INTRODUCTION AUX LANGAGES ORIENTÉS OBJET

Daniel MILLOT

Cet article donne une présentation générale des langages orientés objet, un aperçu de Smalltalk-80, quelques indications rapides sur les stratégies d'implémentation et pose la question de l'intérêt pédagogique de ces langages.

INTRODUCTION

La tortue Logo réagit à un certain nombre de messages prédéfinis que peut lui adresser l'utilisateur (avance, lève-crayon, etc.) et possède un certain nombre de caractéristiques modifiables (position, cap, couleur...) ; c'est un objet logiciel pour lequel l'utilisateur peut définir son propre langage d'interaction (ses procédures) et déclencher une exécution par invocation de l' un des motsclés qu'il aura introduits. Cette démarche est généralisée dans les langages orientés objet (LOO).

La programmation orientée objet (POO) prend ses origines [1] dans la simulation sur ordinateur de phénomènes réels, l'idée étant d'associer à chaque objet réel un objet logiciel, boîte noire capable de réagir à un certain nombre de sollicitations externes ; l'exécution d'un programme donne lieu à l'installation d'un système d'objets et à des échanges de messages entre eux.

Le monde de la POO est un univers uniformément peuplé d'objets qui communiquent entre eux par envoi de messages. On peut se représenter ces objets comme de petits ordinateurs autonomes disposant chacun d'une puissance de calcul et de la possibilité de stocker des données.

Un modèle d'objet étant défini, autant d'exemplaires de ce modèle que nécessaires peuvent être créés en cours d'exécution ; on parle de classe (le modèle) et d'instances (les exemplaires). Programmer dans un LOO, c'est :

- créer des classes,

- créer des objets instances de ces classes,

- spécifier une suite de messages échangés entre ces objets.

Les classes sont organisées en une hiérarchie : les objets d'une sous-classe héritent des caractéristiques des sur-classes, ce qui facilite la réutilisation de logiciel.

Grâce au cycle test/modification très court qu'elle permet, la POO convient bien à l'élaboration rapide de prototypes. Ses atouts (modularité, réutilisation, structuration, flexibilité, facilité de développement...) expliquent l'effet de mode tout à fait mérité dont elle bénéficie actuellement; on lui doit un nombre important de réalisations comme les systèmes de fenêtrage, le Macintosh, etc., et Hypercard est l'un de ses derniers avatars.

I. PHILOSOPHIE DES LANGAGES ORIENTÉS OBJETS

Elle tient pour l'essentiel dans les quatre mots : objet, message, classe, héritage.

I.1. Les objets

Les objets sont des entités dynamiques qui apparaissent, évoluent et peuvent éventuellement disparaître au cours de l'exécution d'un programme; passifs lors de leur création, ils ne sont activés que par réception de messages, et retournent à l'état passif après l'exécution du code correspondant au message reçu.

Chaque objet dispose de données privées (i.e. invisibles de l'extérieur, accessibles uniquement par l'objet) et de procédures pour manipuler ces données. La seule façon d'inspecter ou de modifier l'état interne d'un objet est de lui envoyer l'un des messages prévus par l'interface de communications déclarée au niveau de sa classe ; l'objet devient alors actif et exécute une de ses procédures (appelées méthodes), qui peut accéder à ses données (appelées variables d'instance).

L'implémentation du comportement des objets d'une classe (le code des méthodes) est inaccessible aux autres objets ; la structure des objets d'une classe peut changer sans affecter leurs interactions avec les instances des autres classes si l'on conserve la même interface de communication.

Tous les objets sont traités de la même façon : qu'ils soient primitifs (entiers, booléens...) ou introduits par l'utilisateur (constante, image écran, fenêtre, processus...), ils ont le même statut.

Un objet peut occuper une place mémoire importante, et il est nécessaire de pouvoir récupérer la mémoire occupée par des objets devenus inaccessibles ; c'est le rôle du garbage collector que l'on trouve dans les systèmes Lisp ou Logo pour la récupération de cellules (instruction recycle de Logo, par exemple).

I.2. Les classes

Très naturellement lorsqu'on dispose d'un grand nombre d'objets, il est commode de les classer selon leurs propriétés, regroupant des objets présentant des caractères communs. C'est ce que permettent les LOO avec le concept de classe: les objets d'une classe donnée ont tous même structure (même nombre de variables d'instance), comprennent tous les mêmes messages (même interface de communications) et disposent du même code pour les différentes méthodes à leur disposition. Chaque méthode porte un nom (appelé sélecteur) qui permet de la désigner.

La classe est comme le moule à partir duquel seront créés des objets (instances de la classe). Elle ne joue cependant pas le rôle usuel d'un type : un nom peut désigner un objet de n'importe quelle classe, et un même nom peut désigner successivement des objets de classes différentes au cours d'une même exécution.

I.3. Les messages

Toute exécution consiste en des envois de messages. L'instruction O m désigne l'envoi du message m à l'objet O ; lorsqu'il reçoit ce message, O exécute la méthode de sélecteur m de sa classe (c'est-à-dire qu'il envoie à d'autres objets les messages prévus par cette méthode) avant de retourner sa réponse à l'expéditeur du message. Ainsi, l'envoi d'un message provoque généralement une cascade de transmissions et il faut évidemment que cette récursion s'arrête si l'on veut qu'il se passe effectivement quelque chose et qu'une réponse soit apportée au premier message expédié ! C'est là qu'interviennent les objets primitifs, capables de répondre directement sans envoyer de nouveaux messages ; par exemple, l'envoi du message **factorial** à l'objet **5** (instruction qui s'écrit **5 factorial**) provoque le retour de 120, et lorsque l'objet **7** reçoit le message **+** avec comme argument **2**, il retourne immédiatement 9 à l'expéditeur.

La liaison d'un sélecteur de message avec le code correspondant est dynamique. Elle n'est possible que lorsque la classe du receveur est déterminée, donc comme il n'y a pas de typage, seulement au moment de l'exécution. Le même message peut donner lieu à des interprétations sans aucun rapport entre elles selon la classe de l'objet qui le reçoit. Ce polymorphisme qui se résoud à l'exécution est beaucoup plus flexible que la généricité d'Ada dans la mesure où chaque objet peut avoir sa propre interprétation du même message. L'exemple, cité par S. Cook, de l'impression d'un tableau constitué d'éléments de types différents montre comment ceci permet d'introduire de nouveaux types d'objets sans modifier la déclaration du tableau ni retoucher le programme existant ; dans un langage classique procédural, l'impression du tableau A est déclenchée par une instruction for i:=1 to MAX do print(A[i]);

Le type des éléments intervient dans la procédure print sous la forme d'un case procédure print (truc : polytype);

```
begin
  case type(truc) of
        inttype: printint(truc);
        chartype: printchar(truc);
        stringtype: printstring(truc);
        otherwise: error("non imprimable");
  endcase
endproc;
```

où **polytype** désigne l'union de tous les types d'objets imprimables.

Si l'on souhaite incorporer un élément d'un type nouveau dans le tableau A, les définitions de polytype, de la procédure print, ainsi que de toutes les procédures qui manipulent les éléments de A devront être modifiées ; le code correspondant a toute chance d'être disséminé, ce qui peut amener à une recompilation complète. Le compilateur lie en effet le nom print à la procédure print dès qu'il rencontre l'instruction print(A[i]).

Au contraire, dans un LOO, les liaisons sont dynamiques. Au lieu d'avoir une unique procédure print, une méthode de sélecteur print est

attachée à chaque classe, donnant à chaque objet la capacité de répondre au message print; la transmission A[i] print déclenche l'impression de l'élément A[i] quelle que soit sa classe, le nom print n'étant lié au code approprié qu'à la réception du message, selon la classe du receveur.

On peut ainsi introduire de nouveaux types d'objets sans retoucher le programme existant; il suffit de munir la classe de ces nouveaux objets de méthodes pour tous les messages envoyés aux éléments de A.

I.4. L'héritage

Les LOO permettent la réutilisation du code non seulement par le regroupement en classes d'objets similaires, la classe détenant le code utilisé par ses différentes instances, mais aussi par la dérivation de sous-classes à partir de classes existantes. En effet, une nouvelle classe apparaît souvent comme une spécialisation d'une autre classe et il est intéressant d'organiser les classes en une hiérarchie, de la même manière que les biologistes utilisent des classifications pour structurer les familles d'êtres vivants du règne animal ou végétal.

Les objets d'une classe disposent de toutes les variables d'instance ainsi que de toutes les méthodes prévues par les différentes sur-classes de la branche correspondante. Les sélecteurs de méthodes peuvent être surchargés, i.e. être réutilisés au niveau d'une sous-classe et c'est la définition rencontrée la première en remontant la chaîne d'héritage[2] qui est utilisée. Le développement d'un logiciel se fait par dérivation de nouvelles classes à partir d'un système de classes pré-existant.

Pour résumer, on peut dire qu'en dehors de l'uniformité du modèle (tout est objet, et l'envoi de messages est l'unique structure de contrôle), les LOO sont caractérisés par les mécanismes :

- d'instanciation (avec la dichotomie classe/instance),
- d'héritage à travers la hiérarchie des classes.

II. L'EXEMPLE DE SMALLTALK-80

II.1 Les principes

Smalltalk-80, qui est sans doute la réalisation la plus pure d'un langage entièrement conçu avec des objets, repose sur quatre axiomes

(AI) Toute entité est un objet

En Smalltalk, tout est objet : nombres, structures de données, processus, fichiers sur disque, éditeur de texte, compilateur, scheduler... y compris les classes elles-mêmes. Les classes sont donc des objets générateurs, et on enverra en particulier un message à une classe pour lui demander de créer une instance.

(A2) Tout objet est instance d'une classe

Cela est aussi valable pour les classes : chaque classe est l'unique instance d'une métaclasse qui détient les méthodes de la classe (en particulier pour la création des instances). Le système Smalltalk-80 comporte un nombre très important de classes système pré-existantes (qui constituent ce qu'on appelle l'image virtuelle) ; l'utilisateur définit une nouvelle classe en précisant sa position dans l'arbre d'héritage, le nom des variables d'instance rajoutées par rapport à sa surclasse, et les méthodes supplémentaires que pourront exécuter ses objets.

(A3) Toute classe est sous-classe d'une autre classe

Une classe étant donnée, on peut définir une sous-classe en ajoutant des fonctionnalités (sous forme de nouvelles méthodes) ou de la mémoire (des variables d'instance supplémentaires) pour les objets qu'elle instanciera. Les variables et méthodes à la disposition d'un objet ne sont donc pas toutes déclarées au niveau de sa classe : les instances de la sous-classe héritent de toutes les variables d'instance et de toutes les méthodes des sur-classes. L'héritage simple structure la hiérarchie des classes en arbre dont la racine est la classe Object ; cette classe contient les méthodes définissant les comportements les plus élémentaires de tous les objets du système, dont une méthode doesNotUnderstand destinée à récupérer les messages reçus mais non compris par les objets (i.e. non prévus par leur interface de communication).

La figure 1, sur laquelle le nom des classes système est souligné, montre une partie de l'arbre d'héritage de Smalltalk-80 (la flèche se lit « hérite de ... »).

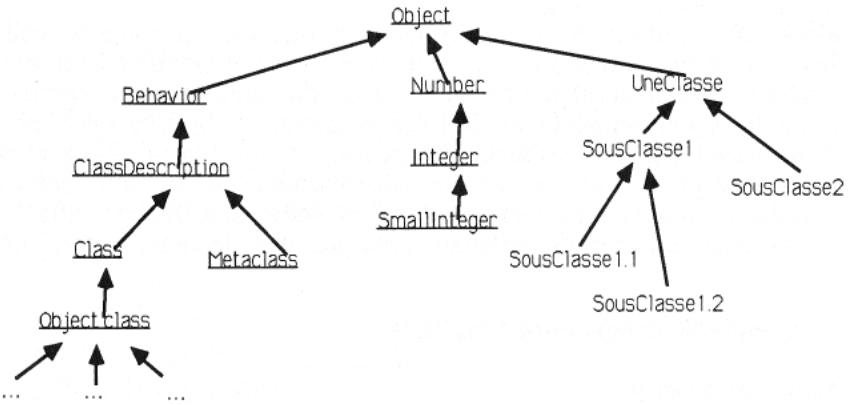

fig. 1 - Arbre d'héritage

(A4) Toute expression est une transmission

Un objet est activé à la réception d'un message; la méthode correspondant au sélecteur du message est tout d'abord recherchée dans la classe de l'objet. Si elle est trouvée, elle est exécutée, sinon la recherche se poursuit dans la super-classe du receveur, et ainsi de suite en remontant l'arbre hiérarchique, éventuellement jusqu'à la classe Object. La liaison entre message et méthode est entièrement dynamique. La méthode déclenchée est la première, correspondant au sélecteur, trouvée lors de l'exploration de l'arbre d'héritage. Il est par ailleurs possible de démarrer la recherche (appelée look-up) à un certain niveau de l'arbre (en utilisant super), ou qu'un objet s'envoie un message à lui-même (avec self).

Un message peut comporter des arguments, et une méthode peut utiliser des variables temporaires, n'existant que pendant son exécution. Tout message retourne une valeur qui désigne un objet (pointeur). Par exemple, l'évaluation de l'expression f ** Form new provoque l'affectation de f avec une instance de la classe Form créée par cette dernière en réponse au message new.

L'exécution d'une méthode se termine donc par le retour explicite d'une valeur (spécifié sous la forme** valeur) ou, lorsque toutes les expressions de la méthode sont évaluées et qu'aucune valeur à retourner n'est précisée, par le retour de self, qui désigne l'objet destinataire du message.

Lors de l'envoi d'un message, l'expéditeur est suspendu jusqu'au retour de la valeur de l'expression dénotant la transmission ; la

sémantique de l'envoi de message est donc celle de l'appel de procédure à distance.

II.2. L'environnement Smalltalk

L'interface utilisateur est une composante fondamentale du système Smalltalk, et nécessite un écran bitmap et une souris (Macintosh, poste de travail Tektronic, station Sun, etc.) ; elle se présente sous la forme d'un environnement interactif qui permet la navigation à travers l'ensemble des classes système (plus de 200) à l'aide d'un browser (fenêtre spéciale). Grâce à de multiples fenêtres et menus déroulants, l'utilisateur a la possibilité de retrouver toute information sur les méthodes existantes, l'utilisation des variables d'instance, la définition des classes ou leur hiérarchie ; il est entièrement pris en charge pour la création de nouvelles classes.

Le développement d'une application est totalement incrémental ; toute nouvelle méthode est immédiatement compilée, et peut être testée et modifiée à volonté, sans attendre la mise au point ou l'écriture du reste de l'application. L'environnement Smalltalk donne aussi la possibilité d'inspecter un objet, de modifier la valeur d'une variable puis de reprendre l'exécution, de faire du pas à pas, et en cas d'erreur (réception d'un message ne correspondant pas à l'interface du receveur et déclenchement de la méthode doesNotUnderstand), d'examiner les différents messages en cours de traitement pour découvrir la cause du dysfonctionnement.

II.3. Un exemple de programme Smalltalk

Il s'agit de la transcription Smalltalk de l'algorithme évoqué par Michel Canal dans son article sur Ada (Bulletin n° 47 de sept 87) permettant de déterminer la suite des nombres premiers.

Trois classes sont nécessaires : Générateur, Filtre et FlotDeSortie. Un objet de la classe Générateur produit 2 puis le flot des naturels impairs (c'est suffisant) en direction d'une série d'objets Filtre, chacun d'eux étant caractérisé par un diviseur premier dont il est chargé de prélever les multiples. Un objet de la classe FlotDeSortie permet l'édition des résultats. Le début de l'exécution du programme est représenté par la figure 2.

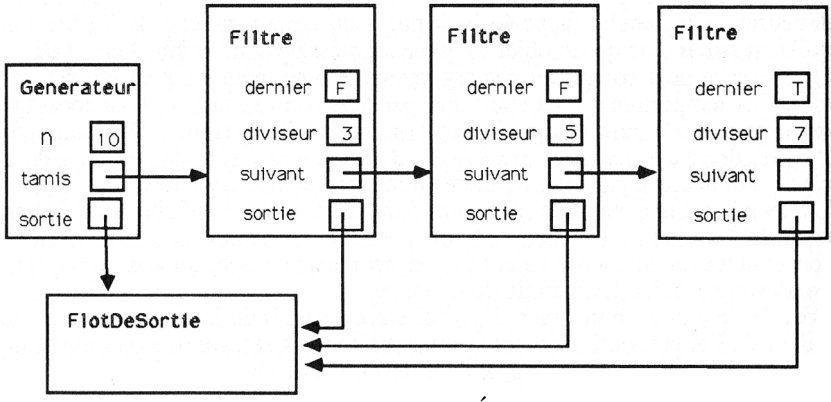

fig. 2 - Crible d'Ératosthène

Le code de la classe Générateur est donné ci-dessous (les sélecteurs de message prédéfinis ont été soulignés ; à noter les deux points (:) utilisés pour annoncer un argument, ainsi que la majuscule initiale des noms de classe) :

```
Object subclass: #Générateur
instanceVariableNames: "n sortie tamis"
Instance methods
init
  sortie ← FlotDeSortie new
produitjusqua: nb
| m |
  sortie nextPutAll: (nb ArintStringBase: 10) ; nextPutAll: " : "cr.
  nb <= 3 ifTrue: sortie nextPutAll: "Faut pas pousser!"]
  ifFalse:[sortie nextPutAll: "2" ; tab.
  tamis ← Filtre premier: 3 vers: sortie.
  n ← 3.
  m ← nb-1.

[n < m] whileTrue: [n <- n + 2. tamis
      aFiltrer: n]].
  ↑ sortie contents
```

Class methods

new

|g|

g ← super new.

↑g init

Le code de la classe Filtre s'écrit Object subclass: #Filtre

instanceVariableNames: "diviseur dernier suivant sortie "

Instance methods

init: unEntier sortie: uneSortie

diviseur ← unEntier.

dernier ← true.

sortie ← uneSortie.

sortie publie: diviseur

aFiltrer: unEntier

unEntier\\diviseur = 0 ifFalse:

[dernier ifTrue: [dernier ← false.

suivant ← Filtre premier: unEntier vers: sortie.

ifFalse: [suivant aFiltrer: unEntier]]

Class methods

premier unEntier vers: uneSortie |f|

f↑ super new.

f init: unEntier sortie: uneSortie.

↑f

Enfin le code de la classe FlotDeSortie est donné par WriteStream subclass: #FlotDeSortie

instance

```
methods
init

  self cr ; cr ; nextPutAll: "Nombres premiers

jusqu'"a " publie: unEntier

  self nextPutAll: (unEntier

printStringBase: 10); tab Class methods

  new

  |e|

  e ←self on: (String new: 10).

  ↑e init
```

L'exécution du programme pour déterminer les nombres premiers jusqu'à 1000 est déclenchée par l'évaluation des instructions suivantes

```
|g|

g ↑ Generateur new.

g produitJusqua: 1000.
```

III. DIFFÉRENTES TECHNIQUES D'IMPLÉMENTATION DES LOO

III.1. À l'aide d'une machine virtuelle

Le système Smalltalk-80 est conçu autour d'une machine virtuelle capable d'exécuter un code intermédiaire (les bytecodes) ; les méthodes sont compilées en bytecodes par le compilateur au fur et à mesure de leur saisie et sont alors exécutables par l'interpréteur.

III.2. Avec un préprocesseur

Afin de permettre une approche objet, une couche objet a été ajoutée à des langages comme Pascal ou C sous la forme d'un pré-processeur (Pascal-Object, C++, Objective C).

III.3. Au-dessus de Lisp

Les fermetures Lisp correspondent assez naturellement au concept d'objet; la transmission de messages est alors traitée par l'évaluateur comme une application fonctionnelle. De nombreux LOO ont donc été construits au-dessus de Lisp: Flavors, avec un mécanisme d'héritage multiple et de combinaison de méthodes, LOOPS et LORE avec un héritage multiple également, LRO2, Objvlisp, AB CL...

IV. INTERNET PÉDAGOGIQUE DES LOO ?

La tortue Logo se comporte manifestement comme un objet qui réagit à un certain nombre de messages prédéfinis (ou que l'utilisateur a introduits) par l'exécution d'un code prédéfini (ou de procédures déclarées par l'utilisateur). L'interprète Logo peut lui-aussi être considéré comme un objet, et le traitement d'erreurs Logo (« comment faire ... ») se présente sous une forme qui rappelle beaucoup le doesNotUnderstand de Smalltalk.

L'intérêt pédagogique de Logo n'est plus à démontrer, et tout donne à penser que cet intérêt vient du «côté objet» de Logo ; la construction progressive de la solution Logo d'un problème par briques successsives que l'on peut tester au fur et à mesure rappelle tout à fait l'élaboration dans un LOO de la définition d'une classe correspondant au comportement souhaité. Les LOO permettent très naturellement de créer des objets, de les animer, de modifier leurs caractéristiques et d'étudier immédiatement l'effet obtenu. Nul doute qu'ils se prêteraient bien au développement de micro-mondes. Par ailleurs, les possibilités de visualisation graphique et la désignation à l'aide de la souris donnent un côté concret et manipulatoire à l'utilisation d'environnements comme celui de Smalltalk qui laisse penser que de jeunes enfants pourraient tirer partie d'une utilisation raisonnable de ce type d'outils. Et que dire de la possibilité de créer sans tout réinventer ? C'est tout de même plus naturel que la démarche la plus répandue en programmation qui consiste à réécrire un autre programme pour chaque nouveau problème sans pouvoir véritablement réutiliser ce qui a été fait précédemment !

Alors, le succès de Logo à l'école préfigure-t-il celui des formidables outils de simulation que sont les LOO ? Peut-être. Faut-il donc se pencher sur de nouveaux langages qui ne seront pas plus la panacée que d'autres ? La question comporte en elle-même sa réponse. Mais il est indéniable qu'il y a en germe dans les LOO ce que pourrait être un

laboratoire de simulation que chacun pourrait utiliser selon ses compétences : utilisation directe du système tel qu'il est, personnalisation de celui-ci pour la création d'un environnement sur mesure, etc.

Daniel MILLOT
Formateur Académie de Dijon

BIBLIOGRAPHIE

Pour ceux qui souhaitent en savoir plus sur - *les LOO en général*

Les Langages Orientés Objets (Concepts, Langages et Applications), C. Bailly, J.F. Challine, P. Gloess, H. Ferri, B. Marchesin (Cepadues-Editions) Languages and object-oriented programming, S. Cook dans Software Engineering Journal (mars 86)

Object Oriented Languages, thème d'un dossier de huit articles différents dans le numéro d'août 86 du magazine Byte

Object-oriented programming : themes and variations, M. Stefik et D.Bobrow dans AI Magazine (janvier 86)

- Smalltalk en particulier

Smalltalk-80, A Mevel et T. Gueguen (Eyrolles)

Smalltalk-80 : The Language and its Implementation, A. Goldberg et D.Robson Addison-Wesley) ... la bible, connue sous le nom de «Blue Book». Rappelons enfin que Smalltalk est disponible sur Macintosh.

NOTES

[1] Simula-67, l'ancêtre des LOO, a été conçu pour permettre, à travers un environnement basé sur Algol-60, la simulation de processus réels (comme des épidémies, des processus industriels...). Il introduit la notion de classe pour regrouper à la fois des variables et un ensemble de procédures pour manipuler ces variables ; des objets d'une classe sont créés dynamiquement (i.e. pendant l'exécution), et ont une existence autonome : ils disposent chacun de leurs propres valeurs pour ces variables, et demeurent dans le système tant qu'on peut les référer. Alan Kay a repris ces idées dans son projet de machine interactive développé au centre de recherches de Xerox à Palo Alto (Californie), et y a adjoint la

transmission de messages aboutissant à la première version du langage Smalltalk en 1972. Les concepts sous-jacents ont ensuite été progressivement utilisés pour d'autres objectifs que la simulation : animations graphiques, systèmes de fenêtrage, systèmes d'exploitation, représentation de connaissances.

[2] L'héritage est dit simple lorsque chaque classe a un seul ancêtre immédiat ; la hiérarchie a la structure d'un arbre. Certains LOO utilisent cependant l'héritage multiple: une classe donnée peut avoir plusieurs ancêtres immédiats ; ainsi, un bateau-jouet hérite à la fois des propriétés des bateaux et de celles des jouets. Avec l'héritage multiple, des conflits entre plusieurs définitions de méthode pour un même sélecteur peuvent apparaître, et le langage doit proposer une façon de résoudre ce délicat problème.