Introduction

L'hégémonie que la programmation par objets exerce aujourd'hui sur la quasi-totalité de l'informatique a quelque chose d'angoissant. Notamment pour ceux qui ont participé à son émergence, qui ont contribué à son ascension, et qui par conséquent sont bien conscients de ses défauts et de ses limitations intrinsèques. Il est donc naturel que le pavé jeté dans la mare par Richard Gabriel « Les objets ont échoué »[1] ait fourni le thème du colloque saluant le départ à la retraite de Jean-François Perrot, à Jussieu le 17 octobre 2003. Le présent numéro spécial contient les textes rédigés par les quatre orateurs principaux, qui ont bien voulu développer par écrit leurs communications, ainsi que deux autres articles qui viennent les compléter.

Le but de cette brève introduction est d'expliciter les liens qui unissent ces six contributions au thème général du colloque.

La programmation par objets, comme les autres styles de programmation, repose sur une avancée technique. De même que la programmation procédurale ne serait pas pensable sans la double gestion des adresses de retour et des environnements dans une pile unique, que la programmation fonctionnelle a pour pilier la représentation des fonctions comme des fermetures, et que la programmation logique tient à la possibilité de manipuler les points de choix, de même la programmation par objets a sa racine dans l'usage conjoint de deux indirections. Lors de l'interprétation d'une transmission (envoi d'un message à un objet récepteur), deux aiguillages ont lieu :

(1) celui qui considère que le nom de la méthode invoquée désigne un corps de procédure à trouver dans la classe du récepteur (c'est « l'indirection bien placée » dont parle François Pachet dans son « point de vue Ikea ») ;

(2) celui qui, lors de l'exécution de ce corps de procédure (au demeurant parfaitement banal), pousse les variables libres à compter leurs adresses à partir de l'adresse du récepteur, fournie comme un argument implicite à l'appel.

Soulignons une fois de plus l'exceptionnelle fécondité de ce double mécanisme. Le point essentiel est l'interprétation des classes comme représentant des concepts, par opposition aux objets qui représentent des instances de concepts. Cette interprétation met au premier plan la dualité extension/intension. La classe telle

1. En version originale : « Objects have failed ». Ce débat à OOPSLA'2002 à Seattle, est documenté par Richard P. Gabriel sur son site web :
http://www.dreamsongs.com/Essays.html et
http://dreamsongs.com/ObjectsHaveFailedNarrative.html

qu'elle est écrite, compilée, sauvegardée sur le disque, est une pure intension – en particulier, elle est une entité finie. Elle est automatiquement considérée comme représentant l'ensemble de ses instances – qui est potentiellement infini. Ce changement de point de vue apparaît en pleine lumière dans l'interprétation de l'héritage, où l'accroissement de l'intension est interprété comme une restriction de l'extension : si on ajoute des traits au discours intensionnel d'une classe *A* (attributs ou méthodes, les deux seules entités exprimables dans nos langages) pour obtenir une classe étendue *B*, on postule que les instances de *B*, qui bénéficient de ces adjonctions, appartiennent toujours à la classe *A*, ce qui entraîne l'inclusion des extensions et permet d'attribuer à *B* la qualité de sous-classe de *A*.

On sait que cette interprétation relève de la pétition de principe. La seule justification logique qu'on peut invoquer en sa faveur est que les instances de la classe étendue *B* se comportent *à toutes fins utiles*, comme les instances de la classe primitive *A*. En disant cela, on entend que leurs comportements sont identiques sauf en cas d'erreur pour cause de non-compréhension (le fameux « message not understood » de Smalltalk) : plus savante, l'instance de *B* comprend plus de messages que l'instance de *A* – mais pour ceux qui sont compris des deux instances, l'interprétation est la même. On retrouve ainsi l'ordre défini naturellement sur les fonctions partielles, mais dès la première redéfinition de méthode cette identité de comportement disparaît. Au mépris de toute justification, et en abandonnant tout espoir de vérification formelle, on continue néanmoins à parler de sous-classes.

Et pourtant, quelle commodité ! Malgré toutes les tentatives pour le réformer, on n'a toujours pas trouvé mieux, ni moins cher. Et c'est l'héritage simple, celui qui traduit le plus directement la mécanique implémentatoire, qui reste le plus utilisé – en dépit des nombreux travaux sur l'héritage multiple, apparemment plus séduisant sur le plan des idées : manque de coïncidence, comme dirait François Pachet.

Bien évidemment une mécanique aussi légère ne peut que s'écrouler sous le poids des projets démesurés que son propre succès a rendu pensables. *Inde irae*. Reste à faire mieux.

Une première réponse est de proposer de nouveaux terrains de jeux à la programmation par objets, et de passer du secteur tertiaire à l'industrie du divertissement[2]. Francis Wolinski et Jean-Pierre Legrand décrivent la fortune des objets dans la banque et l'assurance. Après eux François Pachet part d'une critique radicale pour aller vers des objets difficiles à construire (alors que ceux du tertiaire « étaient déjà là »). Il développe ici les vues qu'il avait présentées dans sa conférence invitée à LMO'03 (Pachet, 2003).

Une autre voie est de remettre en cause les classes dans leur rôle de représentation des concepts. Insistons : quel modèle grossier nous offrent les classes ! Un jeu de variables, une batterie de procédures (utilisant ces variables,

2. Alias *entertainment* : voyez par exemple la récente revue *Computers in Entertainment*, publiée par l'ACM.

grâce à la 2e indirection), et c'est tout ! Comme il est étonnant que cet outil rudimentaire s'avère si efficace en pratique. Encore plus étonnante est la confiance aveugle que les programmeurs peuvent avoir en un outil si imparfait : voyez l'approche dite *Naked Objects*[3] – les *objets tout nus* – qui veut faire l'économie de l'interface homme-machine au profit d'une « présentation directe » des objets métier. Oui, en vérité les objets sont nus, mais pas au sens où l'entendent les thuriféraires des *Naked Objects*...

Parmi les non-dits qu'il faut remettre en cause, figure la relation entre classes et types. L'interprétation extensionnelle des classes entre en résonance avec les habitudes de typage fortement enracinées chez les programmeurs : d'où la tentation de traiter les classes comme des types, tentation si forte qu'elle est devenue un dogme. Même un langage non typé comme Smalltalk éprouve le besoin de se déguiser en « typé dynamiquement ». Hélas, on sait depuis les propos dissonants de Cook (Cook *et al*., 1989) que cette démarche conduit à un échec dès qu'on veut prendre le typage au sérieux.

Dans ces conditions, la critique de l'interprétation des classes comme concepts peut suivre deux voies : celle qui tente une modélisation directe des concepts, et celle qui dissocie les classes et les types. La première est représentée ici par Amedeo Napoli et ses amis, la seconde par Thérèse Hardin et Renaud Rioboo.

Pour prendre la mesure des gouffres que nous franchissons allègrement (*for Fools rush in where Angels fear to tread*), demandons-nous comment une classe désigne un concept.

Elle le fait par voie constructive, en disant comment sont faites les instances dudit concept (de la manière la plus directe, en donnant le plan de leur implantation en mémoire), et ce que font ces instances (en définissant les méthodes sur la base de l'implantation). C'est là une démarche d'ingénieur, de constructeur, compréhensible par tout un chacun en raison de notre héritage culturel d'*homo faber*.

Les logiciens privilégient une toute autre approche, qu'on peut qualifier d'extérieure, fondée sur un calcul des concepts qui permet certes de raisonner, mais ne vous dit jamais comment sont faites les instances. Peut-on passer d'un point de vue à l'autre ? Oui, par exemple grâce aux treillis de Galois, fort étudiés dans notre communauté, de Montréal à Montpellier. Ou en trouvant des algorithmes de classification comme ceux des systèmes de RCO dont nous parle la bande des cinq. Les logiques de description ont poussé l'approche « extérieure » jusqu'aux frontières du calculable. On a pu croire cette voie sans issue, et voici que le web sémantique les remet en selle.

Plus généralement, la déferlante XML vient modifier notre regard sur les objets, comme l'analysent (Euzenat *et al.*, 2003). Les « objets XML » sont de pures données structurées, où les méthodes n'ont plus de place. Ils s'adaptent donc bien aux systèmes de représentation des connaissances. En effet, les méthodes, ces

3. www.nakedobjects.org

entités essentielles pour la programmation effective, se révèlent d'incontrôlables empêcheuses de raisonner. On trouvera un écho de ce problème de fond dans l'article d'Amedeo *et al.* (section 2.4).

Le problème serait-il mal posé ? L'exemple de l'évolution de la systématique et des notions de genre et d'espèce (Lecointre *et al.*, 2001) incite à la prudence. D'ailleurs, pourquoi un programmeur se mêle-t-il d'un débat pour philosophes ? *Sutor, ne supra crepidam...*

Occupons-nous donc d'un problème bien posé : les structures algébriques, les fameux *Groupe, Anneau, Corps,* leurs parents et alliés nous lancent un défi. Comment les représenter de manière opérationnelle, tout en respectant les habitudes des mathématiciens ? Essayez donc, armé de votre langage favori... Thérèse et Renaud ont relevé le défi. Ils le font dans une perspective précise, celle du calcul formel, et ils utilisent un langage qui traite le typage sérieusement, à savoir O'Caml. Leur contribution au débat sur les objets comporte donc à la fois une méthode et un résultat (la notion d'espèce).

Un troisième genre de réponse au constat de Gabriel consiste à regarder vers l'avenir, dans la direction « après les objets ». C'est ce que font les Nantais, Jean Bézivin et Pierre Cointe avec son équipe.

Les mécanismes fondamentaux rappelés ci-dessus, puissants mais simples, montrent leurs limites à bien d'autres points de vue, notamment à l'épreuve des nouveaux enjeux de l'informatique répartie à grande échelle, qui révolutionne notre manière de concevoir et de réutiliser les programmes. On constate :

(1) L'encapsulation apportée par les objets n'est réalisée qu'à moitié. Les méthodes offertes par l'objet sont bien visibles par l'interface d'appel. En revanche, les appels vers d'autres objets restent encapsulés et ne sont pas visibles de l'extérieur (en d'autres termes, ils ne possèdent pas d'interface de sortie), ce qui rend difficile leur reconnexion arbitraire à d'autres objets.

(2) Les objets, bien qu'ils soient de bonnes entités structurelles, sont de piètres entités de déploiement, car ils sont orphelins en l'absence du code de leur classe.

(3) La réutilisation des programmes (en général) est malaisée, car ils sont en général beaucoup trop liés à une implémentation particulière (version du langage, système d'exploitation sous-jacent, etc.).

La réponse aux deux premiers constats est développée par Pierre Cointe et ses collègues. Ils privilégient la notion de composant logiciel, comme alternative ou évolution de la notion d'objet. L'idée intuitive est de considérer le logiciel comme un assemblage de composants génériques, à l'image des composants électroniques. Les composants logiciels conçus pour être autonomes, *i.e.* prêts au déploiement et à l'utilisation (2), proposent ainsi une symétrie retrouvée entre interface d'entrée et interface de sortie (1), reliées par des entités de nature différente, appelées connecteurs (l'analogue des pistes des circuits imprimés). Mais la description des

conditions de fonctionnement des composants électroniques (tension, etc.) est beaucoup plus aisée que pour le logiciel. Ceci conduit à tenter de modéliser et caractériser ces contextes d'utilisation (par exemple, au niveau du contexte englobant *via* les conteneurs des *Enterprise Java Beans*, au niveau des spécifications *via* des contrats non fonctionnels).

Cependant, l'assemblage de composants repose sur une vision essentiellement hiérarchique et fonctionnelle de la décomposition. Elle recouvre mal des préoccupations « transversales » (souvent appelées propriétés non fonctionnelles), telles que la sécurité, la distribution, etc., qui deviennent très importantes voire fondamentales à l'heure de la programmation répartie à grande échelle. Pierre Cointe et son équipe mettent en avant la notion d'aspect comme moyen complémentaire aux composants pour modéliser de telles préoccupations transversales. Les aspects sont décrits plus ou moins indépendamment des différents composants fonctionnels, puis le code complet est composé *via* des « tisseurs d'aspects ». Cette démarche à la fois hiérarchique et transversale est en quelque sorte l'analogue d'une organisation (d'entreprise, d'institut de recherche…) « en trame » incluant départements fonctionnels et services transversaux.

La vision plus radicale en réponse aux troisième constat (3), développée par Jean Bézivin, est de considérer que la racine des problèmes tient à la programmation explicite, qui laisse le logiciel trop dépendant des contingences matérielles (versions, système sous-jacent…). Il défend alors l'approche « Model Driven Architecture (MDA) » dans laquelle on troque des programmes contre des modèles (au sens des modèles exprimés en UML), décrivant les logiciels de manière indépendante des plates-formes de réalisation (dont les caractéristiques cibles sont elles-mêmes décrites *via* des modèles). Une succession de transformation et de projection de modèles est envisagée pour obtenir au final des programmes exécutables. L'approche « tout modèle » pose la question de la possibilité d'exprimer les mécanismes d'implémentation en termes de modèles et également la question du pouvoir d'expression des règles de transformation.

L'interopérabilité des modèles (*via* XML comme métalangage de base et l'architecture de métamodèles Meta-Object Facility (MOF), une forme de transposition des architectures à métaclasses) succède ainsi à l'interopérabilité des objets (à la CORBA), et la complète. L'approche MDA se situe dans un contexte renouvelé, où les échanges à travers le réseau jouent un rôle important. D'autres travaux sont en cours pour métamodéliser et faire interopérer les étapes et les outils du processus même de conception (choix de métamodèles, échanges, documentation, transformation, etc.).

Et maintenant, la parole est aux auteurs !

Jean-François Perrot, LIP6, Paris

Jean-Pierre Briot, LIP6, Paris

Bibliographie

Cook W.R., Hill W., Canning P.S., "Inheritance is not subtyping", *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'89)*, 1989, p. 125-135.

Euzenat J., Napoli A., Baget J.-F., « XML et les objets (Objectif XML) », *Numéro spécial « XML et les objets », L'Objet*, Lavoisier, vol. 9, n° 3, 2003, p. 11-36.

Lecointre G., Le Guyader H., *Classification phylogénétique du vivant*, Belin, 2001.

Pachet F., « Objets et divertissement », *Numéro spécial « LMO'03 », L'Objet*, Lavoisier, vol. 9, n° 1-2, 2003, p. 13.