains(arg);
}

public boolean isValid() {
  return valid;
}

private class ArgsException extends Exception {
}

private abstract class ArgumentMarshaler {
  public abstract void set(String s) throws ArgsException;
  public abstract Object get();
}

private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  private boolean booleanValue = false;

  public void set(String s) {
    booleanValue = true;
  }

  public Object get() {
    return booleanValue;
  }
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
  private String stringValue = "";

  public void set(String s) {
    stringValue = s;
  }

  public Object get() {
    return stringValue;
  }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0;
```

```
    public void set(String s) throws ArgsException {
      try {
        intValue = Integer.parseInt(s);
      } catch (NumberFormatException e) {
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }
}
```

Voilà donc où j'en étais arrivé après le premier remaniement. Malgré tout le travail réalisé, le résultat est un peu décevant. La structure est certainement meilleure, mais il reste toujours des variables au début, une construction horrible de gestion des types dans `setArgument` et toutes ces fonctions `set` laides, sans mentionner le traitement des erreurs. Le travail est loin d'être terminé. Procédons à un nouveau remaniement.

Je voudrais vraiment retirer la construction de gestion des types dans `setArgument` [G23]. J'aimerais que `setArgument` contienne un seul appel à `ArgumentMarshaler.set`. Cela signifie que je dois déplacer `setIntArg`, `setStringArg` et `setBooleanArg` dans la classe dérivée de `ArgumentMarshaler` appropriée. Mais il y a un problème.

Si vous examinez attentivement `setIntArg`, vous noterez qu'elle utilise deux variables d'instances : `args` et `currentArg`. Si je transfère `setIntArg` dans `BooleanArgument Marshaler`, je devrais passer `args` et `currentArgs` en argument de fonction. Ce n'est pas très propre [F1]. Je préférerais passer un seul argument, non deux. Heureusement, il existe une solution simple. Le tableau `args` peut être converti en une liste, puis il suffit de passer un `Iterator` aux fonctions `set`. Je suis arrivé au code suivant en dix étapes, en exécutant à chaque fois tous les tests. Je vous montre simplement le résultat, mais vous ne devriez pas avoir trop de mal à imaginer ces petites étapes.

```
public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private Iterator<String> currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;
  private List<String> argsList;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}
```

```
  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && argsList.size() == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }
---
  private boolean parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
      String arg = currentArgument.next();
      parseArgument(arg);
    }

    return true;
  }
---
  private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      m.set(parameter);
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    try {
      m.set(currentArgument.next());
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
```

Tous ces changements sont simples et permettent aux tests de réussir. Je peux à présent déplacer les fonctions `set` dans les classes dérivées appropriées. Je dois tout d'abord commencer par modifier `setArgument` de la manière suivante :

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
```

```
      if (m == null)
        return false;
      try {
        if (m instanceof BooleanArgumentMarshaler)
          setBooleanArg(m);
        else if (m instanceof StringArgumentMarshaler)
          setStringArg(m);
        else if (m instanceof IntegerArgumentMarshaler)
          setIntArg(m);
        else
          return false;
      } catch (ArgsException e) {
        valid = false;
        errorArgumentId = argChar;
        throw e;
      }
      return true;
    }
```

Cette modification est importante car, pour retirer intégralement la chaîne des `if-else`, la condition d'erreur doit en être sortie.

Je peux à présent déplacer les fonctions `set`. Puisque la fonction `setBooleanArg` est simple, elle va être le premier cobaye. L'objectif est de la modifier de manière à passer par `BooleanArgumentMarshaler`.

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
      if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m, currentArgument);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);

    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }

---

  private void setBooleanArg(ArgumentMarshaler m,
                             Iterator<String> currentArgument)
                             throws ArgsException {
    try {
      m.set("true");
    catch (ArgsException e) {
    }
  }
```

Vous aurez remarqué que je supprime le traitement d'une exception que je viens juste d'ajouter. En effet, au cours d'un remaniement, il est très fréquent d'ajouter du code, pour ensuite le retirer. Les petites modifications très ciblées et l'exécution nécessaire des tests conduisent à de nombreuses allées et venues. Le remaniement ressemble fortement à la résolution d'un Rubik's cube. De très nombreuses petites étapes permettent d'atteindre un objectif plus grand. Chaque étape permet d'effectuer la suivante.

Pourquoi ai-je passé cet `iterator` alors que `setBooleanArg` n'en a certainement pas besoin ? Simplement parce que `setIntArg` et `setStringArg` en auront besoin ! Par ailleurs, puisque je souhaite déplacer ces trois fonctions grâce à une méthode abstraite dans `ArgumentMarshaller`, je dois le passer à `setBooleanArg`.

La fonction `setBooleanArg` est désormais inutile. S'il existait une fonction `set` dans `ArgumentMarshaler`, je pourrais l'invoquer directement. Il est donc temps de créer cette fonction ! La première étape consiste à ajouter la nouvelle méthode abstraite dans `ArgumentMarshaler`.

```
private abstract class ArgumentMarshaler {
  public abstract void set(Iterator<String> currentArgument)
                       throws ArgsException;
  public abstract void set(String s) throws ArgsException;
  public abstract Object get();
}
```

Bien sûr, cela perturbe toutes les classes dérivées. Je dois donc implémenter la nouvelle méthode dans chacune d'elles.

```
private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  private boolean booleanValue = false;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    booleanValue = true;
  }

  public void set(String s) {
    booleanValue = true;
  }

  public Object get() {
    return booleanValue;
  }
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
  }

  public void set(String s) {
    stringValue = s;
  }
```

```
    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
    }

    public void set(String s) throws ArgsException {
      try {
        intValue = Integer.parseInt(s);
      } catch (NumberFormatException e) {
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }
```

Je peux à présent supprimer `setBooleanArg` !

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
      if (m instanceof BooleanArgumentMarshaler)
        m.set(currentArgument);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);

    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }
```

Les tests réussissent tous, et la fonction `set` a été déplacée dans `BooleanArgumentMarshaler` !

La même procédure s'applique pour les `String` et les `Integer`.

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
```

```
      if (m instanceof BooleanArgumentMarshaler)
        m.set(currentArgument);
      else if (m instanceof StringArgumentMarshaler)
        m.set(currentArgument);
      else if (m instanceof IntegerArgumentMarshaler)
        m.set(currentArgument);

    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }

---

  private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";

    public void set(Iterator<String> currentArgument) throws ArgsException {
      try {
        stringValue = currentArgument.next();
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_STRING;
        throw new ArgsException();
      }
    }

    public void set(String s) {
    }

    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      set(parameter);
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }
```

```
    public void set(String s) throws ArgsException {
      try {
        intValue = Integer.parseInt(s);
      } catch (NumberFormatException e) {
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }
```

Toute la construction `if-else` peut enfin être supprimée.

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
      m.set(currentArgument);
      return true;
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
  }
```

Je peux à présent me débarrasser de quelques fonctions inélégantes dans `Integer ArgumentMarshaler` et procéder à un petit nettoyage.

```
  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0

    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        intValue = Integer.parseInt(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_INTEGER;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_INTEGER;
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }
```

Je peux également transformer `ArgumentMarshaler` en une interface.

```
private interface ArgumentMarshaler {
  void set(Iterator<String> currentArgument) throws ArgsException;
  Object get();
}
```

Voyons s'il est facile d'ajouter un nouveau type d'argument dans la structure. Cette opération ne doit demander que quelques modifications isolées. Tout d'abord, j'ajoute un nouveau cas de test qui vérifie que l'argument `double` est correctement pris en charge.

```
public void testSimpleDoublePresent() throws Exception {
  Args args = new Args("x##", new String[] {"-x","42.3"});
  assertTrue(args.isValid());
  assertEquals(1, args.cardinality());
  assertTrue(args.has('x'));
  assertEquals(42.3, args.getDouble('x'), .001);
}
```

Je peux ensuite nettoyer le code d'analyse du schéma et ajouter la détection de `##` qui correspond à un argument de type `double`.

```
private void parseSchemaElement(String element) throws ParseException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (elementTail.length() == 0)
    marshalers.put(elementId, new BooleanArgumentMarshaler());
  else if (elementTail.equals("*"))
    marshalers.put(elementId, new StringArgumentMarshaler());
  else if (elementTail.equals("#"))
    marshalers.put(elementId, new IntegerArgumentMarshaler());
  else if (elementTail.equals("##"))
    marshalers.put(elementId, new DoubleArgumentMarshaler());
  else
    throw new ParseException(String.format(
      "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
}
```

Puis j'écris la classe `DoubleArgumentMarshaler`.

```
private class DoubleArgumentMarshaler implements ArgumentMarshaler {
  private double doubleValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      doubleValue = Double.parseDouble(parameter);
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_DOUBLE;
      throw new ArgsException();
    } catch (NumberFormatException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_DOUBLE;
      throw new ArgsException();
    }
  }
```

```
    public Object get() {
      return doubleValue;
    }
  }
```

Elle m'oblige à ajouter de nouveaux codes d'erreur.

```
  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
    MISSING_DOUBLE, INVALID_DOUBLE}
```

J'ai également besoin d'une fonction `getDouble`.

```
  public double getDouble(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Double) am.get();
    } catch (Exception e) {
      return 0.0;
    }
  }
```

Tous les tests réussissent ! Cet ajout a pratiquement été indolore. Assurons-nous que le traitement des erreurs fonctionne correctement. Le cas de test suivant vérifie qu'une erreur est déclenchée lorsqu'une chaîne invalide est passée à un argument `##`.

```
  public void testInvalidDouble() throws Exception {
    Args args = new Args("x##", new String[] {"-x","Forty two"});
    assertFalse(args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0, args.getInt('x'));
    assertEquals("Argument -x expects a double but was 'Forty two'.",
                 args.errorMessage());
  }

---


  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Should not get here.");
      case UNEXPECTED_ARGUMENT:
        return unexpectedArgumentMessage();
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
                             errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
                             errorArgumentId);
      case INVALID_DOUBLE:
        return String.format("Argument -%c expects a double but was '%s'.",
                             errorArgumentId, errorParameter);
```

```
      case MISSING_DOUBLE:
        return String.format("Could not find double parameter for -%c.",
                             errorArgumentId);
    }
    return "";
  }
```

Les tests réussissent. Le test suivant vérifie que l'absence d'un argument `double` est correctement détectée.

```
  public void testMissingDouble() throws Exception {
    Args args = new Args("x##", new String[]{"-x"});
    assertFalse(args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0.0, args.getDouble('x'), 0.01);
    assertEquals("Could not find double parameter for -x.",
                 args.errorMessage());
  }
```

Comme prévu, ce test réussit. Je l'ai écrit uniquement pour une question d'exhaustivité.

Le code de gestion des exceptions est plutôt laid et ne devrait pas réellement se trouver dans la classe `Args`. Le code lance également une exception `ParseException` qui ne nous appartient pas vraiment. Par conséquent, je vais fusionner toutes les exceptions dans une seule classe `ArgsException` qui sera placée dans son propre module.

```
public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}


  public enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
    MISSING_DOUBLE, INVALID_DOUBLE}
}

---

public class Args {
  ...
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ArgsException.ErrorCode errorCode = ArgsException.ErrorCode.OK;
  private List<String> argsList;

  public Args(String schema, String[] args) throws ArgsException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    valid = parse();
  }
```

```
private boolean parse() throws ArgsException {
  if (schema.length() == 0 && argsList.size() == 0)
    return true;
  parseSchema();
  try {
    parseArguments();
  } catch (ArgsException e) {
  }
  return valid;
}

private boolean parseSchema() throws ArgsException {
  ...
}

private void parseSchemaElement(String element) throws ArgsException {
  ...
  else
    throw new ArgsException(
      String.format("Argument: %c has invalid format: %s.",
                    elementId,elementTail));
}

private void validateSchemaElementId(char elementId) throws ArgsException {
  if (!Character.isLetter(elementId)) {
    throw new ArgsException(
      "Bad character:" + elementId + "in Args format: " + schema);
  }
}

...

private void parseElement(char argChar) throws ArgsException {
  if (setArgument(argChar))
    argsFound.add(argChar);
  else {
    unexpectedArguments.add(argChar);
    errorCode = ArgsException.ErrorCode.UNEXPECTED_ARGUMENT;
    valid = false;
  }
}

...

private class StringArgumentMarshaler implements ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
    try {
      stringValue = currentArgument.next();
    } catch (NoSuchElementException e) {
      errorCode = ArgsException.ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
```

```
      public Object get() {
        return stringValue;
      }
    }

    private class IntegerArgumentMarshaler implements ArgumentMarshaler {
      private int intValue = 0;

      public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
          parameter = currentArgument.next();
          intValue = Integer.parseInt(parameter);
        } catch (NoSuchElementException e) {
          errorCode = ArgsException.ErrorCode.MISSING_INTEGER;
          throw new ArgsException();
        } catch (NumberFormatException e) {
          errorParameter = parameter;
          errorCode = ArgsException.ErrorCode.INVALID_INTEGER;
          throw new ArgsException();
        }
      }

      public Object get() {
        return intValue;
      }
    }

    private class DoubleArgumentMarshaler implements ArgumentMarshaler {
      private double doubleValue = 0;

      public void set(Iterator<String> currentArgument) throws ArgsException {
        String parameter = null;
        try {
          parameter = currentArgument.next();
          doubleValue = Double.parseDouble(parameter);
        } catch (NoSuchElementException e) {
          errorCode = ArgsException.ErrorCode.MISSING_DOUBLE;
          throw new ArgsException();
        } catch (NumberFormatException e) {
          errorParameter = parameter;
          errorCode = ArgsException.ErrorCode.INVALID_DOUBLE;
          throw new ArgsException();
        }
      }

      public Object get() {
        return doubleValue;
      }
    }
  }
```

C'est parfait. À présent, `Args` lance uniquement des exceptions `ArgsException`. En plaçant `ArgsException` dans son propre module, je peux déplacer une grande partie du code de traitement des erreurs dans ce module et le sortir de celui de `Args`. Je dispose

ainsi d'un endroit naturel et évident où placer tout ce code, ce qui m'aidera à nettoyer plus avant le module `Args`.

Le code des exceptions et des erreurs peut désormais être totalement séparé du module `Args` (voir Listings 14.13 à 14.16). Pour cela, il aura fallu environ trente petites étapes, sans oublier de vérifier entre chaque que les tests réussissent.

Listing 14.13 : `ArgsTest.java`

```
package com.objectmentor.utilities.args;

import junit.framework.TestCase;

public class ArgsTest extends TestCase {
  public void testCreateWithNoSchemaOrArguments() throws Exception {
    Args args = new Args("", new String[0]);
    assertEquals(0, args.cardinality());
  }

  public void testWithNoSchemaButWithOneArgument() throws Exception {
    try {
      new Args("", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                   e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }

  public void testWithNoSchemaButWithMultipleArguments() throws Exception {
    try {
      new Args("", new String[]{"-x", "-y"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                   e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }

  }

  public void testNonLetterSchema() throws Exception {
    try {
      new Args("*", new String[]{});
      fail("Args constructor should have thrown exception");
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                   e.getErrorCode());
      assertEquals('*', e.getErrorArgumentId());
    }
  }

  public void testInvalidArgumentFormat() throws Exception {
    try {
```

```
      new Args("f~", new String[]{});
      fail("Args constructor should have throws exception");
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_FORMAT, e.getErrorCode());
      assertEquals('f', e.getErrorArgumentId());
    }
  }

  public void testSimpleBooleanPresent() throws Exception {
    Args args = new Args("x", new String[]{"-x"});
    assertEquals(1, args.cardinality());
    assertEquals(true, args.getBoolean('x'));
  }

  public void testSimpleStringPresent() throws Exception {
    Args args = new Args("x*", new String[]{"-x", "param"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals("param", args.getString('x'));
  }

  public void testMissingStringArgument() throws Exception {
    try {
      new Args("x*", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.MISSING_STRING, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }

  public void testSpacesInFormat() throws Exception {
    Args args = new Args("x, y", new String[]{"-xy"});
    assertEquals(2, args.cardinality());
    assertTrue(args.has('x'));
    assertTrue(args.has('y'));
  }

  public void testSimpleIntPresent() throws Exception {
    Args args = new Args("x#", new String[]{"-x", "42"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42, args.getInt('x'));
  }

  public void testInvalidInteger() throws Exception {
    try {
      new Args("x#", new String[]{"-x", "Forty two"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_INTEGER, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
      assertEquals("Forty two", e.getErrorParameter());
    }

  }
```

```
  public void testMissingInteger() throws Exception {
    try {
      new Args("x#", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.MISSING_INTEGER, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }

  public void testSimpleDoublePresent() throws Exception {
    Args args = new Args("x##", new String[]{"-x", "42.3"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42.3, args.getDouble('x'), .001);
  }

  public void testInvalidDouble() throws Exception {
    try {
      new Args("x##", new String[]{"-x", "Forty two"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_DOUBLE, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
      assertEquals("Forty two", e.getErrorParameter());
    }
  }

  public void testMissingDouble() throws Exception {
    try {
      new Args("x##", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.MISSING_DOUBLE, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }
}
```

Listing 14.14 : `ArgsExceptionTest.java`

```
public class ArgsExceptionTest extends TestCase {
  public void testUnexpectedMessage() throws Exception {
    ArgsException e =
      new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                        'x', null);
    assertEquals("Argument -x unexpected.", e.errorMessage());
  }

  public void testMissingStringMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_STRING,
                                        'x', null);
    assertEquals("Could not find string parameter for -x.", e.errorMessage());
  }
```

```
  public void testInvalidIntegerMessage() throws Exception {
    ArgsException e =
      new ArgsException(ArgsException.ErrorCode.INVALID_INTEGER,
                        'x', "Forty two");
    assertEquals("Argument -x expects an integer but was 'Forty two'.",
                 e.errorMessage());
  }

  public void testMissingIntegerMessage() throws Exception {
    ArgsException e =
      new ArgsException(ArgsException.ErrorCode.MISSING_INTEGER, 'x', null);
    assertEquals("Could not find integer parameter for -x.", e.errorMessage());
  }

  public void testInvalidDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.INVALID_DOUBLE,
                                        'x', "Forty two");
    assertEquals("Argument -x expects a double but was 'Forty two'.",
                 e.errorMessage());
  }

  public void testMissingDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_DOUBLE,
                                        'x', null);
    assertEquals("Could not find double parameter for -x.", e.errorMessage());
  }
}
```

Listing 14.15 : `ArgsException.java`

```
public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public ArgsException(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public ArgsException(ErrorCode errorCode, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
  }

  public ArgsException(ErrorCode errorCode, char errorArgumentId,
                       String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
  }
```

```
  public char getErrorArgumentId() {
    return errorArgumentId;
  }

  public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
  }

  public String getErrorParameter() {
    return errorParameter;
  }

  public void setErrorParameter(String errorParameter) {
    this.errorParameter = errorParameter;
  }

  public ErrorCode getErrorCode() {
    return errorCode;
  }

  public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Should not get here.");
      case UNEXPECTED_ARGUMENT:
        return String.format("Argument -%c unexpected.", errorArgumentId);
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
                             errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
                             errorArgumentId);
      case INVALID_DOUBLE:
        return String.format("Argument -%c expects a double but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_DOUBLE:
        return String.format("Could not find double parameter for -%c.",
                             errorArgumentId);
    }
    return "";
  }

  public enum ErrorCode {
    OK, INVALID_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE}
}
```

Listing 14.16 : `Args.java`

```
public class Args {
  private String schema;
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private Iterator<String> currentArgument;
  private List<String> argsList;

  public Args(String schema, String[] args) throws ArgsException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    parse();
  }

  private void parse() throws ArgsException {
    parseSchema();
    parseArguments();
  }

  private boolean parseSchema() throws ArgsException {
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        parseSchemaElement(element.trim());
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ArgsException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
      marshalers.put(elementId, new DoubleArgumentMarshaler());
    else
      throw new ArgsException(ArgsException.ErrorCode.INVALID_FORMAT,
                              elementId, elementTail);
  }

  private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId)) {
      throw new ArgsException(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                              elementId, null);
    }
  }

  private void parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
```

```
    String arg = currentArgument.next();
    parseArgument(arg);
  }
}

private void parseArgument(String arg) throws ArgsException {
  if (arg.startsWith("-"))
    parseElements(arg);
}

private void parseElements(String arg) throws ArgsException {
  for (int i = 1; i < arg.length(); i++)
    parseElement(arg.charAt(i));
}

private void parseElement(char argChar) throws ArgsException {
  if (setArgument(argChar))
    argsFound.add(argChar);
  else {
    throw new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                            argChar, null);
  }
}

private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    m.set(currentArgument);
    return true;
  } catch (ArgsException e) {
    e.setErrorArgumentId(argChar);
    throw e;
  }
}

public int cardinality() {
  return argsFound.size();
}

public String usage() {
  if (schema.length() > 0)
    return "-[" + schema + "]";
  else
    return "";
}

public boolean getBoolean(char arg) {
  ArgumentMarshaler am = marshalers.get(arg);
  boolean b = false;
  try {
    b = am != null && (Boolean) am.get();
  } catch (ClassCastException e) {
    b = false;
  }
  return b;
}
```

```
  public String getString(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
      return "";
    }
  }

  public int getInt(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
      return 0;
    }
  }


  public double getDouble(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Double) am.get();
    } catch (Exception e) {
      return 0.0;
    }
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }
}
```

La plupart des modifications apportées à la classe `Args` sont des suppressions. Une grande partie du code est simplement déplacée de `Args` vers `ArgsException`. Parfait. Tous les `ArgumentMarshaller` sont également transférés dans leurs propres fichiers. Encore mieux !

Une bonne conception d'un logiciel se fonde énormément sur le partitionnement, c'est-à-dire créer des endroits appropriés où placer les différentes sortes de code. Cette séparation des préoccupations permet d'obtenir un code plus simple à comprendre et à maintenir.

La méthode `errorMessage` de `ArgsException` constitue un bon exemple. Elle transgressait manifestement le principe SRP pour placer la mise en forme du message d'erreur dans `Args`. `Args` ne doit s'occuper que du traitement des arguments, non du format des messages d'erreur. Toutefois, est-ce vraiment sensé de placer le code correspondant dans `ArgsException` ?

Il s'agit en réalité d'un compromis. Les utilisateurs qui n'aiment pas les messages d'erreur fournis par `ArgsException` écriront les leurs. Mais disposer de messages d'erreur déjà préparés n'est pas inintéressant.

À présent, il devrait être clair que la solution finale présentée au début de ce chapitre est à portée de mains. Je vous laisse en exercice les dernières transformations.

Conclusion

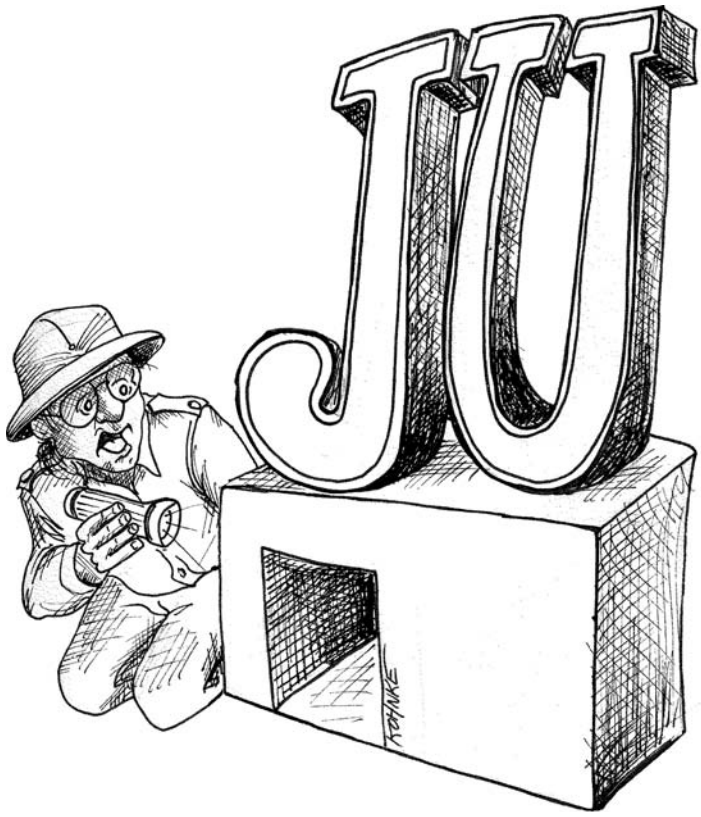
Il ne suffit pas que le code fonctionne. Un code opérationnel masque souvent des dysfonctionnements. Les programmeurs qui se satisfont d'un code simplement opérationnel ne sont pas de véritables professionnels. Ils craignent sans doute de ne pas avoir suffisamment de temps pour améliorer la structure et la conception de leur code, mais je ne suis pas d'accord. Rien n'a un effet aussi négatif sur un projet de développement que du mauvais code. Les plannings erronés peuvent être refaits, les exigences imprécises peuvent être réétudiées, une mauvaise dynamique d'équipe peut être reconstruite. En revanche, le mauvais code pourrit et fermente, puis se transforme en un fardeau qui mène l'équipe à sa perte. J'ai rencontré de nombreuses équipes réduites à progresser péniblement car, dans leur hâte, elles avaient créé un véritable tas de code malveillant qui avait pris les rênes de leur destinée.

Le mauvais code peut évidemment être nettoyé. Toutefois, cette opération est très onéreuse. Plus le code se dégrade, plus les modules s'immiscent les uns dans les autres, en créant de nombreuses dépendances cachées et embrouillées. Retrouver et rompre les anciennes dépendances est une tâche longue et ardue. À l'opposé, conserver un code propre se révèle relativement facile. Si, le matin, vous mettez la pagaille dans un module, il est facile de le nettoyer dans l'après-midi. Mieux encore, si vous avez créé un désordre cinq minutes plus tôt, il est très facile de tout ranger immédiatement.

La solution consiste donc à conserver en permanence un code aussi propre et aussi simple que possible. Ne laissez jamais la gangrène s'installer.

15

Au cœur de JUnit



JUnit fait partie des frameworks Java les plus connus. Sa conception est simple, sa définition, précise et son implémentation, élégante. Mais à quoi ressemble son code ? Dans ce chapitre, nous allons critiquer un exemple tout droit sorti du framework JUnit.

Le framework JUnit

Plusieurs personnes ont travaillé sur JUnit, mais le projet a démarré avec Kent Beck et Eric Gamma dans un avion à destination d'Atlanta. Kent souhaitait apprendre Java et Eric voulait en savoir plus sur le framework de test pour Smalltalk de Kent. "Quoi de plus naturel pour deux geeks enfermés dans un lieu exigu que de sortir leur portable et de commencer à coder ?"[1] Après trois heures de travail à haute altitude, ils avaient posé les bases de JUnit.

Le module que nous allons examiner correspond au code astucieux qui facilite l'identification des erreurs dans la comparaison de chaînes de caractères. Il se nomme `ComparisonCompactor`. Étant donné deux chaînes différentes, comme `ABCDE` et `ABXDE`, il révèle leurs différences en générant une chaîne de la forme `<...B[X]D...>`.

Je pourrais l'expliquer plus en détail, mais les cas de test le font mieux que moi. Examinez le Listing 15.1 et vous comprendrez parfaitement les exigences de ce module. Pendant que vous y êtes, critiquez la structure des tests. Pourraient-ils être plus simples ou plus évidents ?

Listing 15.1 : `ComparisonCompactorTest.java`

```
package junit.tests.framework;

import junit.framework.ComparisonCompactor;
import junit.framework.TestCase;

public class ComparisonCompactorTest extends TestCase {

  public void testMessage() {
    String failure= new ComparisonCompactor(0, "b", "c").compact("a");
    assertTrue("a expected:<[b]> but was:<[c]>".equals(failure));
  }

  public void testStartSame() {
    String failure= new ComparisonCompactor(1, "ba", "bc").compact(null);
    assertEquals("expected:<b[a]> but was:<b[c]>", failure);
  }

  public void testEndSame() {
    String  failure= new ComparisonCompactor(1, "ab", "cb").compact(null);
    assertEquals("expected:<[a]b> but was:<[c]b>", failure);
  }

  public void testSame() {
    String failure= new ComparisonCompactor(1, "ab", "ab").compact(null);
    assertEquals("expected:<ab> but was:<ab>", failure);
  }
```

1. *JUnit Pocket Guide*, Kent Beck, O'Reilly, 2004, page 43.

```
public void testNoContextStartAndEndSame() {
  String failure= new ComparisonCompactor(0, "abc", "adc").compact(null);
  assertEquals("expected:<...[b]...> but was:<...[d]...>", failure);
}

public void testStartAndEndContext() {
  String failure= new ComparisonCompactor(1, "abc", "adc").compact(null);
  assertEquals("expected:<a[b]c> but was:<a[d]c>", failure);
}

public void testStartAndEndContextWithEllipses() {
  String failure=
    new ComparisonCompactor(1, "abcde", "abfde").compact(null);
  assertEquals("expected:<...b[c]d...> but was:<...b[f]d...>", failure);
}

public void testComparisonErrorStartSameComplete() {
  String failure= new ComparisonCompactor(2, "ab", "abc").compact(null);
  assertEquals("expected:<ab[]> but was:<ab[c]>", failure);
}

public void testComparisonErrorEndSameComplete() {
  String failure= new ComparisonCompactor(0, "bc", "abc").compact(null);
  assertEquals("expected:<[]...> but was:<[a]...>", failure);
}

public void testComparisonErrorEndSameCompleteContext() {
  String failure= new ComparisonCompactor(2, "bc", "abc").compact(null);
  assertEquals("expected:<[]bc> but was:<[a]bc>", failure);
}

public void testComparisonErrorOverlapingMatches() {
  String failure= new ComparisonCompactor(0, "abc", "abbc").compact(null);
  assertEquals("expected:<...[]...> but was:<...[b]...>", failure);
}

public void testComparisonErrorOverlapingMatchesContext() {
  String failure= new ComparisonCompactor(2, "abc", "abbc").compact(null);
  assertEquals("expected:<ab[]c> but was:<ab[b]c>", failure);
}

public void testComparisonErrorOverlapingMatches2() {
  String failure=
    new ComparisonCompactor(0, "abcdde", "abcde").compact(null);
  assertEquals("expected:<...[d]...> but was:<...[]...>", failure);
}

public void testComparisonErrorOverlapingMatches2Context() {
  String failure=
    new ComparisonCompactor(2, "abcdde", "abcde").compact(null);
  assertEquals("expected:<...cd[d]e> but was:<...cd[]e>", failure);
}

public void testComparisonErrorWithActualNull() {
  String failure= new ComparisonCompactor(0, "a", null).compact(null);
  assertEquals("expected:<a> but was:<null>", failure);
}
```

```
  public void testComparisonErrorWithActualNullContext() {
    String failure= new ComparisonCompactor(2, "a", null).compact(null);
    assertEquals("expected:<a> but was:<null>", failure);
  }

  public void testComparisonErrorWithExpectedNull() {
    String failure= new ComparisonCompactor(0, null, "a").compact(null);
    assertEquals("expected:<null> but was:<a>", failure);
  }

  public void testComparisonErrorWithExpectedNullContext() {
    String failure= new ComparisonCompactor(2, null, "a").compact(null);
    assertEquals("expected:<null> but was:<a>", failure);
  }

  public void testBug609972() {
    String failure= new ComparisonCompactor(10, "S&P500", "0").compact(null);
    assertEquals("expected:<[S&P50]0> but was:<[]0>", failure);
  }
}
```

À partir de ces tests, j'ai lancé une analyse de couverture de code sur `ComparisonCompactor`. Le code est couvert à 100 %. Chaque ligne de code, chaque instruction `if` et chaque boucle `for` est exécutée par les tests. J'ai ainsi un haut niveau de confiance dans le fonctionnement du code et un grand respect pour le travail des auteurs.

Le code de `ComparisonCompactor` est donné au Listing 15.2. Prenez le temps de le parcourir. Je pense que vous le trouverez parfaitement partitionné, raisonnablement expressif et structurellement simple. Ensuite, nous nous amuserons à chercher la petite bête.

Listing 15.2 : `ComparisonCompactor.java` **(original)**

```
package junit.framework;

public class ComparisonCompactor {

  private static final String ELLIPSIS = "...";
  private static final String DELTA_END = "]";
  private static final String DELTA_START = "[";

  private int fContextLength;
  private String fExpected;
  private String fActual;
  private int fPrefix;
  private int fSuffix;

  public ComparisonCompactor(int contextLength,
                             String expected,
                             String actual) {
    fContextLength = contextLength;
    fExpected = expected;
    fActual = actual;
  }
```

```
public String compact(String message) {
  if (fExpected == null || fActual == null || areStringsEqual())
    return Assert.format(message, fExpected, fActual);

  findCommonPrefix();
  findCommonSuffix();
  String expected = compactString(fExpected);
  String actual = compactString(fActual);
  return Assert.format(message, expected, actual);
}

private String compactString(String source) {
  String result = DELTA_START +
                  source.substring(fPrefix, source.length() -
                  fSuffix + 1) + DELTA_END;
  if (fPrefix > 0)
    result = computeCommonPrefix() + result;
  if (fSuffix > 0)
    result = result + computeCommonSuffix();
  return result;
}

private void findCommonPrefix() {
  fPrefix = 0;
  int end = Math.min(fExpected.length(), fActual.length());
  for (; fPrefix < end; fPrefix++) {
    if (fExpected.charAt(fPrefix) != fActual.charAt(fPrefix))
      break;
  }
}

private void findCommonSuffix() {
  int expectedSuffix = fExpected.length() - 1;
  int actualSuffix = fActual.length() - 1;
  for (;
       actualSuffix >= fPrefix && expectedSuffix >= fPrefix;
       actualSuffix--, expectedSuffix--) {
    if (fExpected.charAt(expectedSuffix) != fActual.charAt(actualSuffix))
      break;
  }
  fSuffix = fExpected.length() - expectedSuffix;
}

private String computeCommonPrefix() {
  return (fPrefix > fContextLength ? ELLIPSIS : "") +
          fExpected.substring(Math.max(0, fPrefix - fContextLength),
          fPrefix);
}

private String computeCommonSuffix() {
  int end = Math.min(fExpected.length() - fSuffix + 1 + fContextLength,
                     fExpected.length());
  return fExpected.substring(fExpected.length() - fSuffix + 1, end) +
         (fExpected.length() - fSuffix + 1 < fExpected.length() -
          fContextLength ? ELLIPSIS : "");
}
```

```
  private boolean areStringsEqual() {
    return fExpected.equals(fActual);
  }
}
```

Vous pourriez émettre quelques critiques sur ce module. Entre autres, certaines expressions sont longues et il contient quelques `+1` étranges. Mais, globalement, ce module est plutôt bon. En effet, il aurait pu ressembler à celui du Listing 15.3.

Listing 15.3 : `ComparisonCompactor.java` **(version sale)**

```
package junit.framework;

public class ComparisonCompactor {
  private int ctxt;
  private String s1;
  private String s2;
  private int pfx;
  private int sfx;

  public ComparisonCompactor(int ctxt, String s1, String s2) {
    this.ctxt = ctxt;
    this.s1 = s1;
    this.s2 = s2;
  }

  public String compact(String msg) {
    if (s1 == null || s2 == null || s1.equals(s2))
      return Assert.format(msg, s1, s2);

    pfx = 0;
    for (; pfx < Math.min(s1.length(), s2.length()); pfx++) {
      if (s1.charAt(pfx) != s2.charAt(pfx))
        break;
    }
    int sfx1 = s1.length() - 1;
    int sfx2 = s2.length() - 1;
    for (; sfx2 >= pfx && sfx1 >= pfx; sfx2--, sfx1--) {
      if (s1.charAt(sfx1) != s2.charAt(sfx2))
        break;
    }
    sfx = s1.length() - sfx1;
    String cmp1 = compactString(s1);
    String cmp2 = compactString(s2);
    return Assert.format(msg, cmp1, cmp2);
  }

  private String compactString(String s) {
    String result =
      "[" + s.substring(pfx, s.length() - sfx + 1) + "]";
    if (pfx > 0)
      result = (pfx > ctxt ? "..." : "") +
        s1.substring(Math.max(0, pfx - ctxt), pfx) + result;
    if (sfx > 0) {
      int end = Math.min(s1.length() - sfx + 1 + ctxt, s1.length());
```

```
        result = result + (s1.substring(s1.length() - sfx + 1, end) +
          (s1.length() - sfx + 1 < s1.length() - ctxt ? "..." : ""));
      }
      return result;
    }

}
```

Même si les auteurs ont laissé ce module dans un bon état de propreté, la règle du boy-scout[2] nous demande de le laisser plus propre que nous l'avons trouvé. Alors, comment pouvons-nous améliorer le code du Listing 15.2 ?

La première chose qui me déplaît est le préfixe `f` ajouté aux variables membres [N6]. Grâce aux environnements de développement actuels, ce codage de la portée est redondant. Par conséquent, supprimons tous les préfixes `f`.

```
private int contextLength;
private String expected;
private String actual;
private int prefix;
private int suffix;
```

Ensuite, le début de la fonction `compact` contient une instruction conditionnelle non encapsulée [G28].

```
public String compact(String message) {
  if (expected == null || actual == null || areStringsEqual())
    return Assert.format(message, expected, actual);

  findCommonPrefix();
  findCommonSuffix();
  String expected = compactString(this.expected);
  String actual = compactString(this.actual);
  return Assert.format(message, expected, actual);
}
```

Pour que les intentions du code soient claires, toutes ces conditions doivent être encapsulées. Nous créons donc une méthode qui corrige ce problème.

```
public String compact(String message) {
  if (shouldNotCompact())
    return Assert.format(message, expected, actual);

  findCommonPrefix();
  findCommonSuffix();
  String expected = compactString(this.expected);
  String actual = compactString(this.actual);
  return Assert.format(message, expected, actual);
}

private boolean shouldNotCompact() {
  return expected == null || actual == null || areStringsEqual();
}
```

2. Voir la section "La règle du boy-scout" au Chapitre 1.

Je n'aime pas vraiment la notation `this.expected` et `this.actual` dans la fonction `compact`. Elle provient du changement de nom de `fExpected` en `expected`. Pourquoi existe-t-il dans cette fonction des variables qui ont les mêmes noms que les variables membres ? S'agit-il d'une autre notion [N4] ? Nous devons choisir des noms dépourvus de toute ambiguïté.

```
String compactExpected = compactString(expected);
String compactActual = compactString(actual);
```

Les formes négatives sont un peu moins faciles à comprendre que les formes positives [G29]. Inversons donc l'instruction `if` et retournons le sens de la conditionnelle.

```
public String compact(String message) {
  if (canBeCompacted()) {
    findCommonPrefix();
    findCommonSuffix();
    String compactExpected = compactString(expected);
    String compactActual = compactString(actual);
    return Assert.format(message, compactExpected, compactActual);
  } else {
    return Assert.format(message, expected, actual);
  }
}

private boolean canBeCompacted() {
  return expected != null && actual != null && !areStringsEqual();
}
```

Le nom de la fonction est étrange [N7]. Si elle compacte les chaînes de caractères, en réalité elle peut ne pas les compacter si `canBeCompacted` retourne `false`. En nommant cette fonction `compact`, nous cachons l'effet secondaire du contrôle des erreurs. Notez également que la fonction retourne un message formaté, pas seulement les chaînes compactées. Elle devrait donc se nommer `formatCompactedComparison`. La lecture du code est ainsi beaucoup plus aisée lorsque l'argument de la fonction est ajouté :

```
public String formatCompactedComparison(String message) {
```

C'est dans le corps de l'instruction `if` que se passe le véritable compactage des chaînes de caractères attendue et réelle. Nous devons donc le transformer en une méthode nommée `compactExpectedAndActual`. Toutefois, nous souhaitons que la fonction `formatCompactedComparison` prenne en charge le formatage. La fonction `compact...` ne doit rien faire d'autre que le compactage [G30]. En conséquence, nous devons la découper :

```
...
private String compactExpected;
private String compactActual;

...

public String formatCompactedComparison(String message) {
  if (canBeCompacted()) {
```

```
      compactExpectedAndActual();
      return Assert.format(message, compactExpected, compactActual);
    } else {
      return Assert.format(message, expected, actual);
    }
  }

  private void compactExpectedAndActual() {
    findCommonPrefix();
    findCommonSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
  }
```

Cette transformation est nécessaire pour que `compactExpected` et `compactActual` deviennent des variables membres. Le fait que les deux dernières lignes de la nouvelle fonction retournent des variables, contrairement aux deux premières, me déplaît. Les conventions employées ne sont pas cohérentes [G11]. Nous devons donc modifier `findCommonPrefix` et `findCommonSuffix` pour qu'elles retournent les valeurs de préfixe et de suffixe.

```
  private void compactExpectedAndActual() {
    prefixIndex = findCommonPrefix();
    suffixIndex = findCommonSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
  }

  private int findCommonPrefix() {
    int prefixIndex = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixIndex < end; prefixIndex++) {
      if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
        break;
    }
    return prefixIndex;
  }

  private int findCommonSuffix() {
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
         actualSuffix--, expectedSuffix--) {
      if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
        break;
    }
    return expected.length() - expectedSuffix;
  }
```

Nous devons également modifier les noms des variables membres pour qu'ils soient plus précis [N1]. En effet, elles sont toutes deux des indices.

Un examen attentif de `findCommonSuffix` révèle un *couplage temporel caché* [G31] : cette méthode dépend du calcul de `prefixIndex` par `findCommonPrefix`. Si ces deux

fonctions sont appelées dans le désordre, une session de débogage difficile nous attend. Pour exposer ce couplage temporel, modifions `findCommonSuffix` afin qu'elle prenne `prefixIndex` en argument.

```
private void compactExpectedAndActual() {
  prefixIndex = findCommonPrefix();
  suffixIndex = findCommonSuffix(prefixIndex);
  compactExpected = compactString(expected);
  compactActual = compactString(actual);
}

private int findCommonSuffix(int prefixIndex) {
  int expectedSuffix = expected.length() - 1;
  int actualSuffix = actual.length() - 1;
  for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
       actualSuffix--, expectedSuffix--) {
    if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
      break;
  }
  return expected.length() - expectedSuffix;
}
```

Cela ne me convient pas vraiment. Passer l'argument `prefixIndex` est quelque peu arbitraire [G32]. Il sert à établir l'ordre, mais n'en explique pas les raisons. Un autre programmeur pourrait annuler notre modification car rien n'indique que ce paramètre est réellement nécessaire. Prenons donc une tactique différente.

```
private void compactExpectedAndActual() {
  findCommonPrefixAndSuffix();
  compactExpected = compactString(expected);
  compactActual = compactString(actual);
}

private void findCommonPrefixAndSuffix() {
  findCommonPrefix();
  int expectedSuffix = expected.length() - 1;
  int actualSuffix = actual.length() - 1;
  for (;
       actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
       actualSuffix--, expectedSuffix--
    ) {
    if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
      break;
  }
  suffixIndex = expected.length() - expectedSuffix;
}

private void findCommonPrefix() {
  prefixIndex = 0;
  int end = Math.min(expected.length(), actual.length());
  for (; prefixIndex < end; prefixIndex++)
    if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
      break;
}
```

Nous remettons `findCommonPrefix` et `findCommonSuffix` sous leur ancienne version, en changeant le nom de `findCommonSuffix` en `findCommonPrefixAndSuffix` et en lui faisant invoquer `findCommonPrefix` avant toute autre opération. La dépendance temporelle des deux fonctions est ainsi établie de manière beaucoup plus fiable que précédemment. Cela révèle également la laideur de `findCommonPrefixAndSuffix`. Procédons à présent à son nettoyage.

```
private void findCommonPrefixAndSuffix() {
  findCommonPrefix();
  int suffixLength = 1;
  for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
    if (charFromEnd(expected, suffixLength) !=
        charFromEnd(actual, suffixLength))
      break;
  }
  suffixIndex = suffixLength;
}

private char charFromEnd(String s, int i) {
  return s.charAt(s.length()-i);}

private boolean suffixOverlapsPrefix(int suffixLength) {
  return actual.length() - suffixLength < prefixLength ||
    expected.length() - suffixLength < prefixLength;
}
```

C'est beaucoup mieux. Nous constatons que `suffixIndex` correspond en réalité à la longueur du suffixe et que son nom n'est donc pas bien choisi. C'est également vrai pour `prefixIndex`, même si dans ce cas "indice" et "longueur" sont synonymes. Toutefois, il est plus cohérent d'utiliser "longueur". Le problème est que la variable `suffixIndex` ne commence pas à zéro ; elle commence à 1 et n'est donc pas une véritable longueur. C'est la raison de la présence de tous ces `+1` dans `computeCommonSuffix` [G33]. Après correction de ce problème, nous obtenons le Listing 15.4.

Listing 15.4 : `ComparisonCompactor.java` **(version intermédiaire)**

```
public class ComparisonCompactor {
...
  private int suffixLength;
...
  private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    suffixLength = 0;
    for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
      if (charFromEnd(expected, suffixLength) !=
          charFromEnd(actual, suffixLength))
        break;
    }
  }
```

```
  private char charFromEnd(String s, int i) {
    return s.charAt(s.length() - i - 1);
  }

  private boolean suffixOverlapsPrefix(int suffixLength) {
    return actual.length() - suffixLength <= prefixLength ||
      expected.length() - suffixLength <= prefixLength;
  }

...
  private String compactString(String source) {
    String result =
      DELTA_START +
      source.substring(prefixLength, source.length() - suffixLength) +
      DELTA_END;
    if (prefixLength > 0)
      result = computeCommonPrefix() + result;
    if (suffixLength > 0)
      result = result + computeCommonSuffix();
    return result;
  }

...
  private String computeCommonSuffix() {
    int end = Math.min(expected.length() - suffixLength +
      contextLength, expected.length()
    );
    return
      expected.substring(expected.length() - suffixLength, end) +
      (expected.length() - suffixLength <
        expected.length() - contextLength ?
        ELLIPSIS : "");
  }
```

Nous avons remplacé les `+1` dans `computeCommonSuffix` par un `1` dans `charFromEnd`, où il est parfaitement sensé, et par deux opérateurs `<=` dans `suffixOverlapsPrefix`, où ils sont parfaitement sensés. Cela nous permet de changer le nom de `suffixIndex` en `suffixLength` et d'améliorer énormément la lisibilité du code.

Il reste cependant un problème. Pendant que je supprimais les `+1`, j'ai remarqué la ligne suivante dans `compactString` :

```
if (suffixLength > 0)
```

Examinez-la dans le Listing 15.4. En principe, puisque la valeur de `suffixLength` est inférieure de 1 à ce qu'elle était auparavant, je devrais changer l'opérateur `>` en `>=`. Mais cela n'a pas de sens. Je comprends *à présent* ce que cela signifie ! C'était probablement un bogue. En réalité, pas vraiment un bogue. Par une analyse plus détaillée, nous voyons que l'instruction `if` évite qu'un suffixe de longueur zéro soit ajouté. Avant notre modification, l'instruction `if` ne servait à rien car `suffixIndex` ne pouvait jamais être inférieur à 1 !

Cela nous amène à nous interroger sur les *deux* instructions `if` dans `compactString` ! Il semblerait que nous puissions les supprimer. Nous les mettons donc en commentaire et exécutons les tests. Ils réussissent ! Nous pouvons ainsi restructurer `compactString` afin d'éliminer des instructions `if` superflues et de simplifier la fonction [G9].

```
private String compactString(String source) {
    return
      computeCommonPrefix() +
      DELTA_START +
      source.substring(prefixLength, source.length() - suffixLength) +
      DELTA_END +
      computeCommonSuffix();
}
```

C'est beaucoup mieux ! Nous voyons à présent que la fonction `compactString` combine simplement des fragments. Nous pourrions probablement rendre le code encore plus propre. De nombreux petits points peuvent être améliorés. Mais, au lieu de tous les détailler, je vous montre simplement le résultat dans le Listing 15.5.

Listing 15.5 : `ComparisonCompactor.java` **(version finale)**

```
package junit.framework;

public class ComparisonCompactor {

  private static final String ELLIPSIS = "...";
  private static final String DELTA_END = "]";
  private static final String DELTA_START = "[";

  private int contextLength;
  private String expected;
  private String actual;
  private int prefixLength;
  private int suffixLength;

  public ComparisonCompactor(
    int contextLength, String expected, String actual
  ) {
    this.contextLength = contextLength;
    this.expected = expected;
    this.actual = actual;
  }

  public String formatCompactedComparison(String message) {
    String compactExpected = expected;
    String compactActual = actual;
    if (shouldBeCompacted()) {
      findCommonPrefixAndSuffix();
      compactExpected = compact(expected);
      compactActual = compact(actual);
    }
    return Assert.format(message, compactExpected, compactActual);
  }
```

```
private boolean shouldBeCompacted() {
  return !shouldNotBeCompacted();
}

private boolean shouldNotBeCompacted() {
  return expected == null ||
         actual == null ||
         expected.equals(actual);
}

private void findCommonPrefixAndSuffix() {
  findCommonPrefix();
  suffixLength = 0;
  for (; !suffixOverlapsPrefix(); suffixLength++) {
    if (charFromEnd(expected, suffixLength) !=
        charFromEnd(actual, suffixLength)
    )
      break;
  }
}

private char charFromEnd(String s, int i) {
  return s.charAt(s.length() - i - 1);
}

private boolean suffixOverlapsPrefix() {
  return actual.length() - suffixLength <= prefixLength ||
    expected.length() - suffixLength <= prefixLength;
}

private void findCommonPrefix() {
  prefixLength = 0;
  int end = Math.min(expected.length(), actual.length());
  for (; prefixLength < end; prefixLength++)
    if (expected.charAt(prefixLength) != actual.charAt(prefixLength))
      break;
}

private String compact(String s) {
  return new StringBuilder()
    .append(startingEllipsis())
    .append(startingContext())
    .append(DELTA_START)
    .append(delta(s))
    .append(DELTA_END)
    .append(endingContext())
    .append(endingEllipsis())
    .toString();
}

private String startingEllipsis() {
  return prefixLength > contextLength ? ELLIPSIS : "";
}

private String startingContext() {
  int contextStart = Math.max(0, prefixLength - contextLength);
```

```
      int contextEnd = prefixLength;
      return expected.substring(contextStart, contextEnd);
    }

    private String delta(String s) {
      int deltaStart = prefixLength;
      int deltaEnd = s.length() - suffixLength;
      return s.substring(deltaStart, deltaEnd);
    }

    private String endingContext() {
      int contextStart = expected.length() - suffixLength;
      int contextEnd =
        Math.min(contextStart + contextLength, expected.length());
      return expected.substring(contextStart, contextEnd);
    }

    private String endingEllipsis() {
      return (suffixLength > contextLength ? ELLIPSIS : "");
    }
  }
```

Le résultat est plutôt agréable. Le module est décomposé en un groupe de fonctions d'analyse et un groupe de fonctions de synthèse. Elles sont organisées de sorte que la définition de chacune apparaît juste après son utilisation. Les fonctions d'analyse se trouvent au début, les fonctions de synthèse, à la fin.

Si vous faites bien attention, vous remarquerez que je suis revenu sur plusieurs choix effectués précédemment dans ce chapitre. Par exemple, dans `formatCompactedComparison`, j'ai à nouveau incorporé des méthodes qui avaient été extraites, et j'ai modifié le sens de l'expression `shouldNotBeCompacted`. Cette manière de procéder est classique. Bien souvent, un remaniement conduit à un autre remaniement qui conduit à annuler le premier. Le remaniement est un processus itératif d'essais et d'erreurs qui converge inévitablement vers un résultat qui semble être le fruit d'un professionnel.

Conclusion

Nous avons donc respecté la règle du boy-scout. Nous avons laissé ce module plus propre que nous l'avons trouvé. Pourtant, il était déjà assez propre. Les auteurs avaient bien fait leur travail. Cependant, aucun module n'est à l'abri d'une amélioration, et nous avons tous la responsabilité de laisser le code dans un meilleur état.

16

Remaniement de SerialDate

Sur la page web **http://www.jfree.org/jcommon/index.php**, vous trouverez la bibliothèque JCommon. Elle contient un paquetage nommé `org.jfree.date`, dans lequel se trouve la classe `SerialDate`. Nous allons étudier cette classe.

David Gilbert est l'auteur de `SerialDate`. Il s'agit manifestement d'un programmeur expérimenté et compétent. Comme nous le verrons, il montre un haut degré de professionnalisme et de discipline dans son code. En fait, il s'agit d'un "bon code", mais que je vais mettre en pièces.

Je ne fais pas cela par méchanceté. Je ne pense pas être meilleur que David au point de prétendre avoir le droit de juger son code. Si vous rencontrez un jour du code que j'ai écrit, je suis certain que vous aurez de nombreuses critiques à formuler.

Je ne veux être ni méchant ni arrogant. Mon intervention se limite exclusivement à une révision professionnelle. Cela ne devrait poser un problème à aucun d'entre nous. Nous devrions même être heureux que quelqu'un se donne du mal à examiner notre code. Ce n'est qu'au travers de critiques comme celle-ci que nous pouvons apprendre. Les médecins le font, les pilotes le font, les hommes de loi le font. Nous, en tant que programmeurs, devons apprendre à le faire.

David n'est pas seulement un bon programmeur, il a également eu le courage et la bonté de donner gratuitement son code à la communauté. Il l'a offert au regard de tous. Il a invité tout le monde à l'employer et à l'examiner. Chapeau !

`SerialDate` (voir Listing B.1 à l'Annexe B) est une classe Java qui représente une date. Pourquoi donc écrire une classe qui représente une date alors que Java propose déjà `java.util.Date` et `java.util.Calendar`, ainsi que bien d'autres ? L'auteur a développé cette classe en réponse à un problème que j'ai moi-même souvent rencontré. Son commentaire Javadoc (ligne 67) l'explique très bien. Nous pouvons ergoter quant à ses intentions, mais j'ai déjà fait face à ce problème et j'accueille volontiers une classe qui manipule des dates, non des heures.

Premièrement, la rendre opérationnelle

La classe `SerialDateTests` propose quelques tests unitaires (voir Listing B.2 à l'Annexe B). Ils réussissent tous. Malheureusement, un rapide examen de leur contenu montre qu'ils ne vérifient pas l'ensemble du code [T1]. Par exemple, en recherchant les utilisations de la méthode `monthCodeToQuarter` (ligne 356), nous constatons qu'elle n'est jamais invoquée [F4]. Les tests unitaires ne la testent donc pas.

À l'aide de Clover, j'ai pu vérifier la couverture des tests unitaires. Cet outil indique qu'ils exécutent uniquement 91 des 185 instructions exécutables dans `SerialDate` (environ 50 %) [T2]. La carte de couverture ressemble donc à un patchwork, avec de grands morceaux de code non exécuté dispersés dans la classe.

Mon objectif était de comprendre totalement cette classe et de la remanier. Je ne pouvais pas procéder sans une meilleure couverture des tests. J'ai donc écrit ma propre suite indépendante de tests unitaires (voir Listing B.4 à l'Annexe B).

Si vous examinez ces tests, vous constaterez que nombre d'entre eux sont en commentaire. Ce sont les tests qui ont échoué. Ils représentent un comportement qui, à mon avis,

devrait être mis en œuvre par `SerialDate`. Au cours du remaniement de `SerialDate`, je vais donc faire en sorte que ces tests réussissent également.

Même en tenant compte des tests en commentaire, Clover rapporte que la nouvelle suite exécute 170 des 185 instructions exécutables (environ 92 %). C'est déjà bien, mais je pense pouvoir faire mieux.

Les premiers tests en commentaire (lignes 23–63) relèvent d'un excès de zèle de ma part. Le programme n'a pas été conçu pour passer ces tests, mais le comportement me semblait évident [G2]. Je ne sais pas vraiment pourquoi la méthode `testWeekdayCodeToString` a été écrite initialement, mais, en raison de sa présence, il semble évident qu'elle ne devrait pas être sensible à la casse. L'écriture de ces tests était simple [T3]. Les faire réussir l'était encore plus. J'ai simplement modifié les lignes 259 et 263 pour utiliser `equalsIgnoreCase`.

J'ai laissé en commentaire les tests des lignes 32 et 45 car je ne savais pas clairement si les abréviations "tues" (mardi) et "thurs" (jeudi) devaient être reconnues.

Les tests des lignes 153 et 154 ne réussissaient pas, alors que, clairement, cela aurait dû être le cas [G2]. Il était facile de corriger ce problème, ainsi que celui des tests de la ligne 163 à 213, en apportant les modifications suivantes à la fonction `stringToMonthCode`.

```
457         if ((result < 1) || (result > 12)) {
                result = -1;
458             for (int i = 0; i < monthNames.length; i++) {
459                 if (s.equalsIgnoreCase(shortMonthNames[i])) {
460                     result = i + 1;
461                     break;
462                 }
463                 if (s.equalsIgnoreCase(monthNames[i])) {
464                     result = i + 1;
465                     break;
466                 }
467             }
468         }
```

Le test en commentaire de la ligne 318 révèle un bogue de la méthode `getFollowingDayOfWeek` (ligne 672). Le 25 décembre 2004 était un samedi. Le samedi suivant était le 1er janvier 2005. Cependant, lors de l'exécution du test, nous constatons que `getFollowingDayOfWeek` retourne le 25 décembre comme étant le samedi qui suit le 25 décembre. Cette réponse est évidemment fausse [G3],[T1]. Ce problème, visible à la ligne 685, représente une erreur classique de condition aux limites [T5]. La ligne devrait être la suivante :

```
685         if (baseDOW >= targetWeekday) {
```

Il est intéressant de noter que cette fonction avait déjà fait l'objet d'une réparation. L'historique des modifications (ligne 43) indique que des bogues avaient été corrigés dans `getPreviousDayOfWeek`, `getFollowingDayOfWeek` et `getNearestDayOfWeek` [T6].

Le test unitaire `testGetNearestDayOfWeek` (ligne 329), qui vérifie le fonctionnement de la méthode `getNearestDayOfWeek` (ligne 705), n'était pas initialement aussi long et exhaustif. Je lui ai ajouté plusieurs cas de test car mes premiers ne passaient pas tous [T6]. Vous pouvez voir le motif de dysfonctionnement en regardant les cas de test en commentaire. Ce motif est révélateur [T7]. Il montre que l'algorithme échoue si le jour le plus proche se trouve dans le futur. Il existe manifestement une erreur aux conditions limites [T5].

Le motif de couverture des tests produite par Clover est également intéressant [T8]. La ligne 719 n'est jamais exécutée ! Autrement dit, l'instruction `if` de la ligne 718 est toujours fausse. En examinant le code, nous constatons qu'elle devrait être vraie. La variable `adjust` est toujours négative et ne peut donc pas être supérieure ou égale à quatre. L'algorithme est donc erroné. Voici la version correcte :

```
int delta = targetDOW - base.getDayOfWeek();
int positiveDelta = delta + 7;
int adjust = positiveDelta % 7;
if (adjust > 3)
  adjust -= 7;

return SerialDate.addDays(adjust, base);
```

Enfin, pour que les tests des lignes 417 et 429 réussissent, il suffit de lancer une exception `IllegalArgumentException` au lieu de retourner une chaîne d'erreur à partir de `weekInMonthToString` et `relativeToString`.

Grâce à ces modifications, tous les tests unitaires réussissent et je pense que la classe `SerialDate` est à présent opérationnelle. Il ne nous reste plus qu'à la remettre en ordre.

Puis la remettre en ordre

Nous allons parcourir la classe `SerialDate` du début à la fin, en l'améliorant au cours de notre balade. Même si le texte ne le mentionne pas, j'exécuterai tous les tests unitaires `JCommon`, y compris mes tests améliorés, après chaque modification apportée. Il faut être certain en permanence que chaque changement décrit fonctionne pour `JCommon`.

En commençant à la ligne 1, nous voyons tout un ensemble de commentaires, avec des informations de licence, des copyrights, des auteurs et un historique des modifications. J'admets que certains aspects légaux doivent être mentionnés et donc que les copyrights et les licences doivent rester. En revanche, l'historique des modifications est abandonné depuis les années 1960. Aujourd'hui, nous disposons pour cela d'outils de gestion du code source. Cet historique doit être supprimé [C1].

La liste des importations qui commence à la ligne 61 peut être raccourcie en utilisant `java.text.*` et `java.util.*` [J1].

La mise en forme HTML de la documentation Javadoc (ligne 67) me gêne énormément. Je n'aime pas avoir un fichier source qui mélange plusieurs langages. Ce commentaire est écrit en *quatre* langages : Java, français, Javadoc et HTML [G1]. En raison de tous les langages employés, il est difficile d'être simple. Par exemple, le positionnement des lignes 71 et 72 est perdu lorsque la documentation Javadoc est générée. Par ailleurs, qui souhaite voir des balises `<ul>` et `<li>` dans le code source ? Une meilleure stratégie consiste à entourer le commentaire global d'une balise `<pre>` afin que la mise en forme du code source soit conservée dans la documentation Javadoc[1].

La ligne 86 contient la déclaration de la classe. Pourquoi cette classe se nomme-t-elle `SerialDate` ? Quel est le sens du mot "serial" ? A-t-il été choisi parce que la classe dérive de `Serializable` ? Cela semble peu probable.

Je vais satisfaire votre curiosité. Je sais pourquoi, tout au moins je pense savoir pourquoi, le mot "serial" intervient dans le nom de la classe. L'indice se trouve dans les constantes `SERIAL LOWER BOUND` et `SERIAL UPPER BOUND` aux lignes 98 et 101. Le commentaire qui débute à la ligne 830 cache un indice encore meilleur. Cette classe se nomme `SerialDate` car elle est implémentée en utilisant un "numéro de série", qui se trouve être le nombre de jours depuis le 30 décembre 1899.

Cela me pose deux problèmes. Premièrement, le terme "numéro de série" n'est pas vraiment correct. Vous pourriez penser que je pinaille, mais l'implémentation correspond plus à un décalage relatif qu'à un numéro de série. Le terme "numéro de série" fait plus penser à une identification de produits qu'à des dates. Pour moi, ce nom n'est donc pas particulièrement descriptif [N1]. Un terme plus approprié pourrait être "ordinal".

Le second problème est plus important. Le nom `SerialDate` sous-entend une implémentation, alors que cette classe est abstraite. Il n'y a donc aucun besoin d'induire quoi que ce soit quant à l'implémentation, et il existe une bonne raison de masquer l'implémentation. Ce nom ne se trouve donc pas au bon niveau d'abstraction [N2]. À mon avis, cette classe devrait simplement se nommer `Date`.

Malheureusement, la bibliothèque Java propose déjà de nombreuses classes nommées `Date`. Ce n'est donc probablement pas un meilleur choix. Puisque cette classe concerne uniquement les jours, non les heures, elle pourrait se nommer `Day`. Toutefois, ce nom est également beaucoup employé en d'autres endroits. Finalement, `DayDate` me paraît un bon compromis.

---

1. Une solution encore meilleure serait que Javadoc présente tous les commentaires de manière préformatée afin qu'ils aient la même apparence dans le code et dans la documentation.

À partir de maintenant, j'emploierai le terme `DayDate`, mais vous ne devez pas oublier que les listings continuent à mentionner `SerialDate`.

Je comprends pourquoi `DayDate` dérive de `Comparable` et `Serializable`. En revanche, pourquoi dérive-t-elle de `MonthConstants` ? La classe `MonthConstants` (voir Listing B.3 à la page 397) n'est qu'un ensemble de constantes statiques finales qui définissent les mois. Hériter des classes qui contiennent des constantes est une vieille astuce que les programmeurs Java employaient afin d'éviter les expressions de la forme `MonthConstants.January`, mais c'est une mauvaise idée [J2]. `MonthConstants` devrait être une énumération.

```
public abstract class DayDate implements Comparable,
                                         Serializable {
  public static enum Month {
    JANUARY(1),
    FEBRUARY(2),
    MARCH(3),
    APRIL(4),
    MAY(5),
    JUNE(6),
    JULY(7),
    AUGUST(8),
    SEPTEMBER(9),
    OCTOBER(10),
    NOVEMBER(11),
    DECEMBER(12);

    Month(int index) {
      this.index = index;
    }

    public static Month make(int monthIndex) {
      for (Month m : Month.values()) {
        if (m.index == monthIndex)
          return m;
      }
      throw new IllegalArgumentException("Invalid month index " + monthIndex);
    }
    public final int index;
  }
```

La transformation de `MonthConstants` en un `enum` conduit à de nombreux changements dans la classe `DayDate` et dans toutes celles qui l'utilisent. Il m'a fallu une heure pour les appliquer. Cependant, toutes les fonctions qui utilisaient un `int` pour représenter un mois prennent à présent un énumérateur de `Month`. Cela signifie que nous pouvons supprimer la méthode `isValidMonthCode` (ligne 326) et tous les contrôles d'erreur sur les codes de mois, comme celui dans `monthCodeToQuarter` (ligne 356) [G5].

À la ligne 91, nous trouvons la variable `serialVersionUID`. Elle sert à contrôler le mécanisme de sérialisation. Si nous la modifions, toute instance de `DayDate` écrite avec

une ancienne version du logiciel ne sera plus lisible et conduira à une exception `InvalidClassException`. Si nous ne déclarons pas de variable `serialVersionUID`, le compilateur la génère automatiquement à notre place, mais elle sera différente chaque fois que nous modifierons le module. Je sais qu'il est souvent recommandé de gérer manuellement cette variable, mais un contrôle automatique de la sérialisation me semble beaucoup plus fiable [G4]. En effet, je préfère largement déboguer l'arrivée d'une exception `InvalidClassException` qu'un comportement étrange dû à un oubli de modification de la variable `serialVersionUID`. Je décide donc de supprimer cette variable, tout au moins pour le moment[2].

Le commentaire de la ligne 93 est, à mon avis, redondant. Les commentaires redondants ne sont que des sources de mensonges et de désinformation [C2]. Par conséquent, je le supprime, lui et tous les commentaires similaires.

Les commentaires des lignes 97 et 100 traitent de numéros de série, et j'ai déjà donné mon point de vue à ce sujet [C1]. Les variables qu'ils décrivent correspondent aux dates limites (la première et la dernière) que `DayDate` peut reconnaître. Tout cela pourrait être un peu plus clair [N1].

```
public static final int EARLIEST_DATE_ORDINAL = 2;     // 01/01/1900
public static final int LATEST_DATE_ORDINAL = 2958465; // 31/12/9999
```

Je ne comprends pas bien pourquoi `EARLIEST DATE ORDINAL` est égal à 2, non à 0. Le commentaire de la ligne 829 y fait allusion, en suggérant que cela vient de la manière dont les dates sont représentées dans Microsoft Excel. Des informations plus détaillées sont fournies dans la classe `SpreadsheetDate` dérivée de `DayDate` (voir Listing B.5 à la page 407). Le commentaire de la ligne 71 décrit parfaitement la question.

Mon souci vient du fait que le problème semble lié à l'implémentation de `SpreadsheetDate` et qu'il n'a aucun rapport avec `DayDate`. J'en conclus que `EARLIEST DATE ORDINAL` et `LATEST DATE ORDINAL` n'appartiennent pas réellement à `DayDate` et que ces constantes doivent être déplacées dans `SpreadsheetDate` [G6].

Grâce à une petite recherche, nous déterminons que ces variables ne sont évidemment employées que dans `SpreadsheetDate`. Rien dans `DayDate` ni dans les autres classes du framework `JCommon`. Par conséquent, je les déplace dans `SpreadsheetDate`.

---

2. Plusieurs relecteurs de ce livre se sont élevés contre cette décision. Ils soutiennent que, dans un framework open-source, il est préférable de contrôler manuellement l'identifiant de sérialisation pour que des modifications mineures du logiciel n'invalident pas les anciennes dates sérialisées. C'est juste. Cependant, le dysfonctionnement, tout incommode qu'il soit, a au moins une cause claire. Par ailleurs, si l'auteur de la classe oublie de mettre à jour l'identifiant, le comportement d'échec est alors indéfini et peut très bien être silencieux. À mon avis, la conclusion de ce débat est qu'il ne faut pas s'attendre à pouvoir bénéficier de la désérialisation entre des versions différentes.

Les variables suivantes, `MINIMUM YEAR SUPPORTED` et `MAXIMUM YEAR SUPPORTED` (lignes 104 et 107), me placent devant un dilemme. Si `DayDate` est une classe abstraite qui ne préfigure aucune implémentation, il semble clair qu'elle ne doit pas fixer une année minimale ou maximale. Je suis à nouveau tenté de déplacer ces variables dans `SpreadsheetDate` [G6]. Toutefois, une rapide recherche de ces variables montre qu'une autre classe les utilise également : `RelativeDayOfWeekRule` (voir Listing B.6 à la page 416). Elles sont employées dans la fonction `getDate`, aux lignes 177 et 178, pour vérifier que l'argument passé à `getDate` est une année valide. Mon dilemme vient donc du fait suivant : un utilisateur d'une classe abstraite a besoin d'informations sur l'implémentation de cette classe.

Nous devons fournir cette information sans polluer la classe `DayDate` elle-même. En général, les informations d'implémentation sont obtenues à partir d'une instance d'une classe dérivée. Cependant, la fonction `getDate` ne reçoit pas en argument une instance de `DayDate`. En revanche, puisqu'elle retourne une telle instance, elle doit bien la créer quelque part. C'est aux lignes 187 à 205 que l'on trouve les indications. L'instance de `DayDate` est créée par l'une des trois fonctions `getPreviousDayOfWeek`, `getNearestDay OfWeek` et `getFollowingDayOfWeek`. En revenant au listing de `DayDate`, nous constatons que ces fonctions (lignes 638–724) retournent toutes une date créée par `addDays` (ligne 571), qui invoque `createInstance` (ligne 808), qui crée une `SpreadsheetDate` [G7] !

En général, les classes de base n'ont pas à connaître leurs classes dérivées. Pour corriger ce problème, nous pouvons employer le motif FABRIQUE ABSTRAITE [GOF] et écrire un `DayDateFactory`. Cette fabrique se chargera de la création des instances de `DayDate` dont nous avons besoin et pourra également répondre aux questions concernant l'implémentation, comme les dates limites.

```
public abstract class DayDateFactory {
  private static DayDateFactory factory = new SpreadsheetDateFactory();
  public static void setInstance(DayDateFactory factory) {
    DayDateFactory.factory = factory;
  }

  protected abstract DayDate _makeDate(int ordinal);
  protected abstract DayDate _makeDate(int day, DayDate.Month month, int year);
  protected abstract DayDate _makeDate(int day, int month, int year);
  protected abstract DayDate _makeDate(java.util.Date date);
  protected abstract int _getMinimumYear();
  protected abstract int _getMaximumYear();

  public static DayDate makeDate(int ordinal) {
    return factory._makeDate(ordinal);
  }

  public static DayDate makeDate(int day, DayDate.Month month, int year) {
    return factory._makeDate(day, month, year);
  }
```

```
  public static DayDate makeDate(int day, int month, int year) {
    return factory._makeDate(day, month, year);
  }

  public static DayDate makeDate(java.util.Date date) {
    return factory._makeDate(date);
  }

  public static int getMinimumYear() {
    return factory._getMinimumYear();
  }

  public static int getMaximumYear() {
    return factory._getMaximumYear();
  }
}
```

Cette classe de fabrique remplace les méthodes `createInstance` par des méthodes `makeDate` aux noms plus appropriés [N1]. Par défaut, elle choisit un `SpreadsheetDateFactory`, mais elle peut être modifiée à tout moment pour employer une fabrique différente. Les méthodes statiques qui renvoient vers des méthodes abstraites utilisent une combinaison des motifs SINGLETON, DÉCORATEUR et FABRIQUE ABSTRAITE [GOF] ; je les trouve particulièrement utiles.

Voici la classe `SpreadsheetDateFactory`.

```
public class SpreadsheetDateFactory extends DayDateFactory {
  public DayDate _makeDate(int ordinal) {
    return new SpreadsheetDate(ordinal);
  }

  public DayDate _makeDate(int day, DayDate.Month month, int year) {
    return new SpreadsheetDate(day, month, year);
  }

  public DayDate _makeDate(int day, int month, int year) {
    return new SpreadsheetDate(day, month, year);
  }

  public DayDate _makeDate(Date date) {
    final GregorianCalendar calendar = new GregorianCalendar();
    calendar.setTime(date);
    return new SpreadsheetDate(
      calendar.get(Calendar.DATE),
      DayDate.Month.make(calendar.get(Calendar.MONTH) + 1),
      calendar.get(Calendar.YEAR));
  }

  protected int _getMinimumYear() {
    return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
  }

  protected int _getMaximumYear() {
    return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
  }
}
```

Vous le constatez, j'ai déjà déplacé les variables `MINIMUM YEAR SUPPORTED` et `MAXIMUM YEAR SUPPORTED` dans la classe `SpreadsheetDate`, à laquelle elles appartiennent [G6].

Le problème suivant de `DayDate` vient des constantes de jours qui commencent à la ligne 109. Elles devraient constituer une autre énumération [J3]. Puisque nous avons déjà rencontré cette situation précédemment, je ne donnerai pas plus de détails. Vous les trouverez dans les listings finaux.

Nous avons ensuite plusieurs tableaux, en commençant par `LAST DAY OF MONTH` à la ligne 140. Mon premier souci concerne les commentaires redondants qui décrivent ces tableaux [C3]. Puisque les noms se suffisent à eux-mêmes, je supprime les commentaires.

Il n'y a aucune raison de ne pas rendre ce tableau privé [G8], puisque la fonction statique `lastDayOfMonth` fournit les données correspondantes.

Le tableau suivant, `AGGREGATE DAYS TO END OF MONTH`, est un tantinet plus mystérieux puisqu'il n'est jamais employé dans le framework `JCommon` [G9]. Je le supprime donc.

C'est la même chose pour `LEAP YEAR AGGREGATE DAYS TO END OF MONTH`.

Le tableau suivant, `AGGREGATE DAYS TO END OF PRECEDING MONTH`, n'est utilisé que dans `SpreadsheetDate` (lignes 434 et 473). Se pose donc la question de son déplacement vers `SpreadsheetDate`. Puisque le tableau n'est pas spécifique à une implémentation, rien ne justifie son déplacement [G6]. Toutefois, `SpreadsheetDate` est la seule implémentation et il est préférable de placer le tableau au plus près de son utilisation [G10].

Pour moi, l'important est d'être cohérent [G11]. Dans ce cas, nous devons rendre le tableau privé et l'exposer par l'intermédiaire d'une fonction comme `julianDateOf LastDayOfMonth`. Mais personne ne semble avoir besoin d'une telle fonction. Par ailleurs, le tableau peut être facilement remis dans `DayDate` si une nouvelle implémentation de `DayDate` en a besoin. Par conséquent, je le déplace.

C'est la même chose pour `LEAP YEAR AGGREGATE DAYS TO END OF MONTH`.

Nous voyons ensuite trois ensembles de constantes qui peuvent être convertis en énumérations (lignes 162–205). Le premier des trois sélectionne une semaine dans un mois. Je le transforme en un `enum` nommé `WeekInMonth`.

```
public enum WeekInMonth {
  FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
  public final int index;

  WeekInMonth(int index) {
    this.index = index;
  }
}
```

Le deuxième ensemble de constantes (lignes 177–187) est un peu plus obscur. Les constantes `INCLUDE NONE`, `INCLUDE FIRST`, `INCLUDE SECOND` et `INCLUDE BOTH` indiquent si les dates aux extrémités d'un intervalle doivent être incluses dans cet intervalle. Mathématiquement, cela se nomme "intervalle ouvert", "intervalle semi-ouvert" et "intervalle fermé". Je pense qu'il est plus clair d'employer la terminologie mathématique [N3] et je transforme donc cet ensemble en un `enum` nommé `DateInterval` avec les énumérateurs `CLOSED`, `CLOSED LEFT`, `CLOSED RIGHT` et `OPEN`.

Le troisième ensemble de constantes (lignes 18–205) décrit si la recherche d'un jour particulier de la semaine doit produire la dernière instance, la suivante ou la plus proche. Trouver le nom approprié est assez difficile. Pour finir, je choisis `Weekday Range`, avec les énumérateurs `LAST`, `NEXT` et `NEAREST`.

Vous pouvez ne pas être d'accord avec les noms retenus. Ils ont un sens pour moi, mais peut-être pas pour vous. Toutefois, l'important est qu'ils sont à présent sous une forme qui permet de les modifier facilement [J3]. Ils ne sont plus passés comme des entiers, mais comme des symboles. Je peux me servir de la fonction "renommer" de mon IDE pour changer les noms, ou les types, sans risquer d'oublier un `1` ou un `2` quelque part dans le code ou laisser une déclaration d'argument `int` mal décrite.

Le champ `description` à la ligne 208 ne semble plus employé par quiconque. Je le supprime donc, ainsi que les méthodes d'accès correspondantes [G9].

Je supprime également le constructeur par défaut à la ligne 213 [G12]. Le compilateur le générera pour nous.

Nous pouvons passer sur la méthode `isValidWeekdayCode` (lignes 216–238) puisque nous l'avons supprimée lorsque nous avons créé l'énumération `Day`.

Nous arrivons donc à la méthode `stringToWeekdayCode` (lignes 242–270). Les commentaires Javadoc qui ne donnent pas plus d'informations que la signature de la méthode sont superflus [C3],[G12]. Le seul intérêt de ces commentaires concerne la description de la valeur de retour `1`. Cependant, puisque nous sommes passés à l'énumération `Day`, le commentaire est en réalité faux [C2]. La méthode lance à présent une exception `IllegalArgumentException`. Je supprime donc le commentaire Javadoc.

Je supprime également les mots-clés `final` dans les arguments et les déclarations de variables. Je n'ai pas l'impression qu'ils ajoutent une réelle valeur, en revanche, ils contribuent au désordre [G12]. Ces suppressions de `final` vont à l'encontre d'une certaine croyance établie. Par exemple, Robert Simmons [Simmons04, p. 73] recommande fortement de "[...] mettre `final` partout dans le code". Je suis clairement en désaccord. Je pense qu'il y a quelques bonnes utilisations de `final`, comme une constante `final`. En dehors de cela, le mot-clé ajoute peu de valeur et participe au désordre.

Si je vois les choses ainsi, c'est probablement dû au fait que les erreurs détectables grâce à l'emploi de `final` sont déjà prises en charge par mes tests unitaires.

Je n'aime pas beaucoup les deux instructions `if` [G5] dans la boucle `for` (lignes 259 et 263). Je les réunis donc en une seule instruction `if` avec l'opérateur `||`. J'utilise également l'énumération `Day` pour contrôler la boucle `for` et apporte quelques autres modifications cosmétiques.

Il me vient à l'esprit que cette méthode n'appartient pas réellement à `DayDate`. Il s'agit en réalité de la fonction d'analyse de `Day`, où je la déplace donc. Cependant, l'énumération `Day` devient alors relativement longue. Puisque le concept de `Day` ne dépend pas de `DayDate`, je sors cette énumération de la classe `DayDate` et la place dans son propre fichier source [G13].

Je déplace également la fonction suivante, `weekdayCodeToString` (lignes 272–286), dans l'énumération `Day` et la nomme `toString`.

```
public enum Day {
  MONDAY(Calendar.MONDAY),
  TUESDAY(Calendar.TUESDAY),
  WEDNESDAY(Calendar.WEDNESDAY),s
  THURSDAY(Calendar.THURSDAY),
  FRIDAY(Calendar.FRIDAY),
  SATURDAY(Calendar.SATURDAY),
  SUNDAY(Calendar.SUNDAY);

  public final int index;
  private static DateFormatSymbols dateSymbols = new DateFormatSymbols();

  Day(int day) {
    index = day;
  }

  public static Day make(int index) throws IllegalArgumentException {
    for (Day d : Day.values())
      if (d.index == index)
        return d;
    throw new IllegalArgumentException(
      String.format("Illegal day index: %d.", index));
  }

  public static Day parse(String s) throws IllegalArgumentException {
    String[] shortWeekdayNames =
      dateSymbols.getShortWeekdays();
    String[] weekDayNames =
      dateSymbols.getWeekdays();

    s = s.trim();
    for (Day day : Day.values()) {
      if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
          s.equalsIgnoreCase(weekDayNames[day.index])) {
        return day;
```

```
        }
      }
      throw new IllegalArgumentException(
        String.format("%s is not a valid weekday string", s));
    }

    public String toString() {
      return dateSymbols.getWeekdays()[index];
    }
  }
```

Il existe deux fonctions `getMonths` (lignes 288–316). La première appelle la seconde, qui, elle, n'est jamais invoquée par une autre fonction. Par conséquent, je les réunis en une seule et les simplifie [G9],[G12],[F4]. Enfin, je choisis un nom plus descriptif [N1].

```
public static String[] getMonthNames() {
  return dateFormatSymbols.getMonths();
}
```

La fonction `isValidMonthCode` (lignes 326–346) n'est plus pertinente depuis la création de l'énumération `Month`. Je la supprime donc [G9].

La fonction `monthCodeToQuarter` (lignes 356–375) "sent l'envie de fonctionnalité" [Refactoring] [G14] et serait probablement mieux dans l'énumération `Month` en tant que méthode nommée `quarter`. J'effectue donc cette transformation.

```
public int quarter() {
  return 1 + (index-1)/3;
}
```

L'énumération `Month` est à présent suffisamment longue pour constituer sa propre classe. Je la sors donc de `DayDate` afin d'être cohérent avec l'énumération `Day` [G11], [G13].

Les deux méthodes suivantes se nomment `monthCodeToString` (lignes 377–426). Nous rencontrons à nouveau une méthode qui appelle sa jumelle avec un indicateur. En général, il est déconseillé de passer un indicateur en argument d'une fonction, en particulier lorsqu'il sélectionne simplement le format de la sortie [G15]. Je renomme, simplifie et restructure ces fonctions, puis les déplace dans l'énumération `Month` [N1], [N3], [C3], [G14].

```
public String toString() {
  return dateFormatSymbols.getMonths()[index - 1];
}

public String toShortString() {
  return dateFormatSymbols.getShortMonths()[index - 1];
}
```

Je renomme la méthode suivante `stringToMonthCode` (lignes 428–472), puis la déplace dans l'énumération `Month` et la simplifie [N1], [N3], [C3], [G12], [G14].

```
public static Month parse(String s) {
  s = s.trim();
  for (Month m : Month.values())
    if (m.matches(s))
      return m;

  try {
    return make(Integer.parseInt(s));
  }
  catch (NumberFormatException e) {}
  throw new IllegalArgumentException("Invalid month " + s);
}

private boolean matches(String s) {
  return s.equalsIgnoreCase(toString()) ||
         s.equalsIgnoreCase(toShortString());
}
```

La méthode `isLeapYear` (lignes 495–517) mérite d'être plus expressive [G16].

```
public static boolean isLeapYear(int year) {
  boolean fourth = year % 4 == 0;
  boolean hundredth = year % 100 == 0;
  boolean fourHundredth = year % 400 == 0;
  return fourth && (!hundredth || fourHundredth);
}
```

La fonction suivante, `leapYearCount` (lignes 519–536), n'appartient pas vraiment à `DayDate`. Personne ne l'invoque, à l'exception de deux méthodes de `SpreadsheetDate`, où je la transfère donc [G6].

La fonction `lastDayOfMonth` (lignes 538–560) utilise le tableau `LAST DAY OF MONTH`, qui appartient en réalité à l'énumération `Month` [G17], vers laquelle je le déplace. Je simplifie également la fonction et la rend un peu plus expressive [G16].

```
public static int lastDayOfMonth(Month month, int year) {
  if (month == Month.FEBRUARY && isLeapYear(year))
    return month.lastDay() + 1;
   else
    return month.lastDay();
}
```

Les choses commencent à devenir plus intéressantes. La fonction suivante se nomme `addDays` (lignes 562–576). Tout d'abord, puisqu'elle manipule les variables de `DayDate`, elle ne doit pas être statique [G18]. Je la convertis donc en méthode d'instance. Ensuite, elle appelle la fonction `toSerial`, qui doit être renommée `toOrdinal` [N1]. Enfin, la méthode peut être simplifiée.

```
public DayDate addDays(int days) {
  return DayDateFactory.makeDate(toOrdinal() + days);
}
```

Les mêmes modifications s'appliquent à `addMonths` (lignes 578–602). Elle doit être une méthode d'instance [G18]. Son algorithme étant un peu compliqué, j'utilise des VARIA-

BLES TEMPORAIRES D'EXPLICATION [Beck97] [G19] pour la rendre plus transparente. Je renomme également la méthode `getYYY` en `getYear` [N1].

```
public DayDate addMonths(int months) {
  int thisMonthAsOrdinal = 12 * getYear() + getMonth().index - 1;
  int resultMonthAsOrdinal = thisMonthAsOrdinal + months;
  int resultYear = resultMonthAsOrdinal / 12;
  Month resultMonth = Month.make(resultMonthAsOrdinal % 12 + 1);
  int lastDayOfResultMonth = lastDayOfMonth(resultMonth, resultYear);
  int resultDay = Math.min(getDayOfMonth(), lastDayOfResultMonth);
  return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
}
```

La fonction `addYears` (lignes 604–626) ne réserve aucune surprise.

```
public DayDate plusYears(int years) {
  int resultYear = getYear() + years;
  int lastDayOfMonthInResultYear = lastDayOfMonth(getMonth(), resultYear);
  int resultDay = Math.min(getDayOfMonth(), lastDayOfMonthInResultYear);
  return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
}
```

La conversion de ces méthodes statiques en méthodes d'instance me chagrine quelque peu. L'expression `date.addDays(5)` montre-t-elle clairement que l'objet `date` n'est pas modifié et qu'une nouvelle instance de `DayDate` est retournée ? Ou bien donne-t-elle l'impression erronée que nous ajoutons cinq jours à l'objet `date` ? Vous pensez sans doute que ce problème n'est pas important, mais un morceau de code semblable au suivant risque d'être trompeur [G20].

```
DayDate date = DateFactory.makeDate(5, Month.DECEMBER, 1952);
date.addDays(7); // Avancer la date d'une semaine.
```

La personne qui lit ce code pensera certainement que `addDays` modifie l'objet `date`. Nous devons donc trouver un nom qui lève cette ambiguïté [N4]. Je choisis les noms `plusDays` et `plusMonths`. Il me semble que les intentions de la méthode sont bien transmises par :

```
DayDate date = oldDate.plusDays(5);
```

Alors que la ligne suivante ne se lit pas suffisamment bien pour que le lecteur comprenne simplement que l'objet `date` soit modifié :

```
date.plusDays(5);
```

Les algorithmes continuent d'être encore plus intéressants. `getPreviousDayOfWeek` (lignes 628–660) fonctionne, mais se révèle un peu complexe. Après un moment de réflexion sur ce qu'elle réalise réellement [G21], je peux la simplifier et employer des variables temporaires d'explication [G19] pour la clarifier. Je la transforme également en méthode d'instance [G18] et retire la méthode d'instance redondante [G5] (lignes 997–1008).

```
public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
  int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;
  if (offsetToTarget >= 0)
    offsetToTarget -= 7;
  return plusDays(offsetToTarget);
}
```

`getFollowingDayOfWeek` (lignes 662–693) subit la même analyse et les mêmes transformations.

```
public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
  int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;
  if (offsetToTarget <= 0)
    offsetToTarget += 7;
  return plusDays(offsetToTarget);
}
```

La fonction suivante se nomme `getNearestDayOfWeek` (lignes 695–726) et nous l'avons déjà corrigée (voir page 288). Cependant, les modifications apportées à ce moment-là ne sont pas cohérentes avec le motif actuel des deux dernières fonctions [G11]. Je me charge donc de la rendre cohérente et d'utiliser des variables temporaires d'explication [G19] pour clarifier l'algorithme.

```
public DayDate getNearestDayOfWeek(final Day targetDay) {
  int offsetToThisWeeksTarget = targetDay.index - getDayOfWeek().index;
  int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
  int offsetToPreviousTarget = offsetToFutureTarget - 7;

  if (offsetToFutureTarget > 3)
    return plusDays(offsetToPreviousTarget);
  else
    return plusDays(offsetToFutureTarget);
}
```

La méthode `getEndOfCurrentMonth` (lignes 728–740) est quelque peu étrange car il s'agit d'une méthode d'instance qui jalouse [G14] sa propre classe en prenant un argument `DayDate`. J'en fais une véritable méthode d'instance et améliore quelques noms.

```
public DayDate getEndOfMonth() {
  Month month = getMonth();
  int year = getYear();
  int lastDay = lastDayOfMonth(month, year);
  return DayDateFactory.makeDate(lastDay, month, year);
}
```

Le remaniement de `weekInMonthToString` (lignes 742–761) se révèle très intéressant. Grâce aux outils de remaniement de mon IDE, je commence par déplacer la méthode vers l'énumération `WeekInMonth` que nous avons créée à la page 294. Puis je renomme la méthode en `toString`. Ensuite, de méthode statique, je la transforme en méthode d'instance. Tous les tests réussissent. (Voyez-vous où je veux en venir ?)

Ensuite, je supprime totalement la méthode ! Cinq assertions échouent (lignes 411–415 du Listing B.4 à la page 399). Je modifie ces lignes pour qu'elles utilisent les noms des

énumérateurs (`FIRST`, `SECOND`...). Tous les tests réussissent. Comprenez-vous pourquoi ? Comprenez-vous également pourquoi chacune de ces étapes est nécessaire ? L'outil de remaniement me permet de vérifier que le code qui appelait `weekInMonthToString` invoque à présent `toString` sur l'énumérateur `weekInMonth`, car tous les énumérateurs implémentent `toString` pour simplement retourner leur nom...

Malheureusement, j'ai été trop astucieux. Aussi élégante que puisse être cette séquence de remaniements, je réalise finalement que les seuls utilisateurs de cette fonction sont les tests que je viens de modifier. Je supprime donc les tests.

Dupe-moi une fois, honte à toi. Dupe-moi deux fois, honte à moi ! Ayant découvert que seuls les tests appellent `relativeToString` (lignes 765–781), je supprime simplement la fonction et ses tests.

Nous arrivons enfin aux méthodes abstraites de cette classe abstraite. La première est évidemment `toSerial` (lignes 838–844), que nous avons renommée `toOrdinal` à la page 298. En la considérant dans ce contexte, je pense que son nom doit être changé en `getOrdinalDay`.

La méthode abstraite suivante est `toDate` (lignes 838–844). Elle convertit un `DayDate` en un `java.util.Date`. Pourquoi cette méthode est-elle abstraite ? Si nous examinons son implémentation dans `SpreadsheetDate` (lignes 198–207 du Listing B.5 à la page 407), nous constatons qu'elle ne dépend absolument pas de l'implémentation de cette classe [G6]. Je la remonte donc dans la hiérarchie.

Les méthodes `getYYYY`, `getMonth` et `getDayOfMonth` sont bien abstraites, mais la méthode `getDayOfWeek` doit également être retirée de `SpreadSheetDate` car elle ne dépend pas d'éléments de code qui ne se trouvent pas dans `DayDate` [G6]. Mais est-ce vraiment le cas ?

Si vous examinez attentivement le code (ligne 247 du Listing B.5 à la page 407), vous constaterez que l'algorithme dépend implicitement de l'origine du jour ordinal (autrement dit, le jour de la semaine du jour 0). Par conséquent, même si cette fonction n'a pas de dépendance physique qui l'empêcherait d'être déplacée dans `DayDate`, elle a bien une dépendance logique.

Les dépendances logiques de ce type m'ennuient [G22]. Si un élément logique dépend de l'implémentation, alors, un élément physique doit également en dépendre. De plus, il me semble que l'algorithme lui-même pourrait être générique et présenter une plus faible dépendance avec l'implémentation [G6].

Je crée donc dans `DayDate` une méthode abstraite nommée `getDayOfWeekForOrdinalZero` et l'implémente dans `SpreadsheetDate` pour qu'elle retourne `Day.SATURDAY`. Je remonte ensuite la méthode `getDayOfWeek` dans `DayDate` et la modifie pour qu'elle appelle `getOrdinalDay` et `getDayOfWeekForOrdinalZero`.

```
public Day getDayOfWeek() {
  Day startingDay = getDayOfWeekForOrdinalZero();
  int startingOffset = startingDay.index - Day.SUNDAY.index;
  return Day.make((getOrdinalDay() + startingOffset) % 7 + 1);
}
```

Lisez attentivement le commentaire des lignes 895 à 899. Cette répétition est-elle vraiment nécessaire ? Comme d'habitude, je supprime ce commentaire, ainsi que les autres.

La méthode suivante se nomme `compare` (lignes 902–913). À nouveau, elle ne devrait pas être abstraite [G6] et je place donc son implémentation dans `DayDate`. Par ailleurs, son nom n'est pas assez descriptif [N1]. Cette méthode retourne en réalité la différence en jours par rapport à la date passée en argument. Je la renomme `daysSince`. De plus, je remarque qu'il n'existe aucun test pour cette méthode. Je les écris.

Les six fonctions suivantes (lignes 915–980) sont des méthodes abstraites qui doivent être implémentées dans `DayDate`. Je les extrais donc de `SpreadsheetDate`.

La dernière fonction, `isInRange` (lignes 982–995), doit également être extraite et remaniée. L'instruction `switch` est un peu laide [G23] et peut être remplacée en déplaçant les cas dans l'énumération `DateInterval`.

```
public enum DateInterval {
  OPEN {
    public boolean isIn(int d, int left, int right) {
      return d > left && d < right;
    }
  },
  CLOSED_LEFT {
    public boolean isIn(int d, int left, int right) {
      return d >= left && d < right;
    }
  },
  CLOSED_RIGHT {
    public boolean isIn(int d, int left, int right) {
      return d > left && d <= right;
    }
  },
  CLOSED {
    public boolean isIn(int d, int left, int right) {
      return d >= left && d <= right;
    }
  };

  public abstract boolean isIn(int d, int left, int right);
}

----------------------------

public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
  int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
  int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
  return interval.isIn(getOrdinalDay(), left, right);
}
```

Nous arrivons ainsi à la fin de `DayDate`. Je fais une nouvelle passe sur l'intégralité de la classe afin de voir si elle sonne bien.

Tout d'abord, le commentaire d'ouverture est obsolète. Je le raccourcis et l'améliore [C2].

Ensuite, je déplace toutes les énumérations restantes dans leur propre fichier [G12].

Puis je déplace la variable statique (`dateFormatSymbols`) et les trois méthodes statiques (`getMonthNames`, `isLeapYear`, `lastDayOfMonth`) dans une nouvelle classe nommée `DateUtil` [G6].

Je déplace les méthodes abstraites au début, là où elles doivent se trouver [G24].

Je renomme `Month.make` en `Month.fromInt` [N1] et procède de même pour toutes les autres énumérations. J'ajoute également une méthode d'accès `toInt()` à toutes les énumérations et rend le champ `index` privé.

Je peux éliminer une redondance intéressante [G5] dans `plusYears` et `plusMonths` en extrayant une nouvelle méthode nommée `correctLastDayOfMonth`. Les trois autres méthodes deviennent également plus claires.

Je supprime le nombre magique `1` [G25], en le remplaçant par `Month.JANUARY.toInt()` ou `Day.SUNDAY.toInt()`. Je passe du temps sur `SpreadsheetDate` pour nettoyer les algorithmes. Le résultat final se trouve dans les Listings B.7, page 419, à B.16, page 427.

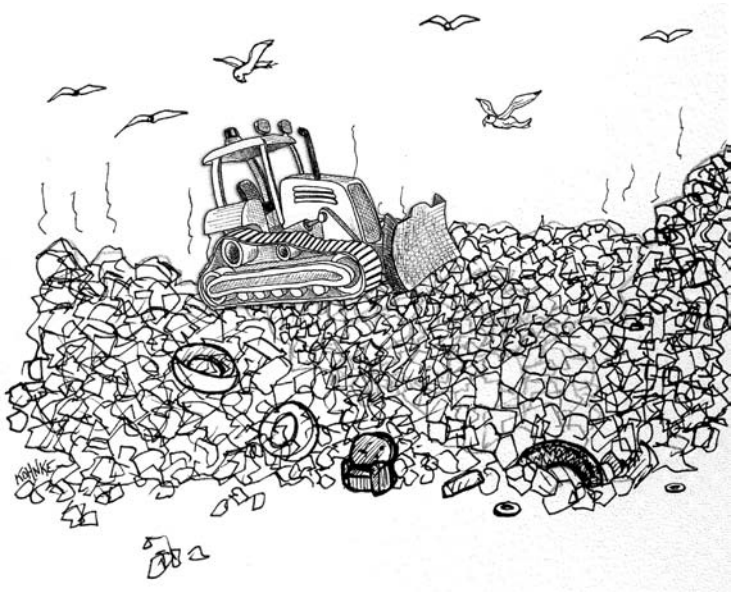
Après analyse, il est intéressant de constater que la couverture du code dans `DayDate` a *baissé*, pour atteindre 84,9 % ! En réalité, le nombre de fonctionnalités testées n'a pas diminué, mais la classe s'est tellement réduite que les quelques lignes non couvertes ont un poids plus important. 45 des 53 instructions exécutables de `DayDate` sont à présent couvertes par les tests. Les lignes non couvertes sont tellement simples que cela ne vaut pas la peine de les tester.

Conclusion

Encore une fois, nous avons respecté la règle du boy-scout. Nous avons laissé ce code plus propre que nous l'avons trouvé. Il a fallu du temps, mais cela en valait la peine. La couverture des tests a augmenté, certains bogues ont été corrigés, le code s'est clarifié et a été réduit. La prochaine personne qui lira ce code le trouvera, avec un peu de chance, plus facile à étudier que nous. Elle sera probablement en mesure de le nettoyer encore un peu.

17

Indicateurs et heuristiques



Dans son ouvrage *Refactoring* [Refactoring], Martin Fowler identifie plusieurs "indicateurs de code" (*Code Smells*[1]). Les indicateurs donnés dans ce chapitre comprennent ceux de Martin, ainsi que plusieurs autres qui me sont dus. Vous trouverez également des heuristiques dont je me sers dans mon métier.

1. N.d.T. : Littéralement "odeurs du code", généralement traduit par "indicateurs de code" ou "symptômes du code". Ces "odeurs" avertissent que quelque chose se passe mal ailleurs. Le flair du programmeur permet de remonter jusqu'au problème.

J'ai constitué cette liste d'indicateurs et d'heuristiques pendant que j'étudiais différents programmes et les remaniais. Chaque fois que j'apportais une modification, je me demandais *pourquoi* j'effectuais ce changement, puis j'en écrivais la raison. Le résultat est une assez longue liste de points qui "sentent" mauvais lorsque je lis du code.

Cette liste doit être lue du début à la fin et servir de référence. Dans l'Annexe C, chaque heuristique fait l'objet d'une référence croisée qui indique où elle a été employée dans ce livre.

Commentaires

C1 : informations inappropriées

Un commentaire ne doit pas contenir des informations dont la véritable place est ailleurs, comme dans le système de gestion du code source, le système de suivi des problèmes ou tout autre système de gestion de documents. Par exemple, les historiques des modifications ne font qu'encombrer les fichiers sources avec de grandes quantités de texte pas vraiment intéressant. En général, les métadonnées, comme les auteurs, la date de dernière modification, les numéros de SPR (*Software Problem Report*), etc., ne doivent pas apparaître dans les commentaires. Ces derniers sont réservés aux notes techniques concernant le code et la conception.

C2 : commentaires obsolètes

Un commentaire ancien, non pertinent et incorrect est obsolète. Les commentaires vieillissent rapidement. Il est préférable de ne pas écrire un commentaire qui deviendra obsolète. Si vous rencontrez un commentaire obsolète, vous devez le mettre à jour ou le supprimer aussi rapidement que possible. Les commentaires obsolètes ont tendance à s'éloigner du code qu'ils sont censés décrire. Ils deviennent des sources d'informations hors de propos et de distractions dans le code.

C3 : commentaires redondants

Un commentaire est redondant lorsqu'il explique quelque chose qui est déjà suffisamment descriptif en soi. Par exemple :

```
i++; // Incrémenter i.
```

Un commentaire Javadoc qui ne dit rien de plus, voire moins, que la signature de la fonction est également redondant :

```
/**
 * @param sellRequest
 * @return
 * @throws ManagedComponentException
 */
```

```
public SellResponse beginSellItem(SellRequest sellRequest)
  throws ManagedComponentException
```

Les commentaires doivent expliquer des choses que le code est incapable d'exprimer par lui-même.

C4 : commentaires mal rédigés

Un commentaire ne vaut la peine d'être écrit que s'il est bien rédigé. Si vous envisagez d'ajouter un commentaire, prenez le temps de vous assurer qu'il s'agit du meilleur commentaire que vous pouvez écrire. Choisissez soigneusement vos mots. Employez une grammaire et une ponctuation correctes. Ne radotez pas. Ne répétez pas l'évident. Soyez concis.

C5 : code mis en commentaire

Les pans de code mis en commentaire me rendent fou. On ne sait jamais depuis quand ils sont en commentaire ni s'ils sont ou non significatifs. Personne ne veut les supprimer car tout le monde suppose que quelqu'un d'autre en a besoin ou envisage de les utiliser.

Le code reste là et se dégrade, en devenant de moins en moins pertinent au fur et à mesure que les jours passent. Il invoque des fonctions qui n'existent plus. Il utilise des variables dont les noms ont changé. Il suit des conventions qui sont depuis longtemps obsolètes. Il pollue les modules dans lesquels il se trouve et distrait celui qui tente de le lire. Le code en commentaire est une *abomination*.

Lorsque vous rencontrez du code en commentaire, *supprimez-le* ! Ne vous inquiétez pas, le système de gestion du code source l'a en mémoire. Si quelqu'un en a réellement besoin, il peut parfaitement consulter une version précédente. Ne tolérez pas que le code mis en commentaire survive.

Environnement

E1 : la construction exige plusieurs étapes

La construction d'un projet doit représenter une seule opération triviale. Vous ne devez pas avoir à extraire de nombreux petits éléments du système de gestion du code source. Vous ne devez pas avoir besoin d'une suite de commandes incompréhensibles ou de scripts dépendant du contexte pour construire chaque élément. Vous ne devez pas avoir à rechercher tous les petits fichiers JAR, XML ou autres dont le système a besoin. Vous *devez* pouvoir extraire le système par une simple commande, puis le construire à l'aide d'une autre commande simple.

```
svn get mySystem
cd mySystem
ant all
```

E2 : les tests exigent plusieurs étapes

Vous devez être en mesure d'exécuter *tous* les tests unitaires à l'aide d'une seule commande. Dans le meilleur des cas, vous pouvez exécuter tous les tests en cliquant sur un bouton de votre IDE. Dans le pire des cas, vous devez pouvoir saisir une seule commande simple dans un shell. L'exécution des tests est fondamentale et tellement importante qu'elle doit pouvoir se faire rapidement, facilement et simplement.

Fonctions

F1 : trop grand nombre d'arguments

Le nombre d'arguments d'une fonction doit être aussi réduit que possible. Il est préférable de n'avoir aucun argument, puis un, deux et trois. Lorsqu'il y a plus de trois arguments, vous devez vous interroger sur leur raison d'être et faire en sorte d'éviter cette situation. (Voir la section "Arguments d'une fonction" à la page 45.)

F2 : arguments de sortie

Les arguments de sortie sont tout sauf intuitifs. Le lecteur s'attend à ce que les arguments soient des entrées, non des sorties. Si votre fonction doit modifier un état, ce doit être celui de l'objet sur lequel elle est invoquée. (Voir la section "Arguments de sortie" à la page 50.)

F3 : arguments indicateurs

Les arguments booléens mettent clairement en évidence que la fonction fait plusieurs choses. Ils sont déroutants et doivent être éliminés. (Voir la section "Arguments indicateurs" à la page 46.)

F4 : fonction morte

Les méthodes qui ne sont jamais appelées doivent être retirées. La conservation du code mort coûte cher. N'ayez pas peur de supprimer une fonction morte : le système de gestion de code source se charge de la conserver.

Général

G1 : multiples langages dans un même fichier source

Avec les environnements de programmation modernes, il est possible d'employer plusieurs langages différents dans un même fichier source. Par exemple, un fichier source Java peut contenir des morceaux de XML, HTML, YAML, JavaDoc, français, JavaScript, etc. De même, en plus du contenu HTML, un fichier JSP peut contenir du code Java, du contenu respectant la syntaxe d'une bibliothèque de balises, des commentaires en français, du contenu Javadoc, du XML, du JavaScript, etc. Cette pratique est au mieux perturbante, et pour le moins imprudemment négligée.

Idéalement, un fichier source ne doit contenir qu'un et un seul langage. De manière plus réaliste, vous devrez probablement en employer plusieurs. Mais vous devez vous donner du mal pour réduire le nombre et l'étendue des langages supplémentaires dans les fichiers sources.

G2 : comportement évident non implémenté

Si l'on respecte le "principe de moindre surprise"[2], toute fonction ou classe doit implémenter les comportements auxquels un autre programmeur peut raisonnablement s'attendre. Par exemple, prenons le cas d'une fonction qui convertit le nom d'un jour en un `enum` qui représente le jour.

```
Day day = DayDate.StringToDay(String dayName);
```

Nous nous attendons à ce que la chaîne `"Monday"` soit convertie en `Day.MONDAY`. Nous pouvons également supposer que les abréviations courantes sont traduites et que la fonction ignore la casse.

Lorsqu'un comportement évident n'est pas implémenté, les lecteurs et les utilisateurs du code ne peuvent plus se fonder sur leurs intuitions concernant les noms des fonctions. Ils n'ont plus confiance dans l'auteur d'origine et doivent se replier sur la lecture des détails du code.

G3 : comportement incorrect aux limites

Il peut sembler évident que le code doive se comporter correctement. Cependant, nous réalisons rarement combien cette évidence est compliquée à mettre en œuvre. Les développeurs écrivent souvent des fonctions qu'ils pensent opérationnelles et font confiance à leur intuition au lieu de faire l'effort de montrer que leur code fonctionne dans tous les cas limites.

---

2. **http://fr.wikipedia.org/wiki/Principe_de_moindre_surprise**.

Rien ne remplace une attention poussée. Chaque condition limite, chaque cas particulier, chaque excentricité et chaque exception peut remettre en question un algorithme élégant et intuitif. *Ne vous fondez pas sur vos intuitions*. Recherchez chaque condition limite et écrivez le test correspondant.

G4 : sécurités neutralisées

Tchernobyl a explosé car le directeur technique avait neutralisé un par un les mécanismes de sécurité. En raison des sécurités, les expérimentations étaient malcommodes à mener. L'expérience n'a pas fonctionné et le monde a vu sa première catastrophe nucléaire civile majeure.

Neutraliser les sécurités présente un risque. Il peut être nécessaire d'exercer un contrôle manuel sur `serialVersionUID`, mais c'est toujours risqué. La désactivation de certains avertissements du compilateur, voire de tous les avertissements, peut faciliter la construction du système, mais au risque de sessions de débogage sans fin. Désactiver les tests qui échouent et se dire que l'on fera en sorte qu'ils réussissent plus tard est aussi mauvais que prétendre que sa carte bancaire représente de l'argent gratuit.

G5 : redondance

Il s'agit de l'une des règles les plus importantes de ce livre, et vous devez la prendre très au sérieux. Pratiquement tous les auteurs qui écrivent des ouvrages sur la conception des logiciels la mentionnent. Dave Thomas et Andy Hunt la nomment principe DRY (*Don't Repeat Yourself*, Ne vous répétez pas) [PRAG]. Kent Beck en fait l'un des principes essentiels de l'Extreme Programming et la désigne par "une fois et une seule". Ron Jeffries classe cette règle en seconde place, juste après la réussite de tous les tests.

Chaque fois que vous recontrez une redondance dans le code, elle représente en réalité une opportunité d'abstraction manquée. Cette redondance peut certainement être convertie en un sous-programme ou une autre classe. Par cette conversion, vous enrichissez le vocabulaire du langage de votre conception. D'autres programmeurs peuvent se servir des outils abstraits que vous créez. Le développement devient plus rapide et moins sujet aux erreurs car vous avez élevé le niveau d'abstraction.

La redondance la plus courante prend la forme de pans de code identiques qui semblent tout droit sortis d'un copier-coller de la part des programmeurs. Ils doivent être remplacés par de simples méthodes.

Une chaîne `switch/case` ou `if/else` qui apparaît de nombreuses fois dans différents modules, en testant toujours les mêmes conditions, est une forme de redondance plus subtile. Ces chaînes doivent être remplacées en utilisant le polymorphisme.

Plus subtil encore, vous pourrez rencontrer des modules qui emploient des algorithmes semblables, mais qui ne partagent aucune ligne de code similaire. Il s'agit encore de

redondance, que vous devez traiter à l'aide des motifs de conception PATRON DE MÉTHODE ou STRATÉGIE [GOF].

La plupart des motifs de conception apparus ces quinze dernières années sont en réalité des solutions bien connues pour éliminer la redondance. Les formes normales de Codd sont également une stratégie de suppression de la redondance dans les schémas de bases de données. L'orienté objet est en soi une stratégie d'organisation des modules et d'élimination de la redondance. C'est également le cas de la programmation structurée.

Je pense que cela doit être clair à présent. Trouvez et éliminez la redondance.

G6 : code au mauvais niveau d'abstraction

Il est important de créer des abstractions qui distinguent les concepts généraux de niveau supérieur des concepts détaillés de niveau inférieur. Pour ce faire, nous créons parfois des classes abstraites pour les concepts de haut niveau et des classes dérivées pour les concepts de bas niveau. Lorsque nous procédons ainsi, nous devons assurer une séparation totale. Nous voulons que *tous* les concepts de bas niveau se trouvent dans les classes dérivées et que *tous* les concepts de haut niveau soient dans la classe de base.

Par exemple, les constantes, les variables ou les fonctions utilitaires qui ne concernent que l'implémentation détaillée ne doivent pas se trouver dans la classe de base. La classe de base n'a pas à connaître leur existence.

Cette règle s'applique également aux fichiers sources, aux composants et aux modules. Pour une bonne conception du logiciel, nous devons séparer les concepts de niveaux différents et les placer dans des conteneurs différents. Ces conteneurs sont parfois des classes de base ou des classes dérivées, d'autres fois des fichiers sources, des modules ou des composants. Quelle que soit la situation, la séparation doit être complète. Il ne faut pas que les concepts de haut niveau et ceux de bas niveau soient mélangés.

Prenons le code suivant :

```
public interface Stack {
  Object pop() throws EmptyException;
  void push(Object o) throws FullException;
  double percentFull();

  class EmptyException extends Exception {}
  class FullException extends Exception {}
}
```

La fonction `percentFull` se trouve au mauvais niveau d'abstraction. Même s'il peut exister plusieurs implémentations de `Stack` dans lesquelles le concept de *remplissage* se comprend, il existe d'autres implémentations *qui ne peuvent tout simplement pas* connaître le degré de remplissage. La fonction serait donc mieux placée dans une interface dérivée, comme `BoundedStack`.

Vous pensez peut-être que l'implémentation pourrait simplement retourner zéro lorsque la pile n'est pas limitée. Vous oubliez sans doute qu'aucune pile n'est vraiment illimitée. Vous ne pouvez pas réellement empêcher une exception `OutOfMemoryException` par un simple test

```
stack.percentFull() < 50.0
```

Si vous implémentez la fonction pour qu'elle retourne 0, vous mentez.

Vous ne pouvez pas mentir ou faire semblant sous prétexte d'une abstraction mal placée. Isoler des abstractions fait partie des plus grandes difficultés pour les développeurs de logiciels, et il n'existe aucun correctif rapide lorsqu'ils se trompent.

G7 : classes de base qui dépendent de leurs classes dérivées

Si les concepts sont partitionnés dans des classes de base et des classes dérivées, c'est principalement pour que les concepts de haut niveau dans une classe de base soient indépendants des concepts de bas niveau dans une classe dérivée. Par conséquent, lorsque nous rencontrons des classes de base qui font référence aux noms de leurs classes dérivées, nous suspectons un problème. En général, les classes de base ne doivent rien connaître de leurs classes dérivées.

Bien entendu, il existe des exceptions à cette règle. Parfois, le nombre de classes dérivées est figé et la classe de base contient du code qui choisit la classe dérivée adéquate. Ce cas se rencontre souvent dans l'implémentation des machines à états finis. Cependant, dans cet exemple précis, les classes dérivées et la classe de base sont fortement couplées et toujours déployées ensemble dans le même fichier jar. De manière générale, nous voulons pouvoir déployer des classes dérivées et des classes de base dans des fichiers jar différents.

En déployant séparément les classes dérivées et les classes de base et en nous assurant que les fichiers jar des classes de base ne connaissent pas le contenu des fichiers jar des classes dérivées, nous pouvons déployer nos systèmes sous forme de composants autonomes et indépendants. Lorsque de tels composants sont modifiés, ils peuvent être redéployés sans avoir à redéployer les composants de base. Autrement dit, l'impact d'une modification est considérablement affaibli et la maintenance des systèmes devient beaucoup plus simple.

G8 : beaucoup trop d'informations

Les modules bien définis offrent des interfaces très petites qui permettent de faire beaucoup avec peu. Les modules mal définis fournissent des interfaces vastes et profondes qui nous obligent à employer plusieurs actions différentes pour réaliser des choses

simples. Avec une interface parfaitement définie, nous dépendons d'un nombre réduit de fonctions et le couplage est faible. Avec une interface médiocre, nous devons invoquer un grand nombre de fonctions et le couplage est élevé.

Les bons développeurs de logiciels apprennent à limiter les points exposés par les interfaces de leurs classes et modules. Moins la classe propose de méthodes, mieux c'est. Moins une fonction connaît de variables, mieux c'est. Moins une classe contient de variables d'instance, mieux c'est.

Cachez vos données. Cachez vos fonctions utilitaires. Cachez vos constantes et vos variables temporaires. Ne créez pas des classes qui offrent un grand nombre de méthodes ou de variables d'instance. Ne créez pas de nombreuses variables et fonctions protégées destinées aux sous-classes. Efforcez-vous de définir des interfaces concises. Maintenez un couplage faible en limitant les informations.

G9 : code mort

Le code mort correspond à du code qui n'est pas exécuté. Vous en trouverez dans le corps d'une instruction `if` qui vérifie une condition qui ne se produit jamais. Vous en trouverez dans le bloc `catch` d'une instruction `try` qui ne lance jamais d'exception. Vous en trouverez dans de petites méthodes utilitaires qui ne sont jamais invoquées ou dans des conditions `switch`/`case` qui ne se présentent jamais.

Au bout d'un certain temps, le code mort commence à "sentir". Plus il est ancien, plus l'odeur est forte et aigre. En effet, le code mort n'est pas totalement mis à jour lorsque la conception change. Il *compile* toujours, mais il ne respecte pas les nouvelles conventions ou les nouvelles règles. Il a été écrit à un moment où le système était *différent*. Lorsque vous rencontrez du code mort, n'hésitez pas. Offrez-lui un enterrement décent en le retirant du système.

G10 : séparation verticale

Les variables et les fonctions doivent être définies au plus près de leur utilisation. Les variables locales doivent être déclarées juste avant leur première utilisation et doivent avoir une portée verticale réduite. Il ne faut pas que les déclarations des variables locales se trouvent à des centaines de lignes de leurs utilisations.

Les fonctions privées doivent être définies juste après leur première utilisation. Elles appartiennent à la portée de la classe globale, mais nous préférons limiter la distance verticale entre les invocations et les définitions. Pour trouver une fonction privée, il faut que nous ayons simplement à regarder vers le bas à partir de sa première utilisation.

G11 : incohérence

Si vous faites quelque chose d'une certaine manière, toutes les choses comparables doivent se faire de cette façon-là. On retrouve là le principe de moindre surprise. Choisissez bien vos conventions et, une fois la décision prise, faites attention à les respecter.

Si, dans une fonction précise, vous utilisez une variable nommée `response` pour contenir un `HttpServletResponse`, vous devez employer de manière cohérente le même nom de variable dans les autres fonctions qui manipulent des objets `HttpServletResponse`. Si vous nommez une méthode `processVerificationRequest`, vous devez choisir un nom semblable, comme `processDeletionRequest`, pour les méthodes qui traitent d'autres types de requêtes.

Lorsqu'elle est appliquée de manière fiable, une simple cohérence de ce genre permet d'obtenir un code plus facile à lire et à modifier.

G12 : désordre

À quoi peut bien servir un constructeur par défaut sans implémentation ? Il ne fait qu'encombrer le code avec des artefacts sans signification. Les variables non utilisées, les fonctions jamais invoquées, les commentaires qui n'apportent aucune information, etc. ne font que contribuer au désordre et doivent être supprimés. Vous devez garder des fichiers sources propres, bien organisés et sans désordre.

G13 : couplage artificiel

Les choses indépendantes ne doivent pas être couplées artificiellement. Par exemple, vous ne devez pas placer les énumérations générales dans des classes spécifiques car l'ensemble de l'application doit alors connaître ces classes spécifiques. C'est également vrai pour les fonctions `static` à usage général déclarées dans des classes spécifiques.

Un couplage artificiel correspond souvent à un couplage entre deux modules qui n'ont aucun rapport direct. Il résulte du placement d'une variable, d'une constante ou d'une fonction dans un endroit commode sur le moment, mais finalement inadapté. Cela révèle une certaine paresse et un manque de soin.

Prenez le temps de déterminer le meilleur emplacement pour la déclaration des fonctions, des constantes et des variables. Ne les déposez pas simplement dans l'endroit le plus pratique, pour les y laisser.

G14 : envie de fonctionnalité

Voilà l'un des indicateurs de code de Martin Fowler [Refactoring]. Les méthodes d'une classe doivent s'intéresser uniquement aux variables et aux fonctions de leur classe, non à celles des autres classes. Lorsqu'une méthode invoque des accesseurs et des muta-

teurs d'un autre objet pour en manipuler les données, elle *envie* la portée de la classe de cet autre objet. Elle voudrait se trouver à l'intérieur de cette autre classe afin d'avoir un accès direct aux variables qu'elle manipule. Par exemple :

```
public class HourlyPayCalculator {
  public Money calculateWeeklyPay(HourlyEmployee e) {
    int tenthRate = e.getTenthRate().getPennies();
    int tenthsWorked = e.getTenthsWorked();
    int straightTime = Math.min(400, tenthsWorked);
    int overTime = Math.max(0, tenthsWorked - straightTime);
    int straightPay = straightTime * tenthRate;
    int overtimePay = (int)Math.round(overTime*tenthRate*1.5);
    return new Money(straightPay + overtimePay);
  }
}
```

La méthode `calculateWeeklyPay` entre dans l'objet `HourlyEmployee` pour obtenir les données qui lui sont nécessaires. Elle *envie* la portée de `HourlyEmployee`. Elle "souhaite" se trouver dans `HourlyEmployee`.

L'envie de fonctionnalité n'est pas souhaitable car elle oblige une classe à exposer sa structure interne à une autre. Toutefois, l'envie de fonctionnalité est parfois un mal nécessaire. Prenons le cas suivant :

```
public class HourlyEmployeeReport {
  private HourlyEmployee employee ;

  public HourlyEmployeeReport(HourlyEmployee e) {
    this.employee = e;
  }

  String reportHours() {
    return String.format(
      "Name: %s\tHours:%d.%1d\n",
      employee.getName(),
      employee.getTenthsWorked()/10,
      employee.getTenthsWorked()%10);
  }
}
```

La méthode `reportHours` envie manifestement la classe `HourlyEmployee`. Mais nous ne voulons pas que `HourlyEmployee` connaisse le format du rapport. Si nous déplaçons la chaîne de format dans la classe `HourlyEmployee`, nous ne respectons pas plusieurs principes de la conception orientée objet[3]. En effet, `HourlyEmployee` serait alors couplée au format du rapport, l'assujettissant ainsi aux modifications de ce format.

3. Plus précisément, le principe de responsabilité unique, le principe ouvert/fermé et le principe de fermeture commune (voir [PPP]).

G15 : arguments sélecteurs

Rien n'est plus abominable qu'un argument `false` à la fin d'un appel de fonction. Que signifie-t-il ? Que se passe-t-il si nous le changeons en `true` ? Non seulement le rôle d'un argument sélecteur est difficile à mémoriser, mais chaque argument sélecteur combine de nombreuses fonctions en une. Les arguments sélecteurs ne sont qu'une manière paresseuse d'éviter de décomposer une longue fonction en plusieurs fonctions plus petites. Prenons la méthode suivante :

```
public int calculateWeeklyPay(boolean overtime) {
  int tenthRate = getTenthRate();
  int tenthsWorked = getTenthsWorked();
  int straightTime = Math.min(400, tenthsWorked);
  int overTime = Math.max(0, tenthsWorked - straightTime);
  int straightPay = straightTime * tenthRate;
  double overtimeRate = overtime ? 1.5 : 1.0 * tenthRate;
  int overtimePay = (int)Math.round(overTime*overtimeRate);
  return straightPay + overtimePay;
}
```

Elle est invoquée avec l'argument `true` lorsque les heures supplémentaires doivent être payées une fois et demie plus cher, et avec un argument `false` lorsqu'elles sont payées le même prix que les heures normales. Si nous rencontrons l'invocation `calculateWeeklyPay(false)` dans le code, il n'est pas certain que nous saurons nous rappeler de sa signification. Toutefois, le vrai problème d'une telle fonction est que son auteur a manqué l'occasion de l'écrire de la manière suivante :

```
public int straightPay() {
  return getTenthsWorked() * getTenthRate();
}

public int overTimePay() {
  int overTimeTenths = Math.max(0, getTenthsWorked() - 400);
  int overTimePay = overTimeBonus(overTimeTenths);
  return straightPay() + overTimePay;
}

private int overTimeBonus(int overTimeTenths) {
  double bonus = 0.5 * getTenthRate() * overTimeTenths;
  return (int) Math.round(bonus);
}
```

Bien entendu, les sélecteurs ne sont pas toujours des booléens. Ils peuvent prendre la forme d'énumérations, d'entiers ou de tout autre type d'argument utilisé pour choisir le comportement de la fonction. En général, il est préférable d'écrire plusieurs fonctions plutôt que passer un code qui définit le comportement d'une fonction.

G16 : intentions obscures

Nous voulons que le code soit aussi expressif que possible. Les expressions à rallonge, la notation hongroise et les nombres magiques masquent les intentions de l'auteur. Par exemple, voici une version possible de la fonction `overTimePay` :

```
public int m_otCalc() {
  return iThsWkd * iThsRte +
    (int) Math.round(0.5 * iThsRte *
      Math.max(0, iThsWkd - 400)
    );
}
```

Elle est sans doute courte et concise, mais elle est virtuellement incompréhensible. Pour faciliter la tâche du lecteur, il est préférable de prendre le temps d'exprimer clairement les intentions du code.

G17 : responsabilité mal placée

Parmi les décisions les plus importantes qu'il doit prendre, le développeur de logiciels doit choisir l'emplacement de son code. Par exemple, où placer la constante `PI` ? Doit-elle rejoindre la classe `Math` ? N'appartiendrait-elle pas à la classe `Trigonometry` ? Ou bien à la classe `Circle` ?

Le principe de moindre surprise intervient également dans cette situation. Le code doit être placé là où le lecteur s'attend naturellement à le trouver. La place de la constante `PI` est aux côtés des fonctions trigonométriques. La constante `OVERTIME RATE` doit être déclarée dans la classe `HourlyPayCalculator`.

Parfois, nous plaçons une fonctionnalité de manière "astucieuse". Nous l'intégrons à la fonction qui nous arrange, mais qui n'est pas nécessairement celle qu'aurait imaginée le lecteur. Supposons, par exemple, que nous devions afficher un rapport avec le nombre total d'heures de travail d'un employé. Nous pouvons additionner toutes ces heures dans le code qui affiche le rapport ou nous pouvons gérer un total dans le code qui reçoit les cartes de pointage.

Pour prendre la bonne décision, il est possible de s'appuyer sur les noms des fonctions. Supposons que notre module de création du rapport possède une fonction nommée `getTotalHours`. Supposons également que le module qui accepte les cartes de pointage contienne une fonction `saveTimeCard`. D'après les noms de ces deux fonctions, laquelle doit calculer le nombre total d'heures ? La réponse est évidente.

Parfois, des questions de performances justifient le calcul du total au moment où les cartes de pointage sont reçues, non lorsque le rapport est affiché. Pas de problème, mais les noms des fonctions doivent refléter ce fonctionnement. Par exemple, le module de gestion des cartes de pointage doit proposer une fonction `computeRunningTotalOf Hours`.

G18 : méthodes statiques inappropriées

`Math.max(double a, double b)` est une bonne méthode statique. Elle n'opère sur aucune instance ; il serait évidemment stupide de devoir écrire `new Math().max(a,b)` ou même `a.max(b)`. Toutes les données utilisées par `max` proviennent de ces deux argu-

ments, non d'un objet "propriétaire". Qui plus est, il n'y a quasiment *aucune chance* que nous voulions une version polymorphe de `Math.max`.

Il nous arrive cependant d'écrire des fonctions statiques qui ne devraient pas l'être. Prenons l'exemple suivant :

```
HourlyPayCalculator.calculatePay(employee, overtimeRate)
```

Cette fonction statique semble raisonnable. Elle n'opère sur aucun objet particulier et prend toutes ses données dans ses arguments. Toutefois, il existe une possibilité raisonnable que cette fonction ait besoin d'être polymorphe. Nous pourrions souhaiter implémenter différents algorithmes de calcul des heures payées, par exemple `OvertimeHourlyPayCalculator` et `StraightTimeHourlyPayCalculator`. Dans ce cas, la fonction ne doit pas être statique. Elle doit être une fonction membre non statique de la classe `Employee`.

De manière générale, vous devez préférer les méthodes non statiques aux méthodes statiques. En cas de doute, optez pour une fonction non statique. Si vous voulez réellement qu'une fonction soit statique, vérifiez qu'il n'y a aucune chance qu'elle ait un jour un comportement polymorphe.

G19 : utiliser des variables explicatives

Kent Beck a écrit sur ce sujet dans son formidable livre *Smalltalk Best Practice Patterns* [Beck97, p. 108], et plus récemment dans son ouvrage tout aussi formidable *Implementation Patterns* [Beck07]. Pour qu'un programme soit lisible, l'une des solutions les plus performantes consiste à décomposer les calculs en valeurs intermédiaires représentées par des variables aux noms significatifs.

Prenons l'exemple suivant extrait de FitNesse :

```
Matcher match = headerPattern.matcher(line);
if(match.find())
{
  String key = match.group(1);
  String value = match.group(2);
  headers.put(key.toLowerCase(), value);
}
```

La simple utilisation de variables explicatives permet de comprendre que le premier groupe qui correspond représente la *clé* (`key`) et que le second groupe représente la *valeur* (`value`).

Il est difficile d'en faire trop. En général, il est préférable d'utiliser trop de variables explicatives que pas assez. Il est étonnant de voir comment un module opaque peut soudainement devenir transparent par une simple décomposition des calculs en valeurs intermédiaires aux noms parfaitement choisis.

G20 : les noms des fonctions doivent indiquer leur rôle

Prenons le code suivant :

```
Date newDate = date.add(5);
```

Doit-on supposer qu'il ajoute cinq jours à la date ? Ou bien s'agit-il de semaines ou d'heures ? L'instance `date` est-elle modifiée ou la fonction retourne-t-elle simplement une nouvelle instance de `Date` sans toucher à l'ancienne ? *En se fondant sur l'appel, il est impossible de dire précisément le rôle de la fonction*.

Si la fonction ajoute cinq jours à la date et modifie `date`, elle doit se nommer `addDaysTo` ou `increaseByDays`. En revanche, si elle retourne une nouvelle date qui correspond à cinq jours plus tard, mais sans modifier l'instance `date`, elle doit se nommer `daysLater` ou `daysSince`.

S'il faut examiner l'implémentation ou la documentation de la fonction pour connaître son rôle, cela signifie qu'un meilleur nom doit être trouvé ou que la fonctionnalité doit être déplacée dans des fonctions ayant des noms appropriés.

G21 : comprendre l'algorithme

Bien souvent, du code très amusant est écrit car le programmeur ne prend pas le temps de comprendre l'algorithme. Il s'attache à créer quelque chose d'opérationnel en ajoutant suffisamment d'instructions `if` et d'indicateurs, sans envisager de s'arrêter pour comprendre ce qui se passe réellement.

La programmation ressemble souvent à une exploration. Vous *pensez* connaître le bon algorithme de mise en œuvre, mais vous finissez par le bricoler, en le retouchant ici et là, jusqu'à ce qu'il "fonctionne". Comment savez-vous qu'il "fonctionne" ? Parce qu'il passe les cas de test, pensez-vous.

Cette approche n'a rien de faux. Il s'agit bien souvent de la seule manière d'arriver à une fonction qui implémente la tâche imaginée. Cependant, il ne suffit pas de laisser les guillemets autour du mot "fonctionne".

Avant de considérer que l'écriture d'une fonction est terminée, vérifiez que vous *comprenez* son fonctionnement. Elle ne doit pas simplement passer tous les tests. Vous devez *savoir*[4] que la solution est correcte.

4. Il y a une différence entre savoir comment le code fonctionne et savoir que l'algorithme réalise le travail demandé. Ne pas être certain de l'adéquation d'un algorithme fait souvent partie de la vie. Ne pas être certain du fonctionnement de son code n'est que de la paresse.

Pour le savoir, l'une des meilleures solutions consiste à remanier la fonction de manière à en obtenir une version si propre et si expressive qu'il devient *évident* qu'elle fonctionne.

G22 : rendre physiques les dépendances logiques

Si un module dépend d'un autre, cette dépendance doit être physique, non simplement logique. Le module dépendant ne doit faire aucune supposition (autrement dit, dépendance logique) sur le module dont il dépend. À la place, il doit demander explicitement à ce module toutes les informations qui lui sont nécessaires.

Imaginons par exemple que vous écriviez une fonction qui affiche un rapport des heures travaillées par des employés. Une classe nommée `HourlyReporter` collecte toutes les données sous une forme commode, puis les passe à `HourlyReportFormatter` pour leur affichage (voir Listing 17.1).

Listing 17.1 : `HourlyReporter.java`

```
public class HourlyReporter {
  private HourlyReportFormatter formatter;
  private List<LineItem> page;
  private final int PAGE_SIZE = 55;

  public HourlyReporter(HourlyReportFormatter formatter) {
    this.formatter = formatter;
    page = new ArrayList<LineItem>();
  }

  public void generateReport(List<HourlyEmployee> employees) {
    for (HourlyEmployee e : employees) {
      addLineItemToPage(e);
      if (page.size() == PAGE_SIZE)
        printAndClearItemList();
    }
    if (page.size() > 0)
      printAndClearItemList();
  }

  private void printAndClearItemList() {
    formatter.format(page);
    page.clear();
  }

  private void addLineItemToPage(HourlyEmployee e) {
    LineItem item = new LineItem();
    item.name = e.getName();
    item.hours = e.getTenthsWorked() / 10;
    item.tenths = e.getTenthsWorked() % 10;
    page.add(item);
  }
```

```
    public class LineItem {
      public String name;
      public int hours;
      public int tenths;
    }
  }
```

Ce code présente une dépendance logique qui n'a pas été rendue physique. Pouvez-vous l'identifier ? Il s'agit de la constante `PAGE SIZE`. Pourquoi la classe `HourlyReporter` devrait-elle connaître la taille de la page ? La taille d'une page est de la responsabilité de la classe `HourlyReportFormatter`.

Le fait de déclarer `PAGE SIZE` dans `HourlyReporter` constitue une responsabilité mal placée [G17] qui conduit cette classe à supposer qu'elle a connaissance de la taille voulue pour la page. Une telle supposition est une dépendance logique. `HourlyReporter` dépend du fait que `HourlyReportFormatter` est en mesure de prendre en charge des pages de taille égale à 55. Si une implémentation de `HourlyReportFormatter` n'accepte pas cette taille, une erreur se produit.

Pour que cette dépendance devienne physique, nous pouvons créer une nouvelle méthode `getMaxPageSize()` dans `HourlyReportFormatter`. Ensuite, `HourlyReporter` appellera cette fonction au lieu d'utiliser la constante `PAGE SIZE`.

G23 : préférer le polymorphisme aux instructions *if/else* ou *switch/case*

Étant donné le sujet du Chapitre 6, cette recommandation pourrait sembler étrange. En effet, j'ai indiqué dans ce chapitre que les instructions `switch` pouvaient être appropriées dans les parties du système où l'ajout de nouvelles fonctions était plus probable que l'ajout de nouveaux types.

Tout d'abord, la plupart des programmeurs emploient les instructions `switch` parce qu'elles constituent une solution évidente, non parce qu'elles sont la bonne solution. L'objectif de cette heuristique est de vous rappeler qu'il faut envisager le polymorphisme avant l'instruction `switch`.

Par ailleurs, les cas de fonctions plus volatiles que les types sont relativement rares. Par conséquent, toute instruction `switch` doit être considérée comme suspecte.

Je suis la règle "UN SWITCH" suivante : *il ne doit pas y avoir plus d'une instruction `switch` pour un type donné de sélection. Les cas de cette instruction `switch` doivent créer des objets polymorphes qui prennent la place d'autres instructions `switch` dans le reste du système*.

G24 : respecter des conventions standard

Chaque équipe doit respecter un standard de codage fondé sur les normes industrielles reconnues. Ce standard doit préciser certains points, comme où déclarer les variables d'instance, comment nommer les classes, les méthodes et les variables, où placer les accolades, etc. L'équipe ne doit pas avoir besoin d'un document pour décrire ces conventions car son code sert d'exemple.

Tous les membres de l'équipe doivent respecter ces conventions. Autrement dit, chacun d'eux doit être suffisamment mûr pour réaliser que l'emplacement des accolades n'a absolument aucune importance à partir du moment où tout le monde s'accorde sur cet emplacement.

Si vous voulez connaître mes conventions, vous les trouverez dans le code remanié des Listings B.7 à B.14 à l'Annexe B.

G25 : remplacer les nombres magiques par des constantes nommées

Il s'agit probablement de l'une des plus anciennes règles du développement de logiciels. Je me souviens l'avoir lue à la fin des années 1960 en introduction des manuels COBOL, FORTRAN et PL/1. En général, il est contre-indiqué d'employer directement des nombres dans le code. Ils doivent être cachés derrière des constantes aux noms parfaitement choisis.

Par exemple, le nombre 86 400 doit être caché derrière la constante `SECONDS PER DAY`. Si vous affichez 55 lignes par page, le nombre 55 doit être caché derrière la constante `LINES PER PAGE`.

Certaines constantes sont si faciles à reconnaître qu'elles n'ont pas besoin d'être cachées derrière une constante nommée si elles sont employées conjointement à du code très descriptif. Par exemple :

```
double milesWalked = feetWalked/5280.0;
int dailyPay = hourlyRate * 8;
double circumference = radius * Math.PI * 2;
```

Avons-nous vraiment besoin des constantes `FEET PER MILE`, `WORK HOURS PER DAY` et `TWO` dans les exemples précédents ? Bien évidemment, le dernier cas est absurde. Il s'agit de formules dans lesquelles les constantes doivent simplement être écrites comme des nombres. Vous pourriez tergiverser quant au cas de `WORK HOURS PER DAY` car les lois ou les conventions pourraient évoluer. En revanche, la formule se lit si bien avec le 8 qu'il est délicat d'ajouter dix-sept caractères supplémentaires à la charge du lecteur. Pour `FEET PER MILE`, le nombre 5 280 est tellement connu et unique que le lecteur (anglophone) le reconnaîtrait même s'il se trouvait seul sur une page sans aucun contexte.

Des constantes comme 3.141592653589793 sont également très connues et facilement reconnaissables. Cependant, les risques d'erreur sont trop importants pour les laisser sous cette forme. Chaque fois que quelqu'un voit 3.1415927535890793, il sait qu'il s'agit de π et n'y prête donc pas une attention particulière. (Avez-vous remarqué l'erreur ?) Par ailleurs, nous ne souhaitons pas que les gens utilisent 3.14, 3.14159 ou 3.142. Par conséquent, c'est une bonne chose que `Math.PI` soit déjà défini pour nous.

L'expression "nombre magique" ne s'applique pas uniquement aux nombres. Elle concerne également tout élément dont la valeur n'est pas autodescriptive. Par exemple :

```
assertEquals(7777, Employee.find("John Doe").employeeNumber());
```

Cette assertion contient deux nombres magiques. Le premier est évidemment 7777, mais sa signification n'est pas évidente. Le second est `"John Doe"` et, encore une fois, les intentions ne sont pas claires.

En réalité `"John Doe"` est le nom de l'employé n° 7777 dans une base de données de test très connue au sein de notre équipe. Tous les membres de l'équipe savent que cette base de données contient plusieurs employés, avec des valeurs et des attributs parfaitement connus. Par ailleurs, `"John Doe"` est également le seul salarié horaire dans cette base de données de test. Par conséquent, le test devrait être écrit de la manière suivante :

```
assertEquals(
  HOURLY_EMPLOYEE_ID,
  Employee.find(HOURLY_EMPLOYEE_NAME).employeeNumber());
```

G26 : être précis

Supposer que la première correspondance est la *seule* correspondance d'une requête est probablement naïf. Utiliser des nombres en virgule flottante pour représenter les devises se révèle pratiquement criminel. Éviter la gestion des verrous et/ou des transactions parce que les mises à jour concurrentes semblent improbables est au mieux un peu fou. Déclarer une variable `ArrayList` alors qu'un `List` est attendu est exagérément contraignant. Rendre toutes les variables `protected` par défaut n'est pas suffisamment contraignant.

Lorsque vous prenez une décision dans votre code, faites en sorte d'être *précis*. Vous devez savoir pourquoi vous l'avez prise et comment traiter les exceptions. Ne lésinez pas sur la précision de vos choix. Si vous décidez d'invoquer une fonction qui peut retourner `null`, assurez-vous de vérifier ce cas. Si vous pensez que votre requête concerne un seul enregistrement dans la base de données, faites en sorte que votre code vérifie qu'il n'y ait pas d'autres enregistrements. Si vous devez manipuler des devises, employez des nombres entiers[5] et appliquez les arrondis appropriés. S'il existe une

5. Ou, mieux encore, une classe `Money` qui utilise des entiers.

possibilité de mises à jour concurrentes, assurez-vous d'implémenter un mécanisme de verrouillage.

Les ambiguïtés et les imprécisions dans le code sont le résultat de désaccords ou d'une paresse. Dans tous les cas, elles doivent être éliminées.

G27 : privilégier la structure à une convention

Appliquez les choix de conception à l'aide d'une structure plutôt qu'une convention. Les conventions de nommage sont bonnes, mais elles sont inférieures aux structures qui, elles, imposent la conformité. Par exemple, les constructions `switch`/`case` avec des énumérations bien nommées sont inférieures aux classes de base avec des méthodes abstraites. Rien n'oblige le programmeur à implémenter à chaque fois l'instruction `switch`/`case` de la même manière. En revanche, les classes de base imposent aux classes concrètes d'implémenter toutes les méthodes abstraites.

G28 : encapsuler les expressions conditionnelles

La logique booléenne est déjà suffisamment complexe à comprendre pour souhaiter ne pas la rencontrer dans le contexte d'une instruction `if` ou `while`. Extrayez des fonctions qui expliquent les intentions de l'expression conditionnelle.

Par exemple, vous devez préférer

```
if (shouldBeDeleted(timer))
```

à

```
if (timer.hasExpired() && !timer.isRecurrent())
```

G29 : éviter les expressions conditionnelles négatives

Les négations sont plus difficiles à comprendre que les affirmations. Par conséquent, lorsque c'est possible, les expressions conditionnelles doivent être données sous une forme positive. Par exemple, vous devez préférer

```
if (buffer.shouldCompact())
```

à

```
if (!buffer.shouldNotCompact())
```

G30 : les fonctions doivent faire une seule chose

Nous sommes souvent tentés de créer des fonctions qui contiennent plusieurs sections réalisant une suite d'opérations. Les fonctions de ce type font plusieurs choses et doivent être converties en plusieurs fonctions plus petites, chacune réalisant une seule chose.

Prenons la fonction suivante :

```
void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
    }
  }
}
```

Ce bout de code réalise trois choses. Il parcourt la liste des employés, vérifie si chacun d'eux doit être payé, puis paye l'employé. Il serait préférable de l'écrire de la manière suivante :

```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```

Chacune de ces fonctions effectue une seule chose (voir la section "Faire une seule chose" au Chapitre 3).

G31 : couplages temporels cachés

Les couplages temporels sont souvent nécessaires, mais ils ne doivent pas être cachés. Les arguments des fonctions doivent être structurés de manière que leur ordre dans les appels soit évident. Prenons le code suivant :

```
public class MoogDiver {
  Gradient gradient;
  List<Spline> splines;

  public void dive(String reason) {
    saturateGradient();
    reticulateSplines();
    diveForMoog(reason);
  }
  ...
}
```

L'ordre des trois fonctions est important. Il faut tout d'abord invoquer `saturateGradient` avant de pouvoir appeler `reticulateSplines`, et alors seulement il est possible d'invoquer `diveForMoog`. Malheureusement, le code n'impose pas ce couplage tempo-

rel. Un autre programmeur pourrait invoquer `reticulateSplines` avant `saturateGradient`, conduisant à une exception `UnsaturatedGradientException`. Voici une meilleure solution :

```
public class MoogDiver {
  Gradient gradient;
  List<Spline> splines;

  public void dive(String reason) {
    Gradient gradient = saturateGradient();
    List<Spline> splines = reticulateSplines(gradient);
    diveForMoog(splines, reason);
  }
  ...
}
```

Le couplage temporel est exposé en créant une séquence d'appels à la manière d'une "chaîne de seaux". Chaque fonction produit un résultat dont la suivante a besoin. Il n'existe donc aucune manière sensée de les invoquer dans le désordre.

Vous pourriez critiquer l'augmentation de la complexité des fonctions induite par cette approche. Cependant, la complexité syntaxique supplémentaire permet de révéler la réelle complexité temporelle de la situation.

Vous remarquerez que j'ai laissé les variables d'instance à leur place. Je suppose qu'elles sont requises par des méthodes privées de la classe. Néanmoins, je veux que les arguments rendent explicite le couplage temporel.

G32 : ne pas être arbitraire

La structure de votre code doit se justifier, et la raison doit en être communiquée par cette structure. Si une structure semble arbitraire, les autres programmeurs pourraient se croire autorisés à la modifier. Si une structure paraît cohérente tout au long du système, les autres programmeurs l'emploieront et conserveront les conventions. Par exemple, j'ai récemment intégré des modifications dans FitNesse et découvert que l'un des contributeurs avait fait la chose suivante :

```
public class AliasLinkWidget extends ParentWidget
{
  public static class VariableExpandingWidgetRoot {
    ...

  ...
}
```

La classe `VariableExpandingWidgetRoot` n'a aucunement besoin de se trouver dans la portée de `AliasLinkWidget`. Par ailleurs, d'autres classes sans rapport utilisent `AliasLinkWidget.VariableExpandingWidgetRoot`. Ces classes n'ont aucunement besoin de connaître `AliasLinkWidget`.

Le programmeur a peut-être placé `VariableExpandingWidgetRoot` dans `AliasWidget` pour des raisons de commodité, ou peut-être qu'il pensait qu'elle devait absolument se trouver dans la portée de `AliasWidget`. Quelle que soit la raison, le résultat est arbitraire. Les classes publiques qui ne sont pas des classes utilitaires d'une autre classe ne doivent pas se trouver dans la portée d'une autre classe. Par convention, elles doivent être publiques au niveau supérieur de leur paquetage.

G33 : encapsuler les conditions aux limites

Les conditions aux limites sont difficiles à suivre. Leur traitement doit se trouver en un seul endroit. Ne les laissez pas se diffuser partout dans le code. Il ne faut pas que des nuées de `+1` et de `1` s'éparpillent çà et là. Prenons le simple exemple suivant tiré de FitNesse :

```
if(level + 1 < tags.length)
{
  parts = new Parse(body, tags, level + 1, offset + endTag);
  body = null;
}
```

Vous aurez remarqué que `level+1` apparaît deux fois. Il s'agit d'une condition aux limites qui doit être encapsulée dans une variable nommée, par exemple, `nextLevel`.

```
int nextLevel = level + 1;
if(nextLevel < tags.length)
{
  parts = new Parse(body, tags, nextLevel, offset + endTag);
  body = null;
}
```

G34 : les fonctions doivent descendre d'un seul niveau d'abstraction

Les instructions d'une fonction doivent toutes se trouver au même niveau d'abstraction, qui doit être un niveau en dessous de l'opération décrite par le nom de la fonction. Il s'agit sans doute de l'heuristique la plus difficile à interpréter et à respecter. Bien que l'idée soit simple, les êtres humains parviennent bien trop facilement à jongler avec des niveaux d'abstraction différents. Prenons, par exemple, le code suivant extrait de FitNesse :

```
public String render() throws Exception
{
  StringBuffer html = new StringBuffer("<hr");
  if(size > 0)
    html.append(" size=\"").append(size + 1).append("\"");
  html.append(">");

  return html.toString();
}
```

Après une courte réflexion, vous comprenez ce qui se passe. Cette fonction construit la balise HTML qui trace une ligne horizontale au travers de la page. La hauteur de la ligne est indiquée par la variable `size`.

En examinant plus attentivement le code, nous constatons que la méthode mélange au moins deux niveaux d'abstraction : une ligne horizontale possède une taille, et la syntaxe de la balise `HR` elle-même. Ce code provient du module `HruleWidget` de FitNesse. Il détecte une suite de quatre tirets ou plus et la convertit en une balise `HR` appropriée. La taille dépend du nombre de tirets.

Voici comment j'ai légèrement remanié ce code. J'ai changé le nom du champ `size` afin que son véritable rôle soit plus clair. Il contient le nombre de tirets supplémentaires.

```
public String render() throws Exception
{
  HtmlTag hr = new HtmlTag("hr");
  if (extraDashes > 0)
    hr.addAttribute("size", hrSize(extraDashes));
  return hr.html();
}

private String hrSize(int height)
{
  int hrSize = height + 1;
  return String.format("%d", hrSize);
}
```

Cette modification sépare parfaitement les deux niveaux d'abstraction. La fonction `render` construit simplement une balise `HR`, sans rien connaître de la syntaxe HTML correspondante. Le module `HtmlTag` s'occupe des détails pénibles de syntaxe.

En réalisant cette intervention, j'ai découvert une erreur subtile. Le code d'origine n'ajoutait pas la barre oblique finale dans la balise `HR`, comme l'exige la norme XHTML. Autrement dit, il génère `<hr>` à la place de `<hr/>`. Le module `HtmlTag` a depuis longtemps été modifié pour respecter la norme XHTML.

La séparation des niveaux d'abstraction fait partie des objectifs les plus importants du remaniement et représente l'un des plus difficiles à atteindre. Prenons le code suivant comme exemple. Il s'agit de ma première tentative de séparation des niveaux d'abstraction dans la méthode `HruleWidget.render`.

```
public String render() throws Exception
{
  HtmlTag hr = new HtmlTag("hr");
  if (size > 0) {
    hr.addAttribute("size", ""+(size+1));
  }
  return hr.html();
}
```

Mon but était, à ce stade, de créer la séparation nécessaire et de faire en sorte que les tests réussissent. J'ai facilement atteint cet objectif, mais la fonction résultante mélangeait encore les niveaux d'abstraction. Dans ce cas, il s'agissait de la construction de la balise `HR` et de l'interprétation et de la mise en forme de la variable `size`. Lorsque nous découpons une fonction en suivant des lignes d'abstraction, nous découvrons fréquemment de nouvelles lignes d'abstraction cachées par la structure précédente.

G35 : conserver les données configurables à des niveaux élevés

Si une constante, comme une valeur par défaut ou une valeur de configuration, est connue et attendue à un haut niveau d'abstraction, ne l'enterrez pas dans une fonction de bas niveau. Exposez-la sous forme d'un argument de cette fonction de bas niveau, invoquée à partir de la fonction de haut niveau. Prenons le code suivant, extrait de FitNesse :

```
public static void main(String[] args) throws Exception
{
  Arguments arguments = parseCommandLine(args);
    ...
}

public class Arguments
{
  public static final String DEFAULT_PATH = ".";
  public static final String DEFAULT_ROOT = "FitNesseRoot";
  public static final int DEFAULT_PORT = 80;
  public static final int DEFAULT_VERSION_DAYS = 14;
  ...
}
```

Les arguments de la ligne de commande sont analysés dès la première ligne d'exécution de FitNesse. Les valeurs par défaut de ces arguments sont fixées au début de la classe `Argument`. Il n'est pas nécessaire de plonger dans les niveaux inférieurs du système pour trouver des instructions comme la suivante :

```
if (arguments.port == 0) // Utiliser 80 par défaut.
```

Les constantes de configuration résident à un niveau très élevé et sont faciles à modifier. Elles sont transmises au reste de l'application. Les niveaux inférieurs de l'application ne possèdent pas les valeurs de ces constantes.

G36 : éviter la navigation transitive

En général, nous ne souhaitons pas qu'un module en sache trop sur ses collaborateurs. Plus précisément, si A collabore avec B et si B collabore avec C, nous ne voulons pas que les modules qui utilisent A connaissent C. Par exemple, l'instruction `a.getB().getC().doSomething();` n'est pas souhaitable.

Nous appelons cela loi de Déméter. Les *Pragmatic Programmers* emploient l'expression "Écrire du code timide" [PRAG, p. 138]. Dans tous les cas, l'idée est de s'assurer que les modules ne connaissent que leurs collaborateurs immédiats et ne sachent pas comment parcourir l'intégralité du système.

Si de nombreux modules utilisent l'instruction `a.getB().getC()` sous une forme ou sous une autre, il sera difficile de modifier la conception et l'architecture pour interposer un module Q entre B et C. En effet, il faudra trouver chaque occurrence de `a.getB().getC()` et la convertir en `a.getB().getQ().getC()`. C'est ainsi que les architectures deviennent rigides. Un trop grand nombre de modules en savent trop sur l'architecture.

À la place, nous voulons que nos collaborateurs directs offrent tous les services dont nous avons besoin. Nous ne devons pas être obligés de parcourir le graphe des objets du système pour trouver la méthode à invoquer. Nous voulons simplement pouvoir écrire :

```
myCollaborator.doSomething()
```

Java

J1 : éviter les longues listes d'importations grâce aux caractères génériques

Si vous utilisez deux classes ou plus d'un paquetage, importez alors l'intégrité du paquetage de la manière suivante :

```
import package.*;
```

Les longues listes d'importations effraient le lecteur. Nous ne voulons pas encombrer le début de nos modules avec 80 lignes d'importations. À la place, nous voulons que les importations communiquent de manière concise la liste des paquetages avec lesquels nous collaborons.

Les importations précises constituent des dépendances rigides, contrairement à l'usage des caractères génériques. Si vous importez une classe spécifique, celle-ci *doit* exister. En revanche, si vous importez un paquetage, aucune classe précise n'est imposée. L'instruction d'importation ajoute simplement le paquetage au chemin suivi pour la recherche des noms. Ces importations ne créent donc pas de véritables dépendances et permettent de garder des modules moins fortement couplés.

Cependant, une longue liste d'importations spécifiques se révèle parfois utile. Par exemple, si vous manipulez du code ancien et souhaitez connaître les classes pour lesquelles vous devez construire des simulacres et des bouchons, vous pouvez parcourir la liste des importations spécifiques et obtenir les noms qualifiés de toutes ces classes, pour ensuite mettre en place les bouchons appropriés. Toutefois, cette utilisation est

plutôt rare. Par ailleurs, la plupart des IDE modernes permettent, en une seule commande, de convertir les importations globales en une liste d'importations spécifiques. Par conséquent, même dans le cas du code ancien, il est préférable d'importer des paquetages entiers.

Les importations globales conduisent parfois à des conflits et à des ambiguïtés de noms. Deux classes de même nom, mais issues de paquetages différents, doivent être importées de manière précise ou être qualifiées lors de leur utilisation. Cela peut être ennuyeux, mais cette situation est suffisamment rare pour préférer les importations globales aux importations spécifiques.

J2 : ne pas hériter des constantes

J'ai rencontré plusieurs fois cette manière de faire et elle m'a toujours gêné. Un programmeur place des constantes dans une interface, puis accède à ces constantes en dérivant de cette interface. Prenons le code suivant :

```
public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  private double hourlyRate;

  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
      hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  ...
}
```

D'où viennent les constantes `TENTHS PER WEEK` et `OVERTIME RATE` ? Peut-être de la classe `Employee` ; voyons voir :

```
public abstract class Employee implements PayrollConstants {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
```

Non, elles ne sont pas là. Dans ce cas, d'où peuvent-elles bien venir ? Examinez attentivement la classe `Employee`. Elle implémente `PayrollConstants`.

```
public interface PayrollConstants {
  public static final int TENTHS_PER_WEEK = 400;
  public static final double OVERTIME_RATE = 1.5;
}
```

Cette pratique est affreuse ! Les constantes se dissimulent au sommet de la hiérarchie d'héritage. L'héritage ne doit pas servir à contourner les règles de portée du langage. À la place, il faut employer une importation statique.

```
import static PayrollConstants.*;

public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  private double hourlyRate;

  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
      hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  ...
}
```

J3 : constantes contre énumérations

Puisque les `enum` font à présent partie du langage (Java 5), servez-vous-en ! Oubliez l'ancienne méthode `public static final int`. La signification des `int` peut disparaître, contrairement à celle des `enum` qui appartiennent à une énumération nommée.

Par ailleurs, examinez attentivement la syntaxe des `enum`. Ils peuvent posséder des méthodes et des champs, ce qui en fait des outils très puissants offrant une expressivité et une souplesse supérieures aux `int`. Pour vous en convaincre, étudiez la variante suivante du code de traitement des payes :

```
public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  HourlyPayGrade grade;

  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
      grade.rate() * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  ...
}

public enum HourlyPayGrade {
  APPRENTICE {
    public double rate() {
      return 1.0;
    }
  },
  LEUTENANT_JOURNEYMAN {
    public double rate() {
      return 1.2;
    }
  },
```

```
    JOURNEYMAN {
      public double rate() {
        return 1.5;
      }
    },
    MASTER {
      public double rate() {
        return 2.0;
      }
    };

    public abstract double rate();
  }
```

Noms

N1 : choisir des noms descriptifs

Ne soyez pas trop rapide dans le choix d'un nom. Assurez-vous qu'il est descriptif. N'oubliez pas que le sens des noms a tendance à dériver avec l'évolution du logiciel. Par conséquent, vous devez vérifier fréquemment la pertinence des noms retenus.

Il ne s'agit pas simplement d'une recommandation de "bien-être". Les noms représentent 90 % de la lisibilité d'un logiciel. Vous devez prendre le temps de faire des choix judicieux et de conserver des noms pertinents. Ils sont trop importants pour être pris à la légère.

Examinez le code suivant. Quelle est sa fonction ? Si je vous montre ce code avec des noms bien choisis, vous le comprendrez parfaitement. En revanche, sous cette forme, il ne s'agit que d'un amoncellement de symboles et de nombres magiques.

```
public int x() {
    int q = 0;
    int z = 0;
    for (int kk = 0; kk < 10; kk++) {
      if (l[z] == 10)
      {
        q += 10 + (l[z + 1] + l[z + 2]);
        z += 1;
      }
      else if (l[z] + l[z + 1] == 10)
      {
        q += 10 + l[z + 2];
        z += 2;
      } else {
        q += l[z] + l[z + 1];
        z += 2;
      }
    }
    return q;
  }
```

Voici comment le code aurait dû être écrit :

```
public int score() {
  int score = 0;
  int frame = 0;
  for (int frameNumber = 0; frameNumber < 10; frameNumber++) {
    if (isStrike(frame)) {
      score += 10 + nextTwoBallsForStrike(frame);
      frame += 1;
    } else if (isSpare(frame)) {
      score += 10 + nextBallForSpare(frame);
      frame += 2;
    } else {
      score += twoBallsInFrame(frame);
      frame += 2;
    }
  }
  return score;
}
```

Cet extrait est en réalité moins complet que le précédent. Vous pouvez néanmoins déduire immédiatement ce qu'il tente de réaliser et ainsi écrire les fonctions manquantes. Les nombres ne sont plus magiques et la structure de l'algorithme est parfaitement descriptive.

Les noms bien choisis ont le pouvoir d'ajouter une description à la structure du code. Ainsi, le lecteur sait ce qu'il doit attendre des autres fonctions du module. Vous pouvez déduire l'implémentation de `isStrike()` en examinant le code précédent. Si vous lisez le code de la méthode `isStrike`, il réalise "pratiquement ce que vous attendiez"[6].

```
private boolean isStrike(int frame) {
  return rolls[frame] == 10;
}
```

N2 : choisir des noms au niveau d'abstraction adéquat

Les noms choisis doivent non pas communiquer une implémentation, mais refléter le niveau d'abstraction de la classe ou de la fonction. Ce n'est pas facile. Une fois encore, les êtres humains sont bien trop aptes à mélanger les niveaux d'abstraction. À chaque examen de votre code, vous trouverez probablement des variables dont les noms se placent à un niveau trop bas. Vous devez saisir l'opportunité de modifier ces noms dès que vous les identifiez. L'amélioration de la lisibilité du code requiert un effort continu. Prenons le cas de l'interface `Modem` :

```
public interface Modem {
  boolean dial(String phoneNumber);
  boolean disconnect();
  boolean send(char c);
```

6. Voir le propos de Ward Cunningham à la page 13.

```
    char recv();
    String getConnectedPhoneNumber();
}
```

À première vue, elle semble parfaite. Toutes les fonctions semblent pertinentes. C'est effectivement le cas pour de nombreuses applications. Cependant, certaines applications pourraient contrôler des modems qui n'ont pas besoin de composer un numéro pour se connecter, par exemple les modems câble ou ADSL actuels pour l'accès à Internet. D'autres modems pourraient se connecter par l'envoi d'un numéro de port à un commutateur au travers d'une connexion USB. Dans ce cas, il est clair que la notion de numéros de téléphone se trouve au mauvais niveau d'abstraction. Voici donc un meilleur choix de noms :

```
public interface Modem {
  boolean connect(String connectionLocator);
  boolean disconnect();
  boolean send(char c);
  char recv();
  String getConnectedLocator();
}
```

À présent, les noms n'ont aucun rapport avec un quelconque numéro de téléphone. Les méthodes peuvent toujours être employées avec des numéros de téléphone, mais également avec d'autres mécanismes de connexion.

N3 : employer si possible une nomenclature standard

Les noms sont plus faciles à comprendre lorsqu'ils se fondent sur une convention ou un usage établi. Par exemple, si vous utilisez le motif DÉCORATEUR, vous devez employer le mot `Decorator` dans les noms des classes correspondantes. Ainsi, `AutoHangupModem Decorator` pourrait être le nom d'une classe qui "décore" un `Modem` en apportant la possibilité de raccrocher automatiquement à la fin d'une session.

Les motifs ne sont pas les seuls standards. En Java, par exemple, les fonctions qui convertissent des objets en chaînes de caractères se nomment souvent `toString`. Il est préférable de suivre ce genre de conventions que d'inventer les vôtres.

Très souvent, les équipes définissent leur propre système standard de nommage dans un projet précis. Eric Evans appelle cela le *langage commun* au projet [DDD]. Le code doit employer systématiquement les termes de ce langage. En résumé, plus vous utilisez des noms ayant une signification spéciale pertinente dans le projet, plus il sera facile au lecteur de savoir ce dont parle le code.

N4 : noms non ambigus

Les noms choisis doivent indiquer de manière non ambiguë le rôle d'une fonction ou d'une variable. Examinons l'exemple suivant extrait de FitNesse :

```
private String doRename() throws Exception
{
  if(refactorReferences)
    renameReferences();
  renamePage();

  pathToRename.removeNameFromEnd();
  pathToRename.addNameToEnd(newName);
  return PathParser.render(pathToRename);
}
```

Le nom de cette fonction ne dit pas ce qu'elle fait, excepté en termes vagues et généraux. Ce problème est accentué par l'existence d'une fonction nommée `renamePage` à l'intérieur de la fonction nommée `doRename` ! Qu'indiquent les noms à propos du rôle différent des deux fonctions ? Absolument rien.

Cette fonction devrait se nommer `renamePageAndOptionallyAllReferences`. Ce nom peut vous sembler un peu long, et c'est bien le cas. Cependant, puisqu'il n'est employé qu'à un seul endroit dans le module, sa valeur d'explication contrebalance sa longueur.

N5 : employer des noms longs pour les portées longues

La longueur d'un nom doit être liée à sa portée. Vous pouvez choisir des noms de variables très courts lorsque leur portée est réduite, mais pour les longues portées vous devez employer des noms longs.

Les noms de variables comme `i` et `j` sont parfaits si leur portée s'étend sur cinq lignes. Examinons l'extrait suivant, tiré d'un ancien jeu de bowling :

```
private void rollMany(int n, int pins)
{
  for (int i=0; i<n; i++)
    g.roll(pins);
}
```

Il est parfaitement clair et serait plus difficile à comprendre si la variable `i` était remplacée par `rollCount`. *A contrario*, les variables et les fonctions dont les noms sont courts perdent leur signification avec l'éloignement. Par conséquent, plus la portée d'un nom est étendue, plus ce nom doit être long et précis.

N6 : éviter la codification

Les noms ne doivent pas contenir une information de type ou de portée. Les préfixes comme `m` et `f` sont superflus dans les environnements actuels. De même, codifier un projet et/ou un sous-système, par exemple avec `vis` (pour un système d'imagerie visuelle), constitue une source de distraction et de redondance. Une fois encore, les environnements actuels apportent toutes ces informations sans avoir besoin de triturer les noms. Ne polluez pas vos noms par une notation hongroise.

N7 : les noms doivent décrire les effets secondaires

Les noms doivent décrire tout ce qu'une fonction, une variable ou une classe est ou fait. Ne cachez pas des effets secondaires à l'aide d'un nom. N'utilisez pas un simple verbe pour décrire une fonction qui ne se limite pas à cette simple action. Par exemple, prenons le code suivant, extrait de TestNG :

```
public ObjectOutputStream getOos() throws IOException {
  if (m_oos == null) {
    m_oos = new ObjectOutputStream(m_socket.getOutputStream());
  }
  return m_oos;
}
```

Cette fonction ne se réduit pas à obtenir un "oos" : elle crée un `ObjectOutputStream` si ce n'est déjà fait. Elle devrait donc se nommer `createOrReturnOos`, par exemple.

Tests

T1 : tests insuffisants

Combien de tests doit contenir une suite ? Malheureusement, la métrique employée par de nombreux programmeurs est "cela devrait suffire". Une suite de tests doit contrôler tout ce qui peut être source de dysfonctionnement. Les tests restent insuffisants tant qu'il existe des conditions qui n'ont pas été explorées par les tests ou des calculs qui n'ont pas été validés.

T2 : utiliser un outil d'analyse de couverture

Les outils d'analyse de couverture signalent tous les trous existant dans votre stratégie de test. Ils permettent de trouver facilement les modules, les classes et les fonctions qui sont insuffisamment testés. La plupart des IDE fournissent une indication visuelle, en marquant en vert les lignes couvertes et en rouge celles non couvertes. Il est ainsi très rapide et très facile de trouver les instructions `if` ou `catch` dont le corps n'est pas vérifié.

T3 : ne pas omettre les tests triviaux

Ils sont faciles à écrire et leur valeur documentaire est plus élevée que leur coût de production.

T4 : un test ignoré est une interrogation sur une ambiguïté

Parfois, nous ne sommes pas certains des détails d'un comportement car les exigences sont floues. Nous pouvons exprimer nos interrogations concernant les exigences sous forme de tests mis en commentaire ou d'un test annoté avec le mot-clé `@Ignore`. Le choix entre ces deux solutions se fonde sur la nécessité pour le point ambigu de compiler ou non.

T5 : tester aux conditions limites

Faites particulièrement attention aux tests aux conditions limites. Bien souvent, la partie centrale d'un algorithme est juste, mais les extrémités ont été mal appréciées.

T6 : tester de manière exhaustive autour des bogues

Les bogues ont tendance à se rassembler. Lorsque vous découvrez un bogue dans une fonction, il est plus sage d'effectuer un test exhaustif de cette fonction. Vous constaterez probablement que le bogue n'était pas seul.

T7 : les motifs d'échec sont révélateurs

Vous pouvez parfois diagnostiquer un problème en recherchant des motifs dans la manière dont les cas de test échouent. Il s'agit d'un autre argument en faveur des cas de test aussi complets que possible. Arrangés de manière juste, les cas de test complets révèlent des motifs.

Supposons, par exemple, que vous ayez remarqué que tous les tests dont l'entrée dépasse cinq caractères échouent. Ou bien que tout test qui passe une valeur négative en second argument d'une fonction échoue. Parfois, le simple fait de voir le motif de rouge et de vert sur le rapport de test suffit à provoquer le déclic qui mène à la solution. Vous en trouverez un exemple intéressant dans la classe `SerialDate` remaniée au Chapitre 16.

T8 : les motifs dans la couverture des tests sont révélateurs

En examinant le code qui est ou n'est pas exécuté par les tests qui réussissent, vous trouverez des indications sur les raisons d'échec des autres tests.

T9 : les tests doivent être rapides

Un test lent est un test qui n'est pas exécuté. Lorsque les échéances se rapprochent, les tests lents sont retirés de la suite. Vous devez donc faire tout votre possible pour garder des tests rapides.

Conclusion

Cette liste d'heuristiques et d'indicateurs ne peut en aucun cas prétendre être complète. Je ne suis même pas certain qu'elle pourrait être complète. L'exhaustivité n'est sans doute pas l'objectif visé, car cette liste ne fait que proposer un système de valeurs.

Ce système de valeurs a été l'objectif et le sujet de cet ouvrage. Un code propre ne s'obtient pas en suivant un ensemble de règles. Vous ne deviendrez pas un artisan du logiciel en apprenant une liste d'heuristiques. Le professionnalisme et le savoir-faire s'acquièrent à partir des valeurs sur lesquelles se fondent les disciplines.

Annexe A

Concurrence II

Par Brett L. Schuchert

Cette annexe revient sur le sujet du Chapitre 13, *Concurrence*, en l'approfondissant. Elle est constituée d'un ensemble de thèmes indépendants que vous pouvez lire dans n'importe quel ordre. Pour permettre une telle lecture, certaines sections présentent une légère redondance.

Exemple client/serveur

Imaginons une simple application client/serveur. Un serveur s'exécute et attend l'arrivée d'un client en écoutant sur une socket. Lorsque le client se connecte, il envoie une requête.

Le serveur

Voici une version simplifiée d'une application serveur. Le code complet de cet exemple est disponible dans la section "Client/serveur monothread" de cette annexe.

```
ServerSocket serverSocket = new ServerSocket(8009);
 while (keepProcessing) {
    try {
        Socket socket = serverSocket.accept();
        process(socket);
    } catch (Exception e) {
        handle(e);
    }
}
```

Cette application attend l'arrivée d'une connexion, traite le message entrant et attend à nouveau que le client suivant se connecte. Voici le code correspondant au client qui se connecte à ce serveur :

```
private void connectSendReceive(int i) {
    try {
        Socket socket = new Socket("localhost", PORT);
        MessageUtils.sendMessage(socket, Integer.toString(i));
        MessageUtils.getMessage(socket);
        socket.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

Quelles sont les performances de ce couple client/serveur ? Comment pouvons-nous décrire de manière formelle ces performances ? Le test suivant vérifie qu'elles sont "acceptables" :

```
@Test(timeout = 10000)
public void shouldRunInUnder10Seconds() throws Exception {
    Thread[] threads = createThreads();
    startAllThreadsw(threads);
    waitForAllThreadsToFinish(threads);
}
```

La phase de configuration est absente de cet exemple simple (voir Listing A.4). Ce test vérifie s'il peut se terminer en dix secondes.

Il s'agit d'un exemple classique de validation du débit d'un système. Celui-ci doit traiter une séquence de requêtes clientes en dix secondes. Tant que le serveur est capable de traiter chaque requête cliente individuelle dans le temps imparti, le test réussit.

Que se passe-t-il lorsque le test échoue ? Hormis développer une sorte de boucle d'obtention des événements, avec un seul thread, il n'y a pas grand-chose à faire pour rendre ce code plus rapide. Est-ce que l'utilisation de plusieurs threads résoudrait le problème ? Peut-être, mais nous devons savoir où le temps est dépensé. Il existe deux possibilités :

- **Entrées/sorties.** Utilisation d'une socket, connexion à une base de données, attente d'une opération d'échange dans la mémoire virtuelle, etc.
- **Processeur.** Calcul numérique, traitement d'une expression régulière, ramasse-miettes, etc.

En général, les systèmes comprennent ces deux types d'opérations, mais l'une a tendance à prédominer. Si le code fait un usage intensif du processeur, une mise à niveau du matériel peut améliorer les performances et permettre à notre test de réussir. Toutefois, puisque le nombre de cycles est limité, l'ajout de threads pour résoudre un problème limité par le processeur ne permettra pas d'accélérer l'exécution.

En revanche, si le traitement est limité par les entrées/sorties, la concurrence peut améliorer l'efficacité. Pendant qu'une partie du système attend que les entrées/sorties soient disponibles, une autre partie peut en profiter pour effectuer une autre opération et ainsi employer plus efficacement le processeur.

Ajouter des threads

Supposons, pour le moment, que le test des performances échoue. Comment pouvons-nous améliorer le débit afin qu'il réussisse à nouveau ? Si la méthode `process` du serveur est limitée par les entrées/sorties, voici une manière d'ajouter le multithread au serveur (changer simplement `processMessage`) :

```
void process(final Socket socket) {
    if (socket == null)
        return;

    Runnable clientHandler = new Runnable() {
        public void run() {
            try {
                String message = MessageUtils.getMessage(socket);
                MessageUtils.sendMessage(socket, "Processed: " + message);
                closeIgnoringException(socket);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };

    Thread clientConnection = new Thread(clientHandler);
    clientConnection.start();
}
```

Supposez que cette modification permette au test de réussir[1] ; le code est terminé, n'est-ce pas ?

Observations concernant le serveur

La nouvelle version du serveur passe le test en un peu plus d'une seconde. Malheureusement, cette solution est un tantinet naïve et apporte son lot de nouveaux problèmes.

Combien de threads notre serveur peut-il créer ? Puisque le code ne fixe aucune limite, nous pouvons potentiellement atteindre celle imposée par la machine virtuelle Java (JVM, *Java Virtual Machine*). Pour les systèmes simples, cela peut convenir. En revanche, quelles sont les conséquences si le système doit prendre en charge de nombreux utilisateurs provenant du réseau public ? Si un trop grand nombre d'utilisateurs se connectent en même temps, le système peut s'arrêter.

Pour le moment, mettons de côté le problème de comportement et examinons les soucis de propreté et de structure de la solution proposée. Combien de responsabilités ce serveur possède-t-il ?

1. Vous pouvez le vérifier vous-même en essayant le code avant et après la modification. Consultez les sections "Client/serveur monothread" et "Client/serveur multithread".

- gestion de la connexion par socket ;
- traitement du client ;
- stratégie de gestion des threads ;
- stratégie d'arrêt du serveur.

Malheureusement, toutes ces responsabilités sont assurées par la fonction `process`. Par ailleurs, le code traverse allègrement plusieurs niveaux d'abstraction différents. Aussi courte que puisse être la fonction `process`, elle doit être repartitionnée.

Puisqu'il existe plusieurs raisons de modifier le serveur, il transgresse le principe de responsabilité unique. Pour qu'un système concurrent soit propre, la gestion des threads doit se trouver en quelques endroits parfaitement maîtrisés. Par ailleurs, tout code qui gère des threads doit être dédié uniquement à cette gestion. Quelle en est la raison ? La résolution des problèmes de concurrence est déjà suffisamment complexe pour qu'il soit inutile d'ajouter en même temps les problèmes liés au code normal.

Si nous créons des classes séparées pour chacune des responsabilités mentionnées précédemment, y compris la gestion des threads, la modification de la stratégie de gestion des threads aura un impact plus faible sur l'ensemble du code et ne s'immiscera pas dans les autres responsabilités. Il sera également plus facile de tester toutes les autres responsabilités sans se préoccuper des questions de multithread. Voici une version qui correspond à cette approche :

```
public void run() {
  while (keepProcessing) {
    try {
      ClientConnection clientConnection = connectionManager.awaitClient();
      ClientRequestProcessor requestProcessor
        = new ClientRequestProcessor(clientConnection);
      clientScheduler.schedule(requestProcessor);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
  connectionManager.shutdown();
}
```

Tous les aspects liés aux threads se trouvent à présent en un seul endroit, c'est-à-dire `clientScheduler`. En cas de problèmes dus à la concurrence, il suffit d'examiner ce seul emplacement :

```
public interface ClientScheduler {
    void schedule(ClientRequestProcessor requestProcessor);
}
```

La stratégie actuelle est facile à mettre en œuvre :

```
public class ThreadPerRequestScheduler implements ClientScheduler {
    public void schedule(final ClientRequestProcessor requestProcessor) {
        Runnable runnable = new Runnable() {
            public void run() {
                requestProcessor.process();
            }
        };

        Thread thread = new Thread(runnable);
        thread.start();
    }
}
```

En ayant isolé toute la gestion des threads en un même endroit, il est beaucoup plus facile de modifier la manière dont ils sont contrôlés. Par exemple, pour utiliser le framework Executor de Java 5, il suffit d'écrire une nouvelle classe et de la brancher (voir Listing A.1).

Listing A.1 : `ExecutorClientScheduler.java`

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class ExecutorClientScheduler implements ClientScheduler {
    Executor executor;

    public ExecutorClientScheduler(int availableThreads) {
        executor = Executors.newFixedThreadPool(availableThreads);
    }

    public void schedule(final ClientRequestProcessor requestProcessor) {
        Runnable runnable = new Runnable() {
            public void run() {
                requestProcessor.process();
            }
        };
        executor.execute(runnable);
    }
}
```

Conclusion

Dans cet exemple précis, nous avons montré que l'ajout de la concurrence permet d'améliorer le débit d'un système, ainsi qu'une manière de valider ce débit grâce à un test. En concentrant tout le code de concurrence dans un petit nombre de classes, nous illustrons la mise en application du principe de responsabilité unique. Dans le cadre de la programmation concurrente, ce point devient particulièrement important en raison de la complexité accrue.

Chemins d'exécution possibles

Examinons la méthode `incrementValue`, qui contient une seule ligne de code Java, sans boucle ni branchement :

```
public class IdGenerator {
  int lastIdUsed;

  public int incrementValue() {
    return ++lastIdUsed;
  }
}
```

Ignorons les problèmes de débordement d'entiers et supposons qu'un seul thread ait accès à une même instance de `IdGenerator`. Dans ce cas, il existe un seul chemin d'exécution et un seul résultat garanti :

- La valeur retournée est égale à la valeur de `lastIdUsed`, qui est supérieure de un à la valeur de cette variable avant l'appel de la méthode.

Que se passe-t-il si nous utilisons deux threads sans modifier la méthode ? Quels sont les résultats possibles si chaque thread appelle une fois `incrementValue` ? Combien existe-t-il de chemins d'exécution possibles ? Tout d'abord, occupons-nous des résultats (nous supposons que la valeur initiale de `lastIdUsed` est 93) :

- Le premier thread obtient la valeur 94, le second thread obtient la valeur 95, `lastIdUsed` vaut 95.
- Le premier thread obtient la valeur 95, le second thread obtient la valeur 94, `lastIdUsed` vaut 95.
- Le premier thread obtient la valeur 94, le second thread obtient la valeur 94, `lastIdUsed` vaut 94.

Le troisième résultat, quoique surprenant, est possible. Pour comprendre comment on peut arriver à ces différents résultats, nous devons connaître le nombre de chemins d'exécution possibles et savoir comment la machine virtuelle Java les exécute.

Nombre de chemins

Pour calculer le nombre de chemins d'exécution existants, nous devons étudier le byte-code généré. L'unique ligne de code Java (`return ++lastIdUsed;`) est convertie en huit instructions de byte-code. L'exécution de chacun des deux threads peut s'intercaler entre chacune de ces huit instructions, à la manière dont le distributeur de cartes au poker les intercale lorsqu'il mélange un sabot[2]. Même avec seulement huit cartes dans chaque main, le nombre de distributions possible est remarquablement grand.

2. Cette comparaison est un tantinet simplifiée. Toutefois, pour notre propos, nous pouvons nous servir de ce modèle simplifié.

Calculer les rangements possibles

Cette note provient d'un échange entre l'Oncle Bob et Brett.

Avec *N* étapes et *T* threads, il existe $T \times N$ étapes au total. Avant chaque étape, il se produit un changement de contexte qui choisit parmi les *T* threads. Chaque chemin peut donc être représenté par une chaîne de chiffres qui représente les changements de contexte. Étant donné les étapes A et B des threads 1 et 2, les six chemins possibles sont 1122, 1212, 1221, 2112, 2121 et 2211. Ou, en citant les étapes, ils sont A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1 et A2B2A1B1. Pour trois threads, la séquence devient 112233, 112323, 113223, 113232, 112233, 121233, 121323, 121332, 123132, 123123...

Ces chaînes présentent la caractéristique suivante : il doit toujours y avoir *N* instances de chaque *T*. Par conséquent, la chaîne 111111 est invalide car elle contient six instances de 1 et aucune instance de 2 et de 3.

Nous voulons les permutations de *N* 1, *N* 2... et *N T*. Il s'agit en réalité des permutations de $N \times T$ choses prises $N \times T$ à la fois, c'est-à-dire $(N \times T)!$, mais en retirant tous les doublons. Nous devons donc calculer le nombre de doublons et soustraire cette valeur de $(N \times T)!$.

Étant donné deux étapes et deux threads, combien existe-t-il de doublons ? Chaque chaîne de quatre chiffres contient deux 1 et deux 2. Chacune de ces paires peut être échangée sans modifier le sens de la chaîne. Nous pouvons inverser les 1 ou les 2, ou aucun. Il existe donc quatre objets isomorphes pour chaque chaîne, ce qui donne trois doublons. Trois de nos quatre possibilités sont des doublons, ou une permutation sur quatre n'est pas un doublon. $4! \times 0.25 = 6$. Le raisonnement semble se tenir.

Combien de doublons avons-nous ? Dans le cas où $N = 2$ et $T = 2$, je peux échanger les 1, les 2 ou les deux. Dans le cas où $N = 2$ et $T = 3$, je peux échanger les 1, les 2, les 3, les 1 et les 2, les 1 et les 3, ou les 2 et les 3. Les échanges correspondent simplement aux permutations de *N*. Supposons que nous ayons *P* permutations de *N*. Le nombre d'arrangements différents de ces permutations est *P****T*.

Par conséquent, le nombre d'objets isomorphes possibles est *N*!***T*. Le nombre de chemins est alors $(T \times N)!/(N!^{**}T)$. Dans notre cas, $T = 2$ et $N = 2$, ce qui donne 6 (24/4).

Pour $N = 2$ et $T = 3$, nous obtenons 720/8 = 90.

Pour $N = 3$ et $T = 3$, nous obtenons 9!/6^3 = 1680.

Dans notre cas simple de *N* instructions à la suite, sans boucles ni instructions conditionnelles, et *T* threads, le nombre total de chemins d'exécution possibles est donné par la formule suivante :

$$\frac{(NT)!}{N!^{T}}$$

Dans notre cas simple d'une seule ligne de code Java, qui équivaut à huit lignes de bytecode et à deux threads, le nombre total de chemins d'exécution possibles est 12 870. Si la variable `lastIdUsed` est de type `long`, chaque lecture/écriture est non plus une mais deux opérations, et le nombre de rangements possibles passe à 2 704 156.

Que se passe-t-il si nous modifions la méthode de la manière suivante ?

```
public synchronized void incrementValue() {
    ++lastIdUsed;
}
```

Le nombre de chemins d'exécution possibles devient égal à 2 pour deux threads, et à *N*! dans le cas général.

Examen plus approfondi

Revenons sur le résultat surprenant de deux threads qui invoquent tous deux la méthode une fois (avant que nous ajoutions `synchronized`) et qui reçoivent la même valeur numérique. Comment est-ce possible ?

Tout d'abord, qu'est-ce qu'une opération atomique ? Une opération atomique peut être définie comme une opération qui ne peut pas être interrompue. Par exemple, dans le code suivant, l'affectation de 0 à la variable `lastid` (ligne 5) est atomique car, conformément au modèle de mémoire de Java, l'affectation d'une valeur sur 32 bits n'est pas interruptible.

```
01: public class Example {
02:     int lastId;
03:
04:     public void resetId() {
05:         value = 0;
06:     }
07:
08:     public int getNextId() {
09:         ++value;
10:     }
11: }
```

Que se passe-t-il si `lastId` est non plus de type `int` mais `long` ? La ligne 5 est-elle toujours atomique ? D'après les spécifications de la JVM, la réponse est non. Elle peut être atomique sur certains processeurs, mais les spécifications de la JVM stipulent que l'affectation d'une valeur de 64 bits nécessite deux affectations de 32 bits. Autrement dit, entre la première et la deuxième affectation de 32 bits, un autre thread peut entrer en scène et modifier l'une des valeurs.

Quid de l'opérateur de préincrémentation (`++`) à la ligne 9 ? Puisqu'il peut être interrompu, il n'est pas atomique. Pour bien comprendre le fonctionnement, examinons en détail le byte-code de ces deux méthodes.

Avant d'aller plus loin, voici trois définitions d'importance :

- **Bloc d'activation.** Chaque invocation de méthode implique un bloc d'activation. Ce dernier comprend l'adresse de retour, les paramètres passés à la méthode et les variables locales qui y sont définies. Cette technique classique est employée pour

définir une pile d'appel, qui est utilisée dans les langages modernes pour les invocations de fonctions/méthodes et permettre les invocations récursives.

- **Variable locale.** Toute variable définie dans la portée de la méthode. Toutes les méthodes non statiques possèdent au moins une variable, `this`, qui représente l'objet courant à l'origine de l'invocation de la méthode, c'est-à-dire l'objet qui a reçu le message le plus récent (dans le thread en cours).
- **Pile des opérandes.** Un grand nombre d'instructions de la machine virtuelle Java prennent des paramètres. C'est dans la pile des opérandes que ces paramètres sont placés. Il s'agit d'une structure de données LIFO (*Last-In, First-Out*) classique.

Voici le byte-code généré pour `resetId()` :

Mnémonique	*Description*	*Pile des opérandes*
`ALOAD 0`	Charge la variable d'indice `0` dans la pile des opérandes. Quelle est cette variable d'indice `0` ? Il s'agit de `this`, c'est-à-dire l'objet courant. Lorsque la méthode a été invoquée, le destinataire du message, une instance de `Example`, a été placé dans le tableau des variables locales du bloc d'activation créé pour l'invocation de la méthode. Il s'agit toujours de la première variable donnée à chaque méthode d'instance.	`this`
`ICONST 0`	Place la valeur constante `0` dans la pile des opérandes.	`this, 0`
`PUTFIELD lastId`	Enregistre la valeur au sommet de la pile (qui vaut `0`) dans la valeur de champ de l'objet désigné par la référence qui se trouve une case après le sommet de la pile (`this`).	<vide>

Ces trois instructions sont garanties atomiques car, bien que le thread qui les exécute puisse être interrompu après l'une ou l'autre, les informations nécessaires à l'instruction `PUTFIELD` (la constante `0` au sommet de la pile et la référence à `this` une case en dessous, ainsi que la valeur de champ) ne peuvent pas être altérées par un autre thread. Par conséquent, lorsque l'affectation se produit, nous sommes certains que la valeur 0 est enregistrée dans la valeur de champ. L'opération est atomique. Les opérandes concernent des informations locales à la méthode, ce qui évite toute interférence entre plusieurs threads.

Si ces trois instructions sont exécutées par dix threads, il existe 4,38679733629e+24 rangements possibles. Cependant, puisqu'il n'y a qu'un seul résultat possible, les diffé-

rents rangements ne sont pas pertinents. Par ailleurs, dans ce cas, le même résultat est garanti pour des entiers longs. Quelle en est la raison ? Les dix threads affectent une valeur constante. Même s'ils s'intercalent les uns entre les autres, le résultat final est identique.

Avec l'opération `++` dans la méthode `getNextId`, nous allons rencontrer quelques problèmes. Supposons que la variable `lastId` contienne la valeur 42 au début de cette méthode. Voici le byte-code correspondant à cette méthode :

Mnémonique	*Description*	*Pile des opérandes*
`ALOAD 0`	Charge `this` sur la pile des opérandes.	`this`
`DUP`	Recopie le sommet de la pile. La pile des opérandes contient à présent deux exemplaires de `this`.	`this`, `this`
`GETFIELD lastId`	Récupère la valeur du champ `lastId` à partir de l'objet désigné par la référence qui se trouve au sommet de la pile (`this`) et place cette valeur dans la pile.	`this`, `42`
`ICONST 1`	Place la constante entière `1` sur la pile.	`this`, `42`, `1`
`IADD`	Additionne (en mode entier) les deux valeurs au sommet de la pile des opérandes et replace le résultat sur la pile.	`this`, `43`
`DUP X1`	Recopie la valeur `43` et la place avant `this`.	`43`, `this`, `43`
`PUTFIELD value`	Enregistre la valeur au sommet de la pile des opérandes, `43`, dans la valeur de champ de l'objet en cours, qui est représenté par la valeur suivante au sommet de la pile des opérandes, `this`.	`43`
`IRETURN`	Retourne la valeur au sommet de la pile.	<vide>

Imaginez que le premier thread termine les trois premières instructions, jusqu'à l'instruction `GETFIELD` incluse, avant d'être interrompu. Un second thread prend le relais et exécute l'intégralité de la méthode, incrémentant ainsi `lastId` de un ; il reçoit 43 en résultat. Ensuite, le premier thread reprend la main ; 42 se trouve toujours sur la pile des opérandes car il s'agissait de la valeur de `lastId` au moment de l'exécution de `GETFIELD`. Ce thread ajoute un, obtient à nouveau 43 et enregistre le résultat. La valeur 43 est également retournée au premier thread. En résultat, l'une des incrémentations est perdue car le premier thread a pris le pas sur le second après que celui-ci a interrompu le premier.

En déclarant synchronisée la méthode `getNexId()`, ce problème est résolu.

Conclusion

La maîtrise du byte-code n'est pas indispensable pour comprendre comment deux threads peuvent se marcher sur les pieds. Si cet exemple ne vous pose pas de difficultés, il illustre les risques d'une exécution multithread. Cette connaissance vous suffira.

Cet exemple trivial montre qu'il est nécessaire de comprendre suffisamment le modèle de mémoire pour savoir ce qui est sûr et ce qui ne l'est pas. Beaucoup pensent que l'opérateur ++ (pré- ou postincrémentation) est atomique, alors que ce n'est clairement pas le cas. Autrement dit, vous devez connaître les éléments suivants :

- l'emplacement des objets/valeurs partagés ;
- le code qui peut provoquer des problèmes de lectures/mises à jour concurrentes ;
- comment empêcher ces problèmes de concurrence.

Connaître sa bibliothèque

Framework *Executor*

Comme l'a montré le code de `ExecutorClientScheduler.java` au Listing A.1, le framework `Executor` proposé par Java 5 permet une exécution complexe basée sur les pools de threads. Cette classe fait partie du paquetage `java.util.concurrent`.

Si vous créez des threads et n'utilisez pas un pool de threads ou utilisez un pool écrit par vos soins, vous devez envisager de passer à `Executor`. Vous obtiendrez un code plus propre, plus facile à suivre et plus concis.

Le framework `Executor` place les threads dans un pool, le redimensionne automatiquement et recrée des threads si nécessaire. Il prend également en charge la notion de *future*, une construction très répandue dans la programmation concurrente. Le framework `Executor` fonctionne avec les classes qui implémentent `Runnable` et celles qui implémentent l'interface `Callable`. Un `Callable` ressemble à un `Runnable`, mais il peut retourner un résultat, ce qui est un besoin fréquent dans les solutions multithreads.

Un *future* se révèle commode lorsque du code doit exécuter plusieurs opérations indépendantes et attendre qu'elles se terminent :

```
public String processRequest(String message) throws Exception {
    Callable<String> makeExternalCall = new Callable<String>() {
        public String call() throws Exception {
            String result = "";
            // Effectuer une requête externe.
            return result;
        }
    };
```

```
        Future<String> result = executorService.submit(makeExternalCall);
        String partialResult = doSomeLocalProcessing();
        return result.get() + partialResult;
    }
```

Dans cet exemple, la méthode commence par exécuter l'objet `makeExternalCall`, puis poursuit son traitement. La dernière ligne invoque `result.get()`, qui bloque jusqu'à ce que l'objet `Future` ait terminé.

Solutions non bloquantes

La machine virtuelle Java 5 tire profit des processeurs modernes, qui prennent en charge les mises à jour fiables non bloquantes. Prenons, par exemple, une classe qui utilise une synchronisation (donc bloquante) pour mettre à jour une valeur de manière sûre vis-à-vis des threads :

```
public class ObjectWithValue {
    private int value;
    public void synchronized incrementValue() { ++value; }
    public int getValue() { return value; }
}
```

Java 5 propose de nouvelles classes pour les situations de ce genre : `AtomicBoolean`, `AtomicInteger` et `AtomicReference` en sont trois exemples, mais il en existe d'autres. Nous pouvons récrire le code précédent en employant une approche non bloquante :

```
public class ObjectWithValue {
    private AtomicInteger value = new AtomicInteger(0);

    public void incrementValue() {
        value.incrementAndGet();
    }
    public int getValue() {
        return value.get();
    }
}
```

Même si cette version emploie un objet à la place d'un type primitif et envoie des messages, comme `incrementAndGet()` à la place de `++`, les performances de cette classe seront quasiment toujours identiques à la version précédente. Dans certains cas, elles pourraient même se révéler légèrement meilleures. En revanche, les cas où elles sont inférieures n'existent virtuellement pas.

Comment est-ce possible ? Les processeurs modernes disposent d'une opération généralement nommée *Compare and Swap* (CAS). Celle-ci est comparable au verrouillage optimiste dans les bases de données, tandis que la version synchronisée s'apparente au verrouillage pessimiste.

Le mot-clé `synchronized` acquiert toujours un verrou, même lorsque aucun autre thread n'essaie de mettre à jour la même valeur. Même si les performances des verrous internes s'améliorent de version en version, ils n'en restent pas moins onéreux.

La version non bloquante fait l'hypothèse que, en général, plusieurs threads ne modifient pas la même valeur suffisamment souvent pour que des problèmes surviennent. À la place, elle détecte efficacement lorsqu'une telle situation s'est produite et recommence l'opération jusqu'à ce que la mise à jour réussisse. Cette détection est presque toujours moins coûteuse que l'obtention d'un verrou, même dans les situations de concurrence moyenne à forte.

Comment la machine virtuelle procède-t-elle ? L'opération CAS est atomique. D'un point de vue logique, cette opération équivaut au code suivant :

```
int variableBeingSet;

void simulateNonBlockingSet(int newValue) {
    int currentValue;
    do {
        currentValue = variableBeingSet
    } while(currentValue != compareAndSwap(currentValue, newValue));
}

int synchronized compareAndSwap(int currentValue, int newValue) {
    if(variableBeingSet == currentValue) {
        variableBeingSet = newValue;
        return currentValue;
    }
    return variableBeingSet;
}
```

Lorsqu'une méthode tente d'actualiser une variable partagée, l'opération CAS vérifie si la variable mise à jour possède toujours la dernière valeur connue. Dans l'affirmative, la variable est modifiée. Dans le cas contraire, la variable n'est pas actualisée car un autre thread est intervenu entre-temps. La méthode qui a tenté la mise à jour, à l'aide de l'opération CAS, constate que la modification n'a pas été effectuée et recommence son opération.

Classes non sûres vis-à-vis des threads

Certaines classes sont intrinsèquement non sûres vis-à-vis des threads, dont :

- `SimpleDateFormat` ;
- connexion aux bases de données ;
- conteneurs de `java.util` ;
- servlets.

Notez également que certaines classes collections possèdent des méthodes individuelles qui ne sont pas sûres vis-à-vis des threads. Par ailleurs, toute opération qui comprend l'invocation de plusieurs méthodes n'est pas sûre vis-à-vis de threads. Par exemple, si vous souhaitez ne pas remplacer un élément dans un `HashTable` lorsqu'il s'y trouve déjà, vous pourriez écrire le code suivant :

```
if(!hashTable.containsKey(someKey)) {
    hashTable.put(someKey, new SomeValue());
}
```

Chaque méthode est individuellement sûre vis-à-vis des threads. Cependant, un autre thread a la possibilité d'insérer une valeur entre les appels à `containsKey` et à `put`. Pour résoudre ce problème, il existe plusieurs solutions.

- Commencez par verrouiller le `HashTable` et assurez-vous que tous les autres utilisateurs du `HashTable` font de même. Il s'agit d'un verrouillage côté client :

```
synchronized(map) {
if(!map.containsKey(key))
    map.put(key,value);
}
```

- Enveloppez le `HashTable` dans son propre objet et employez une API différente. Il s'agit d'un verrouillage côté serveur en utilisant un ADAPTATEUR :

```
public class WrappedHashtable<K, V> {
    private Map<K, V> map = new Hashtable<K, V>();

    public synchronized void putIfAbsent(K key, V value) {
        if (map.containsKey(key))
            map.put(key, value);
    }
}
```

- Employez des collections sûres vis-à-vis des threads :

```
ConcurrentHashMap<Integer, String> map =
  new ConcurrentHashMap<Integer, String>();
map.putIfAbsent(key, value);
```

Les collections du paquetage `java.util.concurrent` disposent d'opérations semblables à `putIfAbsent()` pour prendre en charge de telles opérations.

Impact des dépendances entre méthodes sur le code concurrent

Dans l'exemple simple suivant, des dépendances entre méthodes sont introduites :

```
public class IntegerIterator implements Iterator<Integer>
    private Integer nextValue = 0;

    public synchronized boolean hasNext() {
        return nextValue < 100000;
    }
```

```
    public synchronized Integer next() {
        if (nextValue == 100000)
            throw new IteratorPastEndException();
        return nextValue++;
    }
    public synchronized Integer getNextValue() {
        return nextValue;
    }
}
```

Le code suivant utilise cet `IntegerIterator` :

```
IntegerIterator iterator = new IntegerIterator();
while(iterator.hasNext()) {
    int nextValue = iterator.next();
    // Employer nextValue.
}
```

Si un seul thread exécute ce code, il n'y aura aucun problème. En revanche, que se passe-t-il lorsque deux threads tentent de partager une même instance de `Integer Iterator` en voulant que chacun traite les valeurs qu'il reçoit, mais que chaque élément de la liste ne soit traité que par un seul des deux threads ? La plupart du temps, tout se passe bien ; les threads partagent la liste, en traitant les éléments qui leur sont fournis par l'itérateur et en s'arrêtant une fois l'itération terminée. Cependant, il existe une faible chance que, à la fin de l'itération, une interférence se produise entre les deux threads et qu'elle conduise l'un des deux threads à dépasser la limite de l'itérateur et à lancer une exception.

Voici une illustration du problème : Thread 1 pose la question `hasNext()` et reçoit la réponse `true`. Thread 1 est interrompu et Thread 2 pose la même question, dont la réponse est toujours `true`. Thread 2 invoque ensuite `next()`, qui retourne une valeur comme attendu, mais en ayant pour effet secondaire d'amener `hasNext()` à retourner `false`. Thread 1 reprend, en considérant que `hasNext()` vaut toujours `true`, et invoque ensuite `next()`. Bien que les méthodes individuelles soient synchronisées, le client utilise ***deux*** méthodes.

Ce problème est bien réel et illustre parfaitement ceux que l'on rencontre dans le code concurrent. Dans ce cas particulier, le problème est particulièrement subtil car l'erreur se produit uniquement au cours de la dernière itération. Si les threads sont interrompus au mauvais moment, l'un d'eux peut dépasser la limite de l'itérateur. Ce type de bogue survient généralement bien longtemps après qu'un système a été mis en production et il est difficile à dépister.

Vous avez trois possibilités :

- tolérer la panne ;
- résoudre le problème en modifiant le client : verrouillage côté client ;

- résoudre le problème en modifiant le serveur et par conséquent le client : verrouillage côté serveur.

Tolérer la panne

Parfois, il est possible de faire en sorte que la panne ne provoque aucun dégât. Par exemple, le client précédent peut intercepter l'exception et s'assurer que tout se termine bien. Sincèrement, cette attitude est peu soignée. Elle s'apparente à la résolution de problèmes de fuites de mémoire en redémarrant la machine à minuit.

Verrouillage côté client

Pour que `IntegerIterator` fonctionne parfaitement avec plusieurs threads, modifiez l'exemple de client (et tous les autres clients) de la manière suivante :

```
IntegerIterator iterator = new IntegerIterator();

while (true) {
  int nextValue;
  synchronized (iterator) {
    if (!iterator.hasNext())
      break;
    nextValue = iterator.next();
  }
  doSometingWith(nextValue);
}
```

Chaque client introduit un verrou par l'intermédiaire du mot-clé `synchronized`. Cette redondance va à l'encontre du principe DRY, mais elle peut être nécessaire si le code utilise des outils tiers non sûrs vis-à-vis des threads.

Cette stratégie présente un risque car tous les programmeurs qui utilisent le serveur ne doivent pas oublier de le verrouiller avant de l'invoquer et de le déverrouiller lorsqu'ils ont terminé. Il y a bien longtemps, j'ai travaillé sur un système dans lequel une ressource partagée faisait l'objet d'un verrouillage côté client. Elle était employée dans des centaines d'endroits différents du code. Un programmeur avait oublié de verrouiller la ressource dans l'un de ces endroits.

Il s'agissait d'un système à temps partagé et terminaux multiples sur lequel s'exécutait le logiciel de comptabilité de la section Local 705 du syndicat des camionneurs. L'ordinateur se trouvait dans une salle machine située à 80 km au nord du siège de la section Local 705. Des dizaines d'opérateurs travaillaient sur les terminaux du siège pour la saisie des cotisations. Les terminaux étaient connectés à l'ordinateur au travers de lignes dédiées et de modems half-duplex à 600 bauds. (Je vous avais dit que c'était il y a bien longtemps.)

Environ une fois par jour, l'un des terminaux se bloquait. Il n'y avait aucune raison apparente. Le blocage ne concernait aucun terminal particulier et n'arrivait à aucune heure précise. C'était comme si quelqu'un faisait tourner la roue pour choisir l'heure et le terminal à figer. Parfois, plusieurs terminaux se bloquaient. D'autres fois, plusieurs jours passaient avant un blocage.

Au départ, la seule solution consistait en un redémarrage. Toutefois, il était difficile de coordonner les redémarrages. Il fallait appeler le siège et demander à tous les utilisateurs de terminer leurs tâches en cours sur tous les terminaux. Ce n'est qu'ensuite que nous pouvions procéder à l'arrêt et au redémarrage. Si quelqu'un était au beau milieu d'un travail important qui prenait une ou deux heures, le terminal bloqué le restait simplement pendant tout ce temps.

Après quelques semaines de débogage, nous avons pu déterminer que le problème était provoqué par un compteur de tampon circulaire qui n'était plus synchronisé avec son pointeur. Ce tampon contrôlait la sortie sur le terminal. La valeur du pointeur indiquait que le tampon était vide, alors que le compteur indiquait qu'il était plein. Puisqu'il était vide, rien n'était affiché. Puisqu'il était également plein, rien ne pouvait y être ajouté afin d'être affiché à l'écran.

Nous savions donc pourquoi les terminaux étaient bloqués, mais nous ne savions pas pourquoi le tampon circulaire se désynchronisait. Nous avons bricolé une petite solution pour contourner le problème. Il était possible de lire les boutons du panneau frontal de l'ordinateur (n'oubliez pas, c'était il y a bien longtemps). Nous avons écrit une fonction qui était appelée lors de l'appui sur l'un de ces boutons et qui examinait si un tampon circulaire était à la fois vide et plein. Dans l'affirmative, elle réinitialisait ce tampon à vide. Et voilà, les terminaux bloqués fonctionnaient à nouveau.

Nous n'avions plus à redémarrer le système lors du blocage d'un terminal. La section locale devait simplement nous appeler et nous informer d'un blocage. Il nous suffisait ensuite d'aller dans la salle machine et d'appuyer sur un bouton.

Cependant, les employés de la section locale travaillaient parfois le week-end, ce qui n'était pas notre cas. Nous avons donc planifié une fonction qui vérifiait tous les tampons circulaires une fois par minute et réinitialisait ceux qui étaient à la fois vides et pleins. De cette manière, l'affichage était débloqué avant que la section locale n'ait le temps de téléphoner.

Il nous a fallu plusieurs semaines d'analyse du code assembleur monolithique pour identifier le coupable. Nous avons fait des calculs et déterminé que la fréquence des blocages coïncidait avec une seule utilisation non protégée du tampon circulaire. Il suffisait donc de trouver cette utilisation fautive. Malheureusement, à cette époque-là,

nous ne disposions pas d'outils de recherche, de références croisées ou de toute autre sorte d'aide automatisée. Nous devions simplement nous plonger dans les listings.

En ce froid hiver de 1971 à Chicago, j'ai appris une leçon importante. Le verrouillage côté client est réellement une plaie.

Verrouillage côté serveur

Pour supprimer la redondance, nous apportons les modifications suivantes à `Integer Iterator` :

```
public class IntegerIteratorServerLocked {
    private Integer nextValue = 0;
    public synchronized Integer getNextOrNull() {
        if (nextValue < 100000)
            return nextValue++;
        else
            return null;
    }
}
```

Le code du client doit également être revu :

```
while (true) {
    Integer nextValue = iterator.getNextOrNull();
    if (next == null)
        break;
    // Employer nextValue.
}
```

Dans ce cas, nous changeons l'API de notre classe afin qu'elle soit consciente d'un fonctionnement multithread[3]. Le client doit vérifier `null` au lieu de tester `hasNext()`.

Voici les raisons de préférer, en général, le verrouillage côté serveur :

- Le code redondant est réduit. Le verrouillage côté client impose à chaque client de verrouiller correctement le serveur. En plaçant le code de verrouillage dans le serveur, les clients peuvent employer l'objet sans avoir à écrire du code de verrouillage supplémentaire.
- Les performances peuvent être améliorées. Vous pouvez remplacer un serveur sûr vis-à-vis des threads par un serveur non sûr vis-à-vis des threads dans le cadre d'un déploiement monothread, en évitant ainsi tous les surcoûts.
- Les possibilités d'erreur sont diminuées. Il suffit qu'un seul programmeur oublie de verrouiller pour conduire au dysfonctionnement du système.

3. En réalité, l'interface `Iterator` n'est intrinsèquement pas sûre vis-à-vis des threads. Puisqu'elle n'a jamais été conçue pour être employée par plusieurs threads, ce comportement ne devrait pas être surprenant.

- Une même stratégie est imposée. La stratégie se trouve en un seul endroit, sur le serveur, non en plusieurs emplacements, sur chaque client.
- La portée des variables partagées est réduite. Le client ne les connaît pas ou ne sait pas comment elles sont verrouillées. Tous ces détails sont cachés dans le serveur. En cas de dysfonctionnement, le nombre d'endroits à examiner est plus faible.

Comment pouvez-vous procéder si vous ne disposez pas du code serveur ?

- Utilisez un ADAPTATEUR pour modifier l'API et ajoutez un verrouillage :

```
public class ThreadSafeIntegerIterator {
    private IntegerIterator iterator = new IntegerIterator();

    public synchronized Integer getNextOrNull() {
        if(iterator.hasNext())
            return iterator.next();
        return null;
    }
}
```

- Mieux encore, servez-vous des collections sûres vis-à-vis des threads, avec des interfaces étendues.

Augmenter le débit

Supposons que nous souhaitions nous balader sur Internet en lisant le contenu d'un ensemble de pages à partir d'une liste d'URL. Dès qu'une page est lue, nous l'analysons afin de collecter des statistiques. Lorsque toutes les pages sont lues, nous affichons un rapport récapitulatif.

La classe suivante retourne le contenu d'une page, étant donné une URL.

```
public class PageReader {
  //...
  public String getPageFor(String url) {
    HttpMethod method = new GetMethod(url);

    try {
      httpClient.executeMethod(method);
      String response = method.getResponseBodyAsString();
      return response;
    } catch (Exception e) {
      handle(e);
    } finally {
      method.releaseConnection();
    }
  }
}
```

La classe suivante correspond à l'itérateur qui fournit le contenu des pages en fonction d'un itérateur sur les URL :

```
public class PageIterator {
  private PageReader reader;
  private URLIterator urls;

  public PageIterator(PageReader reader, URLIterator urls) {
    this.urls = urls;
    this.reader = reader;
  }

  public synchronized String getNextPageOrNull() {
    if (urls.hasNext())
      getPageFor(urls.next());
    else
      return null;
  }

  public String getPageFor(String url) {
    return reader.getPageFor(url);
  }
}
```

Une instance de `PageIterator` peut être partagée entre plusieurs threads, chacun utilisant sa propre instance de `PageReader` pour lire et analyser les pages fournies par l'itérateur.

Vous remarquerez que le bloc `synchronized` est très court. Il contient uniquement la section critique de `PageIterator`. Il est toujours préférable de synchroniser la plus petite portion de code possible.

Calculer le débit en mode monothread

Effectuons à présent des calculs simples, en prenant les hypothèses suivantes :

- durée des entrées/sorties pour obtenir une page (moyenne) : 1 seconde ;
- temps d'analyse d'une page (moyenne) : 0,5 seconde ;
- les entrées/sorties occupent 0 % du temps processeur, tandis que l'analyse demande 100 %.

Pour le traitement de *N* pages par un seul thread, le temps d'exécution total est égal à 1,5 seconde × *N*. La Figure A.1 illustre le traitement de treize pages, c'est-à-dire environ 19,5 secondes.

Calculer le débit en mode multithread

S'il est possible de récupérer les pages dans n'importe quel ordre et de les analyser de manière indépendante, nous pouvons employer plusieurs threads pour augmenter le débit. Que se passe-t-il dans le cas de trois threads ? Combien de pages pouvons-nous obtenir en même temps ?

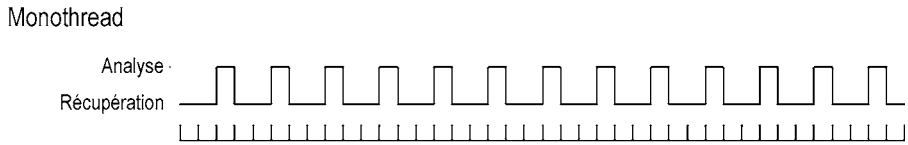


Figure A.1
Version monothread.

La Figure A.2 montre que la solution multithread permet un chevauchement de l'analyse des pages limitée par le processeur et de la lecture des pages limitée par les entrées/sorties. Dans un monde idéal, cela signifie que le processeur est utilisé à 100 %. Chaque lecture d'une page d'une seconde est chevauchée par deux analyses. Il est ainsi possible de traiter deux pages par seconde, ce qui correspond à trois fois le débit de la solution monothread.

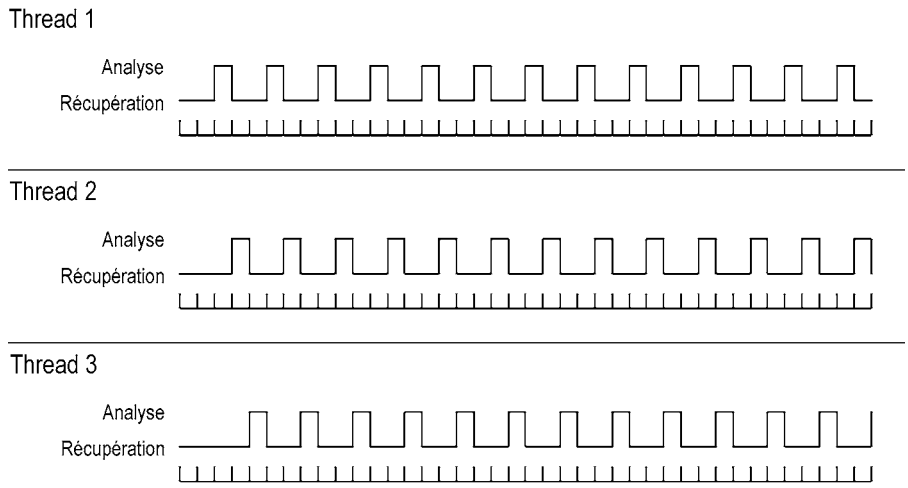


Figure A.2
Trois threads concurrents.

Interblocage

Imaginons une application web avec deux pools de ressources partagées et d'une taille finie :

- un pool de connexions de bases de données pour un travail local sur le stockage des processus ;
- un pool de connexions MQ vers un référentiel principal.

Supposons également que cette application propose les deux opérations suivantes :

- **Création.** Une connexion au référentiel principal et une connexion à une base de données sont obtenues. Le référentiel principal est contacté et un travail local est enregistré dans la base de données des processus.
- **Mise à jour.** Une connexion à la base de données puis au référentiel principal est obtenue. Un travail est écrit dans la base de données des processus, puis envoyé au référentiel principal.

Que se passe-t-il lorsque le nombre d'utilisateurs dépasse la taille des pools ? Considérons que la taille de chaque pool est égale à dix.

- Dix utilisateurs tentent une création. Les dix connexions à la base de données sont obtenues et chaque thread est interrompu après cette opération, avant d'obtenir une connexion au référentiel principal.
- Dix utilisateurs tentent une mise à jour. Les dix connexions au référentiel principal sont obtenues et chaque thread est interrompu après cette opération, avant d'obtenir une connexion à la base de données.
- Les dix threads "création" doivent attendre pour obtenir une connexion au référentiel principal, et les dix threads "mise à jour" doivent attendre pour obtenir une connexion à la base de données.
- Nous obtenons un interblocage. Le système ne peut pas avancer.

Cela peut sembler une situation improbable, mais qui souhaite vraiment un système qui risque de se figer d'un moment à l'autre ? Qui veut déboguer un système dont les symptômes sont aussi difficiles à reproduire ? C'est le genre de problèmes qui surgissent sur le terrain et dont la résolution prend des semaines.

La "solution" classique consiste à introduire des instructions de débogage afin de comprendre ce qui se passe. Bien entendu, ces instructions modifient suffisamment le code pour que l'interblocage se produise dans une situation différente et uniquement au bout de plusieurs mois[4].

Pour résoudre un problème d'interblocage, il est nécessaire d'en comprendre la cause. Pour qu'un interblocage se produise, il faut quatre conditions :

- exclusion mutuelle ;
- détention et attente ;

4. Par exemple, quelqu'un ajoute une sortie de débogage et le problème "disparaît". En réalité, le code de débogage "fixe" le problème pour qu'il reste dans le système.

- pas de préemption ;
- attente circulaire.

Exclusion mutuelle

L'exclusion mutuelle se produit lorsque plusieurs threads doivent employer les mêmes ressources et que ces ressources :

- ne peuvent pas être employées par plusieurs threads à la fois ;
- et sont en nombre limité.

Une connexion à une base de données, un fichier ouvert en écriture, un verrou d'enregistrement et un sémaphore sont des exemples de telles ressources.

Détention et attente

Dès qu'un thread a obtenu une ressource, il ne la libérera pas tant qu'il n'aura pas obtenu toutes les autres ressources dont il a besoin et qu'il n'aura pas terminé son travail.

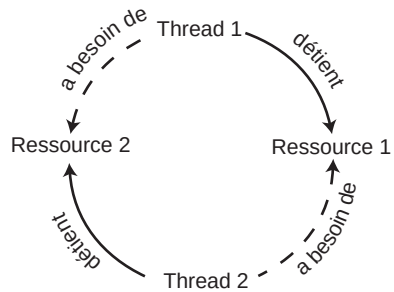
Pas de préemption

Un thread ne peut pas accaparer les ressources détenues par un autre thread. Dès lors qu'un thread détient une ressource, la seule manière pour un autre thread de l'obtenir est que le premier la libère.

Attente circulaire

Imaginons deux threads, T1 et T2, et deux ressources, R1 et R2. T1 détient R1, T2 détient R2. T1 a également besoin de R2, et T2 a besoin de R1. Nous obtenons un fonctionnement semblable à celui illustré par la Figure A.3.

Figure A.3



Ces quatre conditions doivent être satisfaites pour provoquer un interblocage. Il suffit d'annuler l'une de ces conditions pour que l'interblocage soit impossible.

Briser l'exclusion mutuelle

Pour empêcher un interblocage, une stratégie consiste à éviter la condition d'exclusion mutuelle. Pour cela, nous pouvons prendre les dispositions suivantes :

- employer des ressources qui acceptent une utilisation simultanée, par exemple `AtomicInteger` ;
- augmenter le nombre de ressources de manière qu'il soit égal ou supérieur au nombre de threads en concurrence ;
- vérifier que toutes les ressources sont libres avant d'en acquérir une.

Malheureusement, la plupart des ressources sont limitées en nombre et refusent les accès simultanés. Par ailleurs, il n'est pas rare que l'identité de la seconde ressource soit fournie par les résultats des traitements sur la première. Ne soyez pas découragé, il reste encore trois conditions.

Briser la détention et l'attente

Nous pouvons également éliminer un interblocage en refusant d'attendre. Chaque ressource doit être vérifiée avant d'être demandée, et toutes les ressources doivent être libérées et redemandées lorsque l'une d'elles est occupée. Cette approche présente plusieurs risques de problèmes :

- **Famine.** Un thread est incapable d'acquérir les ressources dont il a besoin (il a peut-être besoin d'une combinaison particulière de ressources qui n'est que rarement disponible).
- **Interblocage actif (*livelock*).** Plusieurs threads peuvent entrer en synchronisation, pour tous acquérir une ressource, puis tous libérer une ressource, de manière répétitive. Ce cas se produit particulièrement avec les algorithmes d'ordonnancement simplistes des processeurs (pensez aux périphériques embarqués ou aux algorithmes de distribution des threads simplistes écrits à la main).

Ces deux problèmes peuvent conduire à un débit médiocre. Le premier résulte en une faible utilisation du processeur, tandis que le second résulte en une utilisation élevée et inutile du processeur.

Aussi inefficace que semble cette stratégie, elle est toujours mieux que rien. Il est presque toujours possible de la mettre en œuvre lorsque tout le reste échoue.

Briser la préemption

Pour éviter un interblocage, une autre stratégie consiste à autoriser les threads à enlever des ressources à d'autres threads. En général, cela se fait à l'aide d'un simple mécanisme de requête. Lorsqu'un thread découvre qu'une ressource est occupée, il demande

à son propriétaire de la libérer. Si ce dernier attend également une autre ressource, il la relâche et reprend à zéro.

Cette solution est comparable à l'approche précédente, mais un thread a l'avantage de pouvoir attendre une ressource. Cela diminue le nombre de reprises à zéro. Cependant, sachez que la gestion de toutes ces requêtes peut être complexe.

Briser l'attente circulaire

Il s'agit de la méthode la plus courante pour éviter les interblocages. Avec la plupart des systèmes, il suffit simplement d'établir une simple convention sur laquelle s'accordent toutes les parties.

Dans l'exemple précédent, où Thread 1 a besoin de Ressource 1 et de Ressource 2, et où Thread 2 a besoin de Ressource 2 puis de Ressource 1, le simple fait d'obliger Thread 1 et Thread 2 à allouer les ressources dans le même ordre rend impossible l'attente circulaire.

Plus généralement, si tous les threads s'accordent sur un ordre global des ressources et s'ils les allouent dans cet ordre, l'interblocage est alors impossible. Comme toutes les autres stratégies, celle-ci peut être à l'origine de problèmes :

- L'ordre d'acquisition peut ne pas correspondre à celui d'utilisation. Ainsi, une ressource obtenue au début peut ne pas être utilisée avant la fin. Des ressources peuvent alors être verrouillées plus longtemps que strictement nécessaire.
- Il est parfois impossible d'imposer un ordre sur l'acquisition des ressources. Si l'identifiant de la seconde ressource est donné par une opération effectuée sur la première, l'établissement d'un ordre n'est pas envisageable.

Il existe ainsi de nombreuses manières d'éviter un interblocage. Certaines conduisent à une famine, tandis que d'autres font une utilisation intensive du processeur et font baisser la réactivité. Rien n'est gratuit, tout a un coût !

En isolant la partie liée aux threads dans votre solution, vous pouvez adapter et expérimenter. Vous disposez d'une approche puissante pour acquérir les connaissances qui vous permettront de retenir les meilleures stratégies.

Tester du code multithread

Comment pouvons-nous écrire un test qui montre que le code suivant est défectueux ?

```
01: public class ClassWithThreadingProblem {
02:     int nextId;
03:
04:     public int takeNextId() {
05:         return nextId++;
06:     }
07: }
```

Voici la description d'un test qui permet d'y parvenir :

- mémoriser la valeur actuelle de `nextId` ;
- créer deux threads, qui appellent tous deux une fois `takeNextId()` ;
- vérifier que la valeur de `nextId` est supérieure de deux à la valeur initiale ;
- recommencer jusqu'à rencontrer le cas où `nextId` n'a été incrémenté que de un, non de deux.

Le Listing A.2 présente le code correspondant.

Listing A.2 : `ClassWithThreadingProblemTest.java`

```
01: package example;
02:
03: import static org.junit.Assert.fail;
04:
05: import org.junit.Test;
06:
07: public class ClassWithThreadingProblemTest {
08:     @Test
09:     public void twoThreadsShouldFailEventually() throws Exception {
10:         final ClassWithThreadingProblem classWithThreadingProblem
                = new ClassWithThreadingProblem();
11:
12:         Runnable runnable = new Runnable() {
13:             public void run() {
14:                 classWithThreadingProblem.takeNextId();
15:             }
16:         };
17:
18:         for (int i = 0; i < 50000; ++i) {
19:             int startingId = classWithThreadingProblem.lastId;
20:             int expectedResult = 2 + startingId;
21:
22:             Thread t1 = new Thread(runnable);
23:             Thread t2 = new Thread(runnable);
24:             t1.start();
25:             t2.start();
26:             t1.join();
27:             t2.join();
28:
29:             int endingId = classWithThreadingProblem.lastId;
30:
31:             if (endingId != expectedResult)
32:                 return;
33:         }
34:
35:         fail("Should have exposed a threading issue but it did not.");
36:     }
37: }
```

Ligne	*Description*
10	Créer une seule instance de `ClassWithThreadingProblem`. Nous devons employer le mot-clé `final` car nous utilisons ensuite cette instance dans une classe interne anonyme.
12 16	Créer une classe interne anonyme qui utilise l'unique instance de `ClassWithThreadingProblem`.
18	Exécuter ce code un nombre de fois "suffisant" pour montrer que le code échoue, mais pas trop élevé pour que le test "ne prenne pas trop de temps". Il faut trouver là le bon équilibre. Nous ne voulons pas attendre trop longtemps pour révéler le dysfonctionnement. Le choix de la valeur limite est difficile, mais nous verrons que nous pouvons grandement réduire ce nombre.
19	Mémoriser la valeur initiale. Ce test tente de prouver que le code de `ClassWithThreadingProblem` n'est pas correct. Si ce test réussit, il aura prouvé que le code est invalide. S'il échoue, le test sera incapable d'apporter cette preuve.
20	Nous nous attendons à ce que la valeur finale soit supérieure de deux à la valeur courante.
22 23	Créer deux threads, chacun utilisant l'objet créé aux lignes 12–16. Nous obtenons ainsi deux threads qui tentent d'utiliser notre unique instance de `ClassWithThreadingProblem` et d'entrer en conflit.
24 25	Faire en sorte que les deux threads puissent s'exécuter.
26 27	Attendre que les deux threads soient terminés avant de vérifier les résultats.
29	Enregistrer la valeur finale actuelle.
31 32	Vérifier si `endingId` diffère de la valeur attendue. Dans l'affirmative, l'instruction `return` termine le test – nous avons prouvé que le code est incorrect. Dans le cas contraire, nous essayons à nouveau.
35	Si nous arrivons à cette ligne, le test n'a pas été en mesure de prouver que le code était incorrect en un temps "raisonnable" ; notre code a échoué. Soit le code est correct, soit nous n'avons pas exécuté un nombre d'itérations suffisant pour rencontrer la condition d'échec.

Le test met en place les conditions d'existence d'un problème de mise à jour concurrente. Toutefois, le problème est si rare que le test ne parviendra généralement pas à le détecter.

Pour vraiment détecter le problème, nous devons fixer le nombre d'itérations à plus de un million. Même dans ce cas, et d'après nos essais, en dix exécutions d'une boucle de

un million d'itérations, le problème ne s'est présenté qu'une seule fois. Autrement dit, nous devons fixer le compteur d'itérations à plus de cent millions pour être quasiment certains de rencontrer le problème. Combien de temps sommes-nous prêts à patienter ?

Même si nous réglons le test pour révéler de façon certaine le dysfonctionnement sur une machine, nous devrons probablement l'ajuster à nouveau avec d'autres valeurs pour montrer ce dysfonctionnement sur un autre système d'exploitation, machine ou version de la JVM.

Par ailleurs, notre problème était *simple*. Si, dans cet exemple, il nous est impossible de montrer facilement que le code est incorrect, comment pourrons-nous le détecter dans des problèmes réellement complexes ?

Quelles approches pouvons-nous prendre pour montrer ce dysfonctionnement simple ? Plus important encore, comment pouvons-nous écrire des tests qui révèlent les dysfonctionnements dans du code plus complexe ? Comment pouvons-nous découvrir que notre code contient des défauts lorsque nous ne savons pas où chercher ?

Voici quelques idées :

- Méthode de Monte-Carlo. Les tests doivent être flexibles afin de pouvoir être ajustés. Ensuite, ils doivent être exécutés encore et encore, par exemple sur un serveur de test, en changeant aléatoirement les réglages. Si les tests n'échouent jamais, le code est incorrect. Vous devez écrire ces tests au plus tôt afin qu'un serveur d'intégration continue commence à les exécuter dès que possible. N'oubliez pas de consigner soigneusement les conditions sous lesquelles le test échoue.
- Exécutez les tests sur chaque plate-forme de déploiement cible, de manière répétée et en continu. Plus les tests s'exécutent longtemps sans échec, plus il est probable que
  - le code de production est correct ;
  - ou les tests ne sont pas adaptés à la révélation des problèmes.
- Exécutez les tests sur une machine dont la charge varie. Si vous pouvez simuler des charges proches d'un environnement de production, n'hésitez pas.

Même après avoir mis en place toutes ces solutions, vous pouvez passer à côté des problèmes liés au multithread dans votre code. Les problèmes les plus insidieux sont ceux qui se produisent dans une toute petite intersection, avec une chance sur des milliards. De tels problèmes sont le cauchemar des systèmes complexes.

Outils de test du code multithread

IBM a développé un outil nommé ConTest[5]. Il permet d'instrumenter des classes afin d'augmenter les chances que le code non sûr vis-à-vis des threads échoue.

Nous n'avons aucun lien direct avec IBM ou l'équipe qui a conçu ConTest. Cette information nous a été apportée par un collègue. Nous avons constaté une nette amélioration dans notre capacité à découvrir des problèmes liés aux threads après quelques minutes d'utilisation.

Voici un bref plan d'utilisation de ConTest :

- Écrivez des tests et du code de production, en vous assurant que des tests sont spécifiquement conçus pour simuler plusieurs utilisateurs sous des charges différentes, comme mentionné précédemment.
- Instrumentez les tests et le code de production avec ConTest.
- Exécutez les tests.

Lorsque nous avons instrumenté du code avec ConTest, notre taux de réussite est passé d'approximativement un dysfonctionnement en dix millions d'itérations à environ un dysfonctionnement en *trente* itérations. Voici les valeurs de boucle qui ont permis de découvrir le dysfonctionnement après instrumentation du code : 13, 23, 0, 54, 16, 14, 6, 69, 107, 49, 2. Il est clair que les classes instrumentées échouent beaucoup plus tôt et de manière beaucoup plus systématique.

Conclusion

Ce chapitre s'est aventuré brièvement sur le territoire très vaste et dangereux de la programmation concurrente. Nous nous sommes focalisés sur les approches qui permettent de garder un code concurrent propre, mais, si vous devez développer des systèmes concurrents, vous devez approfondir vos connaissances. Pour cela, nous vous conseillons de commencer par le livre de Doug Lea, *Concurrent Programming in Java: Design Principles and Patterns* [Lea99, p. 191].

Nous avons traité des mises à jour concurrentes et présenté les disciplines de synchronisation et de verrouillage propres qui permettent de les éviter. Nous avons vu comment les threads permettent d'améliorer le débit d'un système limité par les entrées/sorties et présenté des techniques propres qui permettent d'y parvenir. Nous avons mentionné les interblocages et exposé les techniques qui permettent de les éviter de manière propre. Enfin, nous avons étudié les stratégies qui permettent de révéler les problèmes de concurrence grâce à une instrumentation du code.

5. **http://www haifa ibm.com/projects/verification/contest/index.html**.

Code complet des exemples

Client/serveur monothread

Listing A.3 : `Server.java`

```
package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
    ServerSocket serverSocket;
    volatile boolean keepProcessing = true;

    public Server(int port, int millisecondsTimeout) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(millisecondsTimeout);
    }

    public void run() {
        System.out.printf("Server Starting\n");

        while (keepProcessing) {
            try {
                System.out.printf("accepting client\n");
                Socket socket = serverSocket.accept();
                System.out.printf("got client\n");
                process(socket);
            } catch (Exception e) {
                handle(e);
            }
        }
    }

    private void handle(Exception e) {
        if (!(e instanceof SocketException)) {
            e.printStackTrace();
        }
    }

    public void stopProcessing() {
        keepProcessing = false;
        closeIgnoringException(serverSocket);
    }

    void process(Socket socket) {
        if (socket == null)
            return;
```

```
            try {
                System.out.printf("Server: getting message\n");
                String message = MessageUtils.getMessage(socket);
                System.out.printf("Server: got message: %s\n", message);
                Thread.sleep(1000);
                System.out.printf("Server: sending reply: %s\n", message);
                MessageUtils.sendMessage(socket, "Processed: " + message);
                System.out.printf("Server: sent\n");
                closeIgnoringException(socket);
            } catch (Exception e) {
                e.printStackTrace();
            }

        }

        private void closeIgnoringException(Socket socket) {
            if (socket != null)
                try {
                    socket.close();
                } catch (IOException ignore) {
                }
        }

        private void closeIgnoringException(ServerSocket serverSocket) {
            if (serverSocket != null)
                try {
                    serverSocket.close();
                } catch (IOException ignore) {
                }
        }
    }
```

Listing A.4 : `ClientTest.java`

```
package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
    ServerSocket serverSocket;
    volatile boolean keepProcessing = true;

    public Server(int port, int millisecondsTimeout) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(millisecondsTimeout);
    }

    public void run() {
        System.out.printf("Server Starting\n");
```

```
            while (keepProcessing) {
                try {
                    System.out.printf("accepting client\n");
                    Socket socket = serverSocket.accept();
                    System.out.printf("got client\n");
                    process(socket);
                } catch (Exception e) {
                    handle(e);
                }
            }
        }

        private void handle(Exception e) {
            if (!(e instanceof SocketException)) {
                e.printStackTrace();
            }
        }

        public void stopProcessing() {
            keepProcessing = false;
            closeIgnoringException(serverSocket);
        }

        void process(Socket socket) {
            if (socket == null)
                return;

            try {
                System.out.printf("Server: getting message\n");
                String message = MessageUtils.getMessage(socket);
                System.out.printf("Server: got message: %s\n", message);
                Thread.sleep(1000);
                System.out.printf("Server: sending reply: %s\n", message);
                MessageUtils.sendMessage(socket, "Processed: " + message);
                System.out.printf("Server: sent\n");
                closeIgnoringException(socket);
            } catch (Exception e) {
                e.printStackTrace();
            }
         }

        private void closeIgnoringException(Socket socket) {
            if (socket != null)
                try {
                    socket.close();
                } catch (IOException ignore) {
                }
        }

        private void closeIgnoringException(ServerSocket serverSocket) {
            if (serverSocket != null)
                try {
                    serverSocket.close();
                } catch (IOException ignore) {
                }
        }
    }
```

Listing A.5 : `MessageUtils.java`

```
package common;

import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.net.Socket;

public class MessageUtils {
    public static void sendMessage(Socket socket, String message)
            throws IOException {
        OutputStream stream = socket.getOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(stream);
        oos.writeUTF(message);
        oos.flush();
    }

    public static String getMessage(Socket socket) throws IOException {
        InputStream stream = socket.getInputStream();
        ObjectInputStream ois = new ObjectInputStream(stream);
        return ois.readUTF();
    }
}
```

Client/serveur multithread

Pour que le serveur utilise des threads, il suffit simplement de modifier la méthode `process` (les nouvelles lignes sont en gras) :

```
void process(final Socket socket) {
    if (socket == null)
        return;

    Runnable clientHandler = new Runnable() {
        public void run() {
            try {
                System.out.printf("Server: getting message\n");
                String message = MessageUtils.getMessage(socket);
                System.out.printf("Server: got message: %s\n", message);
                Thread.sleep(1000);
                System.out.printf("Server: sending reply: %s\n", message);
                MessageUtils.sendMessage(socket, "Processed: " + message);
                System.out.printf("Server: sent\n");
                closeIgnoringException(socket);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };

    Thread clientConnection = new Thread(clientHandler);
    clientConnection.start();
}
```

Annexe B

org.jfree.date.SerialDate

Listing B.1 : `SerialDate.Java`

```
 1 /* ===========================================================================
 2  * JCommon : bibliothèque libre de classes générales pour Java(tm)
 3  * ===========================================================================
 4  *
 5  * (C) Copyright 2000-2005, par Object Refinery Limited et les Contributeurs.
 6  *
 7  * Site du projet :  http://www.jfree.org/jcommon/index.html
 8  *
 9  * Cette bibliothèque est un logiciel libre ; vous pouvez la redistribuer et/ou
10  * la modifier en respectant les termes de la GNU Lesser General Public License
11  * publiée par la Free Software Foundation, en version 2.1 ou (selon votre choix)
12  * en toute version ultérieure.
13  *
14  * Cette bibliothèque est distribuée en espérant qu'elle sera utile, mais SANS
15  * AUCUNE GARANTIE, sans même la garantie implicite de QUALITÉ MARCHANDE ou
16  * d'ADÉQUATION À UN OBJECTIF PRÉCIS. Pour de plus amples détails, consultez
17  * la GNU Lesser General Public License.
18  *
19  * Vous devez avoir reçu un exemplaire de la GNU Lesser General Public License
20  * avec cette bibliothèque. Dans le cas contraire, merci d'écrire à la Free
21  * Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
22  * MA 02110-1301, USA.
23  *
24  * [Java est une marque ou une marque déposée de Sun Microsystems, Inc.
25  * aux États-Unis et dans d'autres pays.]
26  *
27  * ---------------
28  * SerialDate.java
29  * ---------------
30  * (C) Copyright 2001-2005, par Object Refinery Limited.
31  *
32  * Auteur :            David Gilbert (pour Object Refinery Limited);
33  * Contributeur(s) :   -;
34  *
35  * $Id: SerialDate.java,v 1.7 2005/11/03 09:25:17 mungady Exp $
36  *
```

```
37  * Modifications (depuis le 11-Oct-2001)
38  * ---------------------------------------
39  * 11-Oct-2001 : Réorganisation de la classe et son déplacement dans le nouveau
40  *               paquetage com.jrefinery.date (DG).
41  * 05-Nov-2001 : Ajout de la méthode getDescription() et suppression de la
42  *               classe NotableDate (DG).
43  * 12-Nov-2001 : IBD a besoin d'une méthode setDescription(), à présent que
44  *               la classe NotableDate a disparu (DG). Modification de
45  *               getPreviousDayOfWeek(), getFollowingDayOfWeek() et
46  *               getNearestDayOfWeek() pour corriger les bogues (DG).
47  * 05-Déc-2001 : Correction du bogue dans la classe SpreadsheetDate (DG).
48  * 29-Mai-2002 : Déplacement des constantes de mois dans une interface séparée
49  *               (MonthConstants) (DG).
50  * 27-Aou-2002 : Correction du bogue dans addMonths(), merci à Nalevka Petr (DG).
51  * 03-Oct-2002 : Corrections des erreurs signalées par Checkstyle (DG).
52  * 13-Mar-2003 : Implémentation de Serializable (DG).
53  * 29-Mai-2003 : Correction du bogue dans la méthode addMonths() (DG).
54  * 04-Sep-2003 : Implémentation de Comparable. Mise à jour Javadoc de isInRange (DG).
55  * 05-Jan-2005 : Correction du bogue dans la méthode addYears() (1096282) (DG).
56  *
57  */
58
59 package org.jfree.date;
60
61 import java.io.Serializable;
62 import java.text.DateFormatSymbols;
63 import java.text.SimpleDateFormat;
64 import java.util.Calendar;
65 import java.util.GregorianCalendar;
66
67 /**
68  *  Classe abstraite qui définit nos exigences quant à la manipulation des dates,
69  *  sans lien avec une implémentation précise.
70  *  <P>
71  *  Exigence 1 : concorder au moins avec la gestion des dates par Excel ;
72  *  Exigence 2 : classe immuable ;
73  *  <P>
74  *  Pourquoi ne pas employer java.util.Date ? Nous le ferons lorsque ce sera sensé.
75  *  Parfois, java.util.Date est *trop* précise - elle représente un instant, au
76  *  1/1000me de seconde (la date dépend elle-même du fuseau horaire). Il arrive
77  *  que nous voulions simplement représenter un jour particulier (par exemple, le
78  *  21 janvier 2015) sans nous préoccuper de l'heure, du fuseau horaire ou d'autres
79  *  paramètres. C'est pourquoi nous avons défini SerialDate.
80  *  <P>
81  *  Vous pouvez invoquer getInstance() pour obtenir une sous-classe concrète de
82  *  SerialDate, sans vous inquiéter de l'implémentation réelle.
83  *
84  * @author David Gilbert
85  */
86 public abstract class SerialDate implements Comparable,
87                                              Serializable,
88                                              MonthConstants {
89
90     /** Pour la sérialisation. */
91     private static final long serialVersionUID = -293716040467423637L;
92
93     /** Symboles du format de date. */
```

```
 94     public static final DateFormatSymbols
 95         DATE_FORMAT_SYMBOLS = new SimpleDateFormat().getDateFormatSymbols();
 96
 97     /** Numéro de série pour le 1 janvier 1900. */
 98     public static final int SERIAL_LOWER_BOUND = 2;
 99
100     /** Numéro de série pour le 31 décembre 9999. */
101     public static final int SERIAL_UPPER_BOUND = 2958465;
102
103     /** La première année reconnue par ce format de date. */
104     public static final int MINIMUM_YEAR_SUPPORTED = 1900;
105
106     /** La dernière année reconnue par ce format de date. */
107     public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
108
109     /** Constante utile pour Monday (lundi). Équivaut à java.util.Calendar.MONDAY. */
110     public static final int MONDAY = Calendar.MONDAY;
111
112     /**
113      * Constante utile pour Tuesday (mardi). Équivaut à java.util.Calendar.TUESDAY.
114      */
115     public static final int TUESDAY = Calendar.TUESDAY;
116
117     /**
118      * Constante utile pour Wednesday (mercredi). Équivaut à
119      * java.util.Calendar.WEDNESDAY.
120      */
121     public static final int WEDNESDAY = Calendar.WEDNESDAY;
122
123     /**
124      * Constante utile pour Thursday (jeudi). Équivaut à java.util.Calendar.THURSDAY.
125      */
126     public static final int THURSDAY = Calendar.THURSDAY;
127
128     /** Constante utile pour Friday (vendredi). Équivaut à java.util.Calendar.FRIDAY. */
129     public static final int FRIDAY = Calendar.FRIDAY;
130
131     /**
132      * Constante utile pour Saturday (samedi). Équivaut à java.util.Calendar.SATURDAY.
133      */
134     public static final int SATURDAY = Calendar.SATURDAY;
135
136     /** Constante utile pour Sunday (dimanche). Équivaut à java.util.Calendar.SUNDAY. */
137     public static final int SUNDAY = Calendar.SUNDAY;
138
139     /** Nombre de jours dans chaque mois, pour les années non bissextiles. */
140     static final int[] LAST_DAY_OF_MONTH =
141         {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
142
143     /** Nombre de jours dans une année (non bissextile) jusqu'à la fin de chaque mois. */
144     static final int[] AGGREGATE_DAYS_TO_END_OF_MONTH =
145         {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
146
147     /** Nombre de jours dans une année jusqu'à la fin du mois précédent. */
148     static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
149         {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
150
```

```
151     /** Nombre de jours dans une année bissextile jusqu'à la fin de chaque mois. */
152     static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH =
153         {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
154
155     /**
156      * Nombre de jours dans une année bissextile jusqu'à la fin du mois précédent.
157      */
158     static final int[]
159         LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
160             {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
161
162     /** Constante utile pour désigner la première semaine d'un mois. */
163     public static final int FIRST_WEEK_IN_MONTH = 1;
164
165     /** Constante utile pour désigner la deuxième semaine d'un mois. */
166     public static final int SECOND_WEEK_IN_MONTH = 2;
167
168     /** Constante utile pour désigner la troisième semaine d'un mois. */
169     public static final int THIRD_WEEK_IN_MONTH = 3;
170
171     /** Constante utile pour désigner la quatrième semaine d'un mois. */
172     public static final int FOURTH_WEEK_IN_MONTH = 4;
173
174     /** Constante utile pour désigner la dernière semaine d'un mois. */
175     public static final int LAST_WEEK_IN_MONTH = 0;
176
177     /** Constante utile pour un intervalle. */
178     public static final int INCLUDE_NONE = 0;
179
180     /** Constante utile pour un intervalle. */
181     public static final int INCLUDE_FIRST = 1;
182
183     /** Constante utile pour un intervalle. */
184     public static final int INCLUDE_SECOND = 2;
185
186     /** Constante utile pour un intervalle. */
187     public static final int INCLUDE_BOTH = 3;
188
189     /**
190      * Constante utile pour désigner un jour de la semaine relativement à
191      * une date fixée.
192      */
193     public static final int PRECEDING = -1;
194
195     /**
196      * Constante utile pour désigner un jour de la semaine relativement à
197      * une date fixée.
198      */
199     public static final int NEAREST = 0;
200
201     /**
202      * Constante utile pour désigner un jour de la semaine relativement à
203      * une date fixée.
204      */
205     public static final int FOLLOWING = 1;
206
207     /** Description de la date. */
```

```
208     private String description;
209
210     /**
211      * Constructeur par défaut.
212      */
213     protected SerialDate() {
214     }
215
216     /**
217      * Retourne <code>true</code> si le code entier indiqué représente un jour
218      * de la semaine valide, sinon <code>false</code>.
219      *
220      * @param code  le code dont la validité est testée.
221      *
222      * @return <code>true</code> si le code entier indiqué représente un jour
223      *         de la semaine valide, sinon <code>false</code>.
224      */
225     public static boolean isValidWeekdayCode(final int code) {
226
227         switch(code) {
228             case SUNDAY:
229             case MONDAY:
230             case TUESDAY:
231             case WEDNESDAY:
232             case THURSDAY:
233             case FRIDAY:
234             case SATURDAY:
235                 return true;
236             default:
237                 return false;
238         }
239
240     }
241
242     /**
243      * Convertit la chaîne indiquée en un jour de la semaine.
244      *
245      * @param s  une chaîne représentant le jour de la semaine.
246      *
247      * @return <code>-1</code> si la chaîne ne peut pas être convertie, sinon le
248      *         jour de la semaine.
249      */
250     public static int stringToWeekdayCode(String s) {
251
252         final String[] shortWeekdayNames
253             = DATE_FORMAT_SYMBOLS.getShortWeekdays();
254         final String[] weekDayNames = DATE_FORMAT_SYMBOLS.getWeekdays();
255
256         int result = -1;
257         s = s.trim();
258         for (int i = 0; i < weekDayNames.length; i++) {
259             if (s.equals(shortWeekdayNames[i])) {
260                 result = i;
261                 break;
262             }
263             if (s.equals(weekDayNames[i])) {
264                 result = i;
```

```
265                 break;
266             }
267         }
268         return result;
269 
270     }
271 
272     /**
273      * Retourne une chaîne représentant le jour de la semaine indiqué.
274      * <P>
275      * Il faut trouver une meilleure solution.
276      *
277      * @param weekday  le jour de la semaine.
278      *
279      * @return une chaîne représentant le jour de la semaine indiqué.
280      */
281     public static String weekdayCodeToString(final int weekday) {
282 
283         final String[] weekdays = DATE_FORMAT_SYMBOLS.getWeekdays();
284         return weekdays[weekday];
285 
286     }
287 
288     /**
289      * Retourne un tableau des noms de mois.
290      *
291      * @return un tableau des noms de mois.
292      */
293     public static String[] getMonths() {
294 
295         return getMonths(false);
296 
297     }
298 
299     /**
300      * Retourne un tableau des noms de mois.
301      *
302      * @param shortened  indicateur indiquant que des noms de mois abrégés
303      *                   doivent être retournés.
304      *
305      * @return un tableau des noms de mois.
306      */
307     public static String[] getMonths(final boolean shortened) {
308 
309         if (shortened) {
310             return DATE_FORMAT_SYMBOLS.getShortMonths();
311         }
312         else {
313             return DATE_FORMAT_SYMBOLS.getMonths();
314         }
315 
316     }
317 
318     /**
319      * Retourne true si le code entier indiqué représente un mois valide.
320      *
321      * @param code  le code dont la validité est testée.
322      *
```

```
323      * @return <code>true</code> si le code entier indiqué représente un
324      *         mois valide.
325      */
326     public static boolean isValidMonthCode(final int code) {
327
328         switch(code) {
329             case JANUARY:
330             case FEBRUARY:
331             case MARCH:
332             case APRIL:
333             case MAY:
334             case JUNE:
335             case JULY:
336             case AUGUST:
337             case SEPTEMBER:
338             case OCTOBER:
339             case NOVEMBER:
340             case DECEMBER:
341                 return true;
342             default:
343                 return false;
344         }
345
346     }
347
348     /**
349      * Retourne le trimestre du mois indiqué.
350      *
351      * @param code  le code du mois (1-12).
352      *
353      * @return le trimestre auquel appartient le mois.
354      * @throws java.lang.IllegalArgumentException
355      */
356     public static int monthCodeToQuarter(final int code) {
357
358         switch(code) {
359             case JANUARY:
360             case FEBRUARY:
361             case MARCH: return 1;
362             case APRIL:
363             case MAY:
364             case JUNE: return 2;
365             case JULY:
366             case AUGUST:
367             case SEPTEMBER: return 3;
368             case OCTOBER:
369             case NOVEMBER:
370             case DECEMBER: return 4;
371             default: throw new IllegalArgumentException(
372                 "SerialDate.monthCodeToQuarter: invalid month code.");
373         }
374
375     }
376
377     /**
378      * Retourne une chaîne représentant le mois indiqué.
379      * <P>
```

```
380      * La chaîne retournée correspond à la forme longue du nom du mois
381      * pour les paramètres régionaux par défaut.
382      *
383      * @param month  le mois.
384      *
385      * @return une chaîne représentant le mois indiqué.
386      */
387     public static String monthCodeToString(final int month) {
388
389         return monthCodeToString(month, false);
390
391     }
392
393     /**
394      * Retourne une chaîne représentant le mois indiqué.
395      * <P>
396      * La chaîne retournée correspond à la forme longue ou courte du nom
397      * du mois pour les paramètres régionaux par défaut.
398      *
399      * @param month  le mois.
400      * @param shortened  si <code>true</code>, retourne le nom abrégé
401      *                   du mois.
402      *
403      * @return une chaîne représentant le mois indiqué.
404      * @throws java.lang.IllegalArgumentException
405      */
406     public static String monthCodeToString(final int month,
407                                            final boolean shortened) {
408
409         // Vérifier les arguments...
410         if (!isValidMonthCode(month)) {
411             throw new IllegalArgumentException(
412                 "SerialDate.monthCodeToString: month outside valid range.");
413         }
414
415         final String[] months;
416
417         if (shortened) {
418             months = DATE_FORMAT_SYMBOLS.getShortMonths();
419         }
420         else {
421             months = DATE_FORMAT_SYMBOLS.getMonths();
422         }
423
424         return months[month - 1];
425
426     }
427
428     /**
429      * Convertit une chaîne en un code de mois.
430      * <P>
431      * Cette méthode retourne l'une des constantes JANUARY, FEBRUARY, ...,
432      * DECEMBER qui correspond à la chaîne. Si la chaîne n'est pas reconnue
433      * cette méthode retourne -1.
434      *
435      * @param s  la chaîne à analyser.
436      *
```

```
437      * @return <code>-1</code> si la chaîne ne peut pas être analysée, sinon
438      *         le mois de l'année.
439      */
440     public static int stringToMonthCode(String s) {
441
442         final String[] shortMonthNames = DATE_FORMAT_SYMBOLS.getShortMonths();
443         final String[] monthNames = DATE_FORMAT_SYMBOLS.getMonths();
444
445         int result = -1;
446         s = s.trim();
447
448         // Commencer par convertir la chaîne en un entier (1-12)...
449         try {
450             result = Integer.parseInt(s);
451         }
452         catch (NumberFormatException e) {
453             // Supprimé.
454         }
455
456         // Rechercher ensuite parmi les noms de mois...
457         if ((result < 1) || (result > 12)) {
458             for (int i = 0; i < monthNames.length; i++) {
459                 if (s.equals(shortMonthNames[i])) {
460                     result = i + 1;
461                     break;
462                 }
463                 if (s.equals(monthNames[i])) {
464                     result = i + 1;
465                     break;
466                 }
467             }
468         }
469
470         return result;
471
472     }
473
474     /**
475      * Retourne true si le code entier indiqué représente une semaine du mois valide,
476      * sinon false.
477      *
478      * @param code  le code dont la validité est testée.
479      * @return <code>true</code> si le code entier indiqué représente une semaine
480      *         du mois valide.
481      */
482     public static boolean isValidWeekInMonthCode(final int code) {
483
484         switch(code) {
485             case FIRST_WEEK_IN_MONTH:
486             case SECOND_WEEK_IN_MONTH:
487             case THIRD_WEEK_IN_MONTH:
488             case FOURTH_WEEK_IN_MONTH:
489             case LAST_WEEK_IN_MONTH: return true;
490             default: return false;
491         }
492
493     }
```

```
494
495     /**
496      * Détermine si l'année indiquée est une année bissextile.
497      *
498      * @param yyyy  l'année (dans la plage 1900 à 9999).
499      *
500      * @return <code>true</code> si l'année indiquée est une année bissextile.
501      */
502     public static boolean isLeapYear(final int yyyy) {
503
504         if ((yyyy % 4) != 0) {
505             return false;
506         }
507         else if ((yyyy % 400) == 0) {
508             return true;
509         }
510         else if ((yyyy % 100) == 0) {
511             return false;
512         }
513         else {
514             return true;
515         }
516
517     }
518
519     /**
520      * Retourne le nombre d'années bissextiles entre 1900 et l'année indiquée
521      * COMPRISE.
522      * <P>
523      * Notez que 1900 n'est pas une année bissextile.
524      *
525      * @param yyyy  l'année (dans la plage 1900 à 9999).
526      *
527      * @return le nombre d'années bissextiles entre 1900 et l'année indiquée.
528      */
529     public static int leapYearCount(final int yyyy) {
530
531         final int leap4 = (yyyy - 1896) / 4;
532         final int leap100 = (yyyy - 1800) / 100;
533         final int leap400 = (yyyy - 1600) / 400;
534         return leap4 - leap100 + leap400;
535
536     }
537
538     /**
539      * Retourne le numéro du dernier jour du mois, en tenant compte des années
540      * bissextiles.
541      *
542      * @param month  le mois.
543      * @param yyyy  l'année (dans la plage 1900 à 9999).
544      *
545      * @return le numéro du dernier jour du mois.
546      */
547     public static int lastDayOfMonth(final int month, final int yyyy) {
548
549         final int result = LAST_DAY_OF_MONTH[month];
550         if (month != FEBRUARY) {
```

```
551             return result;
552         }
553         else if (isLeapYear(yyyy)) {
554             return result + 1;
555         }
556         else {
557             return result;
558         }
559
560     }
561
562     /**
563      * Crée une nouvelle date en ajoutant le nombre de jours indiqué à
564      * la date de base.
565      *
566      * @param days  le nombre de jours à ajouter (peut être négatif).
567      * @param base  la date de base.
568      *
569      * @return une nouvelle date.
570      */
571     public static SerialDate addDays(final int days, final SerialDate base) {
572
573         final int serialDayNumber = base.toSerial() + days;
574         return SerialDate.createInstance(serialDayNumber);
575
576     }
577
578     /**
579      * Crée une nouvelle date en ajoutant le nombre de mois indiqué à
580      * la date de base.
581      * <P>
582      * Si la date de base est proche de la fin du mois, le jour du résultat
583      * peut être ajusté :  31 mai + 1 mois = 30 juin.
584      *
585      * @param months  le nombre de mois à ajouter (peut être négatif).
586      * @param base  la date de base.
587      *
588      * @return une nouvelle date.
589      */
590     public static SerialDate addMonths(final int months,
591                                        final SerialDate base) {
592
593         final int yy = (12 * base.getYYYY() + base.getMonth() + months - 1)
594                        / 12;
595         final int mm = (12 * base.getYYYY() + base.getMonth() + months - 1)
596                        % 12 + 1;
597         final int dd = Math.min(
598             base.getDayOfMonth(), SerialDate.lastDayOfMonth(mm, yy)
599         );
600         return SerialDate.createInstance(dd, mm, yy);
601
602     }
603
604     /**
605      * Crée une nouvelle date en ajoutant le nombre d'années indiqué à
606      * la date de base.
607      *
```

```
608      * @param years  le nombre d'années à ajouter (peut être négatif).
609      * @param base  la date de base.
610      *
611      * @return une nouvelle date.
612      */
613     public static SerialDate addYears(final int years, final SerialDate base) {
614
615         final int baseY = base.getYYYY();
616         final int baseM = base.getMonth();
617         final int baseD = base.getDayOfMonth();
618
619         final int targetY = baseY + years;
620         final int targetD = Math.min(
621             baseD, SerialDate.lastDayOfMonth(baseM, targetY)
622         );
623
624         return SerialDate.createInstance(targetD, baseM, targetY);
625
626     }
627
628     /**
629      * Retourne la dernière date qui tombe le jour de la semaine indiqué
630      * AVANT la date de base.
631      *
632      * @param targetWeekday  code du jour de la semaine cible.
633      * @param base  la date de base.
634      *
635      * @return la dernière date qui tombe le jour de la semaine indiqué
636      *         AVANT la date de base.
637      */
638     public static SerialDate getPreviousDayOfWeek(final int targetWeekday,
639                                                    final SerialDate base) {
640
641         // Vérifier les arguments...
642         if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
643             throw new IllegalArgumentException(
644                 "Invalid day-of-the-week code."
645             );
646         }
647
648         // Rechercher la date...
649         final int adjust;
650         final int baseDOW = base.getDayOfWeek();
651         if (baseDOW > targetWeekday) {
652             adjust = Math.min(0, targetWeekday - baseDOW);
653         }
654         else {
655             adjust = -7 + Math.max(0, targetWeekday - baseDOW);
656         }
657
658         return SerialDate.addDays(adjust, base);
659
660     }
661
662     /**
663      * Retourne la première date qui tombe le jour de la semaine indiqué
664      * APRÈS la date de base.
```

```
665      *
666      * @param targetWeekday  code du jour de la semaine cible.
667      * @param base  la date de base.
668      *
669      * @return la première date qui tombe le jour de la semaine indiqué
670      *         APRÈS la date de base.
671      */
672     public static SerialDate getFollowingDayOfWeek(final int targetWeekday,
673                                                     final SerialDate base) {
674
675         // Vérifier les arguments...
676         if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
677             throw new IllegalArgumentException(
678                 "Invalid day-of-the-week code."
679             );
680         }
681
682         // Rechercher la date...
683         final int adjust;
684         final int baseDOW = base.getDayOfWeek();
685         if (baseDOW > targetWeekday) {
686             adjust = 7 + Math.min(0, targetWeekday - baseDOW);
687         }
688         else {
689             adjust = Math.max(0, targetWeekday - baseDOW);
690         }
691
692         return SerialDate.addDays(adjust, base);
693     }
694
695     /**
696      * Retourne la date qui tombe le jour de la semaine indiqué
697      * PROCHE de la date de base.
698      *
699      * @param targetDOW  code du jour de la semaine cible.
700      * @param base  la date de base.
701      *
702      * @return la date qui tombe le jour de la semaine indiqué
703      *         PROCHE de la date de base.
704      */
705     public static SerialDate getNearestDayOfWeek(final int targetDOW,
706                                                   final SerialDate base) {
707
708         // Vérifier les arguments...
709         if (!SerialDate.isValidWeekdayCode(targetDOW)) {
710             throw new IllegalArgumentException(
711                 "Invalid day-of-the-week code."
712             );
713         }
714
715         // Rechercher la date...
716         final int baseDOW = base.getDayOfWeek();
717         int adjust = -Math.abs(targetDOW - baseDOW);
718         if (adjust >= 4) {
719             adjust = 7 - adjust;
720         }
721         if (adjust <= -4) {
```

```
722             adjust = 7 + adjust;
723         }
724         return SerialDate.addDays(adjust, base);
725
726     }
727
728     /**
729      * Avance la date jusqu'au dernier jour du mois.
730      *
731      * @param base  la date de base.
732      *
733      * @return une nouvelle date.
734      */
735     public SerialDate getEndOfCurrentMonth(final SerialDate base) {
736         final int last = SerialDate.lastDayOfMonth(
737             base.getMonth(), base.getYYYY()
738         );
739         return SerialDate.createInstance(last, base.getMonth(), base.getYYYY());
740     }
741
742     /**
743      * Retourne une chaîne qui correspond au code de la semaine du mois.
744      * <P>
745      * Il faut trouver une meilleure solution.
746      *
747      * @param count  code entier représentant la semaine du mois.
748      *
749      * @return une chaîne qui correspond au code de la semaine du mois.
750      */
751     public static String weekInMonthToString(final int count) {
752
753         switch (count) {
754             case SerialDate.FIRST_WEEK_IN_MONTH : return "First";
755             case SerialDate.SECOND_WEEK_IN_MONTH : return "Second";
756             case SerialDate.THIRD_WEEK_IN_MONTH : return "Third";
757             case SerialDate.FOURTH_WEEK_IN_MONTH : return "Fourth";
758             case SerialDate.LAST_WEEK_IN_MONTH : return "Last";
759             default :
760                 return "SerialDate.weekInMonthToString(): invalid code.";
761         }
762
763     }
764
765     /**
766      * Retourne une chaîne représentant la notion 'relative' indiquée.
767      * <P>
768      * Il faut trouver une meilleure solution.
769      *
770      * @param relative  constante représantant la notion 'relative'.
771      *
772      * @return une chaîne représentant la notion 'relative' indiquée.
773      */
774     public static String relativeToString(final int relative) {
775
776         switch (relative) {
777             case SerialDate.PRECEDING : return "Preceding";
778             case SerialDate.NEAREST : return "Nearest";
```

```
779             case SerialDate.FOLLOWING : return "Following";
780             default : return "ERROR : Relative To String";
781         }
782
783     }
784
785     /**
786      * Méthode de fabrique qui retourne une instance d'une certaine sous-classe concrète
787      * de {@link SerialDate}.
788      *
789      * @param day  le jour (1-31).
790      * @param month  le mois (1-12).
791      * @param yyyy  l'année (dans la plage 1900 à 9999).
792      *
793      * @return une instance de {@link SerialDate}.
794      */
795     public static SerialDate createInstance(final int day, final int month,
796                                             final int yyyy) {
797         return new SpreadsheetDate(day, month, yyyy);
798     }
799
800     /**
801      * Méthode de fabrique qui retourne une instance d'une certaine sous-classe concrète
802      * de {@link SerialDate}.
803      *
804      * @param serial  numéro de série du jour (1 janvier 1900 = 2).
805      *
806      * @return une instance de SerialDate.
807      */
808     public static SerialDate createInstance(final int serial) {
809         return new SpreadsheetDate(serial);
810     }
811
812     /**
813      * Méthode de fabrique qui retourne une instance d'une sous-classe de SerialDate.
814      *
815      * @param date  objet date de Java.
816      *
817      * @return une instance de SerialDate.
818      */
819     public static SerialDate createInstance(final java.util.Date date) {
820
821         final GregorianCalendar calendar = new GregorianCalendar();
822         calendar.setTime(date);
823         return new SpreadsheetDate(calendar.get(Calendar.DATE),
824                                    calendar.get(Calendar.MONTH) + 1,
825                                    calendar.get(Calendar.YEAR));
826
827     }
828
829     /**
830      * Retourne le numéro série de la date, où le 1 janvier 1900 = 2 (cela
831      * correspond, presque, au système de numérotation employé dans Microsoft
832      * Excel pour Windows et Lotus 1-2-3).
833      *
834      * @return le numéro série de la date.
835      */
```

```
836     public abstract int toSerial();
837
838     /**
839      * Retourne un java.util.Date. Puisque java.util.Date est plus précis que
840      * SerialDate, nous devons définir une convention pour 'l'heure du jour'.
841      *
842      * @return this sous forme de <code>java.util.Date</code>.
843      */
844     public abstract java.util.Date toDate();
845
846     /**
847      * Retourne une description de la date.
848      *
849      * @return une description de la date.
850      */
851     public String getDescription() {
852         return this.description;
853     }
854
855     /**
856      * Fixe la description de la date.
857      *
858      * @param description  la nouvelle description de la date.
859      */
860     public void setDescription(final String description) {
861         this.description = description;
862     }
863
864     /**
865      * Convertit la date en une chaîne de caractères.
866      *
867      * @return  une représentation de la date sous forme de chaîne.
868      */
869     public String toString() {
870         return getDayOfMonth() + "-" + SerialDate.monthCodeToString(getMonth())
871                                + "-" + getYYYY();
872     }
873
874     /**
875      * Retourne l'année (suppose une plage valide de 1900 à 9999).
876      *
877      * @return l'année.
878      */
879     public abstract int getYYYY();
880
881     /**
882      * Retourne le mois (janvier = 1, février = 2, mars = 3).
883      *
884      * @return le mois de l'année.
885      */
886     public abstract int getMonth();
887
888     /**
889      * Retourne le jour du mois.
890      *
891      * @return le jour du mois.
892      */
893     public abstract int getDayOfMonth();
```

```
894
895     /**
896      * Retourne le jour de la semaine.
897      *
898      * @return le jour de la semaine.
899      */
900     public abstract int getDayOfWeek();
901
902     /**
903      * Retourne la différence (en jours) entre cette date et l'autre date
904      * indiquée.
905      * <P>
906      * Le résultat est positif si cette date se trouve après l'autre date,
907      * et négatif si elle est avant l'autre date.
908      *
909      * @param other  la date servant à la comparaison.
910      *
911      * @return la différence entre cette date et l'autre date.
912      */
913     public abstract int compare(SerialDate other);
914
915     /**
916      * Retourne true si ce SerialDate représente la même date que
917      * le SerialDate indiqué.
918      *
919      * @param other  la date servant à la comparaison.
920      *
921      * @return <code>true</code> si ce SerialDate représente la même date que
922      *         le SerialDate indiqué.
923      */
924     public abstract boolean isOn(SerialDate other);
925
926     /**
927      * Retourne true si ce SerialDate représente une date antérieure au
928      * SerialDate indiqué.
929      *
930      * @param other  la date servant à la comparaison.
931      *
932      * @return <code>true</code> si ce SerialDate représente une date antérieure
933      *         au SerialDate indiqué.
934      */
935     public abstract boolean isBefore(SerialDate other);
936
937     /**
938      * Retourne true si ce SerialDate représente la même date que
939      * le SerialDate indiqué.
940      *
941      * @param other  la date servant à la comparaison.
942      *
943      * @return <code>true<code> si ce SerialDate représente la même date que
944      *         le SerialDate indiqué.
945      */
946     public abstract boolean isOnOrBefore(SerialDate other);
947
948     /**
949      * Retourne true si ce SerialDate représente la même date que
950      * le SerialDate indiqué.
951      *
```

```
952      * @param other  la date servant à la comparaison.
953      *
954      * @return <code>true</code> si ce SerialDate représente la même date que
955      *         le SerialDate indiqué.
956      */
957     public abstract boolean isAfter(SerialDate other);
958
959     /**
960      * Retourne true si ce SerialDate représente la même date que
961      * le SerialDate indiqué.
962      *
963      * @param other  la date servant à la comparaison.
964      *
965      * @return <code>true</code> si ce SerialDate représente la même date que
966      *         le SerialDate indiqué.
967      */
968     public abstract boolean isOnOrAfter(SerialDate other);
969
970     /**
971      * Retourne <code>true</code> si ce {@link SerialDate} se trouve dans
972      * la plage indiquée (INCLUSIF). L'ordre des dates d1 et d2 n'est pas
973      * important.
974      *
975      * @param d1  une date limite de la plage.
976      * @param d2  l'autre date limite de la plage.
977      *
978      * @return un booléen.
979      */
980     public abstract boolean isInRange(SerialDate d1, SerialDate d2);
981
982     /**
983      * Retourne <code>true</code> si ce {@link SerialDate} se trouve dans
984      * la plage indiquée (l'appelant précise si les extrémités sont
985      * incluses). L'ordre des dates d1 et d2 n'est pas important.
986      *
987      * @param d1  une date limite de la plage.
988      * @param d2  l'autre date limite de la plage.
989      * @param include  code qui indique si les dates de début et de fin
990      *                 sont incluses dans la plage.
991      *
992      * @return un booléen.
993      */
994     public abstract boolean isInRange(SerialDate d1, SerialDate d2,
995                                       int include);
996
997     /**
998      * Retourne la dernière date qui tombe le jour de la semaine indiqué
999      * AVANT cette date.
1000     *
1001     * @param targetDOW  code pour le jour de la semaine cible.
1002     *
1003     * @return la dernière date qui tombe le jour de la semaine indiqué
1004     *         AVANT cette date.
1005     */
1006    public SerialDate getPreviousDayOfWeek(final int targetDOW) {
1007        return getPreviousDayOfWeek(targetDOW, this);
1008    }
```

```
1009
1010     /**
1011      * Retourne la première date qui tombe le jour de la semaine indiqué
1012      * APRÈS cette date.
1013      *
1014      * @param targetDOW  code pour le jour de la semaine cible.
1015      *
1016      * @return la première date qui tombe le jour de la semaine indiqué
1017      *         APRÈS cette date.
1018      */
1019     public SerialDate getFollowingDayOfWeek(final int targetDOW) {
1020         return getFollowingDayOfWeek(targetDOW, this);
1021     }
1022
1023     /**
1024      * Retourne la date la plus proche qui tombe le jour de la semaine indiqué.
1025      *
1026      * @param targetDOW  code pour le jour de la semaine cible.
1027      *
1028      * @return la date la plus proche qui tombe le jour de la semaine indiqué.
1029      */
1030     public SerialDate getNearestDayOfWeek(final int targetDOW) {
1031         return getNearestDayOfWeek(targetDOW, this);
1032     }
1033
1034 }
```

Listing B.2 : `SerialDateTest.java`

```
 1 /* ===========================================================================
 2  * JCommon : bibliothèque libre de classes générales pour Java(tm)
 3  * ===========================================================================
 4  *
 5  * (C) Copyright 2000-2005, par Object Refinery Limited et les Contributeurs.
 6  *
 7  * Site du projet :  http://www.jfree.org/jcommon/index.html
 8  *
 9  * Cette bibliothèque est un logiciel libre ; vous pouvez la redistribuer et/ou
10  * la modifier en respectant les termes de la GNU Lesser General Public License
11  * publiée par la Free Software Foundation, en version 2.1 ou (selon votre choix)
12  * en toute version ultérieure.
13  *
14  * Cette bibliothèque est distribuée en espérant qu'elle sera utile, mais SANS
15  * AUCUNE GARANTIE, sans même la garantie implicite de QUALITÉ MARCHANDE ou
16  * d'ADÉQUATION À UN OBJECTIF PRÉCIS. Pour de plus amples détails, consultez
17  * la GNU Lesser General Public License.
18  *
19  * Vous devez avoir reçu un exemplaire de la GNU Lesser General Public License
20  * avec cette bibliothèque. Dans le cas contraire, merci d'écrire à la Free
21  * Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
22  * MA 02110-1301, USA.
23  *
24  * [Java est une marque ou une marque déposée de Sun Microsystems, Inc.
25  * aux États-Unis et dans d'autres pays.]
26  *
```

```
27  * --------------------
28  * SerialDateTests.java
29  * --------------------
30  * (C) Copyright 2001-2005, par Object Refinery Limited.
31  *
32  * Auteur :            David Gilbert (pour Object Refinery Limited);
33  * Contributeur(s) :   -;
34  *
35  * $Id: SerialDateTests.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
36  *
37  * Modifications
38  * -------------
39  * 15-Nov-2001 : Version 1 (DG);
40  * 25-Jui-2002 : Suppression des importations inutiles (DG);
41  * 24-Oct-2002 : Correction des erreurs signalées par Checkstyle (DG);
42  * 13-Mar-2003 : Ajout du test de sérialisation (DG);
43  * 05-Jan-2005 : Ajout du test pour le bogue 1096282 (DG);
44  *
45  */
46
47 package org.jfree.date.junit;
48
49 import java.io.ByteArrayInputStream;
50 import java.io.ByteArrayOutputStream;
51 import java.io.ObjectInput;
52 import java.io.ObjectInputStream;
53 import java.io.ObjectOutput;
54 import java.io.ObjectOutputStream;
55
56 import junit.framework.Test;
57 import junit.framework.TestCase;
58 import junit.framework.TestSuite;
59
60 import org.jfree.date.MonthConstants;
61 import org.jfree.date.SerialDate;
62
63 /**
64  * Quelques tests JUnit pour la classe {@link SerialDate}.
65  */
66 public class SerialDateTests extends TestCase {
67
68     /** Date représentant le 9 novembre. */
69     private SerialDate nov9Y2001;
70
71     /**
72      * Crée un nouveau cas de test.
73      *
74      * @param name  le nom.
75      */
76     public SerialDateTests(final String name) {
77         super(name);
78     }
79
80     /**
81      * Retourne une suite de tests pour l'exécuteur de tests JUnit.
82      *
83      * @return la suite de tests.
```

```
 84      */
 85     public static Test suite() {
 86         return new TestSuite(SerialDateTests.class);
 87     }
 88
 89     /**
 90      * Configure le problème.
 91      */
 92     protected void setUp() {
 93         this.nov9Y2001 = SerialDate.createInstance(9, MonthConstants.NOVEMBER, 2001);
 94     }
 95
 96     /**
 97      * 9 Nov 2001 plus deux mois doit donner 9 Jan 2002.
 98      */
 99     public void testAddMonthsTo9Nov2001() {
100         final SerialDate jan9Y2002 = SerialDate.addMonths(2, this.nov9Y2001);
101         final SerialDate answer = SerialDate.createInstance(9, 1, 2002);
102         assertEquals(answer, jan9Y2002);
103     }
104
105     /**
106      * Cas de test pour un bogue signalé, à présent corrigé.
107      */
108     public void testAddMonthsTo5Oct2003() {
109         final SerialDate d1 = SerialDate.createInstance(5, MonthConstants.OCTOBER, 2003);
110         final SerialDate d2 = SerialDate.addMonths(2, d1);
111         assertEquals(d2, SerialDate.createInstance(5, MonthConstants.DECEMBER, 2003));
112     }
113
114     /**
115      * Cas de test pour un bogue signalé, à présent corrigé.
116      */
117     public void testAddMonthsTo1Jan2003() {
118         final SerialDate d1 = SerialDate.createInstance(1, MonthConstants.JANUARY, 2003);
119         final SerialDate d2 = SerialDate.addMonths(0, d1);
120         assertEquals(d2, d1);
121     }
122
123     /**
124      * Le lundi qui précède le vendredi 9 novembre 2001 doit être le 5 novembre.
125      */
126     public void testMondayPrecedingFriday9Nov2001() {
127         SerialDate mondayBefore = SerialDate.getPreviousDayOfWeek(
128             SerialDate.MONDAY, this.nov9Y2001
129         );
130         assertEquals(5, mondayBefore.getDayOfMonth());
131     }
132
133     /**
134      * Le lundi qui suit le vendredi 9 novembre 2001 doit être le 12 novembre.
135      */
136     public void testMondayFollowingFriday9Nov2001() {
137         SerialDate mondayAfter = SerialDate.getFollowingDayOfWeek(
138             SerialDate.MONDAY, this.nov9Y2001
139         );
140         assertEquals(12, mondayAfter.getDayOfMonth());
```

```
141     }
142
143     /**
144      * Le lundi le plus proche du vendredi 9 novembre 2000 doit être le 12 novembre.
145      */
146     public void testMondayNearestFriday9Nov2001() {
147         SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(
148             SerialDate.MONDAY, this.nov9Y2001
149         );
150         assertEquals(12, mondayNearest.getDayOfMonth());
151     }
152
153     /**
154      * Le lundi le plus proche du 22 janvier 1970 tombe le 19.
155      */
156     public void testMondayNearest22Jan1970() {
157         SerialDate jan22Y1970 = SerialDate.createInstance(22, MonthConstants.JANUARY, 1970);
158         SerialDate mondayNearest=SerialDate.getNearestDayOfWeek(SerialDate.MONDAY, jan22Y1970);
159         assertEquals(19, mondayNearest.getDayOfMonth());
160     }
161
162     /**
163      * Vérifie que la conversion des jours en chaînes retourne le bon résultat. En réalité,
164      * ce résultat dépend des paramètres régionaux et ce test doit être modifié.
165      */
166     public void testWeekdayCodeToString() {
167
168         final String test = SerialDate.weekdayCodeToString(SerialDate.SATURDAY);
169         assertEquals("Saturday", test);
170
171     }
172
173     /**
174      * Teste la conversion d'une chaîne en un jour de la semaine. Ce test échouera si les
175      * paramètres régionaux n'utilisent pas les noms anglais... trouver un meilleur test !
176      */
177     public void testStringToWeekday() {
178
179         int weekday = SerialDate.stringToWeekdayCode("Wednesday");
180         assertEquals(SerialDate.WEDNESDAY, weekday);
181
182         weekday = SerialDate.stringToWeekdayCode(" Wednesday ");
183         assertEquals(SerialDate.WEDNESDAY, weekday);
184
185         weekday = SerialDate.stringToWeekdayCode("Wed");
186         assertEquals(SerialDate.WEDNESDAY, weekday);
187
188     }
189
190     /**
191      * Teste la conversion d'une chaîne en un mois. Ce test échouera si les paramètres
192      * régionaux n'utilisent pas les noms anglais... trouver un meilleur test !
193      */
194     public void testStringToMonthCode() {
195
196         int m = SerialDate.stringToMonthCode("January");
197         assertEquals(MonthConstants.JANUARY, m);
```

```
198
199         m = SerialDate.stringToMonthCode(" January ");
200         assertEquals(MonthConstants.JANUARY, m);
201
202         m = SerialDate.stringToMonthCode("Jan");
203         assertEquals(MonthConstants.JANUARY, m);
204
205     }
206
207     /**
208      * Teste la conversion d'un code de mois en une chaîne.
209      */
210     public void testMonthCodeToStringCode() {
211
212         final String test = SerialDate.monthCodeToString(MonthConstants.DECEMBER);
213         assertEquals("December", test);
214
215     }
216
217     /**
218      * 1900 n'est pas une année bissextile.
219      */
220     public void testIsNotLeapYear1900() {
221         assertTrue(!SerialDate.isLeapYear(1900));
222     }
223
224     /**
225      * 2000 est une année bissextile.
226      */
227     public void testIsLeapYear2000() {
228         assertTrue(SerialDate.isLeapYear(2000));
229     }
230
231     /**
232      * Le nombre d'années bissextiles entre 1900 et 1899, comprise, est 0.
233      */
234     public void testLeapYearCount1899() {
235         assertEquals(SerialDate.leapYearCount(1899), 0);
236     }
237
238     /**
239      * Le nombre d'années bissextiles entre 1900 et 1903, comprise, est 0.
240      */
241     public void testLeapYearCount1903() {
242         assertEquals(SerialDate.leapYearCount(1903), 0);
243     }
244
245     /**
246      * Le nombre d'années bissextiles entre 1900 et 1904, comprise, est 1.
247      */
248     public void testLeapYearCount1904() {
249         assertEquals(SerialDate.leapYearCount(1904), 1);
250     }
251
252     /**
253      * Le nombre d'années bissextiles entre 1900 et 1999, comprise, est 24.
254      */
```

```
255     public void testLeapYearCount1999() {
256         assertEquals(SerialDate.leapYearCount(1999), 24);
257     }
258
259     /**
260      * Le nombre d'années bissextiles entre 1900 et 2000, comprise, est 25.
261      */
262     public void testLeapYearCount2000() {
263         assertEquals(SerialDate.leapYearCount(2000), 25);
264     }
265
266     /**
267      * Sérialise une instance, la restaure et vérifie l'égalité.
268      */
269     public void testSerialization() {
270
271         SerialDate d1 = SerialDate.createInstance(15, 4, 2000);
272         SerialDate d2 = null;
273
274         try {
275             ByteArrayOutputStream buffer = new ByteArrayOutputStream();
276             ObjectOutput out = new ObjectOutputStream(buffer);
277             out.writeObject(d1);
278             out.close();
279
280             ObjectInput in = new ObjectInputStream(
                    new ByteArrayInputStream(buffer.toByteArray()));
281             d2 = (SerialDate) in.readObject();
282             in.close();
283         }
284         catch (Exception e) {
285             System.out.println(e.toString());
286         }
287         assertEquals(d1, d2);
288
289     }
290
291     /**
292      * Teste le bogue 1096282 (à présent corrigé).
293      */
294     public void test1096282() {
295         SerialDate d = SerialDate.createInstance(29, 2, 2004);
296         d = SerialDate.addYears(1, d);
297         SerialDate expected = SerialDate.createInstance(28, 2, 2005);
298         assertTrue(d.isOn(expected));
299     }
300
301     /**
302      *  Divers tests de la méthode addMonths().
303      */
304     public void testAddMonths() {
305         SerialDate d1 = SerialDate.createInstance(31, 5, 2004);
306
307         SerialDate d2 = SerialDate.addMonths(1, d1);
308         assertEquals(30, d2.getDayOfMonth());
309         assertEquals(6, d2.getMonth());
310         assertEquals(2004, d2.getYYYY());
```

```
311
312         SerialDate d3 = SerialDate.addMonths(2, d1);
313         assertEquals(31, d3.getDayOfMonth());
314         assertEquals(7, d3.getMonth());
315         assertEquals(2004, d3.getYYYY());
316
317         SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
318         assertEquals(30, d4.getDayOfMonth());
319         assertEquals(7, d4.getMonth());
320         assertEquals(2004, d4.getYYYY());
321     }
322 }
```

Listing B.3 : `MonthConstants.java`

```
 1 /* ===========================================================================
 2  * JCommon : bibliothèque libre de classes générales pour Java(tm)
 3  * ===========================================================================
 4  *
 5  * (C) Copyright 2000-2005, par Object Refinery Limited et les Contributeurs.
 6  *
 7  * Site du projet :  http://www.jfree.org/jcommon/index.html
 8  *
 9  * Cette bibliothèque est un logiciel libre ; vous pouvez la redistribuer et/ou
10  * la modifier en respectant les termes de la GNU Lesser General Public License
11  * publiée par la Free Software Foundation, en version 2.1 ou (selon votre choix)
12  * en toute version ultérieure.
13  *
14  * Cette bibliothèque est distribuée en espérant qu'elle sera utile, mais SANS
15  * AUCUNE GARANTIE, sans même la garantie implicite de QUALITÉ MARCHANDE ou
16  * d'ADÉQUATION À UN OBJECTIF PRÉCIS. Pour de plus amples détails, consultez
17  * la GNU Lesser General Public License.
18  *
19  * Vous devez avoir reçu un exemplaire de la GNU Lesser General Public License
20  * avec cette bibliothèque. Dans le cas contraire, merci d'écrire à la Free
21  * Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
22  * MA 02110-1301, USA.
23  *
24  * [Java est une marque ou une marque déposée de Sun Microsystems, Inc.
25  * aux États-Unis et dans d'autres pays.]
26  *
27  * -------------------
28  * MonthConstants.java
29  * -------------------
30  * (C) Copyright 2002, 2003, par Object Refinery Limited.
31  *
32  * Auteur :            David Gilbert (pour Object Refinery Limited);
33  * Contributeur(s) :   -;
34  *
35  * $Id: MonthConstants.java,v 1.4 2005/11/16 15:58:40 taqua Exp $
36  *
37  * Modifications
38  * -------------
39  * 29-Mai-2002 : Version 1 (code extrait de la classe SerialDate) (DG);
```

```
40  *
41  */
42
43 package org.jfree.date;
44
45 /**
46  * Constantes utiles pour les mois. Notez qu'elles ne sont PAS équivalentes aux
47  * constantes définies par java.util.Calendar (où JANUARY=0 et DECEMBER=11).
48  * <P>
49  * Utilisées par les classes SerialDate et RegularTimePeriod.
50  *
51  * @author David Gilbert
52  */
53 public interface MonthConstants {
54
55     /** Constante pour January (janvier). */
56     public static final int JANUARY = 1;
57
58     /** Constante pour February (février). */
59     public static final int FEBRUARY = 2;
60
61     /** Constante pour March (mars). */
62     public static final int MARCH = 3;
63
64     /** Constante pour April (avril). */
65     public static final int APRIL = 4;
66
67     /** Constante pour May (mai). */
68     public static final int MAY = 5;
69
70     /** Constante pour June (juin). */
71     public static final int JUNE = 6;
72
73     /** Constante pour July (juillet). */
74     public static final int JULY = 7;
75
76     /** Constante pour August (août). */
77     public static final int AUGUST = 8;
78
79     /** Constante pour September (septembre). */
80     public static final int SEPTEMBER = 9;
81
82     /** Constante pour October (octobre). */
83     public static final int OCTOBER = 10;
84
85     /** Constante pour November (novembre). */
86     public static final int NOVEMBER = 11;
87
88     /** Constante pour December (décembre). */
89     public static final int DECEMBER = 12;
90
91 }
```

Listing B.4 : `BobsSerialDateTest.java`

```
 1 package org.jfree.date.junit;
 2
 3 import junit.framework.TestCase;
 4 import org.jfree.date.*;
 5 import static org.jfree.date.SerialDate.*;
 6
 7 import java.util.*;
 8
 9 public class BobsSerialDateTest extends TestCase {
10
11   public void testIsValidWeekdayCode() throws Exception {
12     for (int day = 1; day <= 7; day++)
13       assertTrue(isValidWeekdayCode(day));
14     assertFalse(isValidWeekdayCode(0));
15     assertFalse(isValidWeekdayCode(8));
16   }
17
18   public void testStringToWeekdayCode() throws Exception {
19
20     assertEquals(-1, stringToWeekdayCode("Hello"));
21     assertEquals(MONDAY, stringToWeekdayCode("Monday"));
22     assertEquals(MONDAY, stringToWeekdayCode("Mon"));
23 //todo    assertEquals(MONDAY,stringToWeekdayCode("monday"));
24 //    assertEquals(MONDAY,stringToWeekdayCode("MONDAY"));
25 //    assertEquals(MONDAY, stringToWeekdayCode("mon"));
26
27     assertEquals(TUESDAY, stringToWeekdayCode("Tuesday"));
28     assertEquals(TUESDAY, stringToWeekdayCode("Tue"));
29 //    assertEquals(TUESDAY,stringToWeekdayCode("tuesday"));
30 //    assertEquals(TUESDAY,stringToWeekdayCode("TUESDAY"));
31 //    assertEquals(TUESDAY, stringToWeekdayCode("tue"));
32 //    assertEquals(TUESDAY, stringToWeekdayCode("tues"));
33
34     assertEquals(WEDNESDAY, stringToWeekdayCode("Wednesday"));
35     assertEquals(WEDNESDAY, stringToWeekdayCode("Wed"));
36 //    assertEquals(WEDNESDAY,stringToWeekdayCode("wednesday"));
37 //    assertEquals(WEDNESDAY,stringToWeekdayCode("WEDNESDAY"));
38 //    assertEquals(WEDNESDAY, stringToWeekdayCode("wed"));
39
40     assertEquals(THURSDAY, stringToWeekdayCode("Thursday"));
41     assertEquals(THURSDAY, stringToWeekdayCode("Thu"));
42 //    assertEquals(THURSDAY,stringToWeekdayCode("thursday"));
43 //    assertEquals(THURSDAY,stringToWeekdayCode("THURSDAY"));
44 //    assertEquals(THURSDAY, stringToWeekdayCode("thu"));
45 //    assertEquals(THURSDAY, stringToWeekdayCode("thurs"));
46
47     assertEquals(FRIDAY, stringToWeekdayCode("Friday"));
48     assertEquals(FRIDAY, stringToWeekdayCode("Fri"));
49 //    assertEquals(FRIDAY,stringToWeekdayCode("friday"));
50 //    assertEquals(FRIDAY,stringToWeekdayCode("FRIDAY"));
51 //    assertEquals(FRIDAY, stringToWeekdayCode("fri"));
52
53     assertEquals(SATURDAY, stringToWeekdayCode("Saturday"));
54     assertEquals(SATURDAY, stringToWeekdayCode("Sat"));
```

```
55 //    assertEquals(SATURDAY,stringToWeekdayCode("saturday"));
56 //    assertEquals(SATURDAY,stringToWeekdayCode("SATURDAY"));
57 //    assertEquals(SATURDAY, stringToWeekdayCode("sat"));
58
59    assertEquals(SUNDAY, stringToWeekdayCode("Sunday"));
60    assertEquals(SUNDAY, stringToWeekdayCode("Sun"));
61 //    assertEquals(SUNDAY,stringToWeekdayCode("sunday"));
62 //    assertEquals(SUNDAY,stringToWeekdayCode("SUNDAY"));
63 //    assertEquals(SUNDAY, stringToWeekdayCode("sun"));
64  }
65
66  public void testWeekdayCodeToString() throws Exception {
67    assertEquals("Sunday", weekdayCodeToString(SUNDAY));
68    assertEquals("Monday", weekdayCodeToString(MONDAY));
69    assertEquals("Tuesday", weekdayCodeToString(TUESDAY));
70    assertEquals("Wednesday", weekdayCodeToString(WEDNESDAY));
71    assertEquals("Thursday", weekdayCodeToString(THURSDAY));
72    assertEquals("Friday", weekdayCodeToString(FRIDAY));
73    assertEquals("Saturday", weekdayCodeToString(SATURDAY));
74  }
75
76  public void testIsValidMonthCode() throws Exception {
77    for (int i = 1; i <= 12; i++)
78      assertTrue(isValidMonthCode(i));
79    assertFalse(isValidMonthCode(0));
80    assertFalse(isValidMonthCode(13));
81  }
82
83  public void testMonthToQuarter() throws Exception {
84    assertEquals(1, monthCodeToQuarter(JANUARY));
85    assertEquals(1, monthCodeToQuarter(FEBRUARY));
86    assertEquals(1, monthCodeToQuarter(MARCH));
87    assertEquals(2, monthCodeToQuarter(APRIL));
88    assertEquals(2, monthCodeToQuarter(MAY));
89    assertEquals(2, monthCodeToQuarter(JUNE));
90    assertEquals(3, monthCodeToQuarter(JULY));
91    assertEquals(3, monthCodeToQuarter(AUGUST));
92    assertEquals(3, monthCodeToQuarter(SEPTEMBER));
93    assertEquals(4, monthCodeToQuarter(OCTOBER));
94    assertEquals(4, monthCodeToQuarter(NOVEMBER));
95    assertEquals(4, monthCodeToQuarter(DECEMBER));
96
97    try {
98      monthCodeToQuarter(-1);
99      fail("Invalid Month Code should throw exception");
100   } catch (IllegalArgumentException e) {
101   }
102 }
103
104 public void testMonthCodeToString() throws Exception {
105   assertEquals("January", monthCodeToString(JANUARY));
106   assertEquals("February", monthCodeToString(FEBRUARY));
107   assertEquals("March", monthCodeToString(MARCH));
108   assertEquals("April", monthCodeToString(APRIL));
109   assertEquals("May", monthCodeToString(MAY));
110   assertEquals("June", monthCodeToString(JUNE));
111   assertEquals("July", monthCodeToString(JULY));
```

```
112     assertEquals("August", monthCodeToString(AUGUST));
113     assertEquals("September", monthCodeToString(SEPTEMBER));
114     assertEquals("October", monthCodeToString(OCTOBER));
115     assertEquals("November", monthCodeToString(NOVEMBER));
116     assertEquals("December", monthCodeToString(DECEMBER));
117
118     assertEquals("Jan", monthCodeToString(JANUARY, true));
119     assertEquals("Feb", monthCodeToString(FEBRUARY, true));
120     assertEquals("Mar", monthCodeToString(MARCH, true));
121     assertEquals("Apr", monthCodeToString(APRIL, true));
122     assertEquals("May", monthCodeToString(MAY, true));
123     assertEquals("Jun", monthCodeToString(JUNE, true));
124     assertEquals("Jul", monthCodeToString(JULY, true));
125     assertEquals("Aug", monthCodeToString(AUGUST, true));
126     assertEquals("Sep", monthCodeToString(SEPTEMBER, true));
127     assertEquals("Oct", monthCodeToString(OCTOBER, true));
128     assertEquals("Nov", monthCodeToString(NOVEMBER, true));
129     assertEquals("Dec", monthCodeToString(DECEMBER, true));
130
131     try {
132       monthCodeToString(-1);
133       fail("Invalid month code should throw exception");
134     } catch (IllegalArgumentException e) {
135     }
136
137   }
138
139   public void testStringToMonthCode() throws Exception {
140     assertEquals(JANUARY,stringToMonthCode("1"));
141     assertEquals(FEBRUARY,stringToMonthCode("2"));
142     assertEquals(MARCH,stringToMonthCode("3"));
143     assertEquals(APRIL,stringToMonthCode("4"));
144     assertEquals(MAY,stringToMonthCode("5"));
145     assertEquals(JUNE,stringToMonthCode("6"));
146     assertEquals(JULY,stringToMonthCode("7"));
147     assertEquals(AUGUST,stringToMonthCode("8"));
148     assertEquals(SEPTEMBER,stringToMonthCode("9"));
149     assertEquals(OCTOBER,stringToMonthCode("10"));
150     assertEquals(NOVEMBER, stringToMonthCode("11"));
151     assertEquals(DECEMBER,stringToMonthCode("12"));
152
153 //todo    assertEquals(-1, stringToMonthCode("0"));
154 //    assertEquals(-1, stringToMonthCode("13"));
155
156     assertEquals(-1,stringToMonthCode("Hello"));
157
158     for (int m = 1; m <= 12; m++) {
159       assertEquals(m, stringToMonthCode(monthCodeToString(m, false)));
160       assertEquals(m, stringToMonthCode(monthCodeToString(m, true)));
161     }
162
163 //    assertEquals(1,stringToMonthCode("jan"));
164 //    assertEquals(2,stringToMonthCode("feb"));
165 //    assertEquals(3,stringToMonthCode("mar"));
166 //    assertEquals(4,stringToMonthCode("apr"));
167 //    assertEquals(5,stringToMonthCode("may"));
168 //    assertEquals(6,stringToMonthCode("jun"));
```

```
169 //    assertEquals(7,stringToMonthCode("jul"));
170 //    assertEquals(8,stringToMonthCode("aug"));
171 //    assertEquals(9,stringToMonthCode("sep"));
172 //    assertEquals(10,stringToMonthCode("oct"));
173 //    assertEquals(11,stringToMonthCode("nov"));
174 //    assertEquals(12,stringToMonthCode("dec"));
175 
176 //    assertEquals(1,stringToMonthCode("JAN"));
177 //    assertEquals(2,stringToMonthCode("FEB"));
178 //    assertEquals(3,stringToMonthCode("MAR"));
179 //    assertEquals(4,stringToMonthCode("APR"));
180 //    assertEquals(5,stringToMonthCode("MAY"));
181 //    assertEquals(6,stringToMonthCode("JUN"));
182 //    assertEquals(7,stringToMonthCode("JUL"));
183 //    assertEquals(8,stringToMonthCode("AUG"));
184 //    assertEquals(9,stringToMonthCode("SEP"));
185 //    assertEquals(10,stringToMonthCode("OCT"));
186 //    assertEquals(11,stringToMonthCode("NOV"));
187 //    assertEquals(12,stringToMonthCode("DEC"));
188 
189 //    assertEquals(1,stringToMonthCode("january"));
190 //    assertEquals(2,stringToMonthCode("february"));
191 //    assertEquals(3,stringToMonthCode("march"));
192 //    assertEquals(4,stringToMonthCode("april"));
193 //    assertEquals(5,stringToMonthCode("may"));
194 //    assertEquals(6,stringToMonthCode("june"));
195 //    assertEquals(7,stringToMonthCode("july"));
196 //    assertEquals(8,stringToMonthCode("august"));
197 //    assertEquals(9,stringToMonthCode("september"));
198 //    assertEquals(10,stringToMonthCode("october"));
199 //    assertEquals(11,stringToMonthCode("november"));
200 //    assertEquals(12,stringToMonthCode("december"));
201 
202 //    assertEquals(1,stringToMonthCode("JANUARY"));
203 //    assertEquals(2,stringToMonthCode("FEBRUARY"));
204 //    assertEquals(3,stringToMonthCode("MAR"));
205 //    assertEquals(4,stringToMonthCode("APRIL"));
206 //    assertEquals(5,stringToMonthCode("MAY"));
207 //    assertEquals(6,stringToMonthCode("JUNE"));
208 //    assertEquals(7,stringToMonthCode("JULY"));
209 //    assertEquals(8,stringToMonthCode("AUGUST"));
210 //    assertEquals(9,stringToMonthCode("SEPTEMBER"));
211 //    assertEquals(10,stringToMonthCode("OCTOBER"));
212 //    assertEquals(11,stringToMonthCode("NOVEMBER"));
213 //    assertEquals(12,stringToMonthCode("DECEMBER"));
214   }
215 
216   public void testIsValidWeekInMonthCode() throws Exception {
217     for (int w = 0; w <= 4; w++) {
218       assertTrue(isValidWeekInMonthCode(w));
219     }
220     assertFalse(isValidWeekInMonthCode(5));
221   }
222 
223   public void testIsLeapYear() throws Exception {
224     assertFalse(isLeapYear(1900));
225     assertFalse(isLeapYear(1901));
```

```
226     assertFalse(isLeapYear(1902));
227     assertFalse(isLeapYear(1903));
228     assertTrue(isLeapYear(1904));
229     assertTrue(isLeapYear(1908));
230     assertFalse(isLeapYear(1955));
231     assertTrue(isLeapYear(1964));
232     assertTrue(isLeapYear(1980));
233     assertTrue(isLeapYear(2000));
234     assertFalse(isLeapYear(2001));
235     assertFalse(isLeapYear(2100));
236   }
237
238   public void testLeapYearCount() throws Exception {
239     assertEquals(0, leapYearCount(1900));
240     assertEquals(0, leapYearCount(1901));
241     assertEquals(0, leapYearCount(1902));
242     assertEquals(0, leapYearCount(1903));
243     assertEquals(1, leapYearCount(1904));
244     assertEquals(1, leapYearCount(1905));
245     assertEquals(1, leapYearCount(1906));
246     assertEquals(1, leapYearCount(1907));
247     assertEquals(2, leapYearCount(1908));
248     assertEquals(24, leapYearCount(1999));
249     assertEquals(25, leapYearCount(2001));
250     assertEquals(49, leapYearCount(2101));
251     assertEquals(73, leapYearCount(2201));
252     assertEquals(97, leapYearCount(2301));
253     assertEquals(122, leapYearCount(2401));
254   }
255
256   public void testLastDayOfMonth() throws Exception {
257     assertEquals(31, lastDayOfMonth(JANUARY, 1901));
258     assertEquals(28, lastDayOfMonth(FEBRUARY, 1901));
259     assertEquals(31, lastDayOfMonth(MARCH, 1901));
260     assertEquals(30, lastDayOfMonth(APRIL, 1901));
261     assertEquals(31, lastDayOfMonth(MAY, 1901));
262     assertEquals(30, lastDayOfMonth(JUNE, 1901));
263     assertEquals(31, lastDayOfMonth(JULY, 1901));
264     assertEquals(31, lastDayOfMonth(AUGUST, 1901));
265     assertEquals(30, lastDayOfMonth(SEPTEMBER, 1901));
266     assertEquals(31, lastDayOfMonth(OCTOBER, 1901));
267     assertEquals(30, lastDayOfMonth(NOVEMBER, 1901));
268     assertEquals(31, lastDayOfMonth(DECEMBER, 1901));
269     assertEquals(29, lastDayOfMonth(FEBRUARY, 1904));
270   }
271
272   public void testAddDays() throws Exception {
273     SerialDate newYears = d(1, JANUARY, 1900);
274     assertEquals(d(2, JANUARY, 1900), addDays(1, newYears));
275     assertEquals(d(1, FEBRUARY, 1900), addDays(31, newYears));
276     assertEquals(d(1, JANUARY, 1901), addDays(365, newYears));
277     assertEquals(d(31, DECEMBER, 1904), addDays(5 * 365, newYears));
278   }
279
280   private static SpreadsheetDate d(int day, int month, int year) {return new
SpreadsheetDate(day, month, year);}
281
```

```
282   public void testAddMonths() throws Exception {
283     assertEquals(d(1, FEBRUARY, 1900), addMonths(1, d(1, JANUARY, 1900)));
284     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(31, JANUARY, 1900)));
285     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(30, JANUARY, 1900)));
286     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(29, JANUARY, 1900)));
287     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(28, JANUARY, 1900)));
288     assertEquals(d(27, FEBRUARY, 1900), addMonths(1, d(27, JANUARY, 1900)));
289
290     assertEquals(d(30, JUNE, 1900), addMonths(5, d(31, JANUARY, 1900)));
291     assertEquals(d(30, JUNE, 1901), addMonths(17, d(31, JANUARY, 1900)));
292
293     assertEquals(d(29, FEBRUARY, 1904), addMonths(49, d(31, JANUARY, 1900)));
294
295   }
296
297   public void testAddYears() throws Exception {
298     assertEquals(d(1, JANUARY, 1901), addYears(1, d(1, JANUARY, 1900)));
299     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(29, FEBRUARY, 1904)));
300     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(28, FEBRUARY, 1904)));
301     assertEquals(d(28, FEBRUARY, 1904), addYears(1, d(28, FEBRUARY, 1903)));
302   }
303
304   public void testGetPreviousDayOfWeek() throws Exception {
305     assertEquals(d(24, FEBRUARY, 2006), getPreviousDayOfWeek(FRIDAY, d(1, MARCH, 2006)));
306     assertEquals(d(22, FEBRUARY, 2006), getPreviousDayOfWeek(WEDNESDAY, d(1, MARCH, 2006)));
307     assertEquals(d(29, FEBRUARY, 2004), getPreviousDayOfWeek(SUNDAY, d(3, MARCH, 2004)));
308     assertEquals(d(29, DECEMBER, 2004), getPreviousDayOfWeek(WEDNESDAY, d(5, JANUARY, 2005)));
309
310     try {
311       getPreviousDayOfWeek(-1, d(1, JANUARY, 2006));
312       fail("Invalid day of week code should throw exception");
313     } catch (IllegalArgumentException e) {
314     }
315   }
316
317   public void testGetFollowingDayOfWeek() throws Exception {
318 //    assertEquals(d(1, JANUARY, 2005),getFollowingDayOfWeek(SATURDAY, d(25, DECEMBER, 2004)));
319     assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(26, DECEMBER, 2004)));
320     assertEquals(d(3, MARCH, 2004), getFollowingDayOfWeek(WEDNESDAY, d(28, FEBRUARY, 2004)));
321
322     try {
323       getFollowingDayOfWeek(-1, d(1, JANUARY, 2006));
324       fail("Invalid day of week code should throw exception");
325     } catch (IllegalArgumentException e) {
326     }
327   }
328
329   public void testGetNearestDayOfWeek() throws Exception {
330     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(16, APRIL, 2006)));
331     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(17, APRIL, 2006)));
332     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(18, APRIL, 2006)));
333     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(19, APRIL, 2006)));
334     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(20, APRIL, 2006)));
335     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(21, APRIL, 2006)));
336     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(22, APRIL, 2006)));
337
338 //todo    assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(16, APRIL, 2006)));
```

```
339     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(17, APRIL, 2006)));
340     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(18, APRIL, 2006)));
341     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(19, APRIL, 2006)));
342     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(20, APRIL, 2006)));
343     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(21, APRIL, 2006)));
344     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(22, APRIL, 2006)));
345
346 //    assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(16, APRIL, 2006)));
347 //    assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(17, APRIL, 2006)));
348     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(18, APRIL, 2006)));
349     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(19, APRIL, 2006)));
350     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(20, APRIL, 2006)));
351     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(21, APRIL, 2006)));
352     assertEquals(d(25, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(22, APRIL, 2006)));
353
354 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(16, APRIL, 2006)));
355 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(17, APRIL, 2006)));
356 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(18, APRIL, 2006)));
357     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(19, APRIL, 2006)));
358     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(20, APRIL, 2006)));
359     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(21, APRIL, 2006)));
360     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(22, APRIL, 2006)));
361
362 //    assertEquals(d(13, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(16, APRIL, 2006)));
363 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(17, APRIL, 2006)));
364 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(18, APRIL, 2006)));
365 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(19, APRIL, 2006)));
366     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(20, APRIL, 2006)));
367     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(21, APRIL, 2006)));
368     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(22, APRIL, 2006)));
369
370 //    assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(16, APRIL, 2006)));
371 //    assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(17, APRIL, 2006)));
372 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(18, APRIL, 2006)));
373 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(19, APRIL, 2006)));
374 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(20, APRIL, 2006)));
375     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(21, APRIL, 2006)));
376     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(22, APRIL, 2006)));
377
378 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(16, APRIL, 2006)));
379 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(17, APRIL, 2006)));
380 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(18, APRIL, 2006)));
381 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(19, APRIL, 2006)));
382 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(20, APRIL, 2006)));
383 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(21, APRIL, 2006)));
384     assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(22, APRIL, 2006)));
385
386     try {
387       getNearestDayOfWeek(-1, d(1, JANUARY, 2006));
388       fail("Invalid day of week code should throw exception");
389     } catch (IllegalArgumentException e) {
390     }
391   }
392
393   public void testEndOfCurrentMonth() throws Exception {
394     SerialDate d = SerialDate.createInstance(2);
395     assertEquals(d(31, JANUARY, 2006), d.getEndOfCurrentMonth(d(1, JANUARY, 2006)));
```

```
396     assertEquals(d(28, FEBRUARY, 2006), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2006)));
397     assertEquals(d(31, MARCH, 2006), d.getEndOfCurrentMonth(d(1, MARCH, 2006)));
398     assertEquals(d(30, APRIL, 2006), d.getEndOfCurrentMonth(d(1, APRIL, 2006)));
399     assertEquals(d(31, MAY, 2006), d.getEndOfCurrentMonth(d(1, MAY, 2006)));
400     assertEquals(d(30, JUNE, 2006), d.getEndOfCurrentMonth(d(1, JUNE, 2006)));
401     assertEquals(d(31, JULY, 2006), d.getEndOfCurrentMonth(d(1, JULY, 2006)));
402     assertEquals(d(31, AUGUST, 2006), d.getEndOfCurrentMonth(d(1, AUGUST, 2006)));
403     assertEquals(d(30, SEPTEMBER, 2006), d.getEndOfCurrentMonth(d(1, SEPTEMBER, 2006)));
404     assertEquals(d(31, OCTOBER, 2006), d.getEndOfCurrentMonth(d(1, OCTOBER, 2006)));
405     assertEquals(d(30, NOVEMBER, 2006), d.getEndOfCurrentMonth(d(1, NOVEMBER, 2006)));
406     assertEquals(d(31, DECEMBER, 2006), d.getEndOfCurrentMonth(d(1, DECEMBER, 2006)));
407     assertEquals(d(29, FEBRUARY, 2008), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2008)));
408   }
409
410   public void testWeekInMonthToString() throws Exception {
411     assertEquals("First",weekInMonthToString(FIRST_WEEK_IN_MONTH));
412     assertEquals("Second",weekInMonthToString(SECOND_WEEK_IN_MONTH));
413     assertEquals("Third",weekInMonthToString(THIRD_WEEK_IN_MONTH));
414     assertEquals("Fourth",weekInMonthToString(FOURTH_WEEK_IN_MONTH));
415     assertEquals("Last",weekInMonthToString(LAST_WEEK_IN_MONTH));
416
417 //todo    try {
418 //      weekInMonthToString(-1);
419 //      fail("Invalid week code should throw exception");
420 //    } catch (IllegalArgumentException e) {
421 //    }
422   }
423
424   public void testRelativeToString() throws Exception {
425     assertEquals("Preceding",relativeToString(PRECEDING));
426     assertEquals("Nearest",relativeToString(NEAREST));
427     assertEquals("Following",relativeToString(FOLLOWING));
428
429 //todo    try {
430 //      relativeToString(-1000);
431 //      fail("Invalid relative code should throw exception");
432 //    } catch (IllegalArgumentException e) {
433 //    }
434   }
435
436   public void testCreateInstanceFromDDMMYYY() throws Exception {
437     SerialDate date = createInstance(1, JANUARY, 1900);
438     assertEquals(1,date.getDayOfMonth());
439     assertEquals(JANUARY,date.getMonth());
440     assertEquals(1900,date.getYYYY());
441     assertEquals(2,date.toSerial());
442   }
443
444   public void testCreateInstanceFromSerial() throws Exception {
445     assertEquals(d(1, JANUARY, 1900),createInstance(2));
446     assertEquals(d(1, JANUARY, 1901), createInstance(367));
447   }
448
449   public void testCreateInstanceFromJavaDate() throws Exception {
450     assertEquals(d(1, JANUARY, 1900),
                   createInstance(new GregorianCalendar(1900,0,1).getTime()));
```

```
451     assertEquals(d(1, JANUARY, 2006),
                     createInstance(new GregorianCalendar(2006,0,1).getTime()));
452   }
453
454   public static void main(String[] args) {
455     junit.textui.TestRunner.run(BobsSerialDateTest.class);
456   }
457 }
```

Listing B.5 : `SpreadsheetDate.java`

```
 1 /* ===========================================================================
 2  * JCommon : bibliothèque libre de classes générales pour Java(tm)
 3  * ===========================================================================
 4  *
 5  * (C) Copyright 2000-2005, par Object Refinery Limited et les Contributeurs.
 6  *
 7  * Site du projet :  http://www.jfree.org/jcommon/index.html
 8  *
 9  * Cette bibliothèque est un logiciel libre ; vous pouvez la redistribuer et/ou
10  * la modifier en respectant les termes de la GNU Lesser General Public License
11  * publiée par la Free Software Foundation, en version 2.1 ou (selon votre choix)
12  * en toute version ultérieure.
13  *
14  * Cette bibliothèque est distribuée en espérant qu'elle sera utile, mais SANS
15  * AUCUNE GARANTIE, sans même la garantie implicite de QUALITÉ MARCHANDE ou
16  * d'ADÉQUATION À UN OBJECTIF PRÉCIS. Pour de plus amples détails, consultez
17  * la GNU Lesser General Public License.
18  *
19  * Vous devez avoir reçu un exemplaire de la GNU Lesser General Public License
20  * avec cette bibliothèque. Dans le cas contraire, merci d'écrire à la Free
21  * Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
22  * MA 02110-1301, USA.
23  *
24  * [Java est une marque ou une marque déposée de Sun Microsystems, Inc.
25  * aux États-Unis et dans d'autres pays.]
26  *
27  * --------------------
28  * SpreadsheetDate.java
29  * --------------------
30  * (C) Copyright 2001-2005, par Object Refinery Limited.
31  *
32  * Auteur :            David Gilbert (pour Object Refinery Limited);
33  * Contributeur(s) :   -;
34  *
35  * $Id: SpreadsheetDate.java,v 1.8 2005/11/03 09:25:39 mungady Exp $
36  *
37  * Modifications
38  * -------------
39  * 11-Oct-2001 : Version 1 (DG);
40  * 05-Nov-2001 : Ajout des méthodes getDescription() et setDescription() (DG);
41  * 12-Nov-2001 : Nom changé de ExcelDate.java en SpreadsheetDate.java (DG);
42  *               Correction d'un bogue de calcul du jour, mois et année à partir
43  *               du numéro de série (DG);
```

```
 44  * 24-Jan-2002 : Correction d'un bogue de calcul du numéro de série à partir du jour,
 45  *                mois et année. Merci à Trevor Hills pour l'avoir signalé (DG);
 46  * 29-Mai-2002 : Ajout de la méthode equals(Object) (SourceForge ID 558850) (DG);
 47  * 03-Oct-2002 : Correction des erreurs signalées par Checkstyle (DG);
 48  * 13-Mar-2003 : Implémentation de Serializable (DG);
 49  * 04-Sep-2003 : Méthodes isInRange() terminées (DG);
 50  * 05-Sep-2003 : Implémentation de Comparable (DG);
 51  * 21-Oct-2003 : Ajout de la méthode hashCode() (DG);
 52  *
 53  */
 54
 55 package org.jfree.date;
 56
 57 import java.util.Calendar;
 58 import java.util.Date;
 59
 60 /**
 61  * Représente une date à l'aide d'un entier, de manière comparable à l'implémentation
 62  * dans Microsoft Excel. La plage des dates reconnues va du 1er janvier 1900 au
 63  * 31 décembre 9999.
 64  * <P>
 65  * Sachez qu'il existe un bogue délibéré dans Excel qui reconnaît l'année 1900
 66  * comme une année bissextile alors que ce n'est pas le cas. Pour plus d'informations,
 67  * consultez l'article Q181370 sur le site web de Microsoft Microsoft :
 68  * <P>
 69  * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
 70  * <P>
 71  * Excel emploie la convention 1er janvier 1900 = 1. Cette classe utilise la
 72  * convention 1er janvier 1900 = 2.
 73  * Par conséquent, le numéro de jour dans cette date sera différent de celui
 74  * donné par Excel pour janvier et février 1900... mais Excel ajoute ensuite un jour
 75  * supplémentaire (29 février 1900, qui n'existe pas réellement !) et, à partir de là,
 76  * les numéros de jours concordent.
 77  *
 78  * @author David Gilbert
 79  */
 80 public class SpreadsheetDate extends SerialDate {
 81
 82     /** Pour la sérialisation. */
 83     private static final long serialVersionUID = -2039586705374454461L;
 84
 85     /**
 86      * Le numéro du jour (1er janvier 1900 = 2, 2 janvier 1900 = 3, ...,
 87      * 31 décembre 9999 = 2958465).
 88      */
 89     private int serial;
 90
 91     /** Le jour du mois (1 à 28, 29, 30 ou 31 selon le mois). */
 92     private int day;
 93
 94     /** Le mois de l'année (1 à 12). */
 95     private int month;
 96
 97     /** L'année (1900 à 9999). */
 98     private int year;
 99
100     /** Description facultative de la date. */
```

```
101     private String description;
102
103     /**
104      * Crée une nouvelle instance de date.
105      *
106      * @param day  le jour (dans la plage 1 à 28/29/30/31).
107      * @param month  le mois (dans la plage 1 à 12).
108      * @param year  l'année (dans la plage 1900 à 9999).
109      */
110     public SpreadsheetDate(final int day, final int month, final int year) {
111
112         if ((year >= 1900) && (year <= 9999)) {
113             this.year = year;
114         }
115         else {
116             throw new IllegalArgumentException(
117                 "The 'year' argument must be in range 1900 to 9999."
118             );
119         }
120
121         if ((month >= MonthConstants.JANUARY)
122                 && (month <= MonthConstants.DECEMBER)) {
123             this.month = month;
124         }
125         else {
126             throw new IllegalArgumentException(
127                 "The 'month' argument must be in the range 1 to 12."
128             );
129         }
130
131         if ((day >= 1) && (day <= SerialDate.lastDayOfMonth(month, year))) {
132             this.day = day;
133         }
134         else {
135             throw new IllegalArgumentException("Invalid 'day' argument.");
136         }
137
138         // Le numéro de série doit être synchronisé avec le jour-mois-année...
139         this.serial = calcSerial(day, month, year);
140
141         this.description = null;
142
143     }
144
145     /**
146      * Constructeur standard - crée un nouvel objet date qui représente
147      * le jour indiqué (dans la plage 2 à 2958465.
148      *
149      * @param serial  le numéro de série du jour (plage : 2 à 2958465).
150      */
151     public SpreadsheetDate(final int serial) {
152
153         if ((serial >= SERIAL_LOWER_BOUND) && (serial <= SERIAL_UPPER_BOUND)) {
154             this.serial = serial;
155         }
156         else {
157             throw new IllegalArgumentException(
```

```
158                 "SpreadsheetDate: Serial must be in range 2 to 2958465.");
159         }
160
161         // Le jour-mois-année doit être synchronisé avec le numéro de série...
162         calcDayMonthYear();
163
164     }
165
166     /**
167      * Retourne la description associée à la date. La date n'est pas obligée
168      * de posséder une description, mais, pour certaines applications, elle
169      * est utile.
170      *
171      * @return la description associée à la date.
172      */
173     public String getDescription() {
174         return this.description;
175     }
176
177     /**
178      * Fixe la description de la date.
179      *
180      * @param description  la description de la date (<code>null</code>
181      *                     est autorisé).
182      */
183     public void setDescription(final String description) {
184         this.description = description;
185     }
186
187     /**
188      * Retourne le numéro série de la date, où le 1 janvier 1900 = 2 (cela
189      * correspond, presque, au système de numérotation employé dans Microsoft
190      * Excel pour Windows et Lotus 1-2-3).
191      *
192      * @return le numéro série de la date.
193      */
194     public int toSerial() {
195         return this.serial;
196     }
197
198     /**
199      * Retourne un <code>java.util.Date</code> équivalent à cette date.
200      *
201      * @return la date.
202      */
203     public Date toDate() {
204         final Calendar calendar = Calendar.getInstance();
205         calendar.set(getYYYY(), getMonth() - 1, getDayOfMonth(), 0, 0, 0);
206         return calendar.getTime();
207     }
208
209     /**
210      * Retourne l'année (suppose une plage valide de 1900 à 9999).
211      *
212      * @return l'année.
213      */
214     public int getYYYY() {
```

```
215         return this.year;
216     }
217
218     /**
219      * Retourne le mois (janvier = 1, février = 2, mars = 3).
220      *
221      * @return le mois de l'année.
222      */
223     public int getMonth() {
224         return this.month;
225     }
226
227     /**
228      * Retourne le jour du mois.
229      *
230      * @return le jour du mois.
231      */
232     public int getDayOfMonth() {
233         return this.day;
234     }
235
236     /**
237      * Retourne un code représentant le jour de la semaine.
238      * <P>
239      * Les codes sont définis dans la classe {@link SerialDate} sous la forme :
240      * <code>SUNDAY</code>, <code>MONDAY</code>, <code>TUESDAY</code>,
241      * <code>WEDNESDAY</code>, <code>THURSDAY</code>, <code>FRIDAY</code> et
242      * <code>SATURDAY</code>.
243      *
244      * @return un code représentant le jour de la semaine.
245      */
246     public int getDayOfWeek() {
247         return (this.serial + 6) % 7 + 1;
248     }
249
250     /**
251      * Teste l'égalité de cette date avec un objet quelconque.
252      * <P>
253      * Cette méthode retourne true UNIQUEMENT si l'objet est une instance de la classe
254      * de base {@link SerialDate} et s'il représente le même jour que ce
255      * {@link SpreadsheetDate}.
256      *
257      * @param object  l'objet à comparer (<code>null</code> est autorisé).
258      *
259      * @return un booléen.
260      */
261     public boolean equals(final Object object) {
262
263         if (object instanceof SerialDate) {
264             final SerialDate s = (SerialDate) object;
265             return (s.toSerial() == this.toSerial());
266         }
267         else {
268             return false;
269         }
270
271     }
```

```
272
273     /**
274      * Retourne un code de hachage pour cette instance.
275      *
276      * @return un code de hachage.
277      */
278     public int hashCode() {
279         return toSerial();
280     }
281
282     /**
283      * Retourne la différence (en jours) entre cette date et l'autre date
284      * indiquée.
285      *
286      * @param other  la date servant à la comparaison.
287      *
288      * @return la différence (en jours) entre cette date et l'autre date
289      *         indiquée.
290      */
291     public int compare(final SerialDate other) {
292         return this.serial - other.toSerial();
293     }
294
295     /**
296      * Implémente la méthode requise par l'interface Comparable.
297      *
298      * @param other  l'autre objet (en général un autre SerialDate).
299      *
300      * @return un entier négatif, zéro ou un entier positif selon que cet objet
301      *         est inférieur à, égal à ou supérieur à l'objet indiqué.
302      */
303     public int compareTo(final Object other) {
304         return compare((SerialDate) other);
305     }
306
307     /**
308      * Retourne true si ce SerialDate représente la même date que
309      * le SerialDate indiqué.
310      *
311      * @param other  la date servant à la comparaison.
312      *
313      * @return <code>true</code> si ce SerialDate représente la même date que
314      *         le SerialDate indiqué.
315      */
316     public boolean isOn(final SerialDate other) {
317         return (this.serial == other.toSerial());
318     }
319
320     /**
321      * Retourne true si ce SerialDate représente une date antérieure
322      * au SerialDate indiqué.
323      *
324      * @param other  la date servant à la comparaison.
325      *
326      * @return <code>true</code> si ce SerialDate représente une date antérieure
327      *         au SerialDate indiqué.
328      */
```

```
329     public boolean isBefore(final SerialDate other) {
330         return (this.serial < other.toSerial());
331     }
332
333     /**
334      * Retourne true si ce SerialDate représente la même date que
335      * le SerialDate indiqué.
336      *
337      * @param other  la date servant à la comparaison.
338      *
339      * @return <code>true</code> si ce SerialDate représente la même date que
340      *         le SerialDate indiqué.
341      */
342     public boolean isOnOrBefore(final SerialDate other) {
343         return (this.serial <= other.toSerial());
344     }
345
346     /**
347      * Retourne true si ce SerialDate représente la même date que
348      * le SerialDate indiqué.
349      *
350      * @param other  la date servant à la comparaison.
351      *
352      * @return <code>true</code> si ce SerialDate représente la même date que
353      *         le SerialDate indiqué.
354      */
355     public boolean isAfter(final SerialDate other) {
356         return (this.serial > other.toSerial());
357     }
358
359     /**
360      * Retourne true si ce SerialDate représente la même date que
361      * le SerialDate indiqué.
362      *
363      * @param other  la date servant à la comparaison.
364      *
365      * @return <code>true</code> si ce SerialDate représente la même date que
366      *         le SerialDate indiqué.
367      */
368     public boolean isOnOrAfter(final SerialDate other) {
369         return (this.serial >= other.toSerial());
370     }
371
372     /**
373      * Retourne <code>true</code> si ce {@link SerialDate} se trouve dans
374      * la plage indiquée (INCLUSIF). L'ordre des dates d1 et d2 n'est pas
375      * important.
376      *
377      * @param d1  une date limite de la plage.
378      * @param d2  l'autre date limite de la plage.
379      *
380      * @return un booléen.
381      */
382     public boolean isInRange(final SerialDate d1, final SerialDate d2) {
383         return isInRange(d1, d2, SerialDate.INCLUDE_BOTH);
384     }
385
```

```
386     /**
387      * Retourne true si ce {@link SerialDate} se trouve dans la plage
388      * indiquée (l'appelant précise si les extrémités sont incluses)
389      * L'ordre des dates d1 et d2 n'est pas important.
390      *
391      * @param d1  une date limite de la plage.
392      * @param d2  l'autre date limite de la plage.
393      * @param include  code qui indique si les dates de début et de fin
394      *                 sont incluses dans la plage.
395      *
396      * @return <code>true</code> si ce SerialDate se trouve dans la plage
397      *         indiquée.
398      */
399     public boolean isInRange(final SerialDate d1, final SerialDate d2,
400                              final int include) {
401         final int s1 = d1.toSerial();
402         final int s2 = d2.toSerial();
403         final int start = Math.min(s1, s2);
404         final int end = Math.max(s1, s2);
405
406         final int s = toSerial();
407         if (include == SerialDate.INCLUDE_BOTH) {
408             return (s >= start && s <= end);
409         }
410         else if (include == SerialDate.INCLUDE_FIRST) {
411             return (s >= start && s < end);
412         }
413         else if (include == SerialDate.INCLUDE_SECOND) {
414             return (s > start && s <= end);
415         }
416         else {
417             return (s > start && s < end);
418         }
419     }
420
421     /**
422      * Calcule le numéro de série pour le jour, le mois et l'année.
423      * <P>
424      * 1er janvier 1900 = 2.
425      *
426      * @param d  le jour.
427      * @param m  le mois.
428      * @param y  l'année.
429      *
430      * @return le numéro de série pour le jour, le mois et l'année.
431      */
432     private int calcSerial(final int d, final int m, final int y) {
433         final int yy = ((y - 1900) * 365) + SerialDate.leapYearCount(y - 1);
434         int mm = SerialDate.AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[m];
435         if (m > MonthConstants.FEBRUARY) {
436             if (SerialDate.isLeapYear(y)) {
437                 mm = mm + 1;
438             }
439         }
440         final int dd = d;
441         return yy + mm + dd + 1;
442     }
```

```
443
444     /**
445      * Calcule le jour, le mois et l'année pour le numéro de série.
446      */
447     private void calcDayMonthYear() {
448
449         // Obtenir l'année à partir de la date.
450         final int days = this.serial - SERIAL_LOWER_BOUND;
451         // Surestimée car nous ignorons les années bissextiles.
452         final int overestimatedYYYY = 1900 + (days / 365);
453         final int leaps = SerialDate.leapYearCount(overestimatedYYYY);
454         final int nonleapdays = days - leaps;
455         // Sous-estimée car nous surestimons les années.
456         int underestimatedYYYY = 1900 + (nonleapdays / 365);
457
458         if (underestimatedYYYY == overestimatedYYYY) {
459             this.year = underestimatedYYYY;
460         }
461         else {
462             int ss1 = calcSerial(1, 1, underestimatedYYYY);
463             while (ss1 <= this.serial) {
464                 underestimatedYYYY = underestimatedYYYY + 1;
465                 ss1 = calcSerial(1, 1, underestimatedYYYY);
466             }
467             this.year = underestimatedYYYY - 1;
468         }
469
470         final int ss2 = calcSerial(1, 1, this.year);
471
472         int[] daysToEndOfPrecedingMonth
473             = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
474
475         if (isLeapYear(this.year)) {
476             daysToEndOfPrecedingMonth
477                 = LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
478         }
479
480         // Obtenir le mois à partir de la date.
481         int mm = 1;
482         int sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
483         while (sss < this.serial) {
484             mm = mm + 1;
485             sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
486         }
487         this.month = mm - 1;
488
489         // Il reste d(+1);
490         this.day = this.serial - ss2
491                    - daysToEndOfPrecedingMonth[this.month] + 1;
492
493     }
494
495 }
```

Listing B.6 : `RelativeDayOfWeekRule.java`

```
 1 /* ===========================================================================
 2  * JCommon : bibliothèque libre de classes générales pour Java(tm)
 3  * ===========================================================================
 4  *
 5  * (C) Copyright 2000-2005, par Object Refinery Limited et les Contributeurs.
 6  *
 7  * Site du projet :  http://www.jfree.org/jcommon/index.html
 8  *
 9  * Cette bibliothèque est un logiciel libre ; vous pouvez la redistribuer et/ou
10  * la modifier en respectant les termes de la GNU Lesser General Public License
11  * publiée par la Free Software Foundation, en version 2.1 ou (selon votre choix)
12  * en toute version ultérieure.
13  *
14  * Cette bibliothèque est distribuée en espérant qu'elle sera utile, mais SANS
15  * AUCUNE GARANTIE, sans même la garantie implicite de QUALITÉ MARCHANDE ou
16  * d'ADÉQUATION À UN OBJECTIF PRÉCIS. Pour de plus amples détails, consultez
17  * la GNU Lesser General Public License.
18  *
19  * Vous devez avoir reçu un exemplaire de la GNU Lesser General Public License
20  * avec cette bibliothèque. Dans le cas contraire, merci d'écrire à la Free
21  * Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston,
22  * MA 02110-1301, USA.
23  *
24  * [Java est une marque ou une marque déposée de Sun Microsystems, Inc.
25  * aux États-Unis et dans d'autres pays.]
26  *
27  * --------------------------
28  * RelativeDayOfWeekRule.java
29  * --------------------------
30  * (C) Copyright 2000-2003, par Object Refinery Limited et les Contributeurs.
31  *
32  * Auteur :            David Gilbert (pour Object Refinery Limited);
33  * Contributeur(s) :   -;
34  *
35  * $Id: RelativeDayOfWeekRule.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
36  *
37  * Modifications (depuis le 26-Oct-2001)
38  * -------------------------------------
39  * 26-Oct-2001 : Paquetage modifié en com.jrefinery.date.*;
40  * 03-Oct-2002 : Correction des erreurs signalées par Checkstyle (DG);
41  *
42  */
43
44 package org.jfree.date;
45
46 /**
47  * Une règle de date annuelle qui retourne une date pour chaque année basée sur (a)
48  * une règle de référence, (b) un jour de la semaine et (c) un paramètre de sélection
49  * (SerialDate.PRECEDING, SerialDate.NEAREST, SerialDate.FOLLOWING).
50  * <P>
51  * Par exemple, le bon vendredi peut être indiqué sous la forme 'the Friday PRECEDING
52  * Easter Sunday'.
53  *
54  * @author David Gilbert
55  */
```

```
56 public class RelativeDayOfWeekRule extends AnnualDateRule {
57
58     /** Référence à la règle de date annuelle sur laquelle cette règle se fonde. */
59     private AnnualDateRule subrule;
60
61     /**
62      * Le jour de la semaine (SerialDate.MONDAY, SerialDate.TUESDAY, etc.).
63      */
64     private int dayOfWeek;
65
66     /** Précise le jour de la semaine (PRECEDING, NEAREST ou FOLLOWING). */
67     private int relative;
68
69     /**
70      * Constructeur par défaut - crée une règle pour le lundi qui suit le 1er janvier.
71      */
72     public RelativeDayOfWeekRule() {
73         this(new DayAndMonthRule(), SerialDate.MONDAY, SerialDate.FOLLOWING);
74     }
75
76     /**
77      * Constructeur standard - construit une règle basée sur la sous-règle fournie.
78      *
79      * @param subrule  la règle qui détermine la date de référence.
80      * @param dayOfWeek  le jour de la semaine relativement à la date de référence.
81      * @param relative  indique *quel* jour de la semaine (précédent, plus proche ou
82      *                  suivant).
83      */
84     public RelativeDayOfWeekRule(final AnnualDateRule subrule,
85             final int dayOfWeek, final int relative) {
86         this.subrule = subrule;
87         this.dayOfWeek = dayOfWeek;
88         this.relative = relative;
89     }
90
91     /**
92      * Retourne la sous-règle (également appelée règle de référence).
93      *
94      * @return la règle de date annuelle qui détermine la date de référence pour
95      *         cette règle.
96      */
97     public AnnualDateRule getSubrule() {
98         return this.subrule;
99     }
100
101    /**
102     * Fixe la sous-règle.
103     *
104     * @param subrule  la règle de date annuelle qui détermine la date de référence pour
105     *                 cette règle.
106     */
107    public void setSubrule(final AnnualDateRule subrule) {
108        this.subrule = subrule;
109    }
110
111    /**
112     * Retourne le jour de la semaine pour cette règle.
```

```
113      *
114      * @return le jour de la semaine pour cette règle.
115      */
116     public int getDayOfWeek() {
117         return this.dayOfWeek;
118     }
119
120     /**
121      * Fixe le jour de la semaine pour cette règle.
122      *
123      * @param dayOfWeek  le jour de la semaine (SerialDate.MONDAY,
124      *                   SerialDate.TUESDAY, etc.).
125      */
126     public void setDayOfWeek(final int dayOfWeek) {
127         this.dayOfWeek = dayOfWeek;
128     }
129
130     /**
131      * Retourne l'attribut 'relative' qui détermine *quel* jour
132      * de la semaine nous intéresse (SerialDate.PRECEDING,
133      * SerialDate.NEAREST ou SerialDate.FOLLOWING).
134      *
135      * @return l'attribut 'relative'.
136      */
137     public int getRelative() {
138         return this.relative;
139     }
140
141     /**
142      * Fixe l'attribut 'relative' (SerialDate.PRECEDING, SerialDate.NEAREST,
143      * SerialDate.FOLLOWING).
144      *
145      * @param relative  détermine *quel* jour de la semaine est sélectionné par
146      *                  cette règle.
147      */
148     public void setRelative(final int relative) {
149         this.relative = relative;
150     }
151
152     /**
153      * Crée un clone de cette règle.
154      *
155      * @return un clone de cette règle.
156      *
157      * @throws CloneNotSupportedException this should never happen.
158      */
159     public Object clone() throws CloneNotSupportedException {
160         final RelativeDayOfWeekRule duplicate
161             = (RelativeDayOfWeekRule) super.clone();
162         duplicate.subrule = (AnnualDateRule) duplicate.getSubrule().clone();
163         return duplicate;
164     }
165
166     /**
167      * Retourne la date générée par cette règle, pour l'année indiquée.
168      *
169      * @param year  l'année (1900 &lt;= year &lt;= 9999).
```

```
170      *
171      * @return la date générée par cette règle pour l'année indiquée (potentiellement
172      *         <code>null</code>).
173      */
174     public SerialDate getDate(final int year) {
175
176         // Vérifier l'argument...
177         if ((year < SerialDate.MINIMUM_YEAR_SUPPORTED)
178             || (year > SerialDate.MAXIMUM_YEAR_SUPPORTED)) {
179             throw new IllegalArgumentException(
180                 "RelativeDayOfWeekRule.getDate(): year outside valid range.");
181         }
182
183         // Calculer la date...
184         SerialDate result = null;
185         final SerialDate base = this.subrule.getDate(year);
186
187         if (base != null) {
188             switch (this.relative) {
189                 case(SerialDate.PRECEDING):
190                     result = SerialDate.getPreviousDayOfWeek(this.dayOfWeek,
191                             base);
192                     break;
193                 case(SerialDate.NEAREST):
194                     result = SerialDate.getNearestDayOfWeek(this.dayOfWeek,
195                             base);
196                     break;
197                 case(SerialDate.FOLLOWING):
198                     result = SerialDate.getFollowingDayOfWeek(this.dayOfWeek,
199                             base);
200                     break;
201                 default:
202                     break;
203             }
204         }
205         return result;
206
207     }
208
209 }
```

Listing B.7 : `DayDate.java` **(version finale)**

```
 1 /* ===========================================================================
 2  * JCommon : bibliothèque libre de classes générales pour Java(tm)
 3  * ===========================================================================
 4  *
 5  * (C) Copyright 2000-2005, par Object Refinery Limited et les Contributeurs.
...
36  */
37 package org.jfree.date;
38
39 import java.io.Serializable;
40 import java.util.*;
41
```

```
42 /**
43  * Une classe abstraite qui représente des dates immuables avec une précision
44  * d'un jour. L'implémentation associe chaque date à un entier qui représente
45  * le nombre ordinal de jours depuis une origine fixée.
46  *
47  * Pourquoi ne pas employer java.util.Date ? Nous le ferons lorsque ce sera sensé.
48  * Parfois, java.util.Date est *trop* précise - elle représente un instant, au
49  * 1/1000e de seconde (la date dépend elle-même du fuseau horaire). Il arrive
50  * que nous voulions simplement représenter un jour particulier (par exemple, le
51  * 21 janvier 2015) sans nous préoccuper de l'heure, du fuseau horaire ou d'autres
52  * paramètres. C'est pourquoi nous avons défini SerialDate.
53  *
54  * Utilisez DayDateFactory.makeDate pour créer une instance.
55  *
56  * @author David Gilbert
57  * @author Robert C. Martin a effectué un remaniement important.
58  */
59
60 public abstract class DayDate implements Comparable, Serializable {
61   public abstract int getOrdinalDay();
62   public abstract int getYear();
63   public abstract Month getMonth();
64   public abstract int getDayOfMonth();
65
66   protected abstract Day getDayOfWeekForOrdinalZero();
67
68   public DayDate plusDays(int days) {
69     return DayDateFactory.makeDate(getOrdinalDay() + days);
70   }
71
72   public DayDate plusMonths(int months) {
73     int thisMonthAsOrdinal = getMonth().toInt() - Month.JANUARY.toInt();
74     int thisMonthAndYearAsOrdinal = 12 * getYear() + thisMonthAsOrdinal;
75     int resultMonthAndYearAsOrdinal = thisMonthAndYearAsOrdinal + months;
76     int resultYear = resultMonthAndYearAsOrdinal / 12;
77     int resultMonthAsOrdinal = resultMonthAndYearAsOrdinal % 12 + Month.JANUARY.toInt();
78     Month resultMonth = Month.fromInt(resultMonthAsOrdinal);
79     int resultDay = correctLastDayOfMonth(getDayOfMonth(), resultMonth, resultYear);
80     return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
81   }
82
83   public DayDate plusYears(int years) {
84     int resultYear = getYear() + years;
85     int resultDay = correctLastDayOfMonth(getDayOfMonth(), getMonth(), resultYear);
86     return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
87   }
88
89   private int correctLastDayOfMonth(int day, Month month, int year) {
90     int lastDayOfMonth = DateUtil.lastDayOfMonth(month, year);
91     if (day > lastDayOfMonth)
92         day = lastDayOfMonth;
93     return day;
94   }
95
96   public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
97     int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
98     if (offsetToTarget >= 0)
```

```
 99      offsetToTarget -= 7;
100    return plusDays(offsetToTarget);
101  }
102
103  public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
104    int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
105    if (offsetToTarget <= 0)
106      offsetToTarget += 7;
107    return plusDays(offsetToTarget);
108  }
109
110  public DayDate getNearestDayOfWeek(Day targetDayOfWeek) {
111    int offsetToThisWeeksTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
112    int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
113    int offsetToPreviousTarget = offsetToFutureTarget - 7;
114
115    if (offsetToFutureTarget > 3)
116      return plusDays(offsetToPreviousTarget);
117    else
118      return plusDays(offsetToFutureTarget);
119  }
120
121  public DayDate getEndOfMonth() {
122    Month month = getMonth();
123    int year = getYear();
124    int lastDay = DateUtil.lastDayOfMonth(month, year);
125    return DayDateFactory.makeDate(lastDay, month, year);
126  }
127
128  public Date toDate() {
129    final Calendar calendar = Calendar.getInstance();
130    int ordinalMonth = getMonth().toInt() - Month.JANUARY.toInt();
131    calendar.set(getYear(), ordinalMonth, getDayOfMonth(), 0, 0, 0);
132    return calendar.getTime();
133  }
134
135  public String toString() {
136    return String.format("%02d-%s-%d", getDayOfMonth(), getMonth(), getYear());
137  }
138
139  public Day getDayOfWeek() {
140    Day startingDay = getDayOfWeekForOrdinalZero();
141    int startingOffset = startingDay.toInt() - Day.SUNDAY.toInt();
142    int ordinalOfDayOfWeek = (getOrdinalDay() + startingOffset) % 7;
143    return Day.fromInt(ordinalOfDayOfWeek + Day.SUNDAY.toInt());
144  }
145
146  public int daysSince(DayDate date) {
147    return getOrdinalDay() - date.getOrdinalDay();
148  }
149
150  public boolean isOn(DayDate other) {
151    return getOrdinalDay() == other.getOrdinalDay();
152  }
153
154  public boolean isBefore(DayDate other) {
155    return getOrdinalDay() < other.getOrdinalDay();
```

```
156   }
157
158   public boolean isOnOrBefore(DayDate other) {
159     return getOrdinalDay() <= other.getOrdinalDay();
160   }
161
162   public boolean isAfter(DayDate other) {
163     return getOrdinalDay() > other.getOrdinalDay();
164   }
165
166   public boolean isOnOrAfter(DayDate other) {
167     return getOrdinalDay() >= other.getOrdinalDay();
168   }
169
170   public boolean isInRange(DayDate d1, DayDate d2) {
171     return isInRange(d1, d2, DateInterval.CLOSED);
172   }
173
174   public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
175     int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
176     int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
177     return interval.isIn(getOrdinalDay(), left, right);
178   }
179 }
```

Listing B.8 : `Month.java` **(version finale)**

```
 1 package org.jfree.date;
 2
 3 import java.text.DateFormatSymbols;
 4
 5 public enum Month {
 6   JANUARY(1), FEBRUARY(2), MARCH(3),
 7   APRIL(4),   MAY(5),      JUNE(6),
 8   JULY(7),    AUGUST(8),   SEPTEMBER(9),
 9   OCTOBER(10),NOVEMBER(11),DECEMBER(12);
10   private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
11   private static final int[] LAST_DAY_OF_MONTH =
12     {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
13
14   private int index;
15
16   Month(int index) {
17     this.index = index;
18   }
19
20   public static Month fromInt(int monthIndex) {
21     for (Month m : Month.values()) {
22       if (m.index == monthIndex)
23         return m;
24     }
25     throw new IllegalArgumentException("Invalid month index " + monthIndex);
26   }
27
28   public int lastDay() {
29     return LAST_DAY_OF_MONTH[index];
```

```
30   }
31
32   public int quarter() {
33     return 1 + (index - 1) / 3;
34   }
35
36   public String toString() {
37     return dateFormatSymbols.getMonths()[index - 1];
38   }
39
40   public String toShortString() {
41     return dateFormatSymbols.getShortMonths()[index - 1];
42   }
43
44   public static Month parse(String s) {
45     s = s.trim();
46     for (Month m : Month.values())
47       if (m.matches(s))
48         return m;
49
50     try {
51       return fromInt(Integer.parseInt(s));
52     }
53     catch (NumberFormatException e) {}
54     throw new IllegalArgumentException("Invalid month " + s);
55   }
56
57   private boolean matches(String s) {
58     return s.equalsIgnoreCase(toString()) ||
59            s.equalsIgnoreCase(toShortString());
60   }
61
62   public int toInt() {
63     return index;
64   }
65 }
```

Listing B.9 : `Day.java` **(version finale)**

```
 1 package org.jfree.date;
 2
 3 import java.util.Calendar;
 4 import java.text.DateFormatSymbols;
 5
 6 public enum Day {
 7   MONDAY(Calendar.MONDAY),
 8   TUESDAY(Calendar.TUESDAY),
 9   WEDNESDAY(Calendar.WEDNESDAY),
10   THURSDAY(Calendar.THURSDAY),
11   FRIDAY(Calendar.FRIDAY),
12   SATURDAY(Calendar.SATURDAY),
13   SUNDAY(Calendar.SUNDAY);
14
15   private final int index;
```

```
16   private static DateFormatSymbols dateSymbols = new DateFormatSymbols();
17
18   Day(int day) {
19     index = day;
20   }
21
22   public static Day fromInt(int index) throws IllegalArgumentException {
23     for (Day d : Day.values())
24       if (d.index == index)
25         return d;
26     throw new IllegalArgumentException(
27       String.format("Illegal day index: %d.", index));
28   }
29
30   public static Day parse(String s) throws IllegalArgumentException {
31     String[] shortWeekdayNames =
32       dateSymbols.getShortWeekdays();
33     String[] weekDayNames =
34       dateSymbols.getWeekdays();
35
36     s = s.trim();
37     for (Day day : Day.values()) {
38       if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
39           s.equalsIgnoreCase(weekDayNames[day.index])) {
40         return day;
41       }
42     }
43     throw new IllegalArgumentException(
44       String.format("%s is not a valid weekday string", s));
45   }
46
47   public String toString() {
48     return dateSymbols.getWeekdays()[index];
49   }
50
51   public int toInt() {
52     return index;
53   }
54 }
```

Listing B.10 : `DateInterval.java` **(version finale)**

```
 1 package org.jfree.date;
 2
 3 public enum DateInterval {
 4   OPEN {
 5     public boolean isIn(int d, int left, int right) {
 6       return d > left && d < right;
 7     }
 8   },
 9   CLOSED_LEFT {
10     public boolean isIn(int d, int left, int right) {
11       return d >= left && d < right;
12     }
13   },
```

```
14   CLOSED_RIGHT {
15     public boolean isIn(int d, int left, int right) {
16       return d > left && d <= right;
17     }
18   },
19   CLOSED {
20     public boolean isIn(int d, int left, int right) {
21       return d >= left && d <= right;
22     }
23   };
24
25   public abstract boolean isIn(int d, int left, int right);
26 }
```

Listing B.11 : `WeekInMonth.java` **(version finale)**

```
 1 package org.jfree.date;
 2
 3 public enum WeekInMonth {
 4   FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
 5   private final int index;
 6
 7   WeekInMonth(int index) {
 8     this.index = index;
 9   }
10
11   public int toInt() {
12     return index;
13   }
14 }
```

Listing B.12 : `WeekdayRange.java` **(version finale)**

```
1 package org.jfree.date;
2
3 public enum WeekdayRange {
4   LAST, NEAREST, NEXT
5 }
```

Listing B.13 : `DateUtil.java` **(version finale)**

```
 1 package org.jfree.date;
 2
 3 import java.text.DateFormatSymbols;
 4
 5 public class DateUtil {
 6   private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
 7
 8   public static String[] getMonthNames() {
 9     return dateFormatSymbols.getMonths();
10   }
```

```
11
12   public static boolean isLeapYear(int year) {
13     boolean fourth = year % 4 == 0;
14     boolean hundredth = year % 100 == 0;
15     boolean fourHundredth = year % 400 == 0;
16     return fourth && (!hundredth || fourHundredth);
17   }
18
19   public static int lastDayOfMonth(Month month, int year) {
20     if (month == Month.FEBRUARY && isLeapYear(year))
21       return month.lastDay() + 1;
22     else
23       return month.lastDay();
24   }
25
26   public static int leapYearCount(int year) {
27     int leap4 = (year - 1896) / 4;
28     int leap100 = (year - 1800) / 100;
29     int leap400 = (year - 1600) / 400;
30     return leap4 - leap100 + leap400;
31   }
32 }
```

Listing B.14 : `DayDateFactory.java` **(version finale)**

```
 1 package org.jfree.date;
 2
 3 public abstract class DayDateFactory {
 4   private static DayDateFactory factory = new SpreadsheetDateFactory();
 5   public static void setInstance(DayDateFactory factory) {
 6     DayDateFactory.factory = factory;
 7   }
 8
 9   protected abstract DayDate _makeDate(int ordinal);
10   protected abstract DayDate _makeDate(int day, Month month, int year);
11   protected abstract DayDate _makeDate(int day, int month, int year);
12   protected abstract DayDate _makeDate(java.util.Date date);
13   protected abstract int _getMinimumYear();
14   protected abstract int _getMaximumYear();
15
16   public static DayDate makeDate(int ordinal) {
17     return factory._makeDate(ordinal);
18   }
19
20   public static DayDate makeDate(int day, Month month, int year) {
21     return factory._makeDate(day, month, year);
22   }
23
24   public static DayDate makeDate(int day, int month, int year) {
25     return factory._makeDate(day, month, year);
26   }
27
28   public static DayDate makeDate(java.util.Date date) {
29     return factory._makeDate(date);
30   }
```

```
31
32   public static int getMinimumYear() {
33     return factory._getMinimumYear();
34   }
35
36   public static int getMaximumYear() {
37     return factory._getMaximumYear();
38   }
39 }
```

Listing B.15 : `SpreadsheetDateFactory.java` **(version finale)**

```
 1 package org.jfree.date;
 2
 3 import java.util.*;
 4
 5 public class SpreadsheetDateFactory extends DayDateFactory {
 6   public DayDate _makeDate(int ordinal) {
 7     return new SpreadsheetDate(ordinal);
 8   }
 9
10   public DayDate _makeDate(int day, Month month, int year) {
11     return new SpreadsheetDate(day, month, year);
12   }
13
14   public DayDate _makeDate(int day, int month, int year) {
15     return new SpreadsheetDate(day, month, year);
16   }
17
18   public DayDate _makeDate(Date date) {
19     final GregorianCalendar calendar = new GregorianCalendar();
20     calendar.setTime(date);
21     return new SpreadsheetDate(
22       calendar.get(Calendar.DATE),
23       Month.fromInt(calendar.get(Calendar.MONTH) + 1),
24       calendar.get(Calendar.YEAR));
25   }
26
27   protected int _getMinimumYear() {
28     return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
29   }
30
31   protected int _getMaximumYear() {
32     return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
33   }
34 }
```

Listing B.16 : `SpreadsheetDate.java` **(version finale)**

```
 1 /* ===========================================================================
 2  * JCommon : bibliothèque libre de classes générales pour Java(tm)
 3  * ===========================================================================
 4  *
```

```
 5  * (C) Copyright 2000-2005, par Object Refinery Limited et les Contributeurs.
 6  *
...
52  *
53  */
54
55 package org.jfree.date;
56
57 import static org.jfree.date.Month.FEBRUARY;
58
59 import java.util.*;
60
61 /**
62  * Représente une date à l'aide d'un entier, de manière comparable à l'implémentation
63  * dans Microsoft Excel. La plage des dates reconnues va du 1er janvier 1900 au
64  * 31 décembre 9999.
65  * <p/>
66  * Sachez qu'il existe un bogue délibéré dans Excel qui reconnaît l'année 1900
67  * comme une année bissextile alors que ce n'est pas le cas. Pour plus d'informations,
68  * consultez l'article Q181370 sur le site web de Microsoft Microsoft :
69  * <p/>
70  * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
71  * <p/>
72  * Excel emploie la convention 1er janvier 1900 = 1. Cette classe utilise la
73  * convention 1er janvier 1900 = 2.
74  * Par conséquent, le numéro de jour dans cette date sera différent de celui
75  * donné par Excel pour janvier et février 1900... mais Excel ajoute ensuite un jour
76  * supplémentaire (29 février 1900, qui n'existe pas réellement !) et, à partir de là,
77  * les numéros de jours concordent.
78  *
79  * @author David Gilbert
80  */
81 public class SpreadsheetDate extends DayDate {
82   public static final int EARLIEST_DATE_ORDINAL = 2;     // 01/01/1900
83   public static final int LATEST_DATE_ORDINAL = 2958465; // 31/12/9999
84   public static final int MINIMUM_YEAR_SUPPORTED = 1900;
85   public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
86   static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
87     {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
88   static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
89     {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
90
91   private int ordinalDay;
92   private int day;
93   private Month month;
94   private int year;
95
96   public SpreadsheetDate(int day, Month month, int year) {
97     if (year < MINIMUM_YEAR_SUPPORTED || year > MAXIMUM_YEAR_SUPPORTED)
98       throw new IllegalArgumentException(
99         "The 'year' argument must be in range " +
100        MINIMUM_YEAR_SUPPORTED + " to " + MAXIMUM_YEAR_SUPPORTED + ".");
101    if (day < 1 || day > DateUtil.lastDayOfMonth(month, year))
102      throw new IllegalArgumentException("Invalid 'day' argument.");
103
104    this.year = year;
105    this.month = month;
```

```
106     this.day = day;
107     ordinalDay = calcOrdinal(day, month, year);
108   }
109
110   public SpreadsheetDate(int day, int month, int year) {
111     this(day, Month.fromInt(month), year);
112   }
113
114   public SpreadsheetDate(int serial) {
115     if (serial < EARLIEST_DATE_ORDINAL || serial > LATEST_DATE_ORDINAL)
116       throw new IllegalArgumentException(
117         "SpreadsheetDate: Serial must be in range 2 to 2958465.");
118
119     ordinalDay = serial;
120     calcDayMonthYear();
121   }
122
123   public int getOrdinalDay() {
124     return ordinalDay;
125   }
126
127   public int getYear() {
128     return year;
129   }
130
131   public Month getMonth() {
132     return month;
133   }
134
135   public int getDayOfMonth() {
136     return day;
137   }
138
139   protected Day getDayOfWeekForOrdinalZero() {return Day.SATURDAY;}
140
141   public boolean equals(Object object) {
142     if (!(object instanceof DayDate))
143       return false;
144
145     DayDate date = (DayDate) object;
146     return date.getOrdinalDay() == getOrdinalDay();
147   }
148
149   public int hashCode() {
150     return getOrdinalDay();
151   }
152
153   public int compareTo(Object other) {
154     return daysSince((DayDate) other);
155   }
156
157   private int calcOrdinal(int day, Month month, int year) {
158     int leapDaysForYear = DateUtil.leapYearCount(year - 1);
159     int daysUpToYear = (year - MINIMUM_YEAR_SUPPORTED) * 365 + leapDaysForYear;
160     int daysUpToMonth = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[month.toInt()];
161     if (DateUtil.isLeapYear(year) && month.toInt() > FEBRUARY.toInt())
162       daysUpToMonth++;
```

```
163     int daysInMonth = day - 1;
164     return daysUpToYear + daysUpToMonth + daysInMonth + EARLIEST_DATE_ORDINAL;
165   }
166
167   private void calcDayMonthYear() {
168     int days = ordinalDay - EARLIEST_DATE_ORDINAL;
169     int overestimatedYear = MINIMUM_YEAR_SUPPORTED + days / 365;
170     int nonleapdays = days - DateUtil.leapYearCount(overestimatedYear);
171     int underestimatedYear = MINIMUM_YEAR_SUPPORTED + nonleapdays / 365;
172
173     year = huntForYearContaining(ordinalDay, underestimatedYear);
174     int firstOrdinalOfYear = firstOrdinalOfYear(year);
175     month = huntForMonthContaining(ordinalDay, firstOrdinalOfYear);
176     day = ordinalDay - firstOrdinalOfYear - daysBeforeThisMonth(month.toInt());
177   }
178
179   private Month huntForMonthContaining(int anOrdinal, int firstOrdinalOfYear) {
180     int daysIntoThisYear = anOrdinal - firstOrdinalOfYear;
181     int aMonth = 1;
182     while (daysBeforeThisMonth(aMonth) < daysIntoThisYear)
183       aMonth++;
184
185     return Month.fromInt(aMonth - 1);
186   }
187
188   private int daysBeforeThisMonth(int aMonth) {
189     if (DateUtil.isLeapYear(year))
190       return LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
191     else
192       return AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
193   }
194
195   private int huntForYearContaining(int anOrdinalDay, int startingYear) {
196     int aYear = startingYear;
197     while (firstOrdinalOfYear(aYear) <= anOrdinalDay)
198       aYear++;
199
200     return aYear - 1;
201   }
202
203   private int firstOrdinalOfYear(int year) {
204     return calcOrdinal(1, Month.JANUARY, year);
205   }
206
207   public static DayDate createInstance(Date date) {
208     GregorianCalendar calendar = new GregorianCalendar();
209     calendar.setTime(date);
210     return new SpreadsheetDate(calendar.get(Calendar.DATE),
211                                Month.fromInt(calendar.get(Calendar.MONTH) + 1),
212                                calendar.get(Calendar.YEAR));
213
214   }
215 }
```

Annexe C

Référence des heuristiques

Voici les références croisées des indicateurs et des heuristiques. Toutes les autres références croisées peuvent être supprimées.

C1 ..... 16-288, 16-291, 17-306
C2 ..... 16-291, 16-295, 16-303, 17-306
C3 ..... 16-294, 16-295, 16-297, 17-306
C4 17-307
C5 17-307
E1 17-307
E2 17-308
F1 ..... 14-248, 17-308
F2 17-308
F3 17-308
F4 ..... 16-286, 16-297, 17-308
G1 ..... 16-289, 17-309
G2 ..... 16-287, 17-309
G3 ..... 16-287, 17-309
G4 ..... 16-291, 17-310
G5 ..... 9-137, 16-290, 16-296, 16-299, 16-303, 17-310
G6 ..... 6-110, 16-291, 16-292, 16-294, 16-298, 16-301, 16-302, 16-303, 17-311
G7 ..... 16-292, 17-312
G8 ..... 16-294, 17-312
G9 ..... 15-281, 16-294, 16-295, 16-297, 17-313
G10 ..... 5-88, 16-294, 17-313
G11 ..... 15-277, 16-294, 16-297, 16-300, 17-314
G12 ..... 16-295, 16-297, 16-303, 17-314
G13 ..... 16-296, 16-297, 17-314
G14 ..... 16-297, 16-300, 17-314
G15 ..... 16-297, 17-316

Bibliographie

[Alexander] *A Timeless Way of Building*, Christopher Alexander, Oxford University Press, New York, 1979.

[AOSD] Portail du développement de logiciels orienté aspect (*Aspect-Oriented Software Development*), **http://aosd.net**.

[ASM] Page d'accueil d'ASM, **http://asm.objectweb.org/**.

[AspectJ] **http://eclipse.org/aspectj**.

[Beck07] *Implementation Patterns*, Kent Beck, Addison-Wesley, 2007.

[Beck97] *Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.

[BeckTDD] *Test Driven Development*, Kent Beck, Addison-Wesley, 2003.

[CGLIB] Bibliothèque de génération de code (*Code Generation Library*), **http://cglib.sourceforge.net/**.

[Colyer] *Eclipse AspectJ*, Adrian Colyer, Andy Clement, George Hurley et Mathew Webster, Pearson Education, Inc., Upper Saddle River, NJ, 2005.

[DDD] *Domain Driven Design*, Eric Evans, Addison-Wesley, 2003.

[DSL] Langage dédié (*Domain-Specific Language*), **http://fr.wikipedia.org/wiki/Langage_dédié**.

[Fowler] Inversion of Control Containers and the Dependency Injection pattern, **http://martinfowler.com/articles/injection.html**.

[Goetz] *Java Theory and Practice: Decorating with Dynamic Proxies*, Brian Goetz, **http://www.ibm.com/developerworks/java/library/j-jtp08305.html**.

[GOF] *Design Patterns : Catalogue de modèles de conception réutilisables*, Gamma et al., Vuibert, 2002.

[Javassist] Page d'accueil de Javassist, **http://www.csg.is.titech.ac.jp/~chiba/javassist/**.

[JBoss] Page d'accueil de JBoss, **http://jboss.org**.

[JMock] *JMock—A Lightweight Mock Object Library for Java*, **http://jmock.org**.

[Knuth92] *Literate Programming*, Donald E. Knuth, Centre pour l'étude du langage et de l'information, Université Leland Stanford Junior, 1992.

[Kolence] Kenneth W. Kolence, Software physics and computer performance measurements, *Proceedings of the ACM annual conference—Volume 2*, Boston, Massachusetts, pp. 1024–1040, 1972.

[KP78] *The Elements of Programming Style*, 2d. ed., Kernighan et Plaugher, McGraw-Hill, 1978.

[Lea99] *Concurrent Programming in Java: Design Principles and Patterns*, 2d. ed., Doug Lea, Prentice Hall, 1999.

[Mezzaros07] *XUnit Patterns*, Gerard Mezzaros, Addison-Wesley, 2007.

[PPP] *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

[PRAG] *The Pragmatic Programmer*, Andrew Hunt et Dave Thomas, Addison-Wesley, 2000.

[RDD] *Object Design: Roles, Responsibilities, and Collaborations*, Rebecca Wirfs-Brock et al., Addison-Wesley, 2002.

[Refactoring] *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.

[RSpec] *RSpec: Behavior Driven Development for Ruby Programmers*, Aslak Hellesøy et David Chelimsky, Pragmatic Bookshelf, 2008.

[Simmons04] *Hardcore Java*, Robert Simmons, Jr., O'Reilly, 2004.

[SP72] *Structured Programming*, O.-J. Dahl, E. W. Dijkstra et C. A. R. Hoare, Academic Press, Londres, 1972.

[Spring] *The Spring Framework*, **http://www.springframework.org**.

[XPE] *Extreme Programming Explained: Embrace Change*, Kent Beck, Addison-Wesley, 1999.

[WELC] *Working Effectively with Legacy Code*, Addison-Wesley, 2004.

Épilogue

En 2005, alors que je participais à la conférence Agile à Denver, Elisabeth Hedrickson[1] m'a donné un bracelet vert semblable à celui que Lance Armstrong a rendu si populaire. Sur ce bracelet, il était écrit "obsédé par les tests". Je l'ai volontiers et fièrement porté. Depuis que j'avais appris le développement piloté par les tests auprès de Kent Beck en 1999, j'étais effectivement devenu obsédé par le TDD.

Ensuite, une chose étrange s'est produite. Je me suis aperçu que je ne pouvais pas retirer le bracelet. Bien entendu, il était non pas physiquement, mais *moralement* collé. Il annonçait de manière évidente mon éthique professionnelle. Il constituait une indication visible de mon engagement à écrire le meilleur code que je pouvais. En le retirant, j'avais l'impression de trahir mon éthique et mon engagement.

Je porte encore ce bracelet. Lorsque j'écris du code, il se trouve dans mon champ de vision périphérique. Il me rappelle constamment ma promesse de faire de mon mieux pour écrire du code propre.

1. **http://www.qualitytree.com/**.

Index

B

C

D

F

J

L

N

O

P

R

S

T

Référence

CODER proprement

Nettoyez votre code et devenez plus performant !

Si un code sale peut fonctionner, il peut également compromettre la pérennité d'une entreprise de développement de logiciels. Chaque année, du temps et des ressources sont gaspillés à cause d'un code mal écrit. Toutefois, ce n'est pas une fatalité.

Grâce à cet ouvrage, vous apprendrez à rédiger du bon code, ainsi qu'à le nettoyer « à la volée », et vous obtiendrez des applications plus robustes, plus évolutives et donc plus durables. Concret et pédagogique, ce manuel se base sur les bonnes pratiques d'une équipe de développeurs aguerris réunie autour de Robert C. Martin, expert logiciel reconnu. Il vous inculquera les valeurs d'un artisan du logiciel et fera de vous un meilleur programmeur.

Coder proprement est décomposé en trois parties. La première décrit les principes, les pratiques et les motifs employés dans l'écriture d'un code propre. La deuxième est constituée de plusieurs études de cas à la complexité croissante. Chacune d'elles est un exercice de nettoyage : vous partirez d'un exemple de code présentant certains problèmes, et l'auteur vous expliquera comment en obtenir une version saine et performante. La troisième partie, enfin, sera votre récompense. Son unique chapitre contient une liste d'indicateurs éprouvés par l'auteur qui vous seront précieux pour repérer efficacement les défauts de votre code.

Après avoir lu ce livre, vous saurez

- faire la différence entre du bon et du mauvais code ;
- écrire du bon code et transformer le mauvais code en bon code ;
- choisir des noms, des fonctions, des objets et des classes appropriés ;
- mettre en forme le code pour une lisibilité maximale ;
- implémenter le traitement des erreurs sans perturber la logique du code ;
- mener des tests unitaires et pratiquer le développement piloté par les tests.

Véritable manuel du savoir-faire en développement agile, cet ouvrage est un outil indispensable à tout développeur, ingénieur logiciel, chef de projet, responsable d'équipe ou analyste des systèmes dont l'objectif est de produire un meilleur code.

À propos de l'auteur :

Robert C. Martin est développeur professionnel depuis 1970 et consultant logiciel international depuis 1990. Il est fondateur et directeur général de Object Mentor, Inc., une équipe de consultants expérimentés qui dispense auprès de clients du monde entier des conseils dans plusieurs domaines de l'informatique, comme C++, Java, C#, Ruby, l'orienté objet, les motifs de conception, UML, les méthodes agiles et l'eXtreme Programming.

Programmation

Niveau : Tous niveaux
Configuration : Multiplate-forme

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr

ISBN : 978-2-7440-4104-4

9 782744 041044