Hugues Bersini

La programmation ORIENTÉE OBJET

Cours et exercices en UML2, Python, PHP, C#, C++ et Java (y compris Android)

7e édition

Code source des exemples du livre

EYROLLES

La programmation ORIENTÉE OBJET

7e édition

Le manuel indispensable à tout étudiant en informatique (IUT, écoles spécialisées, écoles d'ingénieurs)

Cette septième édition de l'ouvrage *L'orienté objet* décortique l'ensemble des mécanismes de la programmation objet (classes et objets, interactions entre classes, envois de messages, encapsulation, héritage, polymorphisme, interface, multithreading, sauvegarde des objets, programmation distribuée, modélisation...) en les illustrant d'exemples empruntant aux technologies les plus populaires : Java (y compris pour Android) et C#, C++, Python, PHP, UML 2, mais aussi les services web, Corba, les bases de données objet, différentes manières de résoudre la mise en correspondance relationnel/objet dont le langage innovant de requête objet Linq et enfin les design patterns.

Chaque chapitre est introduit par un dialogue vivant, à la manière du maître et de l'élève, et se complète de nombreux exercices en UML 2, Java (y compris Android), Python, PHP, C# et C++.

À qui s'adresse cet ouvrage ?

- Ce livre sera lu avec profit par tous les étudiants de disciplines informatiques liées à l'approche objet (programmation orientée objet, modélisation UML, Java [y compris pour Android], Python, PHP, C#/C++...) et pourra être utilisé par leurs enseignants comme matériel de cours.
- Il est également destiné à tous les développeurs qui souhaitent approfondir leur compréhension des concepts objet sous-jacents au langage qu'ils utilisent.

Sur le site www.editions-eyrolles.com

Le code source des exercices et leurs corrections sont fournis sur le site d'accompagnement www.editions-eyrolles.com/dl/0067399

Membre de l'Académie Royale de Belgique, **Hugues Bersini** enseigne l'informatique et la programmation aux facultés polytechnique et Solvay de l'université Libre de Bruxelles, dont il dirige le laboratoire d'Intelligence Artificielle. Il est l'auteur de très nombreuses publications (systèmes complexes, génie logiciel, sciences cognitives et bioinformatique).

Sommaire

Principes de base : quel objet pour l'informatique ? • Un objet sans classe... n'a pas de classe • Du faire savoir au savoir-faire... du procédural à l'OO • Ici Londres : les objets parlent aux objets • Collaboration entre classes • Méthodes ou messages ? • L'encapsulation des attributs • Les classes et leur jardin secret • Vie et mort des objets • UML • Héritage • Redéfinition des méthodes • Abstraite, cette classe est sans objet • Clonage, comparaison et affectation d'objets • Interfaces • Distribution gratuite d'objets : pour services rendus sur le réseau • Multithreading • Programmation événementielle • Persistance d'objets • Et si on faisait un petit flipper ? • Les graphes • Petite chimie OO amusante • Programmation Java sous Android • Design patterns

www.editions-eyrolles.com

La programmation ORIENTÉE OBJET

DANS LA MÊME COLLECTION

C. DELANNOY. – **S'initier à la programmation et à l'orienté objet.**
Avec des exemples en C, C++, C#, Python, Java et PHP.
N°14011, 2e édition, juillet 2016, 360 pages.

A. TASSO. – **Le livre de Java premier langage.**
N°14384, 11e édition, mars 2016, 598 pages.

C. DELANNOY. – **Programmer en Java.**
N°11889, 9e édition, octobre 2016, 948 pages.

C. SOUTOU. – **Programmer avec MySQL.**
N°14302, 4e édition, octobre 2015, 480 pages.

E. SARRION. – **Programmation avec Node.js, Express.js et MongoDB.**
N°13994, septembre 2014, 586 pages.

S. CONCHON, J-C. FILLIÂTRE – **Apprendre à programmer avec Ocaml.**
N°13678, septembre 2014, 430 pages.

C. DELANNOY. – **Programmer en langage C++.**
N°14008, 8e édition, 2011-2014, 820 pages.

C. DELANNOY. – **Exercices en langage C++.**
N°12201, 3e édition, 2007, 336 pages.

Retrouvez nos bundles (livres papier + e-book) et livres numériques sur
http://izibook.eyrolles.com

Hugues Bersini

La programmation ORIENTÉE OBJET

Cours et exercices en UML2, Python, PHP, C#, C++ et Java (y compris Android)

7e édition

EYROLLES

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

Cet ouvrage est la septième édition du livre de Hugues Bersini et Ivan Wellesz, paru à l'origine sous le titre *L'orienté objet*, puis sous le titre *La programmation orientée objet* lors de la quatrième édition (G12441).

Remerciements à Anne Bougnoux pour sa relecture.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'éditeur ou du Centre Français d'Exploitation du Droit de Copie, 20, rue des Grands-Augustins, 75006 Paris.
© Groupe Eyrolles, 2002, 2004, 2007, 2009, 2011, 2013 et 2017 pour la présente édition
ISBN : 978-2-212-67399-9

Avant-propos

Aux tout débuts de l'informatique, le fonctionnement interne des processeurs décidait de la seule manière efficace de programmer un ordinateur. Alors que l'on acceptait tout programme comme une suite logique d'instructions, il était admis que l'organisation du programme et la nature même de ces instructions ne pouvaient s'éloigner de la façon dont le processeur les exécutait : pour l'essentiel, des modifications de données mémorisées, des glissements de ces données d'un emplacement mémoire à un autre, des opérations d'arithmétique et de logique élémentaire, et de possibles ruptures de séquence ou branchements.

La mise au point d'algorithmes complexes, dépassant les simples opérations mathématiques et les simples opérations de stockage et de récupérations de données, obligea les informaticiens à effectuer un premier saut dans l'abstrait, en inventant un style de langage dit procédural, auquel appartiennent les langages Fortran, Cobol, Basic, Pascal, C... Les codes écrits dans ces langages sont devenus indépendants des instructions élémentaires propres à chaque type de processeur. Grâce à eux, les informaticiens ont pris quelques distances par rapport aux processeurs (en ne travaillant plus directement à partir des adresses mémoire et en évitant la manipulation directe des instructions élémentaires) et ont élaboré une écriture de programmes plus proche de la manière naturelle de poser et de résoudre les problèmes. Il est incontestablement plus simple d'écrire : c = a + b qu'une suite d'instructions telles que `"load a, reg1"`, `"load b, reg2"`, `"add reg3, reg1, reg2"`, `"move c, reg3"`, ayant pourtant la même finalité. Une opération de traduction automatique, dite de compilation, se charge de traduire le programme, écrit au départ avec ces nouveaux langages, dans les instructions élémentaires seules comprises par le processeur.

La montée en abstraction permise par ces langages de programmation présente un double avantage : une facilité d'écriture et de résolution algorithmique, ainsi qu'une indépendance accrue par rapport aux différents types de processeur existant aujourd'hui sur le marché. Le programmeur se trouve libéré des détails d'implémentation machine et peut se concentrer sur la logique du problème et ses voies de résolution.

Plus les problèmes à affronter gagnaient en complexité – comptabilité, jeux automatiques, compréhension et traduction des langues naturelles, aide à la décision, bureautique, conception et enseignement assistés, programmes graphiques, etc. –, plus l'architecture et le fonctionnement des processeurs semblaient contraignants. Il devenait vital d'inventer des mécanismes informatiques simples à mettre en œuvre pour réduire cette complexité et rapprocher encore plus l'écriture de programmes des manières humaines de poser et résoudre les problèmes.

Avec l'intelligence artificielle, l'informatique s'inspira de notre mode cognitif d'organisation des connaissances, comme un ensemble d'objets conceptuels entrant dans un réseau de dépendance et se structurant de manière taxonomique. Avec la systémique ou la bioinformatique, l'informatique nous révéla qu'un ensemble d'agents au fonctionnement élémentaire, mais s'influençant mutuellement, peut produire un comportement émergent d'une surprenante complexité lorsqu'on observe le système dans sa globalité. Dès lors, pour comprendre jusqu'à reproduire ce comportement par le biais informatique, la meilleure approche consiste en une découpe adéquate du système en ses parties et une attention limitée au fonctionnement de chacune d'entre elles.

Tout cela mis ensemble (la nécessaire distanciation par rapport au fonctionnement du processeur, la volonté de rapprocher la programmation du mode cognitif de résolution de problème, les percées de l'intelligence artificielle et de la bio-informatique, le découpage comme voie de simplification des systèmes apparemment complexes) conduisit graduellement à un deuxième type de langages de programmation, fêtant ses 55 ans d'existence (l'antiquité à l'échelle informatique) : les langages orientés objet, tels Simula, Smalltalk, C++, Eiffel, Java, C#, Delphi, Power Builder, Python et bien d'autres...

L'orientation objet (OO) en quelques mots

À la différence de la programmation procédurale, un programme écrit dans un langage objet répartit l'effort de résolution de problèmes sur un ensemble d'objets collaborant par envoi de messages. Chaque objet se décrit par un ensemble d'attributs (partie statique) et de méthodes portant sur ces attributs (partie dynamique). Certains de ces attributs étant l'adresse des objets avec lesquels les premiers coopèrent, il leur est possible de déléguer certaines des tâches à leurs collaborateurs. Le tout s'opère en respectant un principe de distribution des responsabilités on ne peut plus simple, chaque objet s'occupant de ses propres attributs. Lorsqu'un objet exige de s'informer sur les attributs d'un autre ou de les modifier, il charge cet autre de s'acquitter de cette tâche. En effet, chaque objet expose à ses interlocuteurs un mode d'emploi restreint, une carte de visite limitée aux seuls services qu'il est apte à assurer et continuera à rendre dans le temps, malgré de possibles modifications dans la réalisation concrète de ces services.

Cette programmation est fondamentalement distribuée, modularisée et décentralisée. Pour autant qu'elle respecte également des principes de confinement et d'accès limité (dits d'encapsulation, l'objet n'expose qu'une partie restreinte de ses services), cette répartition modulaire a également l'insigne avantage de favoriser la stabilité des développements. En effet, elle restreint au maximum les conséquences de modifications apportées au code au cours du temps : seuls les objets concernés sont modifiés, pas leurs interlocuteurs, même si le comportement de ces derniers dépend indirectement des fonctionnalités affectées.

Ces améliorations, résultant de la prise de conscience des problèmes posés par l'industrie du logiciel (complexité accrue et stabilité dégradée), ont enrichi la syntaxe des langages objet. Un autre mécanisme de modularisation inhérent à l'orienté objet est l'héritage, qui permet à la programmation de refléter l'organisation taxonomique de notre connaissance en une hiérarchie de concepts du plus au moins général. À nouveau, cette organisation modulaire en objets génériques et plus spécialistes est à l'origine d'une simplification de la programmation, d'une économie d'écriture et de la création de

zones de code aux modifications confinées. Tant cet héritage que la répartition des tâches entre les objets autorisent une décomposition plus naturelle des problèmes, une réutilisation facilitée des codes déjà existants (tout module peut se prêter à plusieurs assemblages) et une maintenance facilitée et allégée de ces derniers. L'orientation objet s'impose, non pas comme une panacée universelle, mais comme une évolution naturelle de la programmation procédurale qui facilite l'écriture de programmes, les rendant plus gérables, plus compréhensibles, plus stables et mieux réexploitables.

L'orienté objet inscrit la programmation dans une démarche somme toute très classique pour affronter la complexité de quelque problème qui soit : une découpe naturelle et intuitive en des parties plus simples. A fortiori, cette découpe sera d'autant plus intuitive qu'elle s'inspire de notre manière « cognitive » de découper la réalité qui nous entoure. L'héritage, reflet fidèle de notre organisation cognitive, en est le témoignage le plus éclatant. L'approche procédurale rendait cette découpe moins naturelle, plus « forcée ». Si de nombreux adeptes de la programmation procédurale sont en effet conscients qu'une manière incontournable de simplifier le développement d'un programme complexe est de le découper physiquement, ils souffrent de l'absence d'une prise en compte naturelle et syntaxique de cette découpe dans les langages de programmation utilisés. Dans un programme imposant, l'OO aide à tracer les pointillés que les ciseaux doivent suivre là où il semble le plus naturel de les tracer : au niveau du cou, des épaules ou de la ceinture, et non pas au niveau des sourcils, des biceps ou des mollets. De surcroît, cette pratique de la programmation incite à cette découpe suivant deux dimensions orthogonales : horizontalement, les classes se déléguant mutuellement un ensemble de services, verticalement, les classes héritant entre elles d'attributs et de méthodes installés à différents niveaux d'une hiérarchie taxonomique. Pour chacune de ces dimensions, reproduisant fidèlement nos mécanismes cognitifs de conceptualisation, en plus de simplifier l'écriture des codes, il est important de faciliter la récupération de ces parties dans de nouveaux contextes et d'assurer la robustesse de ces parties aux changements survenus dans d'autres. Un code OO, idéalement, sera aussi simple à créer qu'à maintenir, récupérer et faire évoluer.

Il n'est pas pertinent d'opposer le procédural à l'OO car, in fine, toute programmation des méthodes (c'est-à-dire la partie active des classes et des objets) reste totalement tributaire des mécanismes procéduraux. On y rencontre des variables, des arguments, des boucles, des fonctions et leurs paramètres, des instructions conditionnelles, tout ce que l'on trouve classiquement dans les boîtes à outils procédurales. L'OO vient plutôt compléter le procédural, en lui superposant un système de découpe plus naturel et facile à mettre en œuvre. Pour preuve, les langages procéduraux comme le C, Cobol ou, plus récemment, PHP, se sont relativement aisément enrichis d'une couche dite OO sans que cette addition ne remette sérieusement en question l'existant.

Cependant, l'effet de cette couche additionnelle ne se limite pas à quelques structures de données supplémentaires afin de mieux organiser les informations manipulées par le programme. Il va bien au-delà. C'est toute une manière de concevoir un programme et la répartition de ses parties fonctionnelles qui est en jeu. Les fonctions et les données ne sont plus d'un seul tenant mais éclatées en un ensemble de modules reprenant chacun à son compte une sous-partie de ces données et les seules fonctions qui les manipulent. Il faut réapprendre à programmer en s'essayant au développement d'une succession de micro-programmes et au couplage soigné et réduit au minimum de ces micro-programmes.

En découpant 1 000 lignes de code en 10 modules de 100 lignes, le gain est bien plus que linéaire, car il est extraordinairement plus simple de programmer 100 lignes plutôt que 1 000. En substance, la programmation OO pourrait reprendre à son compte ce slogan altermondialiste : « agir localement, penser globalement ».

Se pose alors la question de tactique didactique, très controversée dans l'enseignement de l'informatique aujourd'hui, sur l'ordre dans lequel enseigner procédural et OO. De nombreux enseignants, soutenus en cela par de très nombreux manuels, considèrent qu'il faut d'abord passer par un enseignement intensif et une maîtrise parfaite du procédural, avant de faire le grand saut vers l'OO. Mais vingt-cinq années d'enseignement de la programmation à des étudiants de tous âges et de toutes conditions (issus des sciences humaines ou exactes) nous ont convaincus qu'il n'y a aucun ordre à donner. De même qu'historiquement, l'OO est né quasiment en même temps que le procédural et en complément de celui-ci, l'OO doit s'enseigner conjointement et en complément du procédural. Il faut enseigner les instructions de contrôle en même temps que la découpe en classes. L'enseignement de la programmation doit mélanger à loisir la perception « micro » des mécanismes procéduraux à la vision « macro » de la découpe en objets. Aujourd'hui, tout projet informatique de dimension conséquente débute par une analyse des différentes classes qui le constituent. Il faut aborder l'enseignement de la programmation tout comme débute la prise en charge de ce type de projet, en enseignant au plus vite la manière dont ces classes et les objets qui en résultent opèrent à l'intérieur d'un programme.

Ces dernières années, compétition oblige, l'orienté objet s'est trouvé à l'origine d'une explosion de technologies différentes, mais toutes intégrant à leur manière ses mécanismes de base : classes, objets, envois de messages, héritage, encapsulation, polymorphisme... Ainsi sont apparus de nombreux langages de programmation proposant des syntaxes dont les différences sont soit purement cosmétiques, soit plus subtiles. Ils sont autant de variations sur les thèmes créés par leurs trois principaux précurseurs : Simula, Smalltalk et C++.

L'OO a également conduit à repenser trois des chapitres les plus importants de l'informatique de ces deux dernières décennies :

- tout d'abord, le besoin de développer une méthode de modélisation graphique standardisée débouchant sur un niveau d'abstraction encore supplémentaire (on ne programme plus en écrivant du code mais en dessinant un ensemble de diagrammes, le code étant créé automatiquement à partir de ceux-ci ; c'est le rôle joué par UML 2) ;
- ensuite, les applications informatiques distribuées (on ne parlera plus d'applications distribuées mais d'objets distribués, et non plus d'appels distants de procédures mais d'envois de messages à travers le réseau) ;
- enfin, le stockage des données, qui doit maintenant compter avec les objets.

Chaque fois, plus qu'un changement de vocabulaire, un changement de mentalité sinon de culture s'impose.

Les grands acteurs de l'orienté objet

Aujourd'hui, l'OO est omniprésent. Microsoft par exemple, a développé un nouveau langage informatique purement objet (C#). Il a très intensément contribué au développement d'un système d'informatique distribuée, basé sur des envois de messages d'ordinateur à ordinateur (les services web) et a plus récemment proposé un nouveau langage d'interrogation des objets (LINQ), qui s'interface naturellement avec le monde relationnel et le monde XML. Tous les langages informatiques intégrés dans sa nouvelle plate-forme de développement, .Net (aux dernières nouvelles, ils seraient 22), visent à une

uniformisation (y compris les nouvelles versions de Visual Basic et Visual C++) en intégrant les mêmes briques de base de l'OO. Aboutissement considérable s'il en est, il devient très simple de faire communiquer ou hériter entre elles des classes écrites dans des langages différents.

Plusieurs années auparavant, Sun (racheté depuis par Oracle) avait conçu Java, une création déterminante car elle fut à l'origine de ce nouvel engouement pour une manière de programmer qui pourtant existait depuis toujours sans que les informaticiens dans leur ensemble en reconnaissent l'utilité ni la pertinence. Sun a également créé RMI, Jini et sa propre version des services web, tous basés sur les technologies OO. Ces mêmes services web font l'objet de développements tout autant aboutis chez HP ou IBM. À la croisée de Java et du Web (originellement la raison, sinon du développement de Java, du moins de son succès), on découvre une importante panoplie d'outils de développement et de conception de sites web dynamiques. Depuis, Java est devenu le langage de prédilection pour de nombreuses applications d'entreprise et plus récemment pour le développement d'applications tournant sur les smartphones et tablettes dotés du système Android, maintenu par Google.

IBM et Borland, avec Rational et Together, menaient la danse en matière d'outils d'analyse du développement logiciel, avec la mise au point de puissants environnements UML. Chez IBM, la plate-forme logicielle Eclipse est sans doute, à ce jour, une des aventures Open Source les plus abouties en matière d'OO. Comme environnement de développement Java, Eclipse est aujourd'hui le plus prisé et le plus usité et cherche à gagner son pari « d'éclipser » tous les autres. Borland a rendu Together intégrable tant dans Visual Studio.Net que dans Eclipse, comme outil synchronisant au mieux et au plus la programmation et la réalisation des diagrammes UML. De son côté, Apple encourage les développeurs à passer du langage Objective C, langage de prédilection jusqu'à présent dans l'univers Apple, au dernier né, Swift (tous deux langages OO), pour la conception d'applications censées s'exécuter tant sur les ordinateurs que les tablettes ou smartphones proposés par la célèbre marque à la pomme.

Enfin, l'OMG, organisme de standardisation du monde logiciel, n'a pas pour rien la lettre O comme initiale. UML et Corba sont ses premières productions : la version OO de l'analyse logicielle et la version OO de l'informatique distribuée. Cet organisme plaide de plus en plus pour un développement informatique détaché des langages de programmation ainsi que des plates-formes matérielles, par l'utilisation intensive des diagrammes UML. Partant de ces mêmes diagrammes, les codes seraient créés automatiquement dans un langage choisi et en adéquation avec la technologie voulue.

Le pari d'UML est osé et encore très largement controversé, mais l'évolution de l'informatique au cours des ans a toujours confié à des mécanismes automatisés le soin de prendre en charge des détails qui éloignaient le programmeur de sa mission première : penser et résoudre son problème.

Objectifs de l'ouvrage

Toute pratique économe, fiable et élégante de Java, C++, C#, Python, PHP ou UML requiert, pour débuter, une bonne maîtrise des mécanismes de base de l'OO. Et, pour y pourvoir, rien n'est mieux que d'expérimenter les technologies OO dans ces différentes versions, comme un bon conducteur qui se sera frotté à plusieurs types de véhicules, un bon skieur à plusieurs styles de skis et un guitariste à plusieurs modèles de guitares.

Plutôt qu'un voyage en profondeur dans l'un ou l'autre de ces multiples territoires, ce livre vous propose d'explorer plusieurs d'entre eux, mais en tentant à chaque fois de dévoiler ce qu'ils recèlent de commun. Car ce sont ces ressemblances qui constituent les briques fondamentales de l'OO. Nous pensons que la mise en parallèle de C++, Java, C#, Python, PHP et UML est une voie privilégiée pour l'extraction de ces mécanismes de base.

Il nous a paru pour cette raison indispensable de discuter et comparer la façon dont ces cinq langages de programmation gèrent, par exemple, l'occupation mémoire par les objets, leur manière d'implémenter le polymorphisme ou la programmation dite « générique », pour en comprendre in fine toute la problématique et les subtilités indépendamment de l'une ou l'autre implémentation. Ajoutez une couche d'abstraction, ainsi que le permet UML, et cette compréhension ne pourra que s'en trouver renforcée. Chacun de ces cinq langages offre des particularités amenant les praticiens de l'un ou l'autre à le prétendre supérieur aux autres : la puissance du C++, la compatibilité Windows et l'intégration XML de C#, l'anti-Microsoft et le leadership de Java en matière de serveurs d'entreprise, les vertus pédagogiques et l'aspect « scripting » de Python, le succès incontestable de PHP pour la mise en place simplifiée d'une solution web dynamique et la capacité de s'interfacer aisément avec les bases de données. Nous nous désintéresserons ici complètement de ces querelles de clochers, a fortiori car notre projet pédagogique nous conduit bien davantage à nous pencher sur ce qui les réunit plutôt que ce qui les différencie. C'est leur multiplicité qui a présidé à cet ouvrage et qui en fait, nous l'espérons, son originalité. Nous n'allons pas nous en plaindre et défendons en revanche l'idée que le choix définitif de l'un ou l'autre de ces langages dépend davantage d'habitude, d'environnement professionnel ou d'enseignement, de questions sociales et économiques et surtout de la raison concrète de cette utilisation (pédagogie, performance machine, adéquation web ou base de données…).

Quelques amabilités glanées dans *Masterminds of Programming*

Bjarne Stroustrup (créateur du C++) : « *J'avais prédit que s'il voulait percer, Java serait contraint de croître significativement en taille et en complexité. Il l'a fait* ».
Guido van Rossum (créateur de Python) : « *Je dis qu'une ligne de Python, de Ruby, de Perl ou de PHP équivaut à 10 lignes de Java* ».
Tom Love (co-créateur d'Objective-C, le langage OO de prédilection pour le développement des applications Apple et plus récemment IPhone, IPod et autres smartphones) : « *Tant Objective-C que C++ sont nés au départ du langage C. Dans le premier cas, ce fut un petit langage, simple, élégant, net et bien défini ; dans l'autre, ce fut une abomination hyper compliquée et présentant de véritables défauts de conceptions* ».
James Gosling (créateur de Java) : « *Les pointeurs en C++ sont un désastre, une véritable incitation à programmer de manière erronée* » et « *C# a tout pompé sur Java, à l'exception de la sûreté et de la fiabilité par la prise en charge de pointeurs dangereux qui m'apparaissent comme grotesquement stupides* ».
Anders Hejlsberg (créateur de C#) : « *Je ne comprends pas pourquoi Java a choisi de ne pas évoluer. Si vous regardez l'histoire de l'industrie, tout n'est qu'une question d'évolution. À la minute où vous arrêtez d'évoluer, vous signez votre arrêt de mort* ».
James Rumbaugh (un des trois concepteurs d'UML) : « *Je pense qu'utiliser UML comme générateur de code est une idée exécrable. Il n'y a rien de magique au sujet d'UML. Si vous pouvez créer du code à partir des diagrammes, alors il s'agit d'un langage de programmation. Or UML n'est pas du tout conçu comme un langage de programmation* ».
Bertrand Meyer (créateur d'Eiffel et défendant la programmation OO dite par « contrats ») : « *Je ne comprends pas comment l'on peut programmer sans prendre le temps et la responsabilité de se demander ce que les éléments du programme ont la charge de faire. C'est une question à poser à Gosling, Stroustrup, Alan Kay ou Hejlsberg* ».

Il y a quelques années, un livre intitulé *Masterminds of Programming* (O'Reilly, 2009) et compilant un ensemble d'entretiens avec les créateurs des langages de programmation, nous a convaincu du bien-fondé de ce type de démarche comparative. Il apparaît en effet, au vu de la guerre des mots à laquelle se livrent ses créateurs, qu'aucun langage de programmation ne peut vraiment s'appréhender sans partiellement le comparer à d'autres. En fait, tous se positionnent dans une forme de rupture ou de replacement par rapport à ses prédécesseurs.

Enfin, nous souhaitions que cet ouvrage, tout en restant suffisamment détaché de toute technologie, couvre l'essentiel des problèmes posés par la mise en œuvre des objets en informatique : leur stockage sur le disque dur, leur interfaçage avec les bases de données, leur fonctionnement en parallèle et leur communication à travers Internet. Ce faisant nous acceptons de perdre un peu en précision, et il nous apparaît nécessaire de mettre en garde le lecteur. Ce livre n'aborde aucune des technologies en profondeur, mais les approche toutes dans ce qu'elles ont de commun et qui se devrait de survivre pour des siècles et des siècles…

Plan de l'ouvrage

Les 24 chapitres de ce livre peuvent se répartir en cinq grandes parties.

Le **premier chapitre** constitue une partie en soi car il a pour importante mission d'initier aux briques de base de la programmation orientée objet, sans aucun développement technique : une première esquisse, teintée de sciences cognitives, et toute en intuition, des éléments essentiels de la pratique OO.

La **deuxième partie** intègre les quatorze chapitres suivants. Il s'agit pour chacun d'entre eux de décrire, plus techniquement cette fois, ces briques de base : objet, classe (**chapitres 2 et 3**), messages et communication entre objets (**chapitres 4, 5 et 6**), encapsulation (**chapitres 7 et 8**), gestion mémoire des objets (**chapitre 9**), modélisation objet (**chapitre 10**), héritage et polymorphisme (**chapitres 11 et 12**), classe abstraite (**chapitre 13**), clonage et comparaison d'objets (**chapitre 14**), interface (**chapitre 15**).

Chacune de ces briques est illustrée par des exemples en Java, C#, C++, Python, PHP et UML. Nous y faisons le pari que cette mise en parallèle est la voie la plus naturelle pour la compréhension des mécanismes de base : extraction du concept par la multiplication des exemples.

La **troisième partie** reprend, dans le droit fil des ouvrages dédiés à l'un ou l'autre langage objet, des notions jugées plus avancées : les objets distribués, Corba, RMI, les services web (**chapitre 16**), le multithreading ou programmation parallèle (ou concurrentielle, **chapitre 17**), la programmation événementielle (**chapitre 18**) et enfin la sauvegarde des objets sur le disque dur, y compris l'interfaçage entre les objets et les bases de données relationnelles (**chapitre 19**). Là encore, le lecteur se trouvera le plus souvent en présence de plusieurs versions dans les cinq langages de ces mécanismes.

La **quatrième partie** décrit plusieurs projets de programmation objet totalement aboutis, tant en UML qu'en Java, C# ou Python. Elle inclut d'abord le **chapitre 20**, décrivant la modélisation objet d'un petit flipper et d'un petit tennis, ainsi que les problèmes de conception orientée objet que cette modélisation pose. Le **chapitre 21**, lié au **chapitre 22**, décrit la manière dont les objets peuvent s'organiser en liste liée ou en graphe, mode de mise en relation et de regroupement des objets

que l'on retrouve abondamment dans toute l'informatique. Le **chapitre 22** contient la programmation d'un réacteur chimique créant de nouvelles molécules à partir de molécules de base, tout en suivant à la trace l'évolution de la concentration des molécules dans le temps. La chimie – une chimie élémentaire acquise bien avant l'université – nous est apparue être une plate-forme pédagogique idéale pour l'assimilation des concepts objets. Le **chapitre 23** est la véritable innovation de la septième version de l'ouvrage. Il généralise, pour l'environnement de développement Android, certains des programmes animés Java présentés dans le chapitre 20. Ceci donnera la possibilité au lecteur d'expérimenter ces codes Java tant sur son ordinateur que sur son smartphone Android.

Enfin, la **dernière partie** se ramène au seul **chapitre 24**, dans lequel sont présentées plusieurs recettes de conception OO, résolvant de manière fort élégante un ensemble de problèmes récurrents dans la réalisation de programmes. Ces recettes de conception, dénommées *Design patterns*, sont devenues fort célèbres dans la communauté OO. Leur compréhension s'inscrit dans la suite logique de l'enseignement des briques et des mécanismes de base de l'OO. Elle fait souvent la différence entre l'apprenti et le compagnon parmi les programmeurs OO.

À qui s'adresse ce livre ?

Cet ouvrage est sans nul doute destiné à un public assez large : industriels, enseignants et étudiants, mais sa vocation première n'en reste pas moins une initiation à la programmation orientée objet.

Ce livre sera un compagnon d'étude enrichissant pour les étudiants qui comptent la programmation objet dans leur cursus d'étude (et toutes technologies s'y rapportant : Java, Android, C++, C#, Python, PHP, Corba, RMI, services web, UML, LINQ). Il devrait les aider, le cas échéant, à évoluer de la programmation procédurale à la programmation objet, pour aller ensuite vers toutes les technologies s'y rapportant.

Nous ne pensons pas, en revanche, que ce livre peut seul prétendre à constituer une porte d'entrée dans le monde de la programmation tout court. Comme dit précédemment, nous estimons qu'il est idéal d'aborder en même temps les mécanismes OO et procéduraux. Pour des raisons évidentes de place et car les librairies informatiques en regorgent déjà, nous avons fait l'impasse d'un enseignement de base des mécanismes procéduraux : variables, boucles, instructions conditionnelles, éléments fondamentaux et compagnons indispensables à l'assimilation de l'OO. Nous pensons, dès lors, que ce livre sera plus facile à aborder pour des lecteurs ayant déjà un peu de pratique de la programmation dite procédurale, et ce, dans un quelconque langage de programmation. Finalement, précisons que s'il ne pretend pas être exhaustif – et bien qu'à sa 7e édition – il résiste assez bien au temps. Ses pages ne jaunissent pas trop et, tout comme la plupart des technologies qu'il recense, il reste rétrocompatible avec ses versions précédentes. Il suit les évolutions de toutes ces technologies, même si celles-ci restent délibérément accrochées aux principes OO qui en font son sujet et, nous l'espérons, son attrait principal.

Table des matières

CHAPITRE 1

Principes de base : quel objet pour l'informatique ? 1

Le trio <entité, attribut, valeur> 2
Stockage des objets en mémoire 2
Types primitifs 4
Le référent d'un objet 5
Plusieurs référents pour un même objet 6
L'objet dans sa version passive 7
L'objet et ses constituants 7
Objet composite 8
Dépendance sans composition 9
L'objet dans sa version active 9
Activité des objets 9
Les différents états d'un objet 9
Les changements d'état : qui en est la cause ? 10
Comment relier les opérations et les attributs ? 10
Introduction à la notion de classe 11
Méthodes et classes 11
Sur quel objet précis s'exécute la méthode ? 13
Des objets en interaction 14
Comment les objets communiquent 14
Envoi de messages 15
Identification des destinataires de message . 15
Des objets soumis à une hiérarchie 16
Du plus général au plus spécifique 16
Dépendance contextuelle du bon niveau taxonomique 17
Polymorphisme 18
Héritage bien reçu 18
Exercices 19

CHAPITRE 2

Un objet sans classe… n'a pas de classe 21

Constitution d'une classe d'objets 22
Définition d'une méthode de la classe : avec ou sans retour 23
Identification et surcharge des méthodes par leur signature 24
La classe comme module fonctionnel 26
Différenciation des objets par la valeur des attributs 26
Le constructeur 26
Mémoire dynamique, mémoire statique . . . 28
La classe comme garante de son bon usage . 29
La classe comme module opérationnel 30
Mémoire de la classe et mémoire des objets 30
Méthodes de la classe et des instances 32
Un premier petit programme complet dans les cinq langages 32
En Java 33
EN C# 35
En C++ 37
En Python 40
En PHP 42
La classe et la logistique de développement . 45
Classes et développement de sous-ensembles logiciels 45
Classes, fichiers et répertoires 45
Exercices 46

CHAPITRE 3
Du faire savoir au savoir-faire… du procédural à l'OO 51
Objectif objet : les aventures de l'OO 52
Argumentation pour l'objet 52
Transition vers l'objet 52
Mise en pratique 53
Simulation d'un écosystème 53
Analyse 53
Analyse procédurale 53
Fonctions principales 54
Conception 55
Conception procédurale 55
Conception objet 55
Conséquences de l'orientation objet 56
Les acteurs du scénario 56
Indépendance de développement et dépendance fonctionnelle 57
Petite allusion (anticipée) à l'héritage 58
La collaboration des classes deux à deux .. 58

CHAPITRE 4
Ici Londres : les objets parlent aux objets.............................. 59
Envois de messages 60
Association de classes 61
Dépendance de classes 63
Réaction en chaîne de messages 66
Exercices 67

CHAPITRE 5
Collaboration entre classes .. 69
Pour en finir avec la lutte des classes 70
La compilation Java : effet domino 70
En C#, Python, PHP ou C++ 71
De l'association unidirectionnelle à l'association bidirectionnelle 73
Auto-association 76
Paquets et espaces de noms 78
Exercices 81

CHAPITRE 6
Méthodes ou messages ?...... 83
Passage d'arguments prédéfinis dans les messages 84
En Java 84
Résultat 84
En C# 85
Résultat 86
En C++ 86
Résultat 87
En Python 87
Résultat 88
En PHP 88
Passage d'argument objet dans les messages 89
En Java 89
Résultat 90
En C# 91
Résultat 92
En PHP 93
En C++ 94
Résultat passage par valeur 95
Résultat passage par référent 95
En Python 96
Résultat 96
Une méthode est-elle d'office un message ? . 97
Même message, plusieurs méthodes 97
Nature et rôle, type, classe et interface : quelques préalables 97
Interface : liste de signatures de méthodes disponibles 98
Des méthodes strictement internes 99
La mondialisation des messages 100
Message sur Internet 100
L'informatique distribuée 101
Exercices 101

CHAPITRE 7
L'encapsulation des attributs 107

Accès aux attributs d'un objet 108
Accès externe aux attributs 108
Cachez ces attributs que je ne saurais voir 108
Encapsulation des attributs 109
En Java 109
En C++ 110
En C# 111
En PHP 112
En Python 113
Encapsulation : pour quoi faire ? 114
Pour préserver l'intégrité des objets 114
La gestion d'exception 115
Pour cloisonner leur traitement 117
Pour pouvoir faire évoluer leur traitement en douceur 117
La classe : enceinte de confinement 119
Exercices 119

CHAPITRE 8
Les classes et leur jardin secret 121
Encapsulation des méthodes 122
Interface et implémentation 122
Toujours un souci de stabilité 123
Signature d'une classe : son interface 125
Les niveaux intermédiaires d'encapsulation 126
Classes amies 126
Une classe dans une autre 127
Utilisation des paquets 128
Exercices 130

CHAPITRE 9
Vie et mort des objets......... 131
Question de mémoire 132
Un rappel sur la mémoire RAM 132
L'OO coûte cher en mémoire 133
Qui se ressemble s'assemble : le principe de localité 133
Les objets intermédiaires 134
Mémoire pile 134
En C++ 136
En C# 137
Disparaître de la mémoire comme de la vie réelle 138
Mémoire tas 139
C++ : le programmeur est le seul maître à bord 139
La mémoire a des fuites 141
En Java, C#, Python et PHP : la chasse au gaspi 143
En Java 143
En C# 144
En PHP 145
Le ramasse-miettes (ou garbage collector) 145
Des objets qui se mordent la queue 146
En Python 147
Exercices 150

CHAPITRE 10
UML....................................... 155
Diagrammes UML 156
Représentation graphique standardisée . . 156
Du tableau noir à l'ordinateur 157
Programmer par cycles courts en superposant les diagrammes 159
Diagramme de classe et diagramme de séquence 160
Diagramme de classe 161
Une classe 161
En Java : UML1.java 161
En C# : UML1.cs 162
En C++ : UML1.cpp 162
En Python : UML1.py 163
En PHP : UML1.php 164
Similitudes et différences entre les langages 164
Association entre classes 165
En Java : UML 2.java 165
En C# : UML 2.cs 166

En C++ : UML 2.cpp 167
En Python : UML 2.py 168
En PHP : UML2.php 169
Similitudes et différences entre les langages 169
Pas d'association sans message 170
Rôles et cardinalité 171
Dépendance entre classes 179
Composition 180
En Java 181
UML3.java 182
UML3bis.java 184
En C# 185
UML3.cs 186
UML3bis.cs 187
En C++ 188
UML3.cpp 189
UML3bis.cpp 191
En Python 192
UML3.py 192
UML3bis.py 193
En PHP 194
Classe d'association 195
Les paquets 196
Les bienfaits d'UML 197
Un premier diagramme de classe de l'écosystème 197
Des joueurs de football qui font leurs classes 197
Les avantages des diagrammes de classes . 197
Un diagramme de classe simple à faire, mais qui décrit une réalité complexe à exécuter . 199
Procéder de manière modulaire et incrémentale 200
Diagramme de séquence 200
Diagramme d'états-transitions 206
Exercices 210

CHAPITRE 11
Héritage 217
Comment regrouper les classes dans des superclasses ? 218
Héritage des attributs 219
Pourquoi l'addition de propriétés ? 222
L'héritage : du cognitif aux taxonomies . . 222
Interprétation ensembliste de l'héritage . . 223
Qui peut le plus peut le moins 224
Héritage ou composition ? 224
Économiser en ajoutant des classes ? 225
Héritage des méthodes 226
Code Java 229
Code C# 230
Code C++ 231
Code Python 233
Code PHP 234
La recherche des méthodes dans la hiérarchie 235
Encapsulation protected 236
Héritage et constructeurs 237
Premier code Java 237
Deuxième code Java 238
Troisième code Java : le plus logique et le bon 239
En C# 240
En C++ 241
En Python 242
En PHP 242
Héritage public en C++ 243
Le multihéritage 244
Ramifications descendantes et ascendantes 244
Multihéritage en C++ et Python 245
Code C++ illustrant le multihéritage 246
Code Python illustrant le multihéritage . . . 247
Des méthodes et attributs portant un même nom dans des superclasses distinctes 248
Code C++ illustrant un premier problème lié au multihéritage 248
En Python 250

Plusieurs chemins vers une même superclasse 251
Code C++ : illustrant un deuxième problème lié au multihéritage 252
L'héritage virtuel 254
Exercices 255

CHAPITRE 12

Redéfinition des méthodes 261

La redéfinition des méthodes 262
Beaucoup de verbiage mais peu d'actes véritables 263
Un match de football polymorphique . . . 264
La classe Balle 265
En Java 265
En C++ 266
En C# 266
En Python 266
En PHP 266
La classe Joueur 266
En Java 266
En C++ 267
En C# 268
En Python 269
En PHP 270
Précisons la nature des joueurs 270
En Java 271
En C++ 273
En C# 274
En Python 275
En PHP 275
Passons à l'entraîneur 276
En Java 276
En C++ 277
En C# 278
En Python 278
En PHP 278
Passons maintenant au bouquet final . . . 279
En Java 279
Un même ordre mais une exécution différente 280
C++ : un comportement surprenant 282
Polymorphisme : uniquement possible dans la mémoire tas 285
En C# 286
En Python 288
En PHP 289
Quand la sous-classe doit se démarquer pour marquer 289
Les attaquants participent à un casting . . . 290
Éviter les « mauvais castings » 291
En C++ 292
En C# 293
Le casting a mauvaise presse 293
Redéfinition et encapsulation 295
Exercices 296

CHAPITRE 13

Abstraite, cette classe est sans objet....................... 307

De Canaletto à Turner 308
Des classes sans objet 308
Du principe de l'abstraction à l'abstraction syntaxique 309
Classe abstraite 311
new et abstract incompatibles 312
Abstraite de père en fils 312
Un petit exemple dans quatre langages de programmation 313
En Java 313
En C# 314
En PHP 315
En C++ 316
L'abstraction en Python 317
Un petit supplément de polymorphisme . . 318
Les enfants de la balle 318
Cliquez frénétiquement 318
Le Paris-Dakar 320
Le polymorphisme en UML 320
Exercices 322
Exercice 13.11 330
Exercice 13.12 331

CHAPITRE 14
Clonage, comparaison et affectation d'objets 333

Introduction à la classe Object 334
Une classe à compétence universelle 334
Code Java illustrant l'utilisation de la classe Vector et innovation de Java 5 334
Nouvelle version du code 335
Décortiquons la classe Object 336
Test d'égalité de deux objets 338
Code Java pour expérimenter la méthode equals(Object o) 338
Égalité en profondeur 341
Le clonage d'objets 343
Code Java pour expérimenter la méthode clone() 343
Égalité et clonage d'objets en Python 347
Code Python pour expérimenter l'égalité et le clonage 347
Égalité et clonage d'objets en PHP 348
Code PHP pour expérimenter l'égalité et le clonage 348
Égalité, clonage et affectation d'objets en C++ 350
Code C++ illustrant la duplication, la comparaison et l'affectation d'objets ... 350
Traitons d'abord la mémoire tas 354
Surcharge de l'opérateur d'affectation ... 356
Comparaisons d'objets 356
La mémoire pile 357
Surcharge de l'opérateur de comparaison . 357
Dernière étape 358
Code C++ de la classe O1 créé automatiquement par Rational Rose 359
En C#, un cocktail de Java et de C++ 361
Pour les structures 361
Pour les classes 361
Code C# 361
Exercices 367

CHAPITRE 15
Interfaces 369

Interfaces : favoriser la décomposition et la stabilité 370
Java, C# et PHP : interface et héritage ... 370
Les trois raisons d'être des interfaces 372
Forcer la redéfinition 372
Code Java illustrant l'interface Comparable 373
Code Java illustrant l'interface ActionListener 374
Code Java illustrant l'interface KeyListener 376
Permettre le multihéritage 378
La carte de visite de l'objet 378
Code Java 379
Code C# 381
Code PHP 384
Les interfaces dans UML 2 385
En C++ : fichiers .h et fichiers .cpp 387
Interfaces : du local à Internet 390
Exercices 391

CHAPITRE 16
Distribution gratuite d'objets : pour services rendus sur le réseau....................... 395

Objets distribués sur le réseau : pourquoi ? 396
Faire d'Internet un ordinateur géant 396
Répartition des données 397
Répartition des utilisateurs et des responsables 397
Peer-to-peer 398
L'informatique ubiquitaire 399
Robustesse 400
RMI (Remote Method Invocation) 400
Côté serveur 401
Côté client 402
RMIC : stub et skeleton 404
Lancement du registre 405
Corba (Common Object Request Broker Architecture) 406

Un standard : ça compte 407
IDL 407
Compilateur IDL vers Java 408
Côté client 409
Côté serveur 411
Exécutons l'application Corba 412
Corba n'est pas polymorphique 413
Ajoutons un peu de flexibilité à tout cela . 414
Corba : invocation dynamique versus invocation statique 415
Jini 415
XML : pour une dénomination universelle des services 416
Les services web sur .Net 418
Code C# du service 418
WDSL 420
Création du proxy 420
Code C# du client 421
Soap (Simple Object Access Protocol) . . 422
Invocation dynamique sous .Net 422
Invocation asynchrone en .Net 424
Mais où sont passés les objets ? 426
Un annuaire des services XML universel : UDDI 428
Services web versus RMI et Corba 429
Services web versus Windows Communication Foundation (WCF) . . . 429
Exercices 430

CHAPITRE 17
Multithreading 433
Informatique séquentielle 435
Multithreading 436
Implémentation en Java 437
Implémentation en C# 440
Implémentation en Python 442
L'effet du multithreading sur les diagrammes de séquence UML . . . 443
Du multithreading aux applications distribuées 444
Des threads équirépartis 445
En Java 445
En C# 446
En Python 446
Synchroniser les threads 447
En Java 448
En C# 449
En Python 452
Exercices 455

CHAPITRE 18
Programmation événementielle.................... 459
Des objets qui s'observent 460
En Java 461
La plante 461
Du côté du prédateur 462
Du côté de la proie 463
Finalement, du côté de la Jungle 463
Résultat 464
En C# : les délégués 464
Généralités sur les délégués dans .Net . . . 464
Retour aux observateurs et observables . . . 468
Tout d'abord, la plante 468
Du côté du prédateur 469
Du côté de la proie 470
Finalement, du côté de la Jungle 470
En Python : tout reste à faire 473
Un feu de signalisation plus réaliste 475
En Java 476
Exercices 477

CHAPITRE 19
Persistance d'objets 479
Sauvegarder l'état entre deux exécutions . . 480
Et que dure le disque dur 480
Quatre manières d'assurer la persistance des objets 480
Simple sauvegarde sur fichier 481
Utilisation des streams ou flux 481

Qui sauve quoi ? 482
En Java 482
En C# 484
En C++ 485
En Python 486
En PHP 487
Sauvegarder les objets sans les dénaturer : la sérialisation 488
En Java 489
En C# 491
En Python 492
Contenu des fichiers de sérialisation : illisible 493
Les bases de données relationnelles 493
SQL 494
Une table, une classe 494
Comment interfacer Java et C# aux bases de données 496
En Java 497
En C# 499
Relations entre tables et associations entre classes 501
Relation 1-n 502
Relation n-n 504
Dernier problème : l'héritage 505
Réservation de places de spectacles 506
Les bases de données relationnelles-objet 511
SQL3 512
Les bases de données orientées objet 514
OQL 515
Django et Python 516
LINQ 518
Premier exemple de LINQ agissant sur une collection d'objets 519
Second exemple de LINQ agissant sur une base de données relationnelle 521
Exercices 524

CHAPITRE 20
Et si on faisait un petit flipper ? 527
Généralités sur le flipper et les GUI 528
Une petite animation en C# 533
Une version simplifiée de cette même animation en Python 539
Retour au Flipper 540
Code Java du Flipper 543
Un petit tennis 553

CHAPITRE 21
Les graphes 561
Le monde regorge de réseaux 562
Tout d'abord : juste un ensemble d'objets . 563
Liste liée 565
En Java 567
En C++ 569
La généricité en C++ 571
La généricité en Java et C# 575
Passons aux graphes 581
Exercices 586

CHAPITRE 22
Petite chimie OO amusante 591
Pourquoi de la chimie OO ? 592
Chimie computationnelle 592
Chimie comme plate-forme didactique . . 592
Une aide à la modélisation chimique 592
Les diagrammes de classes du réacteur chimique 593
La classe Composant_Chimique 593
Les classes Composant_Neutre et Composant_Charge 594
Les trois sous-classes de composants neutres . 595
Les trois sous-classes de composants chargés . 597
La classe NœudAtomique 598
La classe NœudMoléculaire 599
La classe Liaison 599
Le graphe moléculaire 599

Les règles de canonisation 601
Les réactions chimiques 602
Une première réaction de croisement 603
Une autre réaction de croisement un peu plus sophistiquée 603
Une réaction d'ouverture de liaison 603
Réaction de transfert de charge 604
Réaction de type Ion-Molecule 604
Comment est calculée la cinétique réactionnelle 604
La classe Reaction 605
Les sous-classes de Reaction 606
Quelques résultats du simulateur 606
Première simulation : une seule molécule produite 607
Deuxième simulation : plusieurs molécules produites 607
Troisième simulation : De nombreuses molécules complexes produites 608
Pourquoi un tel simulateur ? 609
La simulation immunologique en OO ? . . 609
Petite introduction au fonctionnement du système immunitaire 609
Le diagramme UML d'états-transitions . 611

CHAPITRE 23
Programmation Java sous Android **615**
Un petit calculateur élémentaire 617
Animation avec balles colorées 620
Jeu Canon 628

CHAPITRE 24
Design patterns **645**
Introduction aux design patterns 646
De l'architecte à l'archiprogrammeur 646
Les patterns « trucs et ficelles » 647
Le pattern Singleton 648
Le pattern Adaptateur 649
Les patterns Patron de méthode, Proxy, Observer et Memento 650
Le pattern Flyweight 651
Les patterns Builder et Prototype 652
Le pattern Façade 654
Les patterns qui se jettent à l'OO 655
Le pattern Command 656
Le pattern Décorateur 659
Le pattern Composite 662
Le pattern Chain of responsabililty 662
Les patterns Strategy, State et Bridge . . . 663

Index **669**

1

Principes de base : quel objet pour l'informatique ?

Ce chapitre présente les briques de base de la conception et de la programmation orientée objet (OO). Il s'agit pour l'essentiel des notions d'objet, de classe, de message et d'héritage. À ce stade, aucun approfondissement technique n'est effectué. Les quelques extraits de code seront écrits dans un pseudo langage très proche de Java. De simples petits exercices de pensée composent une mise en bouche, toute en intuition, des éléments essentiels de la pratique OO.

SOMMAIRE

- Introduction à la notion d'objet
- Introduction à la notion et au rôle du référent
- L'objet dans sa version passive et active
- Introduction à la notion de classe
- Les interactions entre objets
- Introduction aux notions d'héritage et de polymorphisme

DOCTUS — *Tu as l'air bien remonté, aujourd'hui !*

CANDIDUS — *Je cherche un objet, mon vieux ! Je le cherche partout ! C'est ce sacré objet logiciel dont tout le monde parle, mais il est aussi insaisissable que le Yéti…*

DOC. — *Laisse-moi t'expliquer. Au commencement… il y avait l'ordinateur, tout juste né. C'est nous qui étions à son service pour le pouponner. Aujourd'hui, il comprend beaucoup mieux ce qu'on attend de lui. On peut lui parler en adulte, lui expliquer les choses d'une façon plus structurée…*

CAND. — *…Veux-tu dire qu'il comprend ce que nous voulons si nous le lui spécifions ?*

DOC. — *Doucement ! Je dis simplement que notre bébé est aujourd'hui capable de manipuler lui-même les informations qu'on lui confie. Il a ses propres méthodes d'utilisation et de rangement. Il ne veut même plus qu'on touche à ses jouets.*

Jetons un rapide coup d'œil par la fenêtre ; nous apercevons… des voitures, des passants, des arbres, un immeuble, un avion… Cette simple perception est révélatrice d'un ensemble de mécanismes cognitifs des plus subtils, dont la compréhension est une porte d'entrée idéale dans le monde de l'informatique orientée objet. En effet, pourquoi n'avoir pas cité « l'air ambiant », la « température », la « bonne athmosphère » ou encore « la lumière du jour », que l'on perçoit tout autant ? Pourquoi les premiers se détachent-ils de cette toile de fond parcourue par nos yeux ?

Tout d'abord, leur structure est singulière, compliquée ; ils présentent une forme alambiquée, des dimensions particulières, parfois une couleur uniforme et distincte du décor qui les entoure. Nous dirons que chacun se caractérise par un ensemble « d'attributs » structuraux, prenant pour chaque objet une valeur particulière : une des voitures est rouge et plutôt longue, ce passant est grand, assez vieux, courbé, etc. Ces attributs structuraux – et leur présence conjointe dans les objets – sont la raison première de l'attrait perceptif qu'ils exercent. C'est aussi la raison de la segmentation et de la nette séparation perceptive qui en résulte, car si les voitures et les arbres se détachent en effet de l'arrière plan, nous ne les confondons en rien.

Le trio <entité, attribut, valeur>

Nous avons l'habitude de décrire le monde qui nous entoure à l'aide de ce trio que les informaticiens se plaisent à nommer <entité (ou objet), attribut, valeur> ; par exemple <voiture, couleur, rouge>, <voiture, marque, peugeot>, <passant, taille, grande>, <passant, âge, 50>. Les objets (la voiture et le passant) vous sautent aux yeux parce que chacun se caractérise par un ensemble d'attributs (couleur, âge, taille), prenant une valeur particulière, uniforme « sur tout l'objet ».

La nature des attributs est telle qu'ils servent à définir un nombre important d'objets, pourtant très différents. La voiture a une taille comme le passant. L'arbre a une couleur comme la voiture. Le monde des attributs est beaucoup moins diversifié que celui des objets. C'est une des raisons qui nous permettent de regrouper les objets en classes et sous-classes, comme nous le découvrirons plus tard.

Que ce soit comme résultat de nos perceptions ou dans notre pratique langagière, les attributs et les objets jouent des rôles très différents. Les attributs structurent nos perceptions et ils servent, par exemple, sous forme d'adjectifs, à qualifier les noms qui les suivent ou les précèdent. La première conséquence de cette simple faculté de découpage cognitif sur l'informatique d'aujourd'hui est la suivante :

Objets, attributs, valeurs

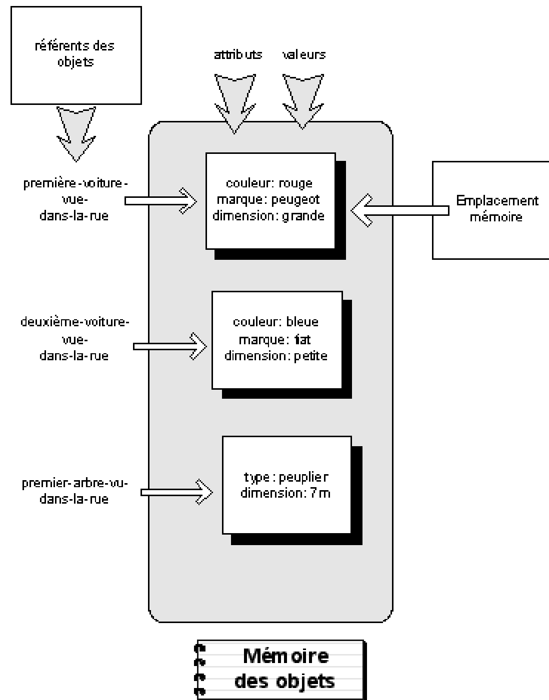
Il est possible dans tous les langages informatiques de stocker et de manipuler des objets en mémoire, comme autant d'ensembles de couples attribut/valeur.

Stockage des objets en mémoire

Dans la figure qui suit, nous voyons apparaître ces différents objets dans la mémoire de l'ordinateur. Chacun occupe un espace mémoire qui lui est propre et alloué lors de sa création. Pour se faciliter la

vie, les informaticiens ont admis un ensemble de « types primitifs » d'attributs, dont ils connaissent à l'avance la taille requise pour encoder la valeur.

Figure 1–1
Les objets informatiques et leur stockage en mémoire



Il s'agit, par exemple, de types comme *réel*, qui occupera 64 bits d'espace (dans le codage des réels en base 2 et selon une standardisation reconnue), ou *entier*, qui en occupera 32 (là encore par sa traduction en base 2), ou *caractère*, qui occupera 16 bits dans le format « unicode ou UTF-16 » (qui code ainsi chacun des caractères de la majorité des écritures répertoriées et les plus pratiquées dans le monde). Dans notre exemple, les dimensions seraient typées en tant qu'entier ou réel. Tant la couleur que la marque pourraient se réduire à une valeur numérique (ce qui ramènerait l'attribut à un entier) choisie parmi un ensemble fini de valeurs possibles, indiquant chacune une couleur ou une marque. Dès lors, le mécanisme informatique responsable du stockage de l'objet « saura », à la simple lecture structurelle de l'objet, quel est l'espace mémoire requis par son stockage.

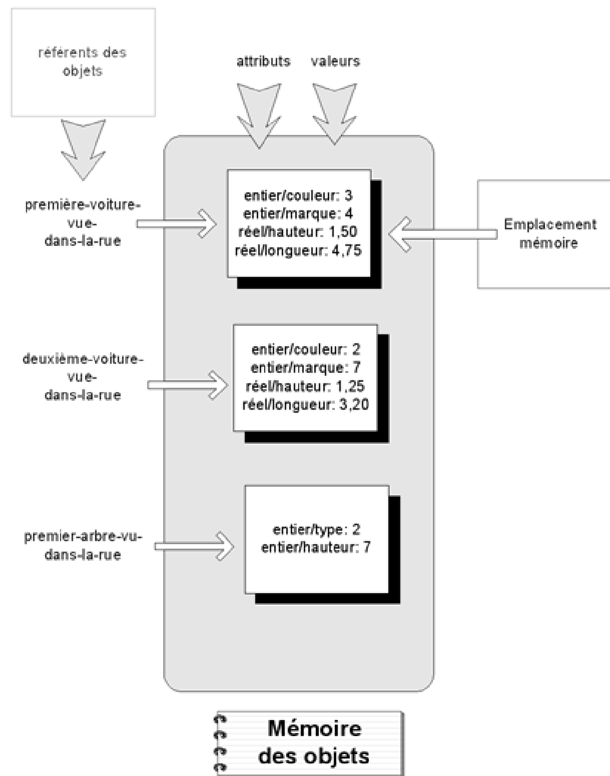
La place de l'objet en mémoire

Les objets sont structurellement décrits par un premier ensemble d'attributs de type primitif, tels qu'entier, réel ou caractère, qui sert à déterminer précisément l'espace qu'ils occupent en mémoire.

Types primitifs

À l'aide de ces types primitifs, le stockage en mémoire des objets se transforme comme reproduit sur la figure 1-2. Ce mode de stockage de données est une caractéristique récurrente en informatique, présent dans pratiquement tous les langages de programmation et dans les bases de données dites relationnelles. Dans ces dernières, chaque objet devient un enregistrement.

Figure 1–2
Les objets avec leur nouveau mode de stockage où chaque attribut est d'un type dit « primitif » ou « prédéfini », comme entier, réel, caractère…



Ainsi, les voitures sont stockées dans des bases à l'aide de leurs couples attribut/valeur ; elles sont gérées par un concessionnaire automobile (comme montré à la figure 1-3).

Figure 1–3
Table d'une base de données relationnelle de voitures, avec quatre attributs et six enregistrements

Marque	Modele	Serie	Numero
Renault	18	RL	4698 SJ 45
Renault	Kangoo	RL	4568 HD 16
Renault	Kangoo	RL	6576 VE 38
Peugeot	106	KID	7845 ZS 83
Peugeot	309	chorus	7647 ABY 82
Ford	Escort	Match	8562 EV 23

> **Bases de données relationnelles**
>
> Il s'agit du mode de stockage des données sur support permanent le plus répandu en informatique. Les données sont stockées en tant qu'enregistrements dans des tables, par le biais d'un ensemble de couples attribut/valeur, dont une clé primaire essentielle à la singularisation de chaque enregistrement. La clé prend une valeur unique par enregistrement. Des relations sont ensuite établies entre les tables par un mécanisme de jonction entre la clé primaire de la première table et la clé dite étrangère de celle à laquelle on désire la relier. Le fait, par exemple, qu'un conducteur puisse posséder plusieurs voitures se traduit en relationnel par la présence dans la table « voiture » d'une clé étrangère reprenant les valeurs de la clé primaire de la table « conducteurs ». La disparition de ces clés dans la pratique OO fait de la sauvegarde des objets dans ces tables un problème épineux de l'informatique d'aujourd'hui, comme nous le verrons au chapitre 19.

Les arbres, quant à eux, chacun également avec son couple attribut/valeur, s'enregistrent dans des bases de données gérées par un botaniste.

Notez d'ores et déjà que, bien qu'il s'agisse de trois objets différents, ils peuvent être repris sous la forme de deux classes : « Voiture » comprend deux objets et « Arbre » un seul. C'est la classe qui se charge de définir le type et le nombre des attributs qui la caractérisent. Les deux voitures seront distinctes uniquement parce qu'elles présentent des valeurs différentes pour ces attributs (« couleur », « marque », « hauteur », « longueur »).

Cette façon de procéder n'a rien de novateur et n'est en rien à l'origine de cette pratique informatique désignée comme orientée objet. La simple opération de stockage et de manipulation d'objets en soi et pour soi, en conformité avec un modèle que nous désignerons par « classe », n'est pas ce qui distingue fondamentalement l'informatique orientée objet de celle désignée comme « procédurale » ou « fonctionnelle ». Nous la retrouvons dans pratiquement tous les langages informatiques. Patience ! Nous allons y venir…

De tout temps également, les mathématiciens, physiciens ou autres scientifiques ont manipulé des objets mathématiques caractérisés par un ensemble de couples attribut/valeur. Ainsi, un point dans un espace à trois dimensions se caractérise par les valeurs réelles prises par ses attributs x,y,z. Lorsque ce point bouge, on peut y adjoindre trois nouveaux attributs pour représenter sa vitesse. Il en est de même pour une espèce animale (nombre de représentants, type d'alimentation...), un atome (nombre atomique, densité), une molécule (atomes la composant, concentration au sein d'un mélange), la santé économique d'un pays (PIB par habitant, balance commerciale, taux d'inflation), des étudiants (nom, prénom, adresse, année d'étude, filière d'études), etc.

Le référent d'un objet

Observons à nouveau les figures 1-1 et 1-2. Chaque objet est nommé et ce nom doit être son unique identifiant. Comme c'est en le nommant que nous accédons à l'objet, il est clair que ce nom ne peut être partagé par plusieurs objets. En informatique, le nom correspondra de manière univoque à l'adresse physique de l'objet en mémoire. Rien n'est plus unique qu'une adresse mémoire. Le nom « première-voiture-vue-dans-la-rue » est en fait une variable informatique, que nous appellerons référent par la suite, stockée également en mémoire, mais dans un espace dédié uniquement aux noms symboliques. À cette variable, on affecte comme valeur l'adresse physique de l'objet que ce nom symbolique désigne. En général, dans la plupart des ordinateurs aujourd'hui, l'adresse mémoire se compose de 64 bits, ce qui permet de stocker jusqu'à 2^{64} informations différentes.

Espace mémoire

Ces dernières années, de plus en plus de processeurs ont fait le choix d'une architecture à 64 bits au lieu de 32, ce qui implique notamment une révision profonde de tous les mécanismes d'adressage dans les systèmes d'exploitation. Depuis, les informaticiens peuvent se sentir à l'aise face à l'immensité de l'espace d'adressage qui s'ouvre à eux : 2^{64}, soit 18 446 744 073 709 551 616 octets.

Référent vers un objet unique

Le nom d'un objet informatique, ce qui le rend unique, est également ce qui permet d'y accéder physiquement. Nous appellerons ce nom le « référent de l'objet ». L'information reçue et contenue par ce référent n'est rien d'autre que l'adresse mémoire où cet objet se trouve stocké.
Un référent est une variable informatique particulière, associée à un nom symbolique, codée sur 64 bits et contenant l'adresse physique d'un objet informatique.

Plusieurs référents pour un même objet

Un même objet peut-il porter plusieurs noms ? Plusieurs référents, qui contiennent tous la même adresse physique, peuvent-ils désigner en mémoire un même objet ? Oui, s'il est nécessaire de nommer, donc d'accéder à l'objet, dans des contextes différents et qui s'ignorent mutuellement.

Dans la vie courante, rien n'interdit à plusieurs personnes, tout en désignant le même objet, de le nommer de manière différente. Il n'y a qu'un seul Hugues Bersini, mais il est « professeur d'informatique » pour ses étudiants, « papa » pour ses enfants, « chéri » pour l'heureuse élue, « ce gêneur » pour son voisin à qui il vient de ravir la dernière place de parking de la rue… Autant de référents pour une seule adresse physique. Les noms des objets seront distincts, car utilisés dans des contextes distincts qui, pour l'essentiel, s'ignorent. C'est aussi faisable sinon fondamental en informatique orientée objet, grâce à ce mécanisme puissant et souple de référence informatique, dénommé adressage indirect par les informaticiens et qui permet, sans difficulté, d'offrir plusieurs voies d'accès à un même objet mémoire. Comme la pratique orientée objet s'accompagne d'une découpe en objets et que chacun d'entre eux peut être sollicité par plusieurs autres qui « s'ignorent » entre eux, il est capital que ces derniers puissent désigner le premier à leur guise, en lui donnant un nom plus conforme à l'utilisation qu'ils en feront.

Adressage indirect

C'est la possibilité pour une variable, non pas d'être associée directement à une donnée, mais plutôt à une adresse physique d'un emplacement contenant, lui, cette donnée. Il devient possible de différer le choix de cette adresse pendant l'exécution du programme, tout en utilisant naturellement la variable. Et plusieurs de ces variables peuvent alors pointer vers un même emplacement car partageant la même adresse. Une telle variable, dont la valeur est une adresse, est dénommée un pointeur en C et C++.

Nous verrons dans la section suivante qu'un attribut d'un objet peut servir de référent vers un autre objet, et qu'il y a là un mécanisme idéal pour que ces deux objets communiquent. Mais n'allons pas trop vite en besogne…

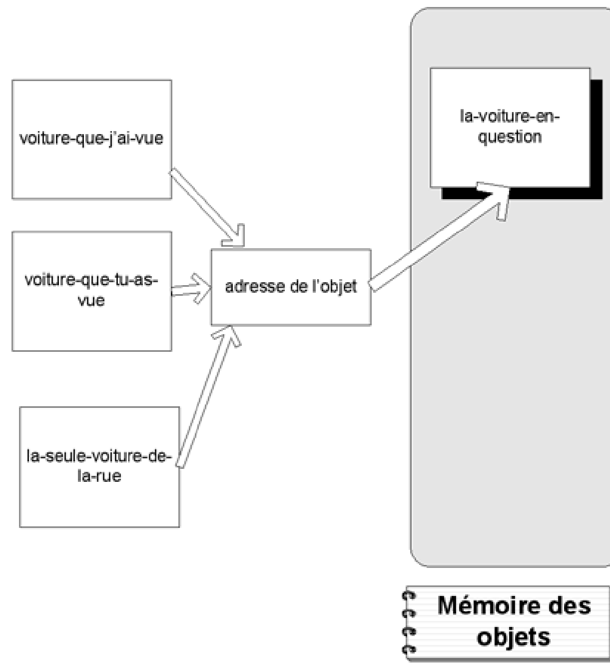


Figure 1–4
Plusieurs référents désignent un même objet grâce au mécanisme informatique d'adressage indirect.

Plusieurs référents pour un même objet

Lorsqu'on écrit un programme orienté objet, on accède couramment à un même objet par plusieurs référents, créés dans différents contextes d'utilisation. Cette multiplication des référents est un élément déterminant de la gestion mémoire associée à l'objet.
On acceptera à ce stade-ci qu'il est utile qu'un objet séjourne en mémoire tant qu'il est possible de le référer. Sans référent, un objet est inaccessible. Le jour où vous n'êtes même plus un numéro dans aucune base de données, vous êtes mort (socialement, tout au moins).
En C++, le programmeur peut effacer un objet à partir de n'importe lequel de ses référents (par exemple « delete voiture-que-j'ai-vue »). On vous laisse dès lors deviner vers quoi pointeront les autres référents (taxés de « pointeurs fous »), pour que vous preniez conscience d'une des difficultés majeures et une des sources d'erreur les plus cruelles (car ses effets sont imprévisibles) inhérentes à la programmation dans ce vénérable langage.

L'objet dans sa version passive

L'objet et ses constituants

Voyons plus précisément ce qui amène notre perception à privilégier certains objets plutôt que d'autres. Certains d'entre eux se révèlent être une composition subtile d'autres objets, tout aussi présents que les premiers, mais que, pourtant, il ne vous est pas venu à l'idée de citer lors de notre première démonstra-

tion. Vous avez dit « la voiture » et non pas « la roue de la voiture » ou « la portière » ; vous avez dit « l'arbre » et non pas « la branche » ou « le tronc de l'arbre ». De nouveau, c'est l'agrégat qui vous saute aux yeux et non pas toutes ses parties. Vous savez pertinemment que l'objet « voiture » ne peut fonctionner en l'absence de ses objets « roues » ou « moteur ». Néanmoins, pour citer ce que vous observiez, vous avez fait l'impasse sur les différentes parties constitutives des objets relevés.

Première distinction : ce qui est utile à soi et ce qui l'est aux autres

L'orienté objet, pour des raisons pratiques que nous évoquerons par la suite, encourage à séparer, dans la description de tout objet, la partie utile pour tous les autres objets qui y recourront, de la partie nécessaire à son fonctionnement propre. Il faut séparer physiquement ce que les autres objets doivent savoir d'un objet donné, afin de solliciter ses services, de ce que ce dernier requiert pour son fonctionnement, c'est-à-dire la mise en œuvre de ces mêmes services.

Objet composite

Tant que vous n'aurez pas besoin du garagiste, vous ne vous préoccuperez pas des roues ni du moteur de votre objet « voiture ». Que les objets s'organisent entre eux en composite et composants est une donnée de notre réalité que les informaticiens ont jugé important de reproduire. Comme indiqué dans la figure suivante, un objet stocké en mémoire peut être placé à l'intérieur de l'espace mémoire réservé à un autre.

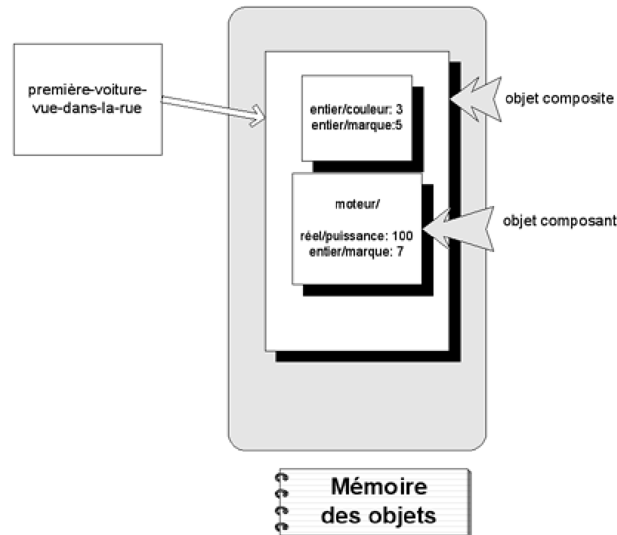


Figure 1–5
L'objet moteur devient un composant de l'objet voiture.

Son accès ne sera dès lors possible qu'à partir de celui qui lui offre cette hospitalité et, s'il le fait, c'est qu'il sait que sa propre existence conditionne celle de son hôte. L'objet moteur, dans ce cas, n'existe que comme attribut de l'objet voiture : si vous vous débarrassez de la voiture, vous vous débarrasserez dans le même temps de son moteur. C'est à la voiture qu'incombera la prise en charge de son moteur : création comme destruction.

> **Une composition d'objets**
>
> Entre eux, les objets peuvent entrer dans une relation de type composition, où certains se trouvent contenus dans d'autres et ne sont accessibles qu'à partir de ces autres. Leur existence dépend entièrement de celle des objets qui les contiennent.

Dépendance sans composition

Vous comprendrez aisément que ce type de relation entre objets ne suffit pas pour décrire fidèlement la réalité qui nous entoure. En effet, si la voiture possède bien un moteur, elle peut également contenir des passagers, qui n'aimeraient pas être portés disparus lors de la mise à la casse de la voiture... D'autres modes de mise en relation entre objets devront être considérés, qui permettent à un premier de se connecter facilement à un deuxième, mais sans que l'existence de celui-ci ne soit entièrement conditionnée par l'existence du premier. C'est ce type d'association qui nous réunit, vous, lecteurs, et nous, auteurs et nous vous souhaitons sincèrement de ne pas nous accompagner le jour du dernier soupir.

L'objet dans sa version active

Activité des objets

Afin de poursuivre cette petite introspection cognitive dans le monde de l'informatique orientée objet, jetez à nouveau un coup d'œil par la fenêtre et décrivez-nous quelques scènes observées : « Une voiture s'arrête à un feu rouge », « Les passants traversent la route », « Un passant entre dans un magasin », « Un oiseau s'envole de l'arbre ». Que dire de toutes ces observations bouleversantes que vous venez d'énoncer ? D'abord, que les objets ne se bornent pas à être statiques. Ils se déplacent, changent de forme, de couleur, d'humeur, souvent suite à une interaction directe avec d'autres objets. La voiture s'arrête car le feu est devenu rouge et elle redémarre dès qu'il passe au vert. Les passants traversent quand les voitures s'arrêtent. L'épicier dit « bonjour » au client qui ouvre la porte de son magasin (et quelquefois le client lui répond). Les objets inertes sont par essence bien moins intéressants que ceux qui se modifient constamment. Certains batraciens ne détectent leur nourriture favorite que si elle est en mouvement : placez-la, immobile, devant eux et l'animal ne la verra simplement pas. Ainsi, l'objet sera d'autant plus riche d'intérêt qu'il est sujet à des transitions d'états nombreuses et variées.

Les différents états d'un objet

Les objets changent donc d'état, continûment, mais tout en préservant leur identité, en restant ces mêmes objets qu'ils ont toujours été. Les objets sont dynamiques, la valeur de leurs attributs change dans le temps, soit par des mécanismes qui leur sont propres (tel le changement des feux de signalisation), soit en raison d'une interaction avec un autre objet (comme dans le cas de la voiture qui s'arrête au feu rouge).

Du point de vue informatique, rien n'est plus simple que de modifier la valeur d'un attribut. Il suffit de se rendre dans la zone mémoire occupée par cet attribut et de remplacer la valeur qui s'y trouve actuelle-

ment stockée par une nouvelle valeur. La mise à jour d'une partie de sa mémoire, par l'exécution d'une instruction appropriée, est une des opérations les plus fréquemment effectuées par un ordinateur. Le changement d'un attribut n'affecte en rien l'adresse de l'objet, donc son identité, tout comme vous restez la même personne, humeur changeante ou non. L'objet, en fait, préservera cette identité jusqu'à sa pure et simple suppression de la mémoire informatique. Pour l'ordinateur : « partir, c'est mourir tout à fait ». L'objet naît, vit une succession de changements d'états et finit par disparaître de la mémoire.

Changement d'états

Le cycle de vie d'un objet, lors de l'exécution d'un programme orienté objet, se limite à une succession de changements d'états, jusqu'à sa disparition pure et simple de la mémoire centrale.

Les changements d'état : qui en est la cause ?

Qui est donc responsable des changements de valeur des attributs ? Qui a la charge de rendre les objets moins inertes qu'ils n'apparaissent à première vue ? Qui se charge de les faire évoluer et, ce faisant, de les rendre un tant soit peu attrayants ? Reprenons l'exemple des feux de signalisation et des voitures évoqué plus haut et, comme indiqué à la figure 1-6, stockons-en un de chaque sorte dans la mémoire de l'ordinateur (nous supposerons que la couleur est bien représentée par un entier ne prenant que les valeurs 1, 2 ou 3, et la vitesse par un simple entier). Installons dans cette même mémoire, mais un peu plus loin, deux opérations chargées de changer l'une la couleur du feu et l'autre la vitesse de la voiture. Dans la mémoire dite centrale, RAM ou vive, d'un ordinateur, ne se trouvent toujours installés que ces deux types d'information : des données et des instructions qui utilisent et modifient ces données, rien d'autre. Comme les attributs, chaque opération se doit d'être nommée afin de pouvoir y accéder ; nos deux opérations s'appelleront respectivement `change` et `changeVitesse (int nV)`. Ces deux opérations sont banales : pour le feu, il s'agit d'incrémenter l'entier couleur et de ramener sa valeur à 1 dès qu'il atteint 4 (le feu reproduit constamment le même cycle), et pour la voiture de changer sa vitesse en fonction de l'argument transmis en paramètre (l'entier `nV`, pour autant qu'il ne dépasse pas la limite autorisée). Ces deux opérations triviales mettront un peu d'animation dans la rue.

Comment relier les opérations et les attributs ?

Alors que nous comprenons bien l'installation en mémoire, tant de l'objet que de l'opération qui pourra le modifier, ce qui nous apparaît moins évident, c'est la liaison entre les deux. Ainsi, pour le feu, comment l'opération et les deux instructions de changement (l'incrémentation et le test), installées dans la mémoire à droite, savent-elles qu'elles portent sur le feu de signalisation installé, lui, dans la mémoire à gauche ? Plus concrètement encore, comment l'opération `change`, qui incrémente et teste un entier, sait-elle que cet entier est, de fait, celui qui code la couleur du feu et non « l'âge du capitaine » ? De même, comment l'opération `changeVitesse` sait-elle que c'est bien sur la vitesse de la voiture qu'elle doit porter, et non pas sur un feu, qu'elle essaierait de pousser à 120. La réponse à cette question vous fait entrer de plain-pied dans le monde de l'orienté objet…

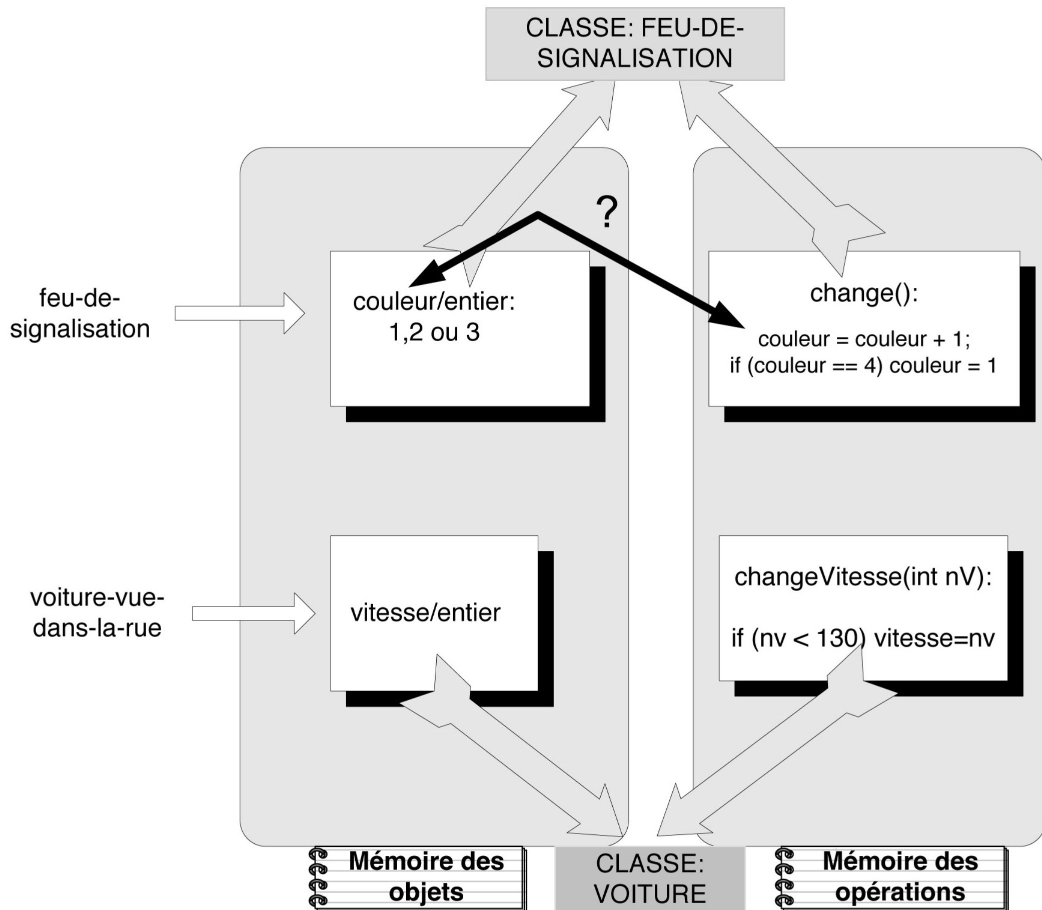


Figure 1–6 Le changement d'état du feu de signalisation par l'entremise de l'opération « change » et celui de la voiture par l'opération « changeVitesse(int nV) »

Introduction à la notion de classe

Méthodes et classes

Place à la réponse. Elle s'articule en deux temps. Concentrons-nous uniquement sur le feu de signalisation (pour la voiture, le raisonnement est en tout point analogue). Dans un premier temps, il faudra que l'opération `change` – que nous appellerons dorénavant *méthode* `change` – ne soit attachée qu'à des objets de type `feu de signalisation`. Seuls ces objets possèdent l'entier sur lequel peut s'exercer cette

méthode. Appliquer la méthode `change` sur tout autre type d'objet, tel que la voiture ou l'arbre, n'a pas de sens, car on ne saurait de quel entier il s'agit. De surcroît, ce double incrément pour revenir à la valeur de base est totalement dénué de signification pour ces autres objets. Le subterfuge qui associe la méthode avec l'objet qui lui correspond consiste à les unir tous deux par les liens, non pas du mariage, mais de la « classe ». Deux classes sont à l'œuvre figure 1-6 : le `Feu-De-Signalisation` et la `Voiture`.

> **Classe**
>
> Une nouvelle structure de données voit le jour en OO : la *classe*, qui, de fait, a pour principal rôle d'unir en son sein tous les attributs de l'objet et toutes les opérations les concernant. Ces opérations sont appelées *méthodes* et regroupent un ensemble d'instructions portant sur les attributs de l'objet.
>
> Pour les programmeurs en provenance du procédural, les attributs de la classe sont comme des arguments implicites passés à la méthode ou encore des variables dont la portée d'action se limite à la seule classe. Les méthodes s'occupent de toute la partie active de ce petit programme, que nous appellerons désormais « classe » et qui tend à ne reprendre qu'un concept unique de la réalité perçue.

La classe devient ce contrat logiciel qui lie les attributs de l'objet et les méthodes qui les utilisent. Par la suite, tout objet devra impérativement respecter ce qui est dit par sa classe, sinon gare au gardien des rites : le compilateur !

Comme c'est l'usage en informatique, s'agissant de variables manipulées, on parlera dorénavant de l'objet comme d'une *instance* de sa classe, et de la classe comme du *type* de cet objet.

On voit l'intérêt, bien sûr, de garder une définition de la classe séparée mais partagée par toutes les instances de celles-ci. Non seulement c'est la classe qui détermine les attributs (leur type) sur lesquels les méthodes peuvent opérer mais, plus encore, seules les méthodes déclarées dans la classe pourront de facto manipuler les attributs des objets typés par cette classe.

La classe `Feu-de-signalisation` pourrait être définie plus ou moins comme suit :

```
class Feu-de-signalisation {
  int couleur ;
  change() {
    couleur = couleur + 1 ;
    if (couleur ==4) couleur = 1 ;
  }
}
```

> **Type entier**
>
> Le type primitif entier est souvent appelé dans les langages de programmation `int` (pour *integer*), le type réel `double` ou `float`, le caractère `char`. Eh oui ! L'anglais reste l'espéranto de l'informatique !

Chaque objet `feu de signalisation` répondra de sa classe, en faisant en sorte, dès sa naissance, de n'être modifié que par les méthodes qui y sont déclarées. Pour notre exemple, nous n'avons défini que `change`, mais il pourrait y en avoir bien d'autres comme `met-le-feu-en-stand-by`, `change-la-durée-d'une-des-couleurs` ; il faudrait alors ajouter quelques attributs comme la durée de chaque couleur.

Un objet existe par l'entremise de ses attributs et se modifie uniquement par l'entremise de ses méthodes (et nous disons bien « ses »).

Sur quel objet précis s'exécute la méthode ?

Dans un second temps, il faudra signaler à la méthode `change` (qui, grâce à la définition de la classe, sait qu'elle opère exclusivement sur des feux de signalisation) lequel, parmi tous les feux possibles et stockés en mémoire, est celui qu'il est nécessaire de changer. Cela se fera par le simple appel de la méthode sur l'objet en question et, plus encore, par l'écriture d'une instruction de programmation de type `feu-de-signalisation.change()`

Nous appliquons la méthode `change()` (nous expliquerons plus tard la raison d'être des parenthèses) sur l'objet `feu-de-signalisation`. Dans cette instruction, c'est le point qui établit la liaison entre l'objet précis et la méthode à exécuter. N'oubliez pas que le référent `feu-de-signalisation` possède effectivement l'adresse de l'objet et donc celle de l'attribut entier-couleur sur lequel la méthode `change()` doit s'appliquer. Pour la voiture, on écrirait par exemple `voiture.changeVitesse(120)`, ce qui aurait pour effet de régler à 120 la vitesse de la voiture en question.

Lier la méthode à l'objet

On lie la méthode `f(x)` à l'objet `a` sur lequel elle doit s'appliquer, au moyen d'une instruction comme `a.f(x)`. Par cette écriture, la méthode `f(x)` sait comment accéder aux seuls attributs de l'objet `a` qu'elle peut manipuler.

Différencier langage orienté objet et langage manipulant des objets

De nombreux langages de programmation, surtout de scripts pour le développement web (JavaScript, VB Script), rendent possible l'exécution de méthodes sur des objets dont les classes préexistent au développement. Le programmeur ne crée jamais de nouvelles classes mais se contente d'exécuter les méthodes de celles-ci sur des objets. Supposons par exemple que vous vouliez agrandir une police (*font*) particulière de votre page web. Vous écrirez `f.setSize(16)` mais jamais dans votre code vous n'aurez créé la classe `Font` (vous utilisez l'objet `f` issu de cette classe) ni sa méthode `setSize()`. Vous vous limitez à les utiliser comme vous utilisez les bibliothèques d'un quelconque langage de programmation. Les classes regroupées dans ces bibliothèques auront été développées par d'autres programmeurs et mises à votre disposition.

La programmation orientée objet débute dès que vous incombe la création de nouvelles classes, même si lors de l'écriture de vos codes, vous vous reposez largement sur des classes écrites par autrui. C'est bien de programmation orientée classe qu'il s'agirait plutôt.

Pour certains, dès lors, et au contraire de ce que nous venons d'affirmer, un langage comme Javascript en deviendrait, lui, vraiment orienté objet. Ce n'est pas tant qu'il soit impossible d'écrire de nouvelles classes, mais plutôt que tous les objets soient en même temps des classes. En effet, il est possible en Javascript de créer directement un objet à partir d'un autre, suivant un mécanisme ressemblant à l'héritage et généralement appelé prototypage. C'est également le modèle des langages Io et Self, largement reconnus comme orientés objet. Déterminer quels langages peuvent effectivement se targuer d'être orientés objet est devenu une question plutôt délicate à trancher.