Conception et programmation orientées objet

Bertrand Meyer

Traduit de l'anglais par Pierre Jouvelot

 ISBN : 978-2-212-12270-1

EYROLLES

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.