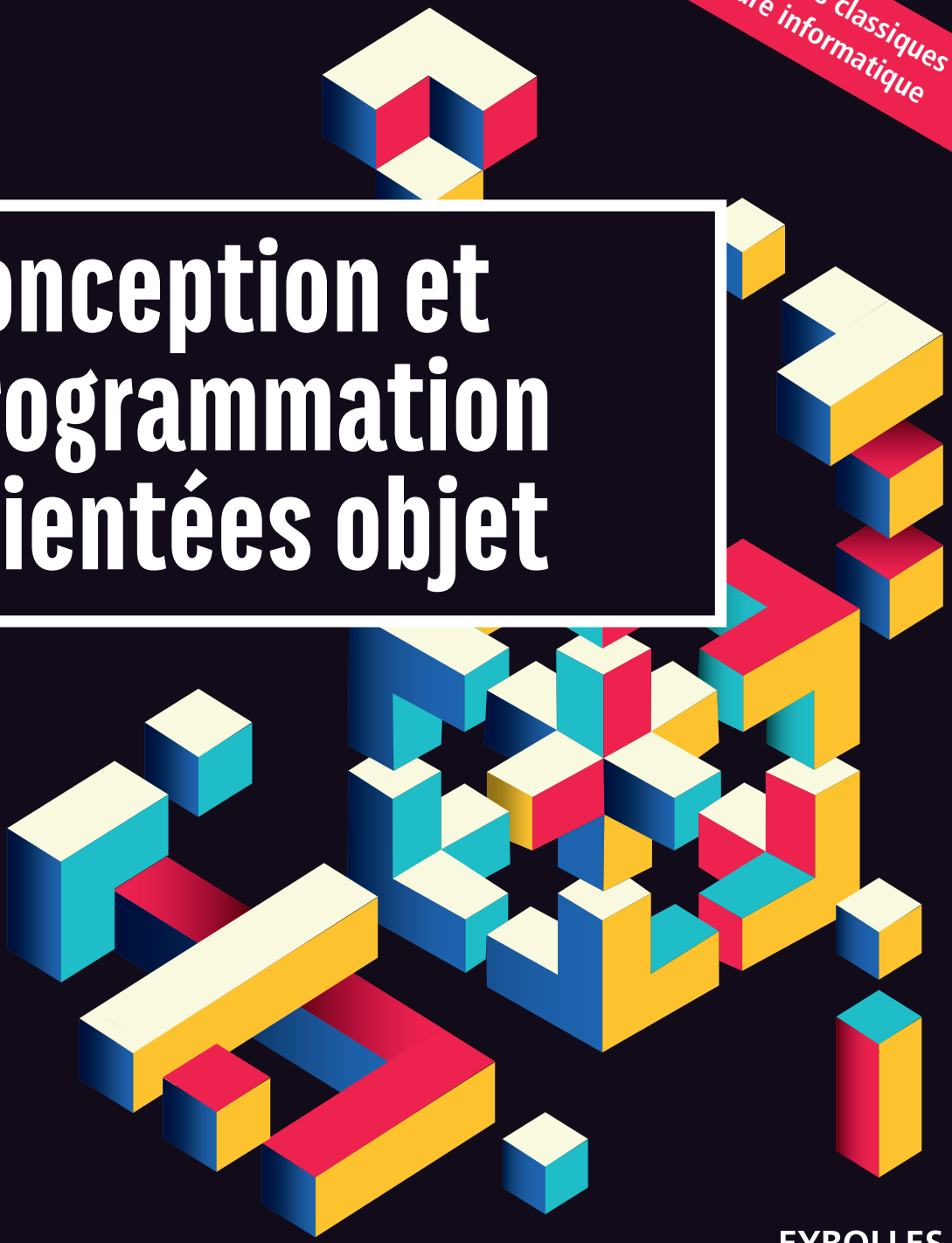
Bertrand Meyer

Un des plus grands classiques de la littérature informatique

Conception et programmation orientées objet



EYROLLES

Conception et programmation orientées objet

CHEZ LE MÊME ÉDITEUR

P. Roques. – **UML 2 par la pratique.** *Cours et exercices.*
N°67565, 8e édition, 400 pages environ, à paraître en 2018 (collection Noire).

P. Roques. – **Mémento UML 2.4.**
N°14356, 3e édition, 2015, 14 pages.

H. Bersini, I. Wellesz. – **La programmation orientée objet.**
Cours et exercices en UML 2 avec Python, PHP, Java, C#, C++.
N°67399, 7e édition, 2017, 672 pages (collection Noire).

C. Delannoy. – **Programmer en langage C++.**
N°67386, 9e édition, 2017, 880 pages (collection Noire).

C. Delannoy. – **Programmer en Java** (*couvre Java 9*).
N°67536, 10e édition, 2017, 920 pages (collection Noire).

P. Martin, J. Pauli, C. Pierre de Geyer et E. Daspet. – **PHP 7 avancé.**
N°14357, 2016, 728 pages.

G. Swinnen. – **Apprendre à programmer avec Python 3.**
N°13434, 3e édition, 2012, 456 pages (collection Noire).

Bertrand Meyer

Conception et programmation orientées objet

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Traduction autorisée de l'ouvrage en langue anglaise intitulé :
Object-Oriented Software Construction, 2nd Edition, Prentice Hall, 1997,
ISBN : 978-0-136291-55-8

Traduction de l'anglais : Pierre Jouvelot
Validation technique : Éric Bezault

La présente édition en langue française est parue initialement en 2000 sous l'ISBN 978-2-212-09111-3, puis en 2007 sous l'ISBN 978-2-212-12270-1. À l'occasion de ce septième tirage, elle bénéficie d'une nouvelle couverture. Le texte reste inchangé.

Sommaire

Partie A
Les problèmes *1*

1. La qualité du logiciel • 3

2. Critères d'orientation objet • 23

Partie B
La route de l'orientation objet *39*

3. Modularité • 41

4. Approches de la réutilisabilité • 69

5. Vers la technologie objet • 105

6. Types abstraits de données • 125

Partie C
Techniques orientées objet *167*

7. La structure statique : les classes • 169

14. Introduction à l'héritage • 445

15. Héritage multiple • 503

16. Techniques d'héritage • 551

17. Typage • 593

22. Comment trouver les classes • 697

23. Principes de conception des classes • 723

24. Bien utiliser l'héritage • 781

25. Techniques utiles • 841

26. Un penchant pour le style • 845

27. Analyse orientée objet • 873

28. Le processus de construction logicielle • 893

29. Enseigner la méthode • 905

PARTIE E
ASPECTS AVANCÉS *919*

30. Concurrence, répartition, client-serveur et Internet • 921

31. Persistance d'objets et bases de données • 1003

32. Quelques techniques OO pour applications graphiques interactives • 1029

Partie F
Appliquer la méthode dans divers langages et environnements *1045*

33. Programmation OO et Ada • 1047

34. Émulation de la technologie objet dans les environnements non OO • 1067

35. De Simula à Java et au-delà : principaux langages et environnements OO • 1081

Partie G
Faire les choses bien *1109*

36. Un environnement orienté objet • 1111

Épilogue : le langage dévoilé • 1129

Partie H
ANNEXES 1131

A. Extraits des bibliothèques Base • 1133

B. Généricité et héritage • 1135

C. Glossaire de la technologie objet • 1155

D. Bibliographie • 1167

Index • 1193

Préface

Née dans le bleu glacé des fjords de la Norvège ; amplifiée (par une incompréhensible aberration des courants sous-marins) près des côtes quelque peu grisâtres du Pacifique californien ; considérée par certains comme un typhon, par d'autres comme un tsunami, et par d'autres encore comme une tempête dans une tasse de thé — une vague de fond vient de déferler sur les rives du monde informatique.

"Objet" est le terme à la mode, venu concurrencer et souvent remplacer "structuré" dans le rôle de synonyme haute technologie de "bel et bon". Comme il est de règle en pareil cas, le terme n'a pas le même sens pour tous ceux qui l'emploient ; toute aussi inévitable est la réaction en trois temps qui menace l'introduction d'un nouveau principe méthodologique : (1) "c'est trivial " ; (2) "ça ne peut pas marcher" ; (3) "c'est toujours ainsi que j'ai travaillé". (Ordre variable selon l'individu.)

Mettons tout de suite les choses au net, de peur que le lecteur, un seul instant, soupçonne l'auteur de la moindre timidité vis-à-vis de son propre sujet : je ne considère pas la technologie objet comme une mode passagère ; je pense qu'elle est loin d'être triviale (tout en m'étant bien sûr efforcé de la rendre aussi limpide que j'ai pu) ; je sais qu'elle marche et je suis convaincu qu'elle est non seulement différente des techniques qui dominent la pratique du logiciel — y compris certains des principes encore enseignés par les manuels universitaires — mais, dans une certaine mesure, incompatible avec elles. Je crois au demeurant que la technologie objet offre à l'industrie du logiciel le potentiel d'une transformation en profondeur ; je suis persuadé qu'elle s'est installée pour longtemps et je constate que personne, jusqu'ici, n'a rien proposé ni même esquissé qui puisse sérieusement prétendre à la remplacer. J'espère enfin que le lecteur, au fur et à mesure de sa progression au travers des pages qui suivent, partagera un peu mon enthousiasme pour cette fascinante approche de l'analyse, de la conception et de l'implémentation du logiciel.

"Approche de l'analyse, de la conception et de l'implémentation du logiciel". Pour présenter la technologie objet, ce livre prend résolument le point de vue du génie logiciel, c'est-à-dire des méthodes, outils et techniques destinés à la production industrielle de logiciel de qualité. Ce n'est pas la seule perspective possible : on s'est aussi intéressé aux objets pour leurs contributions à l'intelligence artificielle, au prototypage d'applications, à la programmation expérimentale, aux applications parallèles et distribuées, aux bases de données. La plupart de ces domaines seront traités (les deux derniers, en particulier, font chacun l'objet d'un chapitre détaillé), mais l'objectif principal reste la question fondamentale du génie logiciel : comment apporter au développement de logiciel l'amélioration fondamentale dont notre industrie, et son

enseignement universitaire, ont si profondément besoin ? L'apport de la technologie objet peut ici – on le découvrira au fil de son apprentissage et de son application – se révéler déterminant.

Structure, fiabilité, épistémologie et taxonomie

Réduite à l'essentiel, la technologie objet est la combinaison de quatre idées : une méthode de structuration, une discipline de fiabilité, un principe épistémologique et une technique de classification.

La *méthode de structuration* guide la décomposition des systèmes en composants, et la réutilisation de ces composants. Un système logiciel effectue certaines actions sur des objets de certains types ; pour obtenir des produits flexibles et réutilisables, il vaut mieux déterminer leur structure à partir des types d'objets qu'à partir des actions. Cette observation conduit à un concept remarquable par sa puissance et l'étendue de son champ d'application : la notion de classe, qui, en technologie objet, sert de base aussi bien à la structure modulaire des logiciels qu'au système de typage sous-jacent.

La *discipline de fiabilité* apporte une réponse radicale à ce qui est peut-être le problème central de notre métier : comment construire du logiciel qui fasse exactement ce qu'il est censé faire. L'idée est de traiter un système logiciel comme une collection de composants qui collaborent de la même façon que les entreprises prospères : en inscrivant les termes de cette collaboration dans des **contrats** qui définissent explicitement et précisément les obligations et avantages applicables à chacune des parties. Le principe de la *conception par contrat*, qui conduit à repenser profondément les techniques de construction, de test et de documentation du logiciel, sert de fil conducteur à l'ensemble de cet ouvrage.

Les types abstraits de données sont décrits au chapitre 6, lequel évoque également certains aspects épistémologiques apparentés.

Le *principe épistémologique* concerne le problème de description des classes. Avec la technologie objet, les objets décrits par une classe ne sont définis que par ce que nous pouvons faire avec eux : les opérations (appelées également *caractéristiques*) et les propriétés formelles de ces opérations (les contrats). Cette idée est exprimée de manière formelle dans la théorie des **types abstraits de données**, évoquée en détail dans un chapitre de ce livre. Elle a des implications profondes, qui vont parfois au-delà du logiciel, et explique pourquoi nous ne devons pas nous limiter au concept naïf d'"objet" véhiculé par le sens usuel de ce mot. Dans le domaine de la modélisation des systèmes d'information, il est couramment admis de pouvoir faire l'hypothèse d'"une réalité externe" qui existerait indépendamment de tout programme y faisant référence ; pour le développeur orienté objet, une telle notion n'a pas de sens, puisque la réalité n'existe pas indépendamment de ce que vous voulez en faire. (De manière plus précise, qu'elle existe ou non est une question sans fondement, car nous ne connaissons que ce que nous pouvons utiliser, et ce que nous savons d'une chose est entièrement défini par la manière dont nous pouvons l'utiliser.)

La *technique de classification* découle de l'observation selon laquelle tout travail intellectuel systématique, et le raisonnement scientifique en fait partie, impose la définition de taxonomies du domaine étudié. Le logiciel ne fait pas exception à cette règle, et la méthode orientée objet s'appuie fortement sur une discipline de classification appelée **héritage**.

Simple mais puissant

Les quatre concepts de classe, de contrat, de type abstrait de données et d'héritage soulèvent bon nombre de questions. Comment allons-nous trouver et décrire les classes ? Comment les programmes vont-ils manipuler les classes et les objets correspondants (les *instances* de ces classes) ? Quelles sont les relations possibles entre classes ? Comment allons-nous pouvoir profiter des similarités qui peuvent exister entre diverses classes ? Quels sont les rapports entre ces idées et les préoccupations clés que sont, pour le génie logiciel, l'extensibilité, la facilité d'utilisation et l'efficacité ?

Les réponses à ces questions reposent sur un éventail limité mais puissant de techniques permettant de produire des logiciels réutilisables, extensibles et fiables : le polymorphisme et la liaison dynamique ; une nouvelle approche des types et de la vérification de type ; la généricité, contrainte ou non ; la rétention d'information ; les assertions ; la gestion sûre des exceptions ; le ramasse-miettes automatique. Des techniques efficaces d'implémentation ont été développées pour permettre d'appliquer ces idées à la réussite de projets, grands et petits, soumis aux contraintes sévères du développement logiciel commercial. Les techniques orientées objet ont également eu un impact considérable sur les interfaces utilisateur et les environnements de développement, permettant de produire des systèmes interactifs bien meilleurs qu'auparavant. Toutes ces idées importantes seront étudiées en détail, de façon à munir le lecteur d'outils immédiatement applicables à une vaste panoplie de problèmes.

Organisation de l'ouvrage

Dans les pages qui suivent, nous aborderons les méthodes et techniques de la construction logicielle orientée objet. Cette présentation est divisée en six parties.

La partie A est une introduction et un rapide survol du sujet. Elle débute avec l'évocation de cet aspect essentiel qu'est la qualité logicielle, avant de poursuivre avec une brève présentation des principales caractéristiques techniques de la méthode. Cette partie forme presque un petit livre en soi, fournissant une première approche de l'orientation objet pour lecteurs pressés. *Chapitres 1 à 2.*

La partie B adopte un rythme plus posé. Intitulée "La route de l'orientation objet", elle prend le temps de décrire les aspects méthodologiques qui conduisent aux concepts OO fondamentaux. L'accent est mis sur la modularité : ce qui est nécessaire pour concevoir des structures adéquates pour la construction de systèmes en vraie grandeur. Elle s'achève par une présentation des types abstraits de données, fondement mathématique de la technologie objet. Les mathématiques mises en jeu ici sont élémentaires, et les lecteurs peu férus de mathématiques pourront se limiter aux idées de base, mais la présentation fournit le fondement théorique nécessaire à une totale compréhension des principes OO. *Chapitres 3 à 6.*

La partie C forme le noyau technique du livre. Elle présente, un à un, les composants techniques centraux de la méthode : les classes ; les objets et le modèle exécutif associé ; les aspects de gestion de la mémoire ; la généricité et le typage ; la conception par contrat, les assertions et les exceptions ; l'héritage, les concepts associés de polymorphisme et de liaison dynamique et leurs nombreuses applications. *Chapitres 7 à 18.*

Chapitres 19 à 29. La partie D évoque les aspects logiques, en mettant l'accent sur l'analyse et la conception. Au travers de plusieurs études de cas approfondies, elle présente certains *schémas de conception* (*design patterns*) fondamentaux, et répond à certaines questions clés : comment trouver les classes, comment utiliser l'héritage à bon escient et comment concevoir des bibliothèques réutilisables. Elle débute par une réflexion plus philosophique sur les exigences intellectuelles des méthodologistes et autres donneurs de leçons ; elle s'achève avec un résumé du processus logiciel utilisé lors de développements OO (le modèle du cycle de vie) et une évocation de la meilleure technique pour enseigner la méthode, que ce soit dans l'industrie ou en milieu académique.

Chapitres 30 à 32. La partie E explore certains sujets plus sophistiqués : la concurrence, la distribution, le développement client-serveur et Internet ; la persistance, l'évolution de schéma et les bases de données orientées objet ; la conception de systèmes interactifs munis d'interfaces graphiques (GUI) modernes.

Chapitres 33 à 35. La partie F est un résumé de la manière dont les idées présentées peuvent être implémentées ou, dans certains cas, émulées dans divers langages et environnements. Elle contient, en particulier, une présentation des principaux langages orientés objet, en mettant l'accent sur Simula, Smalltalk, Objective-C, C++, Ada 95 et Java, et une évaluation de la possibilité d'obtenir, dans des langages non OO comme Fortran, Cobol, Pascal, C et Ada, certains avantages de l'orientation objet.

Chapitre 36. La partie G (*le faire bien*) décrit un environnement qui va au-delà de ces solutions et fournit un jeu intégré d'outils permettant d'appliquer les idées de ce livre.

Annexe A. L'annexe A fournit une source de référence complémentaire sur certaines bibliothèques de classes réutilisables importantes évoquées dans le livre, apportant un modèle pour la conception de logiciels réutilisables.

Un livre à hyperliens

Il est souvent amusant de voir les efforts entrepris par les auteurs pour recommander certains chemins de lecture de leur ouvrage, allant parfois jusqu'à introduire des cartes compliquées de parcours — comme si les lecteurs y faisaient attention, et n'étaient pas suffisamment malins pour suivre leur propre voie. Un auteur doit, cependant, avoir le droit de dire dans quelle perspective il a conçu l'ordre des différents chapitres, et quel chemin il avait en tête pour celui qu'Umberto Eco appelle le "lecteur modèle" — à ne pas confondre avec le lecteur réel, connu aussi sous le nom de "vous", fait de chair, de sang et de goûts.

La réponse est, ici, des plus simples. Ce livre raconte une histoire, et suppose que le lecteur modèle suivra cette histoire du début à la fin, en ayant toutefois le loisir d'éviter les sections les plus spécialisées signalées comme "pouvant être sautées en première lecture" et, s'il n'est pas particulièrement intéressé par les aspects mathématiques des choses, d'ignorer certains développements mathématiques clairement désignés comme tels. Le lecteur réel pourra, bien évidemment, souhaiter découvrir à l'avance certains des développements ultérieurs de l'action, ou limiter son attention à quelques épiphénomènes précis ; chaque chapitre a donc été conçu de manière aussi indépendante que possible, de façon que vous puissiez en apprécier le contenu à votre rythme.

Puisque l'histoire présentée ici révèle une vision cohérente du développement logiciel, les sujets qui seront successivement abordés sont fortement liés. Des références croisées ont été introduites dans les marges de l'ouvrage, offrant ainsi des "hyperliens" à la WWW reliant les différentes sections. Mon conseil au lecteur modèle est de les ignorer lors d'une première lecture, sauf pour s'assurer qu'un sujet laissé pendant sera bien traité en profondeur par la suite. Le lecteur réel, qui n'a peut-être pas envie d'être conseillé, peut utiliser les références croisées comme des guides officieux lui permettant de s'accommoder de l'ordre prédéfini des sujets abordés.

Les références croisées seront essentiellement utiles aux lecteurs modèle et réel lors des lectures ultérieures, car elles leur permettront de s'assurer qu'ils maîtrisent un conception orienté objet donnée dans sa globalité et dans ses relations avec les autres composantes de la méthode. Comme les hyperliens d'un document WWW, les références croisées devraient permettre de suivre facilement de telles associations.

La notation

Dans le domaine du logiciel plus que nulle part ailleurs, pensée et langage sont fortement liés. Au fur et à mesure de notre progression, nous développerons soigneusement une notation permettant d'exprimer les concepts orientés objet à tous les niveaux : modélisation, analyse, conception, implémentation, maintenance.

Ici, comme partout dans ce livre, le pronom "nous" ne fait pas référence à "l'auteur" : comme dans le langage ordinaire, "nous" veut dire vous et moi — le lecteur et l'auteur. En d'autres termes, je souhaiterais, au cours du développement de la notation, que vous soyez partie prenante de ce processus de conception.

Cette hypothèse n'est, bien évidemment, pas totalement réaliste, car la notation existait avant que vous commenciez à lire ces pages. Mais elle n'est pas complètement ridicule non plus, car j'espère, tout au long de notre exploration de la méthode orientée objet et de l'examen approfondi des implications de la notation associée, qu'une certaine forme d'inévitabilité vous saisira, de sorte que vous ayez vraiment l'impression d'avoir participé à sa conception.

Cela explique pourquoi, bien que la notation ait existé depuis plus de dix ans et soit, en fait, associée à plusieurs implémentations commerciales, y compris celle provenant de ma propre entreprise (ISE), j'ai minimisé son rôle de langage. (Son nom n'apparaît qu'à un endroit dans le texte même, et plusieurs fois dans la bibliographie.) Ce livre traite de la méthode orientée objet pour réutiliser, analyser, concevoir, implémenter et maintenir le logiciel ; le langage est une part importante et, j'espère, une conséquence naturelle de cette méthode, mais il n'est pas un but en soi.

De plus, le langage n'introduit pas de complications inutiles, ne contenant que ce qui est strictement nécessaire à la méthode. Les étudiants de première année qui l'ont utilisé ont remarqué que "ce n'était pas un langage du tout" — indiquant par là que la notation est en relation directe avec la méthode : apprendre l'une revient à apprendre l'autre, et il y a peu de décorations linguistiques supplémentaires qui viennent s'ajouter aux concepts. La notation contient, effectivement, peu de ces singularités (qui proviennent souvent de circonstances

historiques, de contraintes machine ou d'exigences de compatibilité avec des formalismes plus anciens) qui caractérisent la plupart des langages de programmation d'aujourd'hui. Bien sûr, vous pouvez ne pas être d'accord avec le choix des mots clés (pourquoi `do` plutôt que `begin` ou même `faire` ?) ou préférer l'ajout de points-virgules après chaque instruction. (La syntaxe a été conçue de manière à rendre les points-virgules optionnels.) Mais ce sont des points mineurs. Ce qui compte, c'est la simplicité de la notation et la façon dont elle s'accorde aux concepts. Si vous comprenez la technologie objet, vous connaissez presque déjà cette notation.

La plupart des livres traitant de questions logicielles considèrent le langage comme acquis, qu'il s'agisse d'un langage de programmation ou d'une notation d'analyse ou de conception. Ici, l'approche est différente ; faire intervenir le lecteur lors de la conception impose non seulement d'expliquer le langage, mais aussi de le justifier et d'évoquer d'autres alternatives. La plupart des chapitres de la partie C contiennent une section de "discussion" qui explique les enjeux de la conception de la notation et la manière dont ils ont été relevés. J'ai souvent souhaité, en lisant des descriptions de langages bien connus, que les concepteurs aient indiqué non pas seulement les solutions qu'ils avaient retenues, mais pourquoi ils les avaient choisies et quelles étaient les alternatives qu'ils avaient rejetées. Les discussions franches de ce livre devraient, je l'espère, vous fournir des éléments de réflexion concernant non seulement la conception de langages, mais également la construction logicielle, car ces deux tâches sont étonnamment similaires.

Analyse, conception et implémentation

Il est toujours dangereux d'utiliser une notation qui ressemble extérieurement à un langage de programmation, car cela peut suggérer qu'elle ne concerne que la phase d'implémentation. Cette impression, aussi fausse soit-elle, est difficile à corriger, car on a fréquemment affirmé aux décideurs et autres développeurs qu'il existait un fossé de proportion quasi métaphysique entre l'éther de la conception et de l'analyse et le bas monde de l'implémentation.

"INTÉGRATION ET RÉVERSIBILITÉ", 28.6, page 900.

La technologie orientée objet, quand elle est bien comprise, réduit considérablement ce fossé en insistant sur l'unité primordiale du développement logiciel et non sur les différences inévitables qui séparent les niveaux d'abstraction. Cette approche *intégrée* (*seamless*) de la construction logicielle est l'une des contributions importantes de la méthode et se reflète dans le langage de ce livre, qui vise aussi bien l'analyse et la conception que l'implémentation.

Malheureusement, l'évolution récente du domaine va à l'encontre de ces principes, au travers de deux phénomènes aussi regrettables l'un que l'autre :

- les langages d'implémentation orientés objet qui ne sont pas adaptés à l'analyse, à la conception et, de manière plus générale, au raisonnement de haut niveau ;
- l'analyse et les méthodes de conception orientées objet qui ne couvrent pas l'implémentation (et sont présentées comme "indépendantes du langage", comme s'il s'agissait là d'un signe honorifique et non d'un aveu d'échec).

Ces idées annihilent une grande partie des retombées positives de l'approche. En revanche, la méthode et la notation développées dans ce livre ont pour objectif d'être applicables à l'ensemble du processus de construction logicielle. Un certain nombre de chapitres concernent les aspects

de conception de haut niveau ; l'un est consacré à l'analyse ; d'autres explorent les techniques d'implémentation et les conséquences de la méthode sur les performances.

L'environnement

La construction logicielle repose sur une tétralogie de base : une méthode, un langage, des outils, des bibliothèques. La méthode est au centre de ce livre ; l'aspect langage vient d'être évoqué. De temps en temps, il nous faudra réfléchir à ce que nous devons espérer des outils et des bibliothèques. Pour des raisons pratiques évidentes, ces discussions feront occasionnellement référence à l'environnement orienté objet d'ISE, avec sa panoplie d'outils et de bibliothèques associées.

Le chapitre 36 résume l'environnement.

L'environnement n'est utilisé qu'à titre d'exemple de ce qu'il est possible de faire pour rendre utilisables en pratique, par des développeurs logiciels, les concepts introduits. Soyez bien conscient qu'il existe de nombreux environnements orientés objet, à la fois pour la notation de ce livre et pour d'autres notations et méthodes OO d'analyse, de conception et d'implémentation ; et que les descriptions fournies font référence à l'environnement tel qu'il existait au moment de la rédaction de cet ouvrage (NdT : sa version anglaise), et sont donc sujettes, comme le reste de notre industrie, à de rapides changements — en mieux. D'autres environnements, OO ou non, sont également cités dans ce livre.

Remerciements (et leur quasi absence)

La première édition de ce livre contenait une liste déjà longue de remerciements. J'ai continué, pendant un certain temps, à écrire les noms des personnes qui y avaient contribué par leurs commentaires ou suggestions, puis j'ai perdu le fil. L'aréopage de collègues qui m'ont aidé ou dont j'ai emprunté certaines idées est devenu si vaste que lister leurs noms nécessiterait plusieurs pages, au risque d'en oublier certains importants. Mieux vaut donc les offenser tous un peu qu'en offenser gravement quelques-uns.

Quelques notes dans la marge ou dans les sections bibliographiques en fin de chapitre rendent justice à certaines idées spécifiques, souvent non publiées.

Ces remerciements resteront, pour l'essentiel, collectifs, ce qui n'en diminue en rien ma gratitude. Mes collègues à ISE et SOL ont, pendant des années, été une source quotidienne d'aide inestimable. Les utilisateurs de nos outils nous ont généreusement fourni leurs conseils. Les lecteurs de la première édition ont fourni des milliers de suggestions pour améliorer l'ouvrage. Lors de la préparation de cette nouvelle édition (je devrais plutôt dire de ce nouveau livre), j'ai envoyé des centaines de messages électroniques pour demander de l'aide de toutes sortes : la clarification d'un point de détail, une référence bibliographique, une permission de citation, les détails d'une attribution, l'origine d'une idée, les spécificités d'une notation, l'adresse officielle d'une page Web ; les réponses ont toujours été positives. Au fur et à mesure de l'élaboration des chapitres, ceux-ci étaient distribués par divers moyens, suscitant de nombreux commentaires constructifs (et je dois citer ici les noms des relecteurs nommés par Prentice Hall, Paul Dubois, James McKim et Richard Wiener, qui m'ont fourni d'inestimables conseils et corrections). Au cours des années précédentes, j'ai donné un nombre incalculable de séminaires, conférences et cours sur les sujets abordés dans ce livre et, dans chaque cas, j'ai retiré quelque chose des commentaires de l'auditoire. J'ai apprécié l'humour et l'esprit de mes collègues lors des tables-

rondes organisées lors de conférences, et ai bénéficié de leur sagesse. De courts séjours sabbatiques à l'University of Technology de Sydney et à l'Università degli Studi di Milano m'ont fourni de nouvelles idées — et, dans le premier cas, trois cents étudiants de première année auprès desquels j'ai pu valider certaines de mes idées concernant la façon dont le génie logiciel devrait être enseigné.

La bibliographie volumineuse montre combien les idées et réalisations des autres ont contribué à ce livre. Parmi les influences les plus marquantes figurent la famille des langages de programmation Algol, caractérisée par son élégance syntaxique et sémantique ; les travaux fondateurs sur la programmation structurée, au sens sérieux du terme (Dijkstra-Hoare-Parnas-Wirth-Mills-Gries), et la construction systématique de programmes ; les techniques de spécification formelle, en particulier les leçons inépuisables tirées de la version originale (fin des années soixante-dix) du langage de spécification Z de Jean-Raymond Abrial, sa nouvelle conception de B et le travail de Cliff Jones sur VDM ; les langages de la génération modulaire (en particulier Ada de Ichbiah, CLU de Liskov, Alphard de Shaw, LPG de Bert et Modula de Wirth) ; et Simula 67, où avait été introduite, bien des années auparavant, et pour l'essentiel sans coup férir, la plus grande partie des concepts nécessaires rappelant ainsi le commentaire de Tony Hoare à propos d'Algol 60 : qu'il s'agissait là d'une claire amélioration par rapport à la majorité de ses successeurs.

Préface de la seconde édition

De nombreux événements se sont produits dans le monde orienté objet depuis la première édition d'*OOSC* (le nom sous lequel ce livre (NdT : sa version anglaise) est désormais connu) publiée en 1988. L'explosion de l'intérêt pour la technologie OO suggéré dans la préface de la première édition, reproduite dans les pages qui précèdent sous une forme légèrement étendue, n'était rien en comparaison de ce à quoi nous avons assisté depuis. De nombreux journaux et conférences sont aujourd'hui dédiés à la technologie objet ; Prentice Hall (NdT : éditeur de la version anglaise) propose une série entière de livres consacrés au sujet ; des développements marquants se sont produits dans des domaines comme les interfaces utilisateur, la concurrence et les bases de données ; de nouveaux sujets d'étude ont émergé, comme l'analyse OO et les spécifications formelles ; le calcul réparti, qui était autrefois un sujet spécialisé, devient de plus en plus d'actualité dans les développements actuels, en partie grâce à la croissance d'Internet ; et le Web influence le travail quotidien de tout un chacun.

Et les bonnes nouvelles ne s'arrêtent pas là. Le progrès constant du monde du logiciel fait plaisir à voir — en partie grâce à la diffusion encore incomplète mais irréfutable de la technologie objet. Trop nombreux sont les livres et articles consacrés au génie logiciel qui débutent encore par les lamentations obligées sur la "crise du logiciel" et l'état pitoyable de notre industrie comparée aux *vraies* disciplines d'ingénierie (qui, comme nous le savons tous, ne ratent jamais rien). Un tel aveu n'est pas de mise. Oh, nous avons encore bien du chemin à parcourir, comme le savent trop bien tous ceux qui utilisent des produits logiciels. Mais, étant donné les défis auxquels nous devons faire face, il n'y a pas de raison d'avoir honte de nous ; et nous nous améliorons à chaque instant. L'ambition de ce livre, comme celle de son prédécesseur, est de faciliter ce processus.

Cette seconde édition n'est pas une mise à jour, mais le résultat d'une profonde refonte. Il n'est pas un paragraphe de la version d'origine qui n'ait pas été modifié. (En fait, presque pas une seule ligne.) De nouveaux sujets ont été rajoutés, y compris un chapitre complet sur la concurrence, la répartition, le calcul client-serveur et la programmation Internet ; un autre sur la persistance et les bases de données ; un sur les interfaces utilisateur ; un sur le cycle de vie du logiciel ; de nombreux schémas de conception et techniques d'implémentation ; une exploration en profondeur d'un aspect méthodologique peu documenté auparavant : comment utiliser au mieux, ou ne pas utiliser, l'héritage et éviter de l'utiliser de travers ; des discussions sur de nombreux autres sujets de méthodologie orientée objet ; une présentation extensive de la théorie des types abstraits de données — le fondement mathématique de notre sujet, indispensable à toute compréhension en profondeur de la technologie objet, et rarement présenté en détail dans les livres d'enseignement et les tutoriels ; une présentation de l'analyse OO ; des centaines de nouvelles références bibliographiques et de sites Web ; la description d'un environnement orienté objet complet et de ses concepts sous-jacents ; et des tonnes de nouveaux principes, idées, avertissements, explications, figures, exemples, comparaisons, citations, classes et routines.

Les réactions à *OOSC*-1 ont été si gratifiantes que je sais que les lecteurs attendent beaucoup. J'espère que ce qu'ils trouveront dans *OOSC*-2 les intriguera, leur sera utile et répondra à leurs attentes.

Santa Barbara B.M.
Janvier 1997

À propos de la bibliographie, des sources Internet et des exercices

Ce livre repose sur les contributions antérieures de nombreux auteurs. Pour faciliter la lecture du texte, le renvoi aux sources a été reporté à la fin de chaque chapitre, dans la section “Notes bibliographiques”. Ces notes vous seront une source précieuse d’informations complémentaires.

Les références se présentent sous la forme [*Nom* 19xx], où *Nom* est le nom du premier auteur et fait référence à la bibliographie de l’annexe D. Cette convention a pour seul but de faciliter la lecture et ne diminue en rien le rôle des autres auteurs. La lettre M à la place de *Nom* indique que la publication émane de l’auteur de ce livre ; la liste des publications de l’auteur se trouve dans la seconde partie de la bibliographie.

La bibliographie débute page 1167.

Certaines références bibliographiques apparaissent dans la marge, à côté des paragraphes qui les citent. Cette distinction vise à rendre la bibliographie utilisable en elle-même, comme un pôle de références sur la technologie objet et sur les sujets afférents. Une note en marge et non entrée dans la bibliographie ne signifie en rien un jugement de valeur ; cette répartition est le fruit d’une simple évaluation pragmatique de l’appartenance ou non à une liste de références centrales sur la technologie objet.

Étant donné la volatilité des adresses électroniques, nous ne pouvons garantir que certaines n’aient pas changé au cours du temps, depuis la version originale de ce livre. Les outils de recherche par mots clés sont à votre disposition pour retrouver les références qui seraient erronées.

La plupart des chapitres contiennent des exercices de difficulté variable. L’auteur s’est imposé de ne pas donner de solutions, bien que de nombreux exercices fournissent des suggestions relativement précises. Les raisons qui ont motivé ce choix sont les suivantes : la peur de gâcher le plaisir du lecteur, le fait que de nombreux exercices traitent de problèmes de conception, pour lesquels il existe plusieurs bonnes réponses, enfin le désir de fournir aux enseignants un matériel pédagogique directement utilisable.

Par de pareils objets les âmes sont blessées,
et cela fait venir de funestes pensées.

Molière, *Tartuffe*, Acte III, Scène II.

Partie A

LES PROBLÈMES

La partie A définit les objectifs de notre recherche en mettant l'accent sur la notion de qualité du logiciel et, pour les lecteurs qui ne sont pas contre les annonces préliminaires, fournit un résumé des aspects essentiels de la technologie objet.

1

La qualité du logiciel

L ’ingénierie traque la qualité ; le génie logiciel vise la production de logiciels de qualité. Ce livre présente un ensemble de techniques qui promettent d’améliorer sensiblement la qualité des produits logiciels.

Avant d’étudier ces techniques, nous devons clarifier leurs raisons d’être. La qualité du logiciel résulte de la combinaison de plusieurs facteurs. Ce chapitre analyse certains d’entre eux, dégage les points qui méritent d’être améliorés et esquisse les grands axes possibles de solutions que nous explorerons lors de notre parcours.

1.1 FACTEURS EXTERNES ET INTERNES

Nous souhaitons tous que nos systèmes logiciels soient rapides, fiables, faciles à utiliser, simples à lire, modulaires, structurés et ainsi de suite. Mais ces adjectifs recouvrent deux classes de qualités bien différentes.

D’un côté, nous envisageons des qualités comme la vitesse ou la facilité d’utilisation, dont la présence, ou non, dans un produit logiciel peut être détectée par ses utilisateurs. Ces propriétés peuvent être appelées des facteurs **externes** de qualité.

> Derrière le mot “utilisateur”, nous devrions envisager non seulement les personnes qui interagissent effectivement avec les produits finaux, comme l’agent d’une compagnie aérienne utilisant un système de réservation des vols, mais aussi celles qui achètent le logiciel ou en commandent la réalisation à l’extérieur, comme le décideur d’une compagnie aérienne chargé d’acquérir ou de passer commande de systèmes de réservation des vols. Ainsi, une propriété comme la facilité avec laquelle le logiciel peut être adapté aux changements de spécifications — appelée *extensibilité* ci-après — tombe dans la catégorie des facteurs externes, bien qu’elle puisse ne pas être d’un intérêt immédiat pour les “utilisateurs finaux” que sont, par exemple, les agents de réservation.

Les autres qualités d’un produit logiciel, comme celles d’être modulaire ou facile à lire, sont des facteurs **internes**, seulement perceptibles aux informaticiens professionnels qui ont accès au texte source du logiciel.

En fin de compte, seuls les facteurs externes ont de l’importance. Si j’utilise un navigateur Web ou si j’habite près d’une centrale nucléaire gérée par ordinateur, je me soucie peu de savoir si le programme source est lisible ou modulaire quand le chargement des images prend des heures ou quand une donnée incorrecte fait exploser l’installation. Mais l’ingrédient essentiel permettant d’obtenir ces qualités externes réside dans les facteurs internes : pour que les

utilisateurs puissent jouir de ces qualités visibles, les concepteurs et implémenteurs devront appliquer des techniques garantissant ces qualités cachées.

Les chapitres suivants présentent un ensemble de techniques modernes garantissant la qualité interne. Il n'en faudrait pas pour autant oublier l'essentiel ; les techniques internes ne sont pas une fin en soi, mais un moyen d'atteindre les qualités logicielles externes. Il nous faut donc débuter par une analyse de ces facteurs externes.

1.2 RAPPEL DES FACTEURS EXTERNES

Voici les facteurs de qualité externe les plus importants, ceux que vise en premier la construction de logiciels orientés objet.

Correction

> ***Définition : correction***
>
> La correction est la capacité que possède un produit logiciel de mener à bien sa tâche, telle qu'elle a été définie par sa spécification.

La correction est la qualité essentielle. Si un système ne fait pas ce qu'il est supposé faire, tout le reste — être rapide, posséder une interface utilisateur agréable… — compte peu.

Mais c'est plus facile à dire qu'à faire. Même la première étape est déjà difficile : nous devons être à même de spécifier de manière précise les besoins du système, ce qui est une tâche passablement ardue.

Les méthodes garantissant la correction seront, en général, **conditionnelles**. Un système logiciel sérieux, même petit d'après les canons actuels, touche à tellement de choses qu'il serait impossible de garantir sa correction en traitant tous ses composants et propriétés sur le même plan. De fait, une approche par couches sera nécessaire, chaque couche se reposant sur celles qui lui sont inférieures :

Couches dans le développement logiciel

Dans cette approche conditionnelle de la correction, nous ne nous préoccupons de garantir la correction d'une couche *que dans l'hypothèse où* celles de niveau inférieur sont correctes. C'est la seule technique réaliste, car elle permet de cerner les problèmes et nous laisse nous concentrer, à chaque étape, sur un ensemble restreint de questions. Vous ne pouvez pas, en pratique, vérifier qu'un programme écrit dans un langage de haut niveau X est correct sans supposer que le compilateur utilisé implémente correctement X. Cela ne veut pas forcément dire que vous devez avoir une confiance aveugle dans le compilateur, mais simplement que

vous décomposez le problème en deux : correction du compilateur et correction de votre programme par rapport à la sémantique du langage.

Dans la méthode décrite dans ce livre, on fait intervenir encore plus de niveaux : le développement logiciel reposera sur des bibliothèques de composants réutilisables, qui peuvent être intégrés dans maintes applications différentes.

Couches dans un processus de développement qui favorise la réutilisation

L'approche conditionnelle s'appliquera également ici : nous devrons nous assurer que les bibliothèques sont correctes et, séparément, vérifier la correction de l'application, celle des bibliothèques étant admises.

Beaucoup de praticiens, quand on aborde la question de la correction du logiciel, pensent test et débogage. Nous pouvons être plus ambitieux : dans les prochains chapitres, nous explorerons un certain nombre de techniques, en particulier le typage et les assertions, qui ont pour mission de faciliter la construction de logiciels corrects dès le début — plutôt que de tenter d'atteindre cette correction par débogage. Débogage et test restent, bien sûr, indispensables comme moyen de vérification supplémentaire du résultat.

Il est possible d'aller encore plus loin et d'adopter une approche complètement formelle de la construction de logiciel. Ce livre ne vise pas un tel objectif, comme le laissent sous-entendre les termes prudents de "vérifier", "garantir" et "assurer" que nous avons utilisés ci-dessus à la place de "prouver". Toutefois, plusieurs techniques décrites dans les chapitres suivants découlent directement des travaux sur les techniques mathématiques utilisées dans les spécifications et vérifications formelles de programmes, et nous rapproche ainsi sensiblement de cet idéal de correction.

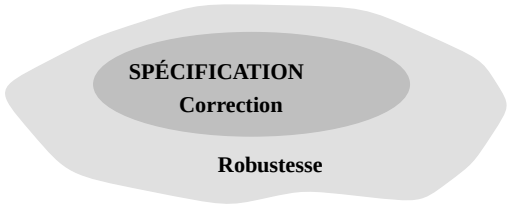
Robustesse

Définition : robustesse

La robustesse est la capacité qu'offrent des systèmes logiciels à réagir de manière appropriée à la présence de conditions anormales.

La robustesse complète la correction. La correction concerne le comportement d'un système dans les cas qui sont conformes à ses spécifications ; la robustesse caractérise ce qui se passe en dehors de cette spécification.

La robustesse par rapport à la correction



Comme le suggère le style littéraire utilisé dans sa définition, la robustesse est une notion par nature plus floue que la correction. Puisque nous nous intéressons ici à des cas qui ne sont pas couverts par la spécification, il n'est pas possible de dire, comme pour la correction, que le système devrait "effectuer son travail" dans de telles conditions ; si ce travail était connu, les cas anormaux deviendraient une partie de la spécification et nous nous retrouverions dans le domaine de la correction.

Sur le traitement des exceptions, voir le chapitre 12.

> Cette définition de "cas anormal" sera à nouveau utile quand nous étudierons le traitement des exceptions. Elle sous-entend que les concepts de normal et d'anormal sont toujours relatifs à une certaine spécification ; un cas anormal correspond simplement à un cas qui n'est pas couvert par la spécification. Si vous étendez les spécifications, les cas qui étaient anormaux deviennent normaux — même s'ils correspondent à des événements, comme l'entrée de données incorrectes, que vous préféreriez éviter. Ici, "normal" ne veut pas dire "désirable", mais simplement "pris en compte lors de la conception du logiciel". Bien qu'il puisse, de prime abord, paraître bizarre de considérer l'entrée d'une donnée erronée comme un cas normal, toute autre approche devrait s'appuyer sur des critères subjectifs et serait donc inutilisable.

Il y aura toujours des cas qui ne seront pas explicitement couverts par une spécification. L'exigence de robustesse a pour objectif de s'assurer que, si de tels cas se présentent, le système ne causera pas de catastrophes ; il devrait fournir des messages d'erreur appropriés ou entrer dans un mode appelé "dégradation harmonieuse".

Extensibilité

Définition : extensibilité

L'extensibilité est la facilité d'adaptation des produits logiciels aux changements de spécifications.

Le logiciel se doit d'être *flexible* et, de fait, l'est, en principe ; rien n'est plus facile à modifier qu'un programme si vous avez accès à son code source. Utilisez simplement votre éditeur de texte.

Le problème de l'extensibilité est un problème d'échelle. Pour de petits programmes, un changement n'est généralement pas une affaire d'état ; mais, au fur et à mesure que le logiciel grossit, il devient de plus en plus difficile à adapter. Un grand système logiciel ressemble, pour ceux qui le maintiennent, à un gigantesque château de cartes dans lequel le retrait d'un élément quelconque peut conduire à l'effondrement de l'édifice tout entier.

Nous avons besoin de l'extensibilité, car la base de tout logiciel repose sur une intervention humaine et son cortège de caprices. Le cas typique des logiciels de gestion (les "systèmes de

gestion de l'information"), dans lesquels la promulgation d'une nouvelle loi ou l'acquisition d'une entreprise peuvent soudainement invalider les hypothèses sur lesquelles repose un système, n'est pas unique ; même dans le cas du calcul numérique, où les lois de la physique ne changent pas d'un mois à l'autre, notre façon de comprendre et de modéliser les systèmes physiques changera.

Les approches traditionnelles du génie logiciel n'ont, jusqu'ici, pas suffisamment pris en compte la notion de changement. Elles ont plutôt privilégié une vision idéale du cycle de vie du logiciel dans laquelle une phase initiale d'analyse gèle, une fois pour toutes, les exigences, le reste du processus étant dévolu à la conception et à l'élaboration d'une solution. Ceci est compréhensible : le premier impératif pour faire progresser cette discipline était de développer des techniques solides permettant d'exprimer et de résoudre des problèmes donnés. On n'envisageait pas les cas où le problème changerait en cours de traitement. Mais, dorénavant, puisque les techniques de base du génie logiciel sont en place, il devient essentiel d'aborder ce point capital. Le changement est omniprésent dans les développements logiciels : changement d'exigences, de notre compréhension de celles-ci, des algorithmes, des modes de représentation de données, des techniques d'implémentation. Prendre en compte le changement est un objectif fondamental de la technologie objet et un thème permanent de ce livre.

Bien que certaines techniques améliorant l'extensibilité puissent être introduites à l'aide de petits exemples ou dans des cours d'initiation, leur pertinence ne se révèle que dans des grands projets. Deux principes sont essentiels à l'amélioration de l'extensibilité :

- *simplicité de conception* : une architecture simple sera toujours plus facile à adapter aux changements.
- *décentralisation* : une modification d'un module aura d'autant moins d'incidence sur les autres modules que celui-ci sera autonome, évitant ainsi le déclenchement d'une réaction en chaîne de changements dans l'ensemble du système.

La méthode orientée objet est, avant toute chose, une méthode d'architecture de systèmes qui aide les concepteurs à concevoir des systèmes dont la structure soit à la fois simple (même dans les grands systèmes) et décentralisée. Simplicité et décentralisation seront des thèmes récurrents des chapitres prochains qui nous conduiront aux principes orientés objet.

Réutilisabilité

Définition : réutilisabilité

La réutilisabilité est la capacité des éléments logiciels à servir à la construction de plusieurs applications différentes.

L'attrait de la réutilisabilité provient du fait que les systèmes logiciels suivent souvent les mêmes modèles ; il devrait être possible d'exploiter ces ressemblances et d'éviter de réinventer des solutions à des problèmes qui se sont déjà posés. En s'inspirant de ces modèles, un élément logiciel réutilisable sera applicable à de nombreux développements différents.

La réutilisabilité influence tous les autres aspects de la qualité du logiciel, car, si le problème de la réutilisabilité est résolu, on aura moins de logiciel à écrire et, donc, plus de temps à

consacrer (à coût constant) à l'amélioration des autres facteurs, comme la correction et la robustesse.

On retrouve ici une problématique que l'approche traditionnelle du cycle de vie du logiciel n'a pas réellement abordée, et ce pour la même raison historique : il faut que vous ayez déjà trouvé les moyens de résoudre un problème avant d'envisager de les appliquer à d'autres problèmes. Mais, avec la croissance du développement logiciel et les efforts déployés pour en faire une véritable industrie, le besoin de réutilisabilité est devenu pressant.

Voir chapitre 4.

La réutilisabilité jouera un rôle central dans les chapitres à venir, l'un d'eux étant, de fait, entièrement dévolu à une analyse fouillée de ce facteur de qualité, ses bénéfices pratiques et les questions qu'il pose.

Compatibilité

> ***Définition : compatibilité***
> La compatibilité est la facilité avec laquelle des éléments logiciels peuvent être combinés à d'autres.

La compatibilité est importante, car nous ne développons pas les éléments logiciels dans le vide : ils doivent interagir entre eux. Mais des problèmes se produisent souvent au cours de ces interactions parce que les modules font des hypothèses différentes sur le reste du monde. Un exemple en est la grande variété de formats de fichiers incompatibles présents dans les systèmes d'exploitation. Un programme ne peut utiliser le résultat d'un autre en entrée que si leurs formats de fichiers sont compatibles.

L'absence de compatibilité peut mener au désastre. En voici un cas extrême :

San Jose (Calif.) Mercury News, 20 Juillet 1992. Cité dans le forum de discussion Usenet "comp.risks", 13.67, Juillet 1992 (extrait).

> *DALLAS — La semaine dernière, AMR, maison-mère d'American Airlines, Inc., a indiqué avoir échoué dans le projet de développement d'un tout nouveau système de réservation commun à l'ensemble des industries du voyage, incluant également les locations de voitures et les chambres d'hôtels.*
>
> *AMR n'a stoppé le développement de son nouveau système de réservation Confirm que quelques semaines après la date à laquelle celui-ci aurait dû commencer à traiter les premières transactions de ses partenaires Budget Rent-A-Car, Hilton Hotels Corp. et Marriott Corp. L'arrêt de ce projet de 125 millions de dollars, étalé sur 4 ans, s'est traduit pour AMR par une provision pour pertes avant impôts de 165 millions de dollars et a fragilisé la réputation de l'entreprise, présentée comme chef de file dans le secteur du voyage.* […]
>
> *Dès janvier, les dirigeants de Confirm ont découvert que le travail de plus de 200 programmeurs, analystes système et ingénieurs n'avait apparemment servi à rien. Les parties essentielles de ce projet pharaonique — sa description tient dans 47 000 pages — avaient été développées séparément, en utilisant des méthodes différentes. Lorsqu'on les a mises ensemble, elles ne pouvaient pas coopérer.*

> *Les développeurs n'ont pas réussi à connecter ensemble les morceaux. Les différents "modules" ne pouvaient pas extraire l'information requise de chaque coté du canal qui les reliait.*
> *AMR Information Services a licencié huit membres confirmés du projet, dont le chef de projet.* […] *Fin juin, Budget et Hilton ont annoncé qu'ils se retiraient du projet.*

La clé de la compatibilité réside dans l'homogénéité de la conception et dans l'élaboration de conventions standardisées pour les communications interprogrammes. Parmi les approches possibles, on trouve :

- Les formats de fichiers standardisés, comme dans le système de fichiers d'Unix où chaque fichier texte est une simple séquence de caractères.
- Les structures de données standardisées, comme dans les systèmes Lisp où toutes les données, y compris les programmes, sont représentées par des arbres binaires (appelés listes dans Lisp).
- Les interfaces utilisateur standardisées, comme avec les diverses versions de Windows, OS/2 et MacOS dans lesquelles tous les outils reposent sur un paradigme unique de communication avec l'utilisateur, basé sur des composants standard comme les fenêtres, les icônes, les menus, etc.

Des solutions plus générales découlent de la définition de protocoles standardisés d'accès aux entités importantes manipulées par le logiciel. C'est l'idée qui conduit aux types abstraits de données et à l'approche orientée objet, ainsi qu'aux protocoles *middleware* comme CORBA et OLE-COM de Microsoft (ActiveX).

Sur les types abstraits de données, voir chapitre 6.

Efficacité

Définition : efficacité

L'efficacité est la capacité d'un système logiciel à utiliser le minimum de ressources matérielles, que ce soit le temps machine, l'espace occupé en mémoire externe et interne, ou la bande passante des moyens de communication.

Le mot "performance" est souvent utilisé comme synonyme d'efficacité. La communauté du logiciel adopte deux attitudes typiques vis-à-vis de l'efficacité :

- Certains développeurs sont obsédés par les questions de performance, ce qui les pousse à dépenser beaucoup d'efforts dans de soi-disant optimisations.
- Mais une autre faction tend à minimiser l'importance de la performance, comme l'illustrent certains dictons, monnaie courante de la profession, comme "Faites bien les choses avant de les faire vite" ou "De toute façon, l'ordinateur de l'année prochaine sera 50 % plus rapide".

Il n'est pas rare de voir la même personne adopter ces deux attitudes à des moments différents, soit une version logicielle de la schizophrénie (Dr. Abstrait contre Mr. Microseconde).

Où se trouve la vérité ? Certes, les développeurs ont souvent montré un intérêt exagéré pour les micro-optimisations. Comme on l'a déjà vu, l'efficacité n'a pas beaucoup d'importance si le logiciel est incorrect (ce qui suggère un nouveau dicton, "*Ne te préoccupe pas de la vitesse à laquelle ça tourne tant que ce n'est pas correct*", qui ressemble au précédent sans en être exactement l'équivalent). Plus généralement, le souci d'efficacité doit être mis en rapport avec

les autres objectifs que sont l'extensibilité et la réutilisabilité, des optimisations poussées pouvant rendre le logiciel si spécialisé qu'il ne serait plus adaptable ou réutilisable. Qui plus est, la puissance toujours croissante du matériel informatique nous permet d'être un peu plus serein quand il s'agit de gagner un dernier octet ou une ultime microseconde.

Tout ceci, pourtant, ne doit pas minimiser l'efficacité. Personne n'aime attendre les réponses d'un système interactif, ni acheter plus de mémoire pour exécuter un programme. Les attitudes cavalières envers l'efficacité tiennent souvent de l'effet de manche ; si le système se révèle finalement si lent et volumineux qu'il en devient inutilisable, ceux qui soutenaient que "la vitesse importe peu" ne seront pas les derniers à se plaindre.

Ce point révèle ce qui me paraît être une caractéristique essentielle du génie logiciel, laquelle n'est d'ailleurs pas près de disparaître : la construction de logiciels est difficile parce qu'elle demande la prise en compte de maintes exigences différentes, certaines d'entre elles, comme la correction, étant abstraites et conceptuelles alors que d'autres, comme l'efficacité, sont concrètes et liées aux propriétés des matériels informatiques.

Pour certains scientifiques, le développement logiciel est une branche des mathématiques ; pour certains ingénieurs, il s'agit d'une technologie appliquée. En fait, c'est un peu les deux à la fois. L'ingénieur logiciel doit concilier les concepts abstraits et leur application concrète, ainsi que les mathématiques du calcul correct et les contraintes de temps et d'espace qui découlent des lois physiques et des limitations de la technologie matérielle actuelle. Faire plaisir à la fois aux anges et aux démons constitue peut-être le défi clé du génie logiciel.

L'accroissement constant de la puissance de calcul, si impressionnante soit-elle, n'est pas une excuse pour mépriser l'efficacité, et ce pour au moins trois raisons :

- Quiconque achète un ordinateur plus gros et plus rapide espère retirer un bénéfice substantiel de cette puissance supplémentaire — pour aborder de nouveaux problèmes, traiter plus rapidement des problèmes anciens ou résoudre des cas plus complexes dans le même laps de temps. Utiliser un nouvel ordinateur pour traiter les problèmes précédents sans gain de temps n'impressionnera personne !
- Une des retombées les plus marquantes de l'accroissement de la puissance de calcul est d'*accentuer* l'avance des algorithmes efficaces par rapport aux autres. Supposons que la nouvelle machine soit deux fois plus rapide que la précédente. Soit n la taille du problème à résoudre et N la valeur maximum de n pouvant être traitée en un temps donné. Alors, si l'algorithme est en O (n), c'est-à-dire s'il tourne en un temps qui est proportionnel à n, la nouvelle machine vous permettra d'aborder des problèmes de taille environ $2 * N$ si N est suffisamment grand. Pour un algorithme en O (n^2) la nouvelle machine ne permettra qu'un accroissement de 41% de N. Un algorithme en O (2^n), similaire à ces algorithmes combinatoires qui font une recherche exhaustive des solutions, incrémenterait seulement N de un — votre retour sur investissement serait, pour le moins, limité.
- Dans certains cas, l'efficacité peut influencer la correction. Une spécification peut exiger que la réponse de l'ordinateur à un événement donné se produise dans un certain délai ; par exemple, un ordinateur de bord doit être prêt à détecter et à traiter un message provenant du détecteur de manette des gaz en un temps suffisamment court pour permettre une action correctrice. Cette relation entre efficacité et correction n'est pas limitée aux seules applications communément considérées comme "temps réel" ; peu de gens seront intéressés

par un modèle de prévision météorologique qui prend vingt-quatre heures pour prévoir le temps du lendemain.

> Voici un autre exemple, peut-être moins crucial, qui m'a souvent embêté : un système de fenêtrage que j'ai utilisé pendant un certain temps était parfois trop lent à détecter que le pointeur de la souris avait été déplacé d'une fenêtre à une autre et, en conséquence, les caractères tapés au clavier, qui concernaient une fenêtre donnée, se retrouvaient parfois dans une autre.
>
> Ici, la limitation de performance entraîne une violation de la spécification, c'est-à-dire de la correction, ce qui, même dans des applications banales de tous les jours, peut avoir des conséquences fâcheuses : pensez à ce qui peut arriver si les deux fenêtres sont utilisées pour envoyer du courrier électronique à deux correspondants différents. Des mariages ont été annulés, et même des guerres déclenchées, pour moins que ça.

Puisque ce livre est centré sur les concepts du génie logiciel orienté objet, seuls quelques passages traitent explicitement de l'impact qu'auront ceux-ci sur les performances. Mais le souci de performance sera omniprésent. Chaque fois que l'étude introduira une solution orientée objet d'un problème, on veillera à ce que cette solution soit non seulement élégante, mais aussi efficace ; chaque fois qu'un nouveau mécanisme orienté objet sera introduit, qu'il s'agisse du ramasse-miettes (ou d'autres approches de gestion mémoire pour calculs orientés objet), de la liaison dynamique, de la généricité ou de l'héritage répété, il le sera avec la garantie de pouvoir être implémenté à un coût raisonnable en temps et espace ; et chaque fois que cela sera approprié, les conséquences sur les performances des techniques étudiées seront mentionnées.

L'efficacité n'est qu'un des facteurs de qualité ; nous ne devrions pas (comme le font certains dans la profession) nous laisser guider par elle. Mais il doit néanmoins être pris en considération, que ce soit lors de la construction d'un système logiciel ou dans la conception d'un langage de programmation. Si vous répudiez la performance, la performance vous répudiera.

Portabilité

Définition : portabilité

La portabilité est la facilité avec laquelle des produits logiciels peuvent être transférés d'un environnement logiciel ou matériel à un autre.

La portabilité traite des différences non seulement entre matériels physiques distincts, mais, plus généralement, entre diverses **machines matérielles-logicielles**, celles que nous programmons en pratique, y compris les systèmes d'exploitation, éventuellement les systèmes de fenêtrage et autres outils de base. Dans le reste de ce livre, le mot "plate-forme" sera utilisé pour désigner un type de machine matérielle-logicielle ; un exemple de plate-forme est "Intel X86 sous Windows NT" (connu sous le nom "Wintel").

Une grande partie des incompatibilités entre plate-formes n'est pas justifiée et, pour un observateur naïf, la seule explication possible consiste parfois à envisager l'existence d'une conspiration visant à brimer l'humanité, en général, et les programmeurs, en particulier. Quelle qu'en soit cependant la cause, cette diversité fait que la portabilité reste une préoccupation majeure des développeurs et des utilisateurs de logiciel.

Facilité d'utilisation

Définition : facilité d'utilisation

La facilité d'utilisation est la facilité avec laquelle des personnes présentant des formations et des compétences différentes peuvent apprendre à utiliser les produits logiciels et à s'en servir pour résoudre des problèmes. Elle recouvre également la facilité d'installation, d'opération et de contrôle.

La définition insiste sur les différences de compétence des utilisateurs potentiels. Cette exigence présente un défi majeur aux concepteurs de logiciel qui se préoccupent de facilité d'utilisation : comment fournir une aide et des explications détaillées à des utilisateurs inexpérimentés, tout en évitant d'ennuyer les utilisateurs plus aguerris qui veulent aller directement à l'essentiel ?

Comme pour beaucoup d'autres qualités étudiées dans ce chapitre, une des clés permettant de faciliter l'utilisation est la simplicité de structure. Un système bien conçu, construit selon une structure claire et bien pensée, sera plus facile à apprendre et à utiliser qu'un autre construit n'importe comment. Cette condition n'est pas suffisante, bien sûr (ce qui est simple et clair pour le concepteur peut être complexe et obscur pour les utilisateurs, en particulier si les explications sont données dans des termes propres au concepteur et non à l'utilisateur), mais cela aide considérablement.

C'est un des domaines dans lesquels la méthode orientée objet est particulièrement productive ; plusieurs techniques OO, qui semblent de prime abord se limiter à la conception et à l'implémentation, fournissent également des idées d'interface qui facilitent la vie des utilisateurs finaux. Les chapitres à venir en fourniront plusieurs exemples.

Voir Wilfred J. Hansen, "User Engineering Principles for Interactive Systems", Proceedings of FJCC 39, AFIPS Press, Montvale (NJ), 1971, pages 523-532.

Les concepteurs logiciels concernés par la facilité d'utilisation devront se méfier du précepte le plus souvent cité dans la littérature afférente aux interfaces homme-machine, et tiré d'un article précurseur de Hansen : ***Connais l'utilisateur***. L'argument est qu'un bon concepteur doit faire l'effort de comprendre la communauté présumée des utilisateurs du système. Ce point de vue ignore une des caractéristiques des systèmes ayant connu un certain succès : ils vont toujours au-delà de leur audience initiale. (Deux exemples anciens et célèbres sont Fortran, conçu comme un outil permettant de résoudre les problèmes d'une petite communauté d'ingénieurs et de scientifiques qui programmaient un IBM 704, et Unix, prévu pour un usage interne aux Bell Laboratories.) Un système conçu pour un groupe spécifique sera fondé sur des hypothèses qui ne seront plus valables pour une audience plus large.

Les bons concepteurs d'interface utilisateur adoptent une politique plus prudente. Ils font le minimum d'hypothèses possibles concernant leurs utilisateurs. Quand vous concevez un système interactif, vous pouvez supposer que les utilisateurs sont des membres de la race humaine et qu'ils savent lire, déplacer une souris, cliquer sur un bouton et taper (lentement) sur un clavier ; pas beaucoup plus. Si le logiciel concerne un domaine d'application spécialisé, vous pouvez peut-être supposer que l'utilisateur est au courant de ses principes de base. Mais même cela est risqué. Pour renverser, en le paraphrasant, le conseil de Hansen :

Principe de conception d'interface utilisateur

Ne prétendez pas connaître l'utilisateur ; vous ne le connaissez pas.

Fonctionnalité

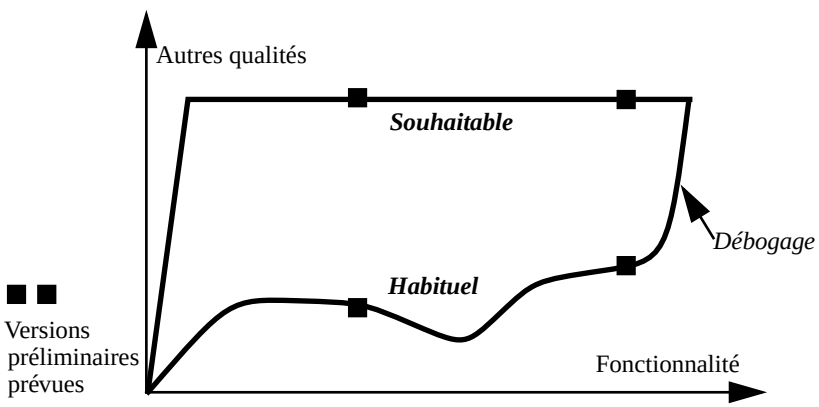
Définition : fonctionnalité
La fonctionnalité est l'étendue des possibilités offertes par un système.

Un des problèmes les plus ardus que rencontre un chef de projet est de déterminer le niveau de fonctionnalité qui suffit. L'incitation à ajouter toujours plus de choses, connue dans le jargon de l'industrie par l'expression de *course aux options* (*featurism*) (souvent "*course larvée aux options*"), est toujours vivace. Ses conséquences sont fâcheuses dans les projets internes, où la pression vient des utilisateurs à l'intérieur d'une même entreprise, et plus encore dans les produits commerciaux, car la partie la plus visible d'une étude comparative effectuée par un journaliste est souvent un tableau récapitulant, côte à côte, les caractéristiques offertes par les produits concurrents.

La course aux options combine, en fait, deux problèmes, l'un plus difficile que l'autre. Le problème le plus simple est la perte de cohérence qui peut résulter de l'ajout de nouveaux gadgets, gênant la facilité d'utilisation. En effet, les utilisateurs ont tendance à se plaindre que tous les "gadgets" (*bells and whistles*) de la nouvelle version d'un produit le rendent extrêmement complexe. Ces commentaires ne devraient pas néanmoins être pris au pied de la lettre, car ces nouveaux gadgets ne viennent pas du néant : ils ont été, la plupart du temps, demandés par les utilisateurs — d'*autres* utilisateurs. Ce qui m'apparaît comme une bagatelle superflue peut être, pour vous, un outil indispensable.

La solution consiste, ici, à s'efforcer de maintenir la cohérence globale du produit en essayant de faire tout tenir dans un moule général. Un bon produit logiciel est basé sur un petit nombre d'idées puissantes ; même s'il y a beaucoup de détails spécifiques, on devrait pouvoir tous les considérer comme dérivant de ces concepts de base. Le "plan général" doit rester visible et tout devrait y avoir sa place.

Le problème le plus difficile est d'éviter d'être tellement obnubilé par les détails que l'on en oublie les autres qualités. Les projets tombent fréquemment dans ce piège, une situation illustrée clairement par Roger Osmond sous la forme des deux parcours possibles vers l'achèvement d'un projet :



***Courbes d'Osmond ; d'après* [Osmond 1995]**

La courbe du bas (en noir) est malheureusement trop fréquente : dans la course frénétique à l'ajout de nouvelles fonctionnalités, le développement perd de vue la qualité globale. La phase finale, dont le but est d'obtenir que tout soit comme il faut, peut être longue et stressante. Si, sous la pression des utilisateurs ou des concurrents, vous êtes obligé d'avancer la diffusion des produits — aux moments indiqués par des carrés noirs sur le dessin — le résultat peut nuire à votre réputation.

Osmond suggère (courbe en grisé), grâce aux techniques d'amélioration de la qualité qu'offre le développement OO, de maintenir un niveau constant de qualité, sous tous ses aspects hormis la fonctionnalité, durant l'ensemble du projet. Vous refusez ainsi de compromettre fiabilité, extensibilité et autres : vous n'introduisez de nouvelles fonctions que lorsque vous êtes satisfait de celles dont vous disposez.

Cette méthode est plus difficile à appliquer du fait des pressions mentionnées, mais fournit un processus de développement logiciel plus efficace et, souvent, un meilleur produit final. Même si le résultat est le même, comme le suggère la figure, il devrait être atteint plus tôt (bien que le dessin ne mentionne pas le temps). Suivre le chemin suggéré ne rend peut-être pas plus facile la décision de lancer des versions préliminaires — les points indiqués par des carrés gris dans le dessin — mais la rend du moins plus simple : il vous suffira d'estimer si ce que vous avez déjà correspond à une part suffisante de l'ensemble des fonctionnalités, permettant d'attirer un consommateur potentiel plutôt que de le faire fuir. La question "est-ce suffisamment bon" (sous-entendu, "est-ce que cela ne va pas se planter") ne devrait pas être un facteur.

Comme le sait n'importe quel lecteur ayant élaboré un projet logiciel, il est plus facile d'énoncer un tel conseil que de le suivre. Mais chaque projet devrait s'efforcer d'appliquer l'approche représentée par la meilleure des deux courbes d'Osmond. Cela s'intègre bien avec le *modèle de groupe*, présenté dans un prochain chapitre comme un modèle général de développement orienté objet rigoureux.

Ponctualité

Définition : ponctualité

La ponctualité est la capacité d'un système logiciel à être livré au moment désiré par ses utilisateurs, ou avant.

La ponctualité est source d'une des plus grandes frustrations de notre industrie. Un superbe produit logiciel qui arrive trop tard peut complètement rater sa cible. C'est également vrai dans d'autres industries, mais peu évoluent aussi vite que celle du logiciel.

"NT 4.0 Beats Clock", Computer-World, vol. 30, n° 30, 22 juillet 1996.

La ponctualité est encore, pour les grands projets, un phénomène peu fréquent. Quand Microsoft a annoncé que la dernière version de son principal système d'exploitation serait livrée, après plusieurs années de travail, avec un mois d'avance, l'événement a suscité un tel intérêt qu'il a fait la une de *ComputerWorld* dans un article évoquant les retards importants de projets plus anciens.

Autres qualités

D'autres qualités, en plus de celles évoquées jusqu'à présent, affectent les utilisateurs de systèmes logiciels, ainsi que ceux qui achètent ces systèmes ou commandent leur développement. En particulier :

- La **vérifiabilité** est la facilité à préparer les procédures de recette, en particulier les jeux de tests, ainsi que les procédures permettant de détecter les erreurs et de les faire remonter, lors des phases de validation et d'opération, aux défauts dont elles proviennent.
- L'**intégrité** est la capacité que présentent certains systèmes logiciels à protéger leurs divers composants (programmes, données) contre les accès et modifications non autorisés.
- La **réparabilité** est la capacité à faciliter la réparation des défauts.
- L'**économie**, cousine de la ponctualité, est la capacité d'un système à être terminé dans les limites de son budget, ou en deçà.

À propos de la documentation

Dans une liste des facteurs de qualité du logiciel, on peut s'attendre à trouver l'exigence d'une bonne documentation. Mais ce n'est pas un facteur de qualité isolé ; le besoin de documentation est plutôt une conséquence des autres facteurs de qualité décrits ci-dessus. Nous pouvons distinguer trois sortes de documentation :

- La documentation *externe*, qui permet aux utilisateurs de maîtriser la puissance du système et de l'utiliser correctement ; c'est une conséquence de la définition de la facilité d'utilisation.
- La documentation *interne*, qui permet aux développeurs logiciels de comprendre la structure et l'implémentation du système ; c'est une conséquence de l'exigence d'extensibilité.
- La documentation d'*interface de modules*, qui permet aux développeurs logiciels de comprendre les fonctionnalités offertes par un module sans avoir à en comprendre l'implémentation ; c'est une conséquence de l'exigence de réutilisabilité. Elle se déduit aussi de l'extensibilité, puisque la documentation d'interface de modules permet de déterminer si un changement donné doit avoir un impact sur un module précis.

Plutôt que de traiter la documentation comme un produit séparé du logiciel proprement dit, il est préférable de rendre le logiciel aussi autodocumenté que possible. Ceci s'applique aux trois types de documentation :

- En incluant des mécanismes d'"aide" en ligne et en suivant des conventions claires et cohérentes d'interface utilisateur, vous simplifiez la tâche des auteurs de manuels utilisateur et autres formes de documentations externes.
- Un bon langage d'implémentation éliminera une grande partie du besoin de documentation interne s'il favorise une expression claire et structurée. Cela sera l'une des exigences principales de la notation orientée objet développée tout le long de ce livre.
- La notation introduira la rétention d'information, ainsi que d'autres techniques (comme les assertions), en vue de séparer l'interface des modules de leur implémentation. Il est alors possible d'utiliser des outils pour produire automatiquement la documentation d'interface de

modules à partir du texte des modules. Cet outil constitue l'un des sujets abordés en détail dans les chapitres suivants.

Toutes ces techniques diminuent le rôle de la documentation traditionnelle, quoique nous ne puissions évidemment pas espérer nous en passer complètement.

Compromis

Dans ce survol des facteurs externes de qualité du logiciel, nous avons rencontré certaines exigences qui sont incompatibles avec d'autres.

Comment peut-on obtenir l'*intégrité* sans introduire des protections diverses, ce qui limitera inévitablement la *facilité d'utilisation* ? L'*économie* paraît souvent s'opposer à la *fonctionnalité*. L'*efficacité* optimale nécessiterait une adaptation parfaite à un environnement matériel et logiciel donné, ce qui est opposé à la *portabilité*, et à une spécification, alors que la *réutilisabilité* incite à résoudre des problèmes plus généraux que celui donné initialement. Les impératifs de *ponctualité* peuvent nous faire pencher vers l'utilisation de techniques de "développement rapide d'applications" (RAD) dont les résultats risquent de ne pas permettre beaucoup d'*extensibilité*.

Bien qu'il soit possible, dans de nombreux cas, de trouver une solution conciliant apparemment ces facteurs contradictoires, vous serez parfois amené à faire des compromis. Trop souvent, les développeurs font ces compromis de manière implicite, sans prendre le temps de considérer les divers aspects et choix disponibles ; l'efficacité tend à être le facteur dominant dans ces décisions impromptues. L'approche basée sur le génie logiciel imposera un effort d'explicitation de ces critères avant qu'un choix ne soit fait en connaissance de cause.

Aussi nécessaires que soient les compromis entre facteurs de qualité, un facteur reste prééminent : la correction. On ne peut justifier de compromettre la correction au profit d'autres intérêts comme l'efficacité. Si le logiciel ne remplit pas sa fonction, le reste est sans utilité.

Préoccupations essentielles

Toutes les qualités évoquées ci-dessus sont importantes. Mais, dans l'état actuel de l'industrie du logiciel, quatre d'entre elles sont primordiales :

- *correction* et *robustesse* : il est encore trop difficile de produire un logiciel sans défauts (bogues) et trop pénible de corriger ces défauts, une fois ceux-ci introduits. Les techniques permettant d'améliorer la correction et la robustesse sont proches : approches plus systématiques de la construction du logiciel ; spécifications plus formelles ; vérifications introduites tout au long du processus de construction du logiciel (et pas seulement lors du test et du débogage après coup) ; meilleurs mécanismes de langage comme le typage statique, les assertions, la gestion automatique de la mémoire et le traitement rigoureux des exceptions, qui permettent aux développeurs d'exprimer des exigences de correction et de robustesse et d'utiliser des outils de détection des incohérences avant que celles-ci n'entraînent des défauts. Du fait de la proximité des concepts de correction et de robustesse, il est commode d'utiliser un mot unique plus général, la **fiabilité**, pour se référer à ces deux facteurs.

- *extensibilité* et *réutilisabilité* : le logiciel devrait être plus facile à modifier ; les éléments logiciels que nous produisons devraient être applicables de manière plus générale et il devrait exister un répertoire plus vaste de composants d'utilisation générale que nous pourrions réutiliser lors du développement de nouveaux systèmes. Ici également, des idées voisines sont utiles pour améliorer ces deux qualités : toute technique qui facilite la production d'architectures plus décentralisées, dans lesquelles les composants sont auto-suffisants et ne communiquent qu'à travers des canaux restreints et bien définis, sera bénéfique. Le terme de **modularité** couvrira à la fois réutilisabilité et extensibilité.

Comme nous le verrons en détail dans les chapitres suivants, la méthode orientée objet peut significativement améliorer ces quatre facteurs de qualité — ce qui explique pourquoi elle est si attirante. Elle contribue de façon significative à d'autres aspects, en particulier :

- *compatibilité* : la méthode favorise un style unifié de conception, ainsi que des interfaces standardisées entre modules et systèmes, ce qui permet de produire des systèmes qui travailleront de concert.
- *portabilité* : avec l'accent mis sur les concepts d'abstraction et de rétention d'information, la technologie objet encourage les concepteurs à distinguer les propriétés de spécification de celles d'implémentation, diminuant ainsi les efforts de portage. Les techniques de polymorphisme et de liaison dynamique rendront même possible la réalisation de systèmes qui s'adaptent automatiquement aux divers composants de la machine matérielle-logicielle, par exemple à des systèmes de fenêtrage ou à des systèmes de gestion de bases de données différents.
- *facilité d'utilisation* : la contribution des outils OO aux systèmes interactifs modernes et, en particulier, à leurs interfaces utilisateur est bien connue, à tel point qu'elle occulte parfois ses autres aspects (les publicistes ne sont pas les seuls à appeler "orienté objet" tout système qui utilise des icones, des fenêtres et des souris).
- *efficacité* : comme indiqué ci-dessus et bien que la puissance supplémentaire des techniques orientées objet puisse paraître, de prime abord, coûteuse, se fier à des composants réutilisables de qualité professionnelle peut souvent améliorer considérablement les performances.
- *ponctualité*, *économie* et *fonctionnalité* : les techniques OO permettent à ceux qui les maîtrisent de produire du logiciel plus rapidement et à un coût moindre ; elles facilitent l'ajout de fonctions et peuvent elles-mêmes en suggérer de nouvelles fonctions.

Malgré toutes ces améliorations, nous devons garder à l'esprit que la méthode orientée objet n'est pas une panacée et que les questions habituelles du génie logiciel restent en grande partie d'actualité. Aider à préciser un problème n'est pas le résoudre.

1.3 DE LA MAINTENANCE LOGICIELLE

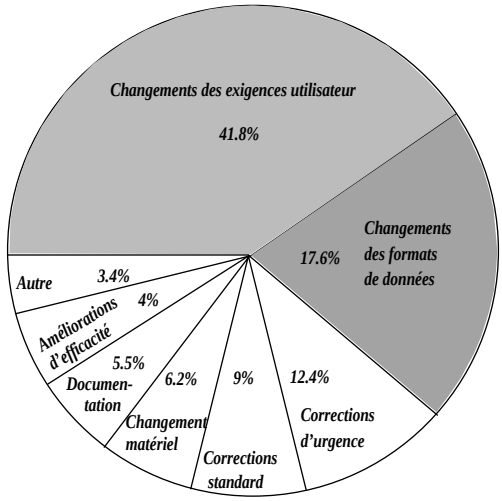
La liste de facteurs ne mentionne pas une qualité souvent citée : la maintenabilité. Pour comprendre pourquoi, nous devons approfondir la notion sous-jacente de maintenance.

La maintenance représente ce qui se passe après qu'un produit logiciel a été diffusé. Les discussions de méthodologie logicielle ont tendance à se focaliser sur la phase de développement ; même chose pour les cours d'introduction à la programmation. Mais on

estime couramment que 70 % du coût du logiciel est dévolu à la maintenance. Aucune étude sur la qualité du logiciel ne peut être complète si elle néglige cet aspect.

Que représente la "maintenance" d'un logiciel ? Une minute de réflexion montre que ce terme n'est pas approprié : un produit logiciel ne s'use pas à force d'usages répétés et ne nécessite donc pas d'être "entretenu" comme doit l'être une voiture ou une télévision. En fait, le mot est utilisé par les gens du logiciel pour désigner tout à la fois des activités nobles et d'autres qui le sont moins. La partie noble représente la modification : quand les spécifications des systèmes informatiques changent, reflétant en cela les modifications du monde extérieur, les systèmes doivent également changer. La partie moins noble est le débogage tardif : éliminer les erreurs qui n'auraient pas dû être là. Le schéma ci-dessous, tiré d'une étude de Lientz et Swanson qui a fait date, illustre ce que recouvre le terme général de maintenance. L'étude a concerné 487 installations qui produisaient du logiciel de toutes sortes ; bien qu'elle soit un peu ancienne, des publications plus récentes ont confirmé l'essentiel des résultats. Elle montre le pourcentage des coûts de maintenance alloués à chacune des activités de maintenance identifiées par les auteurs.

***Répartition des coûts de maintenance. Source* : [Lientz 1980]**



Plus des deux cinquièmes du coût sont dévolus à des modifications ou extensions requises par les utilisateurs. Cela correspond à la partie noble de la maintenance que nous avons décrite ci-dessus, et c'en est aussi la partie inévitable. Il faudrait voir quelle économie pourrait réaliser l'industrie du logiciel si celle-ci construisait ses produits en montrant, dès le début, plus de considération pour l'extensibilité. Nous pouvons légitimement nous attendre, ici, à une aide de la technologie objet.

Pour un autre exemple, voir "Quelle est la longueur de la seconde initiale ?", page 129.

Le second poste, par ordre décroissant de coût, est particulièrement intéressant : l'effet des changements de formats de données. Quand la structure physique des fichiers et autres types de données change, les programmes doivent être adaptés. Par exemple, quand les services postaux américains ont introduit, il y a quelques années, les codes postaux "5 + 4" pour les grandes entreprises (en utilisant neuf chiffres au lieu de cinq), de nombreux programmes qui traitaient des adresses et "savaient" qu'un code postal est formé d'exactement cinq chiffres ont du être réécrits, pour un coût de l'ordre de plusieurs centaines de millions de dollars, selon la presse.

> De nombreux lecteurs auront sans doute reçu les belles brochures présentant un ensemble de conférences — pas un événement unique, mais une séquence de sessions dans plusieurs villes — dédiées au "problème du millénaire" : comment s'y prendre pour mettre à jour la myriade de programmes qui manipulent des dates et dont les auteurs n'ont jamais imaginé un seul instant qu'une date puisse exister au-delà du XX^{e} siècle ? L'effort d'adaptation du code postal est une plaisanterie à côté de ça. Jorge Luis Borges aurait aimé l'idée : étant donné que peu de gens se préoccupent de ce qui se passera le 1er janvier 3000, ceci doit être le sujet le plus minuscule auquel a été consacrée une série de conférences, ou même une seule conférence, dans toute l'histoire de l'humanité, venue et à venir : *un simple chiffre décimal.*

Le problème n'est pas qu'une partie d'un programme connaisse la structure physique des données : c'est inévitable, puisque les données doivent bien finir par être traitées de manière interne. Mais, avec les techniques traditionnelles de conception, cette connaissance est distribuée dans trop de parties du système, ce qui conduit à des changements de programme dont l'importance est difficile à justifier quand la structure physique change — ce qui arrivera forcément. En d'autres termes, si les codes postaux passent de cinq à neuf chiffres ou si la date demande un chiffre de plus, on peut concevoir qu'un programme qui manipule les codes ou les dates doive être adapté ; ce qui n'est pas acceptable, c'est que la connaissance de la longueur exacte de ces données soit placardée à travers tout le programme et que changer cette longueur impose des changements de programmes sans commune mesure avec la taille conceptuelle du changement de la spécification.

La théorie des types abstraits de données fournira la clé de ce problème en permettant aux programmes d'accéder aux données via des propriétés externes plutôt qu'à travers leur implémentation physique.

Le chapitre 6 couvre les types abstraits de données en détail.

Un autre aspect marquant de la distribution des activités est le faible pourcentage (5,5 %) des coûts de documentation. Rappelez-vous qu'il s'agit des coûts des tâches effectuées au moment de la maintenance. Il semble ici — disons plutôt qu'il semblerait, en l'absence de données plus spécifiques — que la documentation d'un projet sera élaborée soit au moment du développement, soit pas du tout. Nous apprendrons à utiliser un style de conception intégrant la documentation dans le logiciel et disposant d'outils spéciaux pour l'extraire.

Les postes suivants de la liste de Lientz et Swanson sont aussi intéressants, quoique moins directement liés au sujet de ce livre. Les réparations de bogues d'urgence (faites en hâte quand un utilisateur prévient qu'un programme ne produit pas les résultats escomptés ou se comporte de manière catastrophique) coûtent plus que les corrections de routine planifiées. Et ce non seulement parce qu'elles doivent être effectuées en conditions de stress, mais aussi parce qu'elles perturbent le processus bien huilé de diffusion des nouvelles versions et peuvent introduire de nouvelles erreurs. Les deux derniers postes ne comptabilisent que des pourcentages faibles :

- L'amélioration de l'efficacité ; il semblerait que, quand un système marche, les chefs de projets et programmeurs sont peu enclins à le perturber dans l'espoir d'améliorer sa performance et préfèrent le laisser tel quel. (Quand on pense au dicton "Faites bien les choses avant de les faire vite", beaucoup de chefs de projet sont probablement déjà contents d'arriver à la première étape.)
- Le "transfert vers un nouvel environnement" correspond également à un pourcentage faible. Une interprétation possible (à nouveau une conjecture, en l'absence de données plus précises) est qu'il y a essentiellement deux sortes de programmes quand on considère la

question de la portabilité, et pas grand-chose en dehors de ces deux extrêmes : certains programmes sont conçus avec un impératif de portabilité et leur portage coûte relativement peu ; les autres sont tellement liés à leur plate-forme d'origine, et seraient tellement difficiles à porter, que les développeurs n'essaient même pas.

1.4 CONCEPTS CLÉS INTRODUITS DANS CE CHAPITRE

- Le but du génie logiciel est de trouver des moyens permettant de construire des logiciels de qualité.
- Plutôt qu'un facteur unique, la qualité dans le logiciel se conçoit d'avantage comme un compromis entre plusieurs objectifs.
- Les facteurs externes, perceptibles aux utilisateurs et clients, devraient être distingués des facteurs internes, perceptibles aux concepteurs et implémenteurs.
- Seuls comptent les facteurs externes, mais ils ne peuvent être obtenus qu'à l'aide des facteurs internes.
- Une liste des facteurs de base de qualité a été présentée. Ceux qui manquent le plus au logiciel d'aujourd'hui, et que la méthode orientée objet vise directement, sont les facteurs de sûreté comme la correction et la robustesse, réunis sous le vocable de fiabilité, et les facteurs qui requièrent des architectures logicielles plus décentralisées : réutilisabilité et extensibilité, connus sous le terme commun de modularité.
- La maintenance du logiciel, qui correspond à une part importante des coûts du logiciel, est pénalisée par les difficultés d'implémentation des changements des produits logiciels et par la dépendance exagérée des programmes envers la structure physique des données qu'ils manipulent.

1.5 NOTES BIBLIOGRAPHIQUES

De nombreux auteurs ont proposé des définitions de la qualité du logiciel. Parmi les premiers articles sur le sujet, deux, en particulier, restent aujourd'hui d'actualité : [Hoare 1972], un article éditorial invité, et [Boehm 1978], résultat d'une des premières études systématiques, par un groupe à TRW.

La distinction entre les facteurs internes et externes a été introduite dans une étude commanditée par l'US Air Force à General Electric en 1977 [McCall 1977]. McCall utilise les termes "facteurs" et "critères" pour ce que nous avons appelé, dans ce chapitre, facteurs externes et internes. La plupart des facteurs introduits dans ce chapitre (mais pas tous) correspondent à ceux de McCall ; l'un de ses facteurs, maintenabilité, a été éliminé car, comme indiqué ci-dessus, il est couvert de manière adéquate par l'extensibilité et la vérifiabilité. L'étude de McCall évoque non seulement les facteurs externes, mais aussi un certain nombre de facteurs internes ("critères"), ainsi que des *métriques*, techniques quantitatives permettant d'estimer le taux de satisfaction des facteurs internes. Avec la technologie objet, pourtant, un certain nombre de ces facteurs internes et métriques sont obsolètes, car trop liés à des pratiques

logicielles anciennes. Étendre cette partie des travaux de McCall aux techniques développées dans ce livre serait un projet utile ; voir la bibliographie et les exercices du chapitre 3.

L'argument concernant l'effet relatif des améliorations des machines sur la complexité des algorithmes est dérivé de [Aho 1974].

Une référence standard en matière de facilité d'utilisation est [Shneiderman 1987], améliorant [Shneiderman 1980] qui était dévolu au sujet plus large de la psychologie logicielle. La page Web du laboratoire de Shneiderman à *http://www.cs.umd.edu/projects/hcil/* contient un grand nombre de références bibliographiques sur ces sujets.

Les courbes d'Osmond sont tirées d'un tutoriel donné par Roger Osmond à TOOLS USA [Osmond 1995]. À noter que la forme donnée dans ce chapitre ne prend pas en compte le temps, permettant ainsi, sur les deux courbes possibles, une vue plus directe du compromis entre la fonctionnalité et les autres qualités, mais n'indiquant pas le risque de retard du projet que présente la courbe en noir. Les courbes originelles d'Osmond sont exprimées en fonction du temps plutôt que de la fonctionnalité.

Le diagramme des coûts de maintenance est dérivé d'une étude de Lientz et Swanson basée sur un questionnaire concernant la maintenance et envoyé à 487 organisations [Lientz 1980]. Voir aussi [Boehm 1979]. Bien que leurs données de base puissent être considérées comme trop spécialisées et obsolètes de nos jours (l'étude était basée sur des applications de gestion de l'information par lot d'environ 23 000 instructions en moyenne, taille importante à l'époque mais qui ne l'est plus de nos jours), les résultats semblent toujours applicables dans leurs grandes lignes. La Software Management Association réalise une revue annuelle de la maintenance ; voir [Dekleva 1992] pour un rapport concernant l'une des ces revues.

Les expressions *programming-in-the-large* et *programming-in-the-small* ont été introduites par [DeRemer 1976].

Pour une discussion générale sur les questions touchant au génie logiciel, voir le livre de Ghezzi, Jazayeri et Mandrioli [Ghezzi 1991]. Un ouvrage sur les langages de programmation par certains de ces auteurs, [Ghezzi 1997], fournit des renseignements supplémentaires sur certaines des questions abordées dans ce livre.