Chapitre 1

INTRODUCTION

Les langages à objets sont apparus depuis quelques années comme un nouveau mode de programmation. Pourtant la programmation par objets date du milieu des années 60 avec Simula, un langage créé par Ole Dahl et Kristen Nygaard en Norvège et destiné à programmer des simulations de processus physiques. Bien que de nombreux travaux de recherche aient eu lieu depuis cette époque, l'essor des langages à objets est beaucoup plus récent. Ce livre essaie de montrer que les langages à objets ne sont pas une mode passagère, et que la programmation par objets est une approche générale de la programmation qui offre de nombreux avantages.

1.1 LE CHAMP DES LANGAGES

Situer les langages à objets dans le vaste champ des langages de programmation n'est pas chose facile tant le terme d'*objet* recouvre de concepts différents selon les contextes. Nous présentons ici plusieurs classifications des langages et situons les langages à objets dans ces différentes dimensions.

Classification selon le mode de programmation

Cette classification présente les principaux modèles de programmation qui sont utilisés aujourd'hui, c'est-à-dire les principaux modèles par lesquels on peut exprimer un calcul.

- *La programmation impérative*, la plus ancienne, correspond aux langages dans lesquels l'algorithme de calcul est décrit explicitement, à l'aide d'instructions telles que l'affectation, le test, les branchements, etc. Pour s'exécuter, cet algorithme nécessite des données, stockées dans des variables auxquelles le programme accède et qu'il peut modifier. La formule de Niklaus Wirth décrit parfaitement cette catégorie de langages :

  programme = algorithme + structure de données.

  On classe dans cette catégorie les langages d'assemblage, Fortran, Algol, Pascal, C, Ada. Cette catégorie de langages est la plus ancienne car elle correspond naturellement au modèle d'architecture des machines qui est à la base des ordinateurs : le modèle de Von Neumann.

- *La programmation fonctionnelle* adopte une approche beaucoup plus mathématique de la programmation. Fondée sur des travaux assez anciens sur le lambda-calcul et popularisée par le langage Lisp, la programmation fonctionnelle supprime la notion de variable, et décrit un calcul par une fonction qui s'applique sur des données d'entrées et fournit comme résultat les données en sortie. De fait, ce type de programmation est plus abstrait car il faut décrire l'algorithme indépendamment des données. Il existe peu de langages purement fonctionnels ; beaucoup introduisent la notion de variable pour des raisons le plus souvent pratiques, car l'abstraction complète des données conduit souvent à des programmes lourds à écrire. Parmi les langages fonctionnels, il faut citer bien sûr Lisp et ses multiples incarnations (CommonLisp, Scheme, Le_Lisp, etc.), ainsi que ML.

- *La programmation logique*, née d'une approche également liée aux mathématiques, la logique formelle, est fondée sur la

description d'un programme sous forme de prédicats. Ces prédicats sont des règles qui régissent le problème décrit ; l'exécution du programme, grâce à l'inférence logique, permet de déduire de nouvelles formules, ou de déterminer si une formule donnée est vraie ou fausse à partir des prédicats donnés. L'inférence logique est tout à fait similaire aux principes qui régissent la démonstration d'un théorème mathématique à l'aide d'axiomes et de théorèmes connus. Le plus célèbre des langages logiques est Prolog.

La programmation par objets est-elle une nouvelle catégorie dans cette classification ? Il est difficile de répondre à cette question. La programmation par objets est proche de la programmation impérative : en effet, là où la programmation impérative met l'accent sur la partie algorithmique de la formule de Wirth, la programmation par objets met l'accent sur la partie structure de données. Néanmoins, ceci n'est pas suffisant pour inclure la programmation par objets dans la programmation impérative, car l'approche des langages à objets s'applique aussi bien au modèle fonctionnel ou logique qu'au modèle impératif.

Classification selon le mode de calcul

Cette classification complète la précédente en distinguant deux modèles d'exécution d'un calcul :

- *Les langages séquentiels* correspondent à une exécution séquentielle de leurs instructions selon un ordre que l'on peut déduire du programme. Ces langages sont aujourd'hui les plus répandus, car ils correspondent à l'architecture classique des ordinateurs dans lesquels on a une seule unité de traitement (modèle de Von Neumann). Les langages à objets sont pour la plupart séquentiels.

- *Les langages parallèles* permettent au contraire à plusieurs instructions de s'exécuter simultanément dans un programme. L'essor de la programmation parallèle est dû à la disponibilité de machines à architecture parallèle. La programmation

parallèle nécessite des langages spécialisés car les langages usuels ne fournissent pas les primitives de communication et de synchronisation indispensables. Or il s'avère que le modèle général de la programmation par objets est facilement parallélisable. Une classe de langages à objets, les langages d'acteurs, sont effectivement des langages parallèles.

Classification selon le typage

Cette classification considère la notion de *type* qui, dans les langages de programmation, est destinée à apporter une sécurité au programmeur : en associant un type à chaque expression d'un programme, on peut déterminer par une analyse statique, c'est-à-dire en observant le texte du programme sans l'exécuter, si le programme est correct du point de vue du système de types. Cela permet d'assurer que le programme ne provoquera pas d'erreur à l'exécution en essayant, par exemple, d'ajouter un booléen à un entier.

- Dans un *langage à typage statique*, on associe, par une analyse statique, un type à chaque expression du programme. C'est le système de types le plus sûr, mais aussi le plus contraignant. Pascal est l'exemple typique d'un langage à typage statique.

- Dans un *langage fortement typé*, l'analyse statique permet de vérifier que l'exécution du programme ne provoquera pas d'erreur de type, sans pour autant être capable d'associer un type à chaque expression. Ceci signifie que les types devront éventuellement être calculés à l'exécution pour contrôler celle-ci. Les langages qui offrent le polymorphisme paramétrique (ou généricité), comme ADA, sont fortement typés. La plupart des langages à objets typés sont fortement typés, et notamment Simula, Modula3 et C++.

- Dans un *langage faiblement typé*, on ne peut assurer, par la seule analyse statique, que l'exécution d'un programme ne provoquera pas d'erreur liée au système de types. Il est donc nécessaire de calculer et de contrôler les types à l'exécution,

ce qui justifie l'appellation de *langage à typage dynamique*. Parmi les langages à objets, Eiffel est faiblement typé car une partie du contrôle de types doit être réalisée à l'exécution.

- Dans un *langage non typé*, la notion de type n'existe pas, et il ne peut donc y avoir de contrôle sur la validité du programme vis-à-vis des types. Ainsi, Lisp est un langage non typé, de même que le langage à objets Smalltalk.

La notion de type apporte une sécurité de programmation indispensable dans la réalisation de gros systèmes. De plus, comme nous allons le voir, elle permet de rendre l'exécution des programmes plus efficace. C'est donc une notion importante, et l'on peut constater que les langages à objets couvrent une grande partie de la classification ci-dessus. C'est ce critère qui va nous servir à structurer la suite de ce livre en distinguant d'un côté les *langages typés* (fortement ou faiblement), d'autre part les *langages non typés*. De nombreux travaux sont consacrés actuellement à l'étude des systèmes de types dans les langages à objets car le concept d'héritage, qui est l'un des fondements des langages à objets, nécessite des systèmes de types qui n'entrent pas dans les cadres étudiés jusqu'à présent.

Classification selon le mode d'exécution

Cette classification porte sur la façon dont l'exécution du programme est réalisée. Le mode d'exécution n'est pas à proprement parler une caractéristique d'un langage, mais une caractéristique de l'*implémentation* d'un langage.

- *Les langages interprétés* permettent à l'utilisateur d'entrer des expressions du langage et de les faire exécuter immédiatement. Cette approche offre l'avantage de pouvoir tester et modifier rapidement un programme au détriment de la vitesse d'exécution. Outre la vitesse d'exécution, les langages interprétés ont généralement pour inconvénient d'être moins sûrs, car de nombreux contrôles sémantiques sont réalisés au fur et à mesure de l'interprétation et non dans

une phase préliminaire de compilation. Beaucoup de langages à objets sont interprétés, et tirent avantage de cette apparente faiblesse pour donner une plus grande souplesse à l'exécution des programmes. Ainsi, une modification ponctuelle d'un programme peut avoir des effets (contrôlés) sur l'ensemble de l'exécution, ce qui permet notamment de construire des environnements sophistiqués pour la mise au point des programmes.

- *Les langages compilés* nécessitent une phase de compilation avant de passer à l'exécution proprement dite d'un programme. Le but de la compilation est essentiellement de produire du code directement exécutable par la machine, donc plus efficace. Pour cela, le compilateur utilise en général les informations qui lui sont fournies par le système de types du langage. C'est ainsi que la plupart des langages typés sont compilés. De manière générale, les langages interprétés sont plus nombreux que les langages compilés car la réalisation d'un compilateur est une tâche complexe, surtout si l'on veut engendrer du code machine efficace. Les langages à objets n'échappent pas à cette règle, et il existe peu de langages à objets compilés.

- *Les langages semi-compilés* ont les caractéristiques des langages interprétés mais, pour une meilleure efficacité, ils utilisent un compilateur de manière invisible pour l'utilisateur. Ce compilateur traduit tout ou partie du programme dans un code intermédiaire ou directement en langage machine. Si le compilateur engendre un code intermédiaire, c'est un interpréteur de ce code qui réalisera l'exécution du programme. Les langages à objets considérés comme interprétés sont souvent semi-compilés, comme Smalltalk ou le langage de prototypes Self.

Classification selon la modularité

Cette dernière classification analyse la façon dont les langages permettent au programmeur de réaliser des modules et d'encapsuler les données. La modularité et l'encapsulation

assurent une plus grande sécurité de programmation et fournissent la base de la réutilisation.

- *Absence de modularité* : le programmeur doit, par des conventions de programmation et de nommage, prendre en charge la modularité. C'est notamment le cas de Pascal qui oblige à donner des noms différents à toutes les variables, procédures et fonctions globales, et ne permet pas de cacher des définitions autrement que par les règles de visibilité (procédures imbriquées), ce qui est insuffisant. Le langage C offre une modularité selon le découpage du programme en fichiers, en permettant de déclarer des variables ou fonctions privées à un fichier. En revanche, il n'offre pas la possibilité d'imbrication des procédures et fonctions.

- *Modularité explicite* : certains langages offrent la notion de module comme concept du langage, que l'on peut composer avec d'autres notions. Par exemple, en Ada, la notion de « package » permet d'implémenter un module, et se combine avec la généricité pour autoriser la définition de modules génériques (paramétrés par des types).

- *Modularité implicite* : les langages à objets fournissent une modularité que l'on peut qualifier d'implicite dans la mesure où celle-ci ne fait pas appel à des structures particulières du langage. Au contraire, la notion de module est implicite et indissociable de celle de classe.

La programmation par objets est une forme de programmation modulaire dans laquelle l'unité de modularité est fortement liée aux structures de données du langage. Par comparaison, le langage Ada offre une modularité beaucoup plus indépendante des structures de données et de traitement du langage. La modularité est souvent considérée comme un atout important pour la réutilisation des programmes. De ce point de vue, les langages à objets permettent une réutilisation bien plus importante que la plupart des autres langages.

La programmation par objets comme un style

De ce tour d'horizon, il faut conclure que la programmation par objets est plus une approche générale de la programmation qu'un type facilement classable. De fait, on voit aujourd'hui de nombreuses « extensions objets » à des langages existant : Pascal Objet, Lisp Objet, Cobol Objet, etc. Si certaines de ces extensions ne sont pas toujours un succès, il est un fait que les principes de la programmation par objets sont applicables dans un grand nombre de contextes.

1.2 HISTORIQUE

La figure 1 présente la généalogie des principaux langages à objets. On y distingue deux pôles autour des langages Simula et Smalltalk.

La famille Simula

Le langage Simula est considéré comme le précurseur des langages à objets. Développé dans les années 60 pour traiter des problèmes de simulation de processus physiques (d'où son nom), Simula a introduit la notion de classe et d'objet, la notion de méthode et de méthode virtuelle. Ces concepts restent à la base des langages à objets typés et compilés, comme on le verra en détail dans le chapitre 3. Simula a été créé dans la lignée du langage Algol, langage typé de l'époque dont Pascal s'est également inspiré.

Les langages de la famille Simula sont des langages impératifs typés et généralement compilés. L'intérêt d'un système de types est de permettre un plus grand nombre de contrôles sémantiques lors de la compilation, et d'éviter ainsi des erreurs qui ne se manifesteraient qu'à l'exécution. Typage et compilation sont deux concepts indépendants : on peut imaginer des langages typés interprétés et des langages non typés compilés. C'est néanmoins rarement le cas, car les informations déduites du

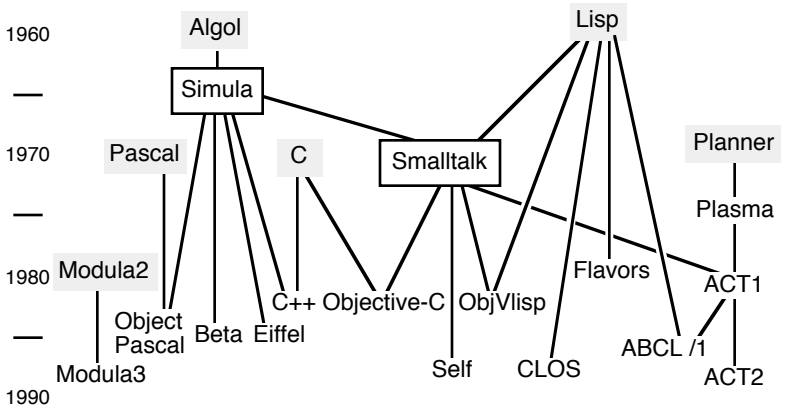

Figure 1 - Généalogie des principaux langages à objets

typage permettent de générer du code plus efficace, ce qui encourage à compiler les langages typés.

Simula a inspiré nombre de langages à objets. Certains d'entre eux sont maintenant largement plus connus et répandus que Simula, comme C++ et Eiffel. Plusieurs langages ont été conçus en ajoutant des concepts des langages à objets à un langage existant. Ainsi, Classcal et son descendant Object Pascal ajoutent des classes au langage Pascal, Modula3 est une révision majeure de Modula2 incluant des objets, C++ est construit à partir de C. Une telle approche présente avantages et inconvénients : il est commode de prendre un langage existant aussi bien pour les concepteurs que pour les utilisateurs car cela évite de tout réinventer ou de tout réapprendre. D'un autre côté, certains aspects du langage de base peuvent se révéler néfastes pour l'adjonction de mécanismes des langages à objets.

La famille Smalltalk

Dans les années 70, sous l'impulsion d'Alan Kay, ont commencé aux laboratoires Xerox PARC des recherches qui ont conduit au langage Smalltalk, considéré par beaucoup comme le

prototype des langages à objets. De fait, c'est Smalltalk plutôt que Simula qui est à l'origine de la vague des langages à objets depuis les années 80.

Smalltalk s'inspire de Simula pour les concepts de classes, d'objets et de méthodes, mais adopte une approche influencée par Lisp pour ce qui concerne l'implémentation : Smalltalk est un langage semi-compilé dans lequel tout est décidé lors de l'exécution. Comme dans Lisp, la souplesse due à l'aspect interprété de l'exécution est largement exploitée par le langage.

D'un point de vue conceptuel, Smalltalk introduit la notion de métaclasse qui n'est pas présente dans Simula. Cette notion permet de donner une description méta-circulaire du langage de telle sorte que l'on peut réaliser assez simplement un interpréteur de Smalltalk en Smalltalk. Cet aspect méta-circulaire est également présent dans Lisp, et dans de nombreux langages interprétés. Dans le cas de Smalltalk, les métaclasses sont accessibles à l'utilisateur de façon naturelle, et permettent de nombreuses facilités de programmation. L'introduction du modèle méta-circulaire dans Smalltalk date de la première version du langage (Smalltalk-72). Les versions suivantes (Smalltalk-76 et Smalltalk-80) ont affiné, amélioré et enrichi ce modèle. D'autres langages inspirés de Smalltalk sont allés plus loin dans cette direction, notamment ObjVLisp et Self.

La meilleure preuve de la puissance de Smalltalk est l'ensemble des programmes réalisés en Smalltalk, et l'intense activité de recherche autour du langage, de ses concepts ou de ses langages dérivés. Parmi les réalisations, il faut noter l'environnement de programmation Smalltalk, réalisé à Xerox PARC et implémenté lui-même en Smalltalk. Cet environnement, qui inclut un système d'exploitation, est le premier environnement graphique a avoir été largement diffusé.

Smalltalk a inspiré de nombreux langages de programmation. Alors que Smalltalk est un langage natif, la plupart des langages qu'il a inspirés sont réalisés au-dessus de Lisp. Il s'avère que Lisp fournit une base au-dessus de laquelle il est facile de

construire des mécanismes d'objets. On peut néanmoins regretter que dans nombre de cas le langage sous-jacent reste accessible, ce qui donne à l'utilisateur deux modes de programmation largement incompatibles : le mode fonctionnel de Lisp et le mode de programmation par objets. Certaines incarnations de Lisp intègrent les notions d'objets de façon presque native, comme Le_Lisp et CLOS (CommonLisp Object System).

Autres familles, autres langages

Si les familles Simula et Smalltalk représentent une grande partie des langages à objets, il existe d'autres langages inclassables, d'autres familles en cours de formation.

Ainsi le langage Objective-C est un *langage hybride* qui intègre le langage C avec un langage à objets de type Smalltalk. Plutôt qu'une intégration, on peut parler d'une coexistence entre ces deux aspects, car on passe explicitement d'un univers à l'autre par des délimiteurs syntaxiques. Les entités d'un univers peuvent être transmises à l'autre, mais elles sont alors des objets opaques que l'on ne peut manipuler. L'intérêt de cette approche est de donner à l'utilisateur un environnement compilé et typé (composante C) et un environnement interprété non typé (composante Smalltalk). Parmi les logiciels réalisés en Objective-C, le plus connu est certainement Interface Builder, l'environnement de construction d'applications interactives de la machine NeXT.

Une autre famille est constituée par les *langages de prototypes*. Ces langages, au contraire des langages à objets traditionnels, ne sont pas fondés sur les notions de classes et d'objets, mais uniquement sur la notion de prototype. Un prototype peut avoir les caractéristiques d'une classe, ou d'un objet, ou bien des caractéristiques intermédiaires entre les deux. D'une certaine façon, ces langages poussent le concept des objets dans ses derniers retranchements et permettent d'explorer de nouveaux paradigmes de programmation.

Une dernière famille est constituée de langages parallèles appelés *langages d'acteurs*. Dans un programme écrit avec un tel langage, chaque objet, appelé acteur, est un processus qui s'exécute de façon autonome, émettant et recevant des messages d'autres acteurs. Lorsqu'un acteur envoie un message à un autre acteur, il peut continuer son activité, sans se soucier de ce qu'il advient du message. Le parallélisme est donc introduit par une simple modification du mode de communication entre les objets. Comparé aux modèles de parallélisme mis en œuvre dans d'autres langages, comme les tâches de Ada, la simplicité et l'élégance du modèle des acteurs est surprenante. Cela fait des langages d'acteurs un moyen privilégié d'exploration du parallélisme.

1.3 PLAN DU LIVRE

Ce livre a pour but de présenter les principes fondamentaux des langages à objets et les principales techniques de programmation par objets. Il ne prétend pas apprendre à programmer avec tel ou tel langage à objets. À cet effet, les exemples sont donnés dans un pseudo-langage à la syntaxe proche de Pascal. Ceci a pour but une présentation plus homogène des différents concepts.

Le prochain chapitre présente les principes généraux qui sont à la base de la programmation par objets. Les trois chapitres suivants présentent les grandes familles de langages à objets : les langages à objets typés, c'est-à-dire les langages de la famille de Simula (chapitre 3) ; les langages à objets non typés, et en particulier Smalltalk (chapitre 4) ; les langages de prototypes et les langages d'acteurs (chapitre 5). Le dernier chapitre présente un certain nombre de techniques usuelles de la programmation par objets et une ébauche de méthodologie.

Une connaissance des principes de base des langages de programmation en général et de Pascal en particulier est supposée dans l'ensemble de l'ouvrage. Les chapitres 3 et 4

sont indépendants. Les lecteurs qui préfèrent Lisp à Pascal peuvent lire le chapitre 4 avant le chapitre 3.