Avant-propos

Tout programme manipule un certain nombre d'*objets* (nombres, arbres, listes, tableaux, objets, etc.). Ces *objets programmatiques* correspondent – plus ou moins – aux *objets imaginaires* que le programmeur a en tête, qui sont des représentations du monde réel ou d'un univers de calcul tout aussi réel que l'autre.

La force de la programmation par objets est de proposer une définition des objets programmatiques qui leur permet de « coller » étroitement aux objets imaginaires.

Notons que nous devons distinguer entre les deux acceptions du mot *objet* : au sens technique de la programmation par objets (défini plus loin) et au sens ordinaire : objets du discours. Le charme de la programmation par objets est précisément dans la facilité avec laquelle on passe de l'un à l'autre[1].

Tout est objet

– L'idée fondamentale de la programmation par objets est de considérer systématiquement des entités complexes qui, pour chacun des « êtres » qui apparaissent dans le programme, regroupent *la structure de données* qui le décrit et *les procédures* qui permettent de le manipuler. Ces entités sont naturellement appelées les *objets* du programme.

La structure de chacune de ces entités est spécifiée par une *classe* : on dit alors que l'objet est une *instance* de sa classe.

– On peut passer sous silence le rôle de la classe et ne considérer que l'objet, sans s'interroger sur sa genèse. On parle alors d'*acteurs* [AGH 97] plutôt que d'objets. Les *frames* [MIN 85] forment une variété particulière d'acteurs compliqués pour satisfaire les besoins de l'intelligence artificielle.

1. Ce passage n'est pas nécessairement bidirectionnel : passer des langages aux programmes est toujours faisable en pratique, l'inverse se heurte à des questions mal comprises de *syntaxe* de langages de programmation.

La distinction entre classe et instance et la mise en œuvre explicite des classes est essentielle pour parler de programmation par objets.

– On pourrait aussi bien parler de *type* au lieu de classe ; c'est essentiellement une affaire de tradition.

Toutefois, une distinction importante se fait sur la nature des classes : sont-elles des objets comme les autres, ou sont-elles d'une essence différente de celle des objets ?

La tradition qui parle de types, notamment de *types abstraits*, considère que les types ne sont pas des objets.

En revanche, la programmation par objets *stricto sensu* postule que les classes sont des objets : *tout est objet*.

– La programmation par objets fonctionne donc systématiquement à deux niveaux : définition de classes (définissant le comportement partagé par les instances) et création d'instances (définissant ce qui distingue les différentes instances entre elles), alors que la programmation traditionnelle restreint et obscurcit cette démarche en imposant de séparer données et procédures.

Ce mécanisme intellectuel n'a rien de nouveau, nous le pratiquons constamment. Les mathématiques et le courant structuraliste nous ont habitués à la démarche en deux temps : « Appelons truc un machin qui. . . que . . . quoi . . . Clairement chose est un truc, etc. » En somme, il s'agit de définir ce dont on va parler avant d'en parler effectivement.

L'apport de la programmation par objets est de proposer des formalismes propres à réaliser concrètement cette démarche intellectuelle, permettant d'écrire *comme on pense*.

C'est la possibilité d'utiliser directement des ressources intellectuelles très répandues qui fait la puissance de la programmation par objets – et qui motive son succès. Elle permet à des programmeurs moins qualifiés d'attaquer des problèmes plus difficiles, de les résoudre en moins de temps et de produire des programmes de meilleure qualité.

En somme, programmer par objets, c'est tout bonnement programmer de façon naturelle. . .

Langages à objets

Tout langage favorisant le style de programmation par objets peut prétendre au titre (envié) de « langage à objets ». Actuellement, les langages à objets foisonnent. Voici un essai de classification.

1) L'ancêtre génial dont tous descendent (voir figure 1) est *Simula-67* [DAH 67], issu lui-même d'*Algol-60*, [PER 59] et [NAU 63]. Tout l'essentiel y est, y compris le

mode d'emploi des objets ! Comme son nom l'indique, ce langage fut conçu pour les applications de simulation de processus complexes.

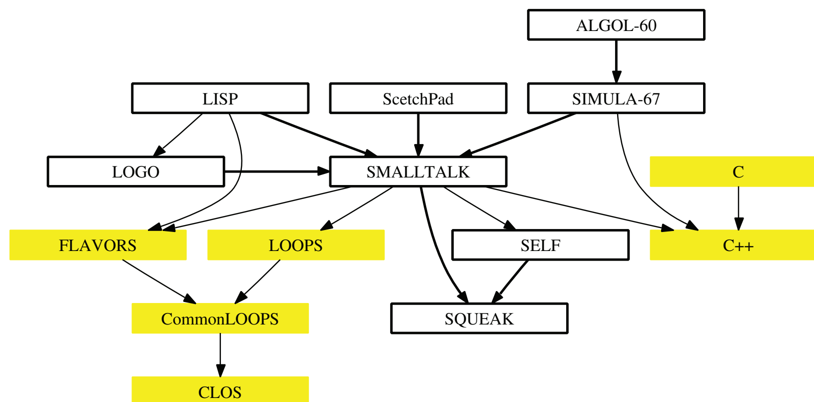

Figure 1. *Esquisse généalogique de Squeak*

On peut voir Simula comme une extension d'Algol-60, auquel sont ajoutés les concepts de classe (sous une forme plus complexe que la notion actuelle) et d'instance. C'est précisément pour pouvoir simuler *commodément* le monde réel que ces notions ont été introduites.

Simula contient aussi un puissant mécanisme de coroutinage qui a (presque) disparu chez ses successeurs.

2) Les extensions de langages impératifs classiques sont apparues à partir de 1983 environ, quand la mode de la programmation par objets a commencé à se répandre sous l'influence de SMALLTALK [GOL 83, GOL 84] et FLAVORS [CAN 82]. Ainsi, à partir du langage C [KER 88] ont été développés les langages C++ [STR 87] et OBJECTIVE-C [COX 87] et à partir de PASCAL [JEN 75] on a vu apparaître le langage OBJECT PASCAL, [JAC 87] et [SCH 86]. Il existe maintenant des extensions objet pour presque tous les goûts dans tous les langages impératifs, par exemple : *object-oriented Forth* [POU 87], *Object Fortran* [REE 91], ou encore *Object-Oriented Cobol* [CHA 96].

On peut les voir comme des versions simplifiées de Simula, bâties sur des substrats différents. Leur clientèle se recrute principalement dans le domaine du g'enie logiciel. Ces langages sont plus des langages à types abstraits que des langages à objets, et devraient se comparer à CLU [LIS 77] et à ADA [DOD 83].

3) Existent ensuite les extensions de LISP [MCC 62]. Il y en a toute une kyrielle : FLAVORS [MOO 80] de la machine Lisp Symbolics, LOOPS [BOB 86] de Xerox PARC et Interlisp, CEYX [HUL 83] et ALCYONE [HUL 85] issues de Le_Lisp, KOOL, [ALB 84] et [ALB 88], de Bull développé en Le_Lisp, OBJVLISP [BRI 87] comme

extension de Vlisp, CLOS, le COMMONLISP OBJECT SYSTEM [BOB 88], et une multitude d'autres (il en naît tous les jours).

Cette floraison est due au fait que la programmation par objets s'acclimate bien en Lisp, en raison de l'incroyable plasticité du langage, et de sa position « culturelle » comme langage principal de l'intelligence artificielle. En effet, ces systèmes sont très souvent à la base de logiciels d'intelligence artificielle complexes, visant la représentation de connaissances et l'écriture de systèmes experts.

Pour des raisons analogues, les extensions de PROLOG [GIA 85] vers les objets se multiplient : LAP [ILI 87], EMICAT, OBJLOG-II [DUG 88], TAO [TAK 83] et autres. Ajoutons à tout cela les langages de *script* plus récents tels que PYTHON [ROS 95], RUBY [MAT 01] ou encore OCAML [LER 07].

4) Enfin, un langage qui se suffit à lui-même, implémenté directement : SMALLTALK, de Xerox PARC. Son père spirituel est Alan Kay et sa machine FLEX [KAY 68], dont les concepts de bases ont été fortement influencés par le programme *SketchPad* de Ivan Sutherland [SUT 63] et le langage LOGO de Seymour Papert [PAP 80]. Il a été développé au Xerox Palo Alto Research Center, d'où est aussi sorti LOOPS.

La version actuelle est SMALLTALK-80. Elle a été précédée par SMALL-TALK-72 et SMALLTALK-76. Une très belle discussion sur l'histoire de SMALLTALK est donnée dans [KAY 93].

C'est un petit langage mais un gros système, très orienté vers la gestion d'un écran bit-map. Initialement, SMALLTALK ne tournait que sur les machines Xerox (1108, 1186), sur SUN et sur les ordinateurs Tektronix 4404 et 4406. Depuis sont apparues des versions pour à peu près tous les ordinateurs et tous les systèmes d'exploitation – il en existe même une version pour la *Playstation* de Sony.

SMALLTALK a exercé une influence profonde sur le développement de la programmation par objets, dont il offre l'exemple le plus achevé, et (à travers le MacIntosh de Apple, développé par des transfuges de l'équipe SMALLTALK originelle) sur toute l'informatique moderne. Sa simplicité conceptuelle et la richesse de son environnement justifient amplement son étude, même si l'on n'envisage pas de l'utiliser par la suite.

Des versions plus petites ont aussi été développées, comme par exemple : SMALLTALK-V par Digitalk ou Little Smalltalk [BUD 88] par Timothy Budd. Cette dernière est particulièrement utile pour se familiariser avec l'implémentation d'interprètes des langages objets. Mais considérons le dernier né de la famille SMALLTALK : SQUEAK [ING 97]. Il a été développé par une équipe centrée autour des concepteurs originaux de SMALLTALK, principalement Alan Kay, Dan Ingalls, Scott Wallace, Ted Kaehler, Jon Maloney, Andreas Raab et Michael Rueger. Il intègre les tout derniers développements des interfaces hommes/machines et il resimplifie la machine virtuelle SMALLTALK, qui – quand même – avait pris de l'âge. Il est distribué en licence *open-source* (allez voir sur www.squeak.org), et tourne sur quasiment tous les ordinateurs. C'est à travers ce langage que nous allons aborder la programmation par objets dans la suite de ce texte.