Apport d'Ada 95 aux paradigmes orientés objet

(article publié aux Journées Internationales des Nouveautés en Génie Logiciel, 13-15 décembre 1994, Paris)

J-P. Rosen
ADALOG
19-21 rue du 8 mai 1945
94110 VANVES

Tel: 01 41 24 31 40
Fax: 01 41 24 07 36
E-m: rosen@adalog.fr

N. Kettani
CR2A
19, avenue Dubonnet
92411 COURBEVOIE

Tel: 01 47 68 97 97
Fax: 01 47 68 87 81
E-m: nkettani@cr2a.fr
(Note: N. Kettani est aujourd'hui chez Rational)

Résumé
La nouvelle norme du langage Ada (Ada 95) apporte à ce dernier l'héritage et les mécanismes de la programmation orientée objet. Les mécanismes utilisés sont originaux et présentent une avancée nouvelle dans le domaine des techniques orientées objet. Cet exposé décrit ces nouvelles possibilités, les compare aux mécanismes existant dans d'autres langages et discute de leur impact sur la conception orientée objet en général.

I Introduction

Ada 95 est la nouvelle version du langage Ada, approuvée par l'ISO le 15 février 1995. Ada 83 n'avait que peu de support *direct* à la notion d'héritage (tout en supportant très bien les approches objet par composition). Il s'en est suivi une querelle de mots pour savoir si Ada était "orienté objet" ou non, et l'utilisation parfois du terme "basé objet" (?) pour décrire les langages tels qu'Ada qui fournissaient indiscutablement les notions d'objet, tout en se passant fort bien de l'héritage... Une des principales extensions d'Ada 95 a consisté à introduire cette notion, ce que nous exposons dans ce papier. Ada peut désormais être qualifié sans arrière-pensée d'"orienté objet".

Notons qu'Ada 95 est donc le premier langage orienté objet standardisé au niveau international. Les processus de validation, qui ont permis la rigoureuse portabilité qui est un des atouts importants d'Ada, sont déjà en place pour Ada 95.

II Le mécanisme d'héritage en Ada 95

L'héritage a été introduit en Ada 95 sans révolutionner les fondements du langage qui ont fait leurs preuves depuis plus de 10 ans, mais au contraire en poursuivant la logique de notions qui existaient déjà: les paquetages, qui sont les unités d'encapsulation et de modularisation, et les types dérivés qui permettent de définir de nouveaux types selon un modèle existant.

II.1 Classes et instances.

Une *classe* est un ensemble de types, dérivés d'un même ancêtre appelé racine de la classe. Cet ancètre (et par voie de conséquence ses descendants) est obligatoirement un type *étiqueté* (*tagged*), c'est-à-dire un type enregistrement (**record**) déclaré avec le mot réservé **tagged**. Ceci n'est pas une restriction, mais une protection supplémentaire. Nous verrons en effet que les mécanismes de l'héritage apportent une plus grande souplesse d'évolution, au prix d'un certain affaiblissement du contrôle (statique) des types. L'utilisateur qui ne souhaite *pas* utiliser ces mécanismes (et c'est le cas par exemple des utilisateurs temps réel) ont la garantie, si le mot **tagged** n'apparaît pas dans leurs programmes, de bénéficier de la même

qualité de vérification statique et de la même efficacité du code généré qu'en Ada 83. On ne paie le prix de la POO que si on l'utilise.

Notons que, comme pour un type non étiqueté, la définition du type n'inclue pas les opérations associées; une classe complète au sens des langages orientés objets (c'est-à-dire un type de données muni des opérations associées) se réalise donc en Ada 95 sous la forme d'un paquetage comportant la définition d'un type étiqueté et d'opérations associées. Pour reprendre l'exemple statistiquement le plus utilisé pour introduire la POO, la classe des objets graphiques s'écrira:

```
package CLASSE_OBJET_GRAPHIQUE is

   type COORDONNEE is range 1..1024;

   type OBJET_GRAPHIQUE is tagged
      record
         X,Y : COORDONNEE;
      end record;

   procedure DEPLACER (Objet : OBJET_GRAPHIQUE;
                       En_X : COORDONNE;
                       En_Y : COORDONNEE);
end CLASSE_OBJET_GRAPHIQUE;

package body CLASSE_OBJET_GRAPHIQUE is
   procedure DEPLACER (Objet : OBJET_GRAPHIQUE;
                       En_X  : COORDONNE;
                       En_Y : COORDONNEE) is
   begin
      Objet := (En_X, En_Y);
   end;

end CLASSE_OBJET_GRAPHIQUE;
```

Avec une telle définition, les coordonnées X et Y sont accessibles de l'extérieur. Nous fournissons cependant une méthode DEPLACER pour modifier ces valeurs[1], car il s'agit d'une opération qui conceptuellement pourrait faire plus que simplement modifier les coordonnées.

Un cercle posséde en plus d'une position, une autre grandeur caractéristique: son rayon. Nous pouvons définir un type CERCLE comme une *extension* (un enrichissement) du type OBJET_GRAPHIQUE:

```
type CERCLE is new OBJET_GRAPHIQUE with
   record
      RAYON : DISTANCE;
   end record;
```

Le type CERCLE est dérivé du type OBJET_GRAPHIQUE: il appartient donc à la classe des objets graphiques, que l'on note en Ada OBJET_GRAPHIQUE'CLASS. Noter la distinction soigneuse faite en Ada entre le type OBJET_GRAPHIQUE (qui ne contient que les objets déclarés avec ce type) et la classe OBJET_GRAPHIQUE'CLASS qui inclue tous les objets du type OBJET_GRAPHIQUE, ou de types dérivés d'OBJET_GRAPHIQUE, tels que CERCLE. Pour bien marquer cette différence, on utilisera l'appellation *type spécifique* pour parler d'un type particulier à l'intérieur d'une classe, que l'on appellera "officiellement" *type à l'échelle de classe*[2].

Comme pour tout type dérivé (même non étiqueté), tous les attributs et méthodes définis pour le type OBJET_GRAPHIQUE sont disponibles pour le type CERCLE. Comme en Ada 83, on peut *redéfinir* lors de la déclaration d'un type les opérations primitives (=méthodes) du type dont elle hérite. Par exemple:

---

[1] Nous reviendrons plus loin sur le problème du déplacement de l'objet.

[2] Traduction lourde, mais fidèle, de l'expression "class wide type".

```
type CERCLE is new OBJET_GRAPHIQUE with
   record
      RAYON : DISTANCE;
   end record;

procedure DEPLACER (Objet : CERCLE;
                    En_X : COORDONNE;
                    En_Y : COORDONNEE) is
begin
   Instructions pour déplacer le cercle
end DEPLACER;
```

Selon les règles normales de la surcharge, un appel à DEPLACER pour un objet de type CERCLE utilisera la méthode définie sur le type correspondant; si une telle méthode n'a pas été explicitement définie, CERCLE disposera quand même (aura hérité) de la méthode DEPLACER définie dans OBJET_GRAPHIQUE.

Le mécanisme décrit au paragraphe précédent n'est pas nouveau à Ada 95; il fonctionne déjà en Ada 83, si ce n'est la possibilité d'étendre un type existant (clause **is new** ... **with record** ... **end record**). Le polymorphisme et la liaison dynamique apparaîssent lorsque l'on introduit des paramètres formels dont le type est à l'échelle de classe. Par exemple, au lieu de récrire la méthode DEPLACER pour chaque OBJET_GRAPHIQUE, nous pouvons fournir une méthode générale:

```
procedure DEPLACER (Objet : OBJET_GRAPHIQUE'CLASS;
                    En_X  : COORDONNE;
                    En_Y  : COORDONNEE) is
begin
   EFFACER(Objet);
   Objet := (En_X, En_Y);
   DESSINER(Objet)
end
```

Noter que le premier paramètre n'est plus défini comme d'un type particulier, mais d'une *classe*; cette procédure pourra donc être appelée en fournissant un paramètre réel appartenant à n'importe quel type de la classe, c'est à dire aussi bien au type OBJET_GRAPHIQUE que CERCLE ou n'importe quel autre type dérivé, directement ou indirectement, de OBJET_GRAPHIQUE[3]. Comme le type effectif n'est plus connu à la compilation, la méthode que doit invoquer l'instruction "EFFACER(Objet)" ne l'est pas non plus: il y a donc *liaison dynamique*. En termes Ada, ce mécanisme peut être vu comme une résolution de surcharge à l'exécution.

Bien entendu, les méthodes EFFACER et DESSINER doivent avoir été définies sur le type OBJET_GRAPHIQUE. Mais la *façon* de dessiner un objet graphique va être particulière à chaque type spécifique, et n'a pas de sens pour la classe générale. On peut alors définir ces méthodes comme *abstraites*. Des méthodes abstraites ne sont autorisées que si le type a été lui-même défini comme abstrait:

```
package CLASSE_OBJET_GRAPHIQUE is

   type OBJET_GRAPHIQUE is abstract tagged
      record
         X,Y : COORDONNEE;
      end record;

   procedure DEPLACER (Objet : OBJET_GRAPHIQUE'CLASS;
                       En_X  : COORDONNE;
                       En_Y  : COORDONNEE);

   procedure DESSINER (Objet : OBJET_GRAPHIQUE) is abstract;

   procedure EFFACER  (Objet : OBJET_GRAPHIQUE) is abstract;

end CLASSE_OBJET_GRAPHIQUE;
```

[3] Remarquer au passage l'*affaiblissement du typage* lié à cette façon de faire: une même procédure est appelable avec des paramètres de types différents.

Un sous-programme muni de la clause "**is abstract**" est dit "abstrait": il n'est pas défini concrètement, il ne sert qu'à marquer que les types dérivés d'OBJET_GRAPHIQUE devront obligatoirement redéfinir ces procédures. Comme il est nécessaire que le client de la classe soit informé de cette propriété, on ne peut définir de sous-programmes abstraits que si le type a lui-même été déclaré **abstract**. Le type OBJET_GRAPHIQUE, ainsi que les types dérivés qui n'auraient pas redéfini les sous-programmes abstraits, sont appelés *types abstraits*: il est interdit de déclarer des objets ou des valeurs de ces types puisque toutes leurs propriétés ne seraient pas définies; ils ne peuvent servir qu'à dériver d'autres types. Dans d'autres langages, on utilise parfois le terme de *méthode retardée* (*deferred*) pour désigner ces sous-programmes. Notre CERCLE va donc devenir:

```
type CERCLE is new OBJET_GRAPHIQUE with
   record
      RAYON : DISTANCE;
   end record;

procedure DESSINER(Objet : CERCLE) is
begin
   Instructions pour dessiner le cercle
end DESSINER;

procedure EFFACER(Objet : CERCLE) is
begin
   Instructions pour effacer le cercle
end EFFACER;
```

Ceci nécessite donc de garder *à l'exécution* la marque du type du paramètre réel associé au paramètre formel: dans la procédure DEPLACER, tout ce que l'on sait du paramètre est qu'il appartient à la classe des OBJET_GRAPHIQUE (qu'il est dérivé directement ou indirectement de OBJET_GRAPHIQUE), or le choix du sous-programme appelé dépend du type *spécifique* du paramètre formel. Cette "marque d'origine" est l'étiquette, le fameux *tag*, ce qui explique qu'il ne soit possible de programmer de cette façon qu'avec des types "étiquetés".

II.2 Objets et pointeurs

Dans certains langages à objet, une même variable du programme peut contenir *tout* objet correspondant à sa classe. En Eiffel par exemple, une variable *X* qui est déclarée comme un objet graphique pourra contenir n'importe que objet graphique, aussi bien un CERCLE qu'un RECTANGLE. Ceci n'est possible que parce que dans ces langages, *tous* les types sont à sémantique d'objet, il n'est pas possible de créer des types abstraits à sémantique de valeur. Au niveau de l'implémentation, toutes les variables ne sont en fait que des pointeurs vers les structures effectives, ce qui nécessite la présence systématique d'opérations de création explicites: un objet ne peut exister simplement parcequ'il est déclaré.

Ada 95 autorise la définition d'objets (variables ou constantes) appartenant à un type à l'échelle de classe, à condition qu'ils soient initialisés. C'est la valeur initiale qui détermine le type effectif. Par exemple:

```
procedure Modifier(Objet : in out OBJET_GRAPHIQUE is
   Variable_de_Travail: OBJET_GRAPHIQUE'Class := Objet;
begin
   ...
end Modifier;
```

Il est possible ainsi d'utiliser tous les mécanismes de la POO sans recourir aux pointeurs, et en particulier sans avoir de pointeurs cachés qui interdiraient quasiment l'utilisation de la POO dans des contextes "temps-réel". Bien entendu, il est possible d'utiliser des pointeurs, mair leur contrôle reste explicite. Ada autorise également des pointeurs sur classe:

```
type PTR_OBJET is access OBJET_GRAPHIQUE'CLASS;
```

Une variable d'un tel type pourra naturellement désigner n'importe quel objet de la *classe*, c'est à dire n'importe quel objet du type OBJET_GRAPHIQUE, CERCLE, RECTANGLE, ou de n'importe quel autre type dérivé de OBJET_GRAPHIQUE. Il est ainsi possible de construire des structures de données hétérogènes.

II.3 Encapsulation

Dans l'exemple précédent, le type OBJET_GRAPHIQUE n'est pas totalement un type de données abstrait, puisque la structure interne est encore visible. Il faut donc en faire un type privé, mais bien entendu il est nécessaire de spécifier que même privé, le type est étiqueté, afin de permettre de l'utiliser en tant que tel dans des dérivations ultérieures. Notre spécification de paquetage va donc devenir:

```
package CLASSE_OBJET_GRAPHIQUE is

   type COORDONNEE is range 1..1024;

   type OBJET_GRAPHIQUE is abstract tagged private;

   procedure POSITIONNER (Objet : OBJET_GRAPHIQUE'CLASS;
                          En_X  : COORDONNEE;
                          En_Y  : COORDONNEE);
   function X_courant (Objet : OBJET_GRAPHIQUE) return COORDONNEE;
   function Y_courant (Objet : OBJET_GRAPHIQUE) return COORDONNEE;

   procedure DEPLACER (Objet : OBJET_GRAPHIQUE'CLASS;
                       En_X  :  COORDONNE;
                       En_Y  : COORDONNEE);

   procedure DESSINER (Objet : OBJET_GRAPHIQUE) is abstract;
   procedure EFFACER  (Objet : OBJET_GRAPHIQUE) is abstract;

private
   type OBJET_GRAPHIQUE is abstract tagged
      record
         X,Y : COORDONNEE
      end record;
end CLASSE_OBJET_GRAPHIQUE;
```

Remarquer que la structure étant devenue cachée, nous avons fourni un constructeur (POSITIONNER) et deux sélecteurs (X_COURANT et Y_COURANT) afin de permettre respectivement de donner une valeur aux coordonnées, ou de retrouver leurs valeurs courantes. De même, il est possible d'étendre de façon privée un type. CERCLE pourrait être défini comme suit:

```
package CLASSE_CERCLE is
   type CERCLE is new OBJET_GRAPHIQUE with private;

private
   type CERCLE is new OBJET_GRAPHIQUE with
      record
         RAYON : DISTANCE;
      end record;

   procedure DESSINER (Objet : CERCLE);
   procedure EFFACER  (Objet : CERCLE);

end CLASSE_CERCLE;
```

Remarquer l'annonce des procédures DESSINER et EFFACER en partie privée: le compilateur sait ainsi que le paquetage va fournir ces sous-programmes, et que donc le type peut ne pas être abstrait. Pourquoi ne pas avoir mis ces déclarations visibles (hors partie privée)? L'utilisateur sait de toutes façons que ces méthodes existent, et seule compte l'abstraction; l'endroit précis de la déclaration n'est finalement pas si important, et nous préférons toujours limiter ce qui est exposé.

II.4 Liaison avec le parent

Lorsque l'on redéfinit des méthodes héritées, il est très fréquent que l'on souhaite simplement ajouter certains traitements au traitement "de base" fourni par la méthode importée. Il faudra donc souvent, dans le corps d'une méthode d'un enfant, appeler la méthode définie par le parent. Ceci se fait naturellement en Ada: il suffit d'appeler le sous-programme correspondant en effectuant une conversion de type. Comme toujours, seule la cohérence des types détermine le sous-programme appelé, le parent ici. Par exemple, nous pouvons avoir un objet CERCLE_POINTE, analogue au cercle, mais où l'on marque le centre du cercle par un point. Nous pouvons le définir ainsi:

```
type CERCLE_POINTE is new CERCLE with null record;
procedure DESSINER (Objet : CERCLE) is
begin
   DESSINER(CERCLE(Objet));
   -- Dessiner le centre
end DESSINER;
procedure EFFACER  (Objet : CERCLE) is
begin
   -- Effacer le centre
   EFFACER(CERCLE(Objet));
end DESSINER;
```

Nous créons un nouveau type CERCLE_POINTE, mais comme il ne comporte pas de nouveau champ, nous devons le signaler par la clause "**with null record**". Bien entendu nous ne redessinons pas tout le cercle: l'avantage de la POO est qu'on ne recode que la différence entre le nouveau comportement et celui que l'on possédait déjà. Par conséquent, le DESSINER d'un CERCLE_POINTE appelle le DESSINER de CERCLE grâce à une conversion de type (et similairement pour effacer). On peut comprendre cet appel comme signifiant: "dessiner ce CERCLE_POINTE en le considérant comme un CERCLE". Noter au passage que le type spécifique de l'appel est directement connu, et qu'il n'y a pas donc de choix dynamique.

Remarquer également que dans le cas d'une hiérarchier complexe, n'importe quel enfant peut choisir d'appeler une des méthodes de n'importe lequel de ses ancètres, simplement en convertissant dans le type approprié.

On voit donc que ce mécanisme, facile à comprendre (comme toujours, seule la cohérence des types détermine le sous-programme effectivement appelé), fournit à Ada une spécificité unique: le choix, *par l'appelant*, d'effectuer un appel spécifique ou dynamique. De plus, on peut appeler directement la méthode associée à un ancêtre précis, non nécessairement le parent immédiat.

II.5 Facettes multiples

Dans l'exemple que nous avons vu, la classe CERCLE héritait des propriétés d'une seule classe, OBJET_GRAPHIQUE. Or il arrive parfois que l'on veuille considérer des objets complexes selon plusieurs vues, plusieurs facettes. Par exemple, on peut définir une classe FIGURE_GEOMETRIQUE permettant d'obtenir diverses caractéristiques (périmètre, surface...) des objets géométriques courants.

Il est bien certain qu'un CERCLE *est* un objet graphique, mais c'*est* également un objet géométrique. Il n'est ni nécessaire, ni souhaitable de rassembler ces deux notions dans une même classe, car elles sont orthogonales: les segments sont des objets graphiques tout à fait respectables, mais cela n'aurait pas de sens de parler de leur périmètre ni de leur surface. Il faudrait donc ajouter à CERCLE les propriétés liées à la notion de FIGURE_GEOMETRIQUE en plus de celles héritées de la classe OBJET_GRAPHIQUE.

Ceci se fait facilement en Ada au moyen de la technique d'*enrichissement de classe* par des génériques. Nous pouvons écrire un générique destiné à fournir des propriétés géométriques à toute autre classe:

```
with Grandeurs; -- Définit Longueur, Surface etc.
generic
   type Objet is tagged limited private;
package Geometrique is
   type Objet_geometrique is abstract new Objet with private;

   use Grandeurs;
   function Aire(L_objet : Objet_Geometrique)      return Surface;
   function Perimetre(L_objet : Objet_Geometrique) return Longueur;

   -- etc...
private
   type Objet_Geometrique is new Objet with null record;
end Geometrique;
```

A partir de là, nous pouvons utiliser ce générique pour créer un nouveau type *enrichi* de propriétés géométriques:

```
package Classe_Graphique_Geometrique is new Geometrique(OBJET_GRAPHIQUE);

subtype Objet_Graphique_Geometrique is
   Classe_Graphique_Geometrique.Objet_Geometrique;
```

Nous considérons maintenant un cercle comme un objet graphique aussi bien que géométrique; bien entendu, il faut définir la méthode pour le dessiner, l'effacer, ainsi que pour calculer l'aire et le périmètre; nous devons donc (re)définir les procédures correspondantes:

```
with GRANDEURS;
package CLASSE_CERCLE is
   type CERCLE is new Objet_Graphique_Geometrique with private;

private
   type CERCLE is new OBJET_GRAPHIQUE with
      record
         RAYON : DISTANCE;
      end record;

   procedure DESSINER (Objet : CERCLE);
   procedure EFFACER  (Objet : CERCLE);

   use Grandeurs;
   function Aire(L_objet : CERCLE) return Surface;
   function Perimetre(L_objet : CERCLE) return Longueur;

end CLASSE_CERCLE;
```

Nous pouvons dériver indifféremment des objets de Objet_Graphique_Geometrique si nous voulons à la fois les deux facettes, ou seulement de Objet_Graphique pour ceux (les segments par exemple) dont les propriétés géométriques n'ont aucun sens.

III Comparaison avec les autres langages.

Nous avons vu précédemment les principales fonctionnalités fournies par Ada pour la classification. Chaque langage dit "orienté objet" offre ses propres mécanismes et ses modalités particulières. Nous allons maintenant discuter des rapports entre la façon de réaliser les concepts de la classification en Ada par rapport à ceux d'autres langages.

III.1 Notion de classe et appel de méthode

Ada n'a pas de construction syntaxique spéciale pour la notion de *classe*: une classe au sens des LOO est réalisée par un paquetage contenant la déclaration d'un type étiqueté et des opérations associées. Les classes sont donc moins visibles en tant que telles que dans les langages "tout OO". En revanche, cette structure présente un certain nombre d'avantages:

- Les notions d'encapsulation et de typage sont orthogonales, alors que les classes d'autres langages servent à la fois aux deux. On voit apparaître alors des "fausses classes" qui ne servent qu'à encapsuler des éléments, sans aucune relation avec la classification en tant que méthode.
- Il n'existe aucun problème pour imbriquer les déclarations. Un paquetage définissant une classe peut définir un autre paquetage de classe imbriquée, une procédure (qu'elle soit ou non une "méthode" d'un type étiqueté) peut contenir ses propres classes locales, etc. Sans vouloir rentrer dans les détails techniques,noter qu'*aucun autre* langage orienté objet n'offre cette possibilité.
- Les opérandes des méthodes sont symétriques. Dans la plupart des langages orientés objets, une méthode est "propriété" d'un objet, et l'appel d'une méthode utilise une syntaxe particulière. Si un objet O appartient à la classe C et qu'on veut appeler sa méthode M, on écrira:

```
O.M;
```

En Ada, une méthode de classe est simplement une procédure qui possède un paramètre du type C, et l'appel s'écrira:

```
M(O);
```

Apparemment, la solution des LOO correspond mieux à la notion de "demander à l'objet O d'exécuter sa méthode M". Cependant, le problème se complique si la méthode doit porter sur plusieurs paramètres de la même classe; l'écriture "LOO" rompt la symétrie:

```
O1.M(O2);
```

alors que celle-ci est conservée en Ada:

```
M(O1, O2);
```

Si maintenant nous définissons une sous-classe S de C; en termes Ada, nous dérivons du type étiqueté C. Avec les LOO classiques, il n'y a qu'un seul terme principal qui détermine la méthode appelée. Par conséquent, la méthode dont héritera un objet de classe D sera une méthode dont le paramètre sera de classe C; en Ada, *tous* les paramètres sont dérivés, ce qui signifie que la procédure M dérivée portera bien sur deux paramètres de type D. Pour prendre un exemple, si nous redéfinissons la comparaison d'égalité:

```
type C is tagged record ...;
function "=" (O1, O2 : C) return Boolean;

type D is new C with ...;
-- on hérite de :
-- function "=" (O1, O2 : D) return Boolean;
```

Dans le cas des autres LOO, on hériterait d'une fonction de comparaison d'un C avec un D, c'est à dire de quelque chose qui n'aurait aucun sens!

Autrement dit, la solution Ada est moins "pure" du point de vue de la théorie des LOO, mais elle est plus sûre et fournit un comportement plus conforme aux attentes de l'utilisateur dès qu'une méthode doit porter sur plus d'un seul objet.

III.2 Variables de classe

Certains LOO autorisent la définition de *variables de classe*, par opposition aux attributs normaux appelés *variables d'instance*. Une variable de classe est, comme son nom l'indique, liée à une classe, et donc partagée par toutes les instances appartenant à la classe; ceci peut être nécessaire par exemple pour réaliser un compteur d'instances de la classe. Sa réalisation en Ada est tout à fait naturelle: il s'agit simplement d'une variable globale du corps de paquetage:

```
package Classe_Objets is
   type Objet is tagged
      record
         ...
      end record;
```

```
      -- Opérations sur Objet
   end Classe_objet;

   package body Classe_Objets is
      type T_Compteur is range 0..1000;
      Compteur : T_Compteur := 0;

      -- Implémentation des opérations
   end Classe_Objets;
```

III.3 Approches de l'héritage multiple

Nous avons vu qu'en Ada, une approche par composition de générique permettait de voir un même objet selon plusieurs facettes différentes. D'autres langages utilisent pour celà la notion d'héritage multiple, qui correspondrait en Ada au fait de pouvoir dériver de plusieurs types à la fois.

La question des avantages et inconvénients de l'héritage multiple est l'objet d'un vaste débat dans la communauté "orientée objet". En effet, si l'héritage multiple est séduisant en théorie, il apporte avec lui le problème dit de l'héritage *répété*: que faire lorsqu'une même méthode est héritée à travers plusieurs chemins différents? La solution choisie par Ada au problème des vues multiples permet de s'affranchir naturellement de ce problème: l'enrichissement des classes se faisant par ajout successif, les nouvelles méthodes remplacent naturellement les anciennes. C'est donc l'ordre d'instanciation (choisi par l'utilisateur) qui détermine la méthode effectivement utilisée.

Enfin, cette solution procure une grande finesse dans la définition des relations entre objets. Si un générique de facette annonce:

```
generic
   type Origine is abstract tagged limited private;
package Facette is ...
```

alors il pourra être utilisé avec n'importe quel type étiqueté. Mais s'il annonce:

```
generic
   type Origine is new Objet_Géométrique with private;
package Facette is ...
```

alors, il ne pourra être instancié que sur un descendant (direct ou indirect) du type Objet_Géométrique. Autrement dit, on peut créer ainsi des facettes qui ne peuvent servir qu'à enrichir certaines classes bien précises (mais qui du coup peuvent tirer parti de la connaissance supplémentaire des propriétés du type ancètre). En héritage multiple classique, cela reviendrait à créer une classe dont on ne pourrait hériter que si l'on héritait simultanément d'une autre classe; un tel type de contrôle serait très difficile à définir.

III.4 Objets actifs

Les paradigmes orientés objet sont fondés sur l'idée qu'une classe est une abstraction d'un ensemble d'objets du monde réel; or la plupart des objets du monde réel sont susceptibles d'évoluer en parallèle. Tout langage orienté objet se doit donc de supporter le parallélisme[4]. Ceci s'obtient en Ada en associant une tâche à une structure de donnée, de façon plus ou moins couplée.

Une première solution consiste à attacher manuellement une tâche à un objet. On peut pour cela utiliser un pointeur désignant l'objet comme discriminant de la tâche. Imaginons par exemple que nous voulions représenter un récipient qui fuit: toutes les 10s., il perd un litre d'eau. Nous lui associons un "moniteur de fuite" qui diminue la quantité périodiquement:

[4] Ce critère semble parfaitement naturel, mais n'est pourtant pas retenu habituellement par les chantres de la programmation orientée objet, car alors ni C++, ni Eiffel ne sauraient être qualifiés de langages à objets.

```
package Compteur_fuyant is
   type Volume is delta 0.01 range 0.0 .. 100.0;
   type Compteur is
      record
         Contenu : Volume := 0;
      end record;

   task type Moniteur(Compteur_associé: access Compteur);
end Compteur_Fuyant;


package body Compteur_fuyant is
   task body Moniteur is
   begin
      loop
         delay 60.0;
         if Contenu > 1.0 then
            Contenu := Contenu - 1.0;
         else
            Contenu := 0.0;
         end if;
      end loop;
   end Moniteur;
end Compteur_Fuyant;


with Compteur_Fuyant; use Compteur_Fuyant;
procedure Test_Compteur is
   Mon_Compteur : aliased Compteur := (Contenu => 100.0);
   Ma_Tache : Moniteur(Mon_Compteur'Access);
begin
   ...
end Test_Compteur;
```

Avec cette solution la tâche "moniteur" est relativement extérieure à l'objet "compteur": rien ne nous empêche de ne *pas* associer de tâche à l'objet, et d'avoir un compteur qui ne fuit pas. Cela peut être le comportement désiré. Si inversement nous voulons être sûrs que tous les compteurs fuient, il faut associer la tâche plus étroitement au compteur. C'est ce que permettent les auto-pointeurs:

```
package Compteur_fuyant is
   type Volume is delta 0.01 range 0.0 .. 100.0;

   type Compteur;
   task type Moniteur(Compteur_associé: access Compteur);

   type Compteur is limited
      record
         Contenu     : Volume := 0;
         Le_Moniteur : Moniteur(Compteur'ACCESS);
      end record;

end Compteur_Fuyant;
```

Remarquer que c'est le nom du *type* "Compteur" qui est utilisé comme préfixe de l'attribut 'ACCESS ici. Tout objet déclaré du type "Compteur" contiendra une tâche "Moniteur", dont le discriminant désignera automatiquement l'objet "Compteur" dont il fait partie. Ces auto-pointeurs ne sont autorisés que dans des types limités: en effet, une affectation détruirait l'invariant que la tâche désigne toujours l'objet englobant. Noter que les auto-pointeurs ne sont pas limités aux tâches: on peut les utiliser pour lier n'importe quels types.

On peut réaliser ainsi des objets réellement actifs. Noter que l'on pourrait ainsi avoir plusieurs tâches associées à un même objet.

III.5 Distribution

Ada 95 inclue dans sa norme un modèle d'exécution distribuée; il est donc également le premier langage strandardisé permettant de répartir des applications de façon indépendante de tout mécanisme du système d'exploitation.

Un exposé du mécanisme de distribution sortirait du cadre de ce papier; on pourra en trouver une discussion dans [Bek94] par exemple. Notons simplement qu'en ce qui concerne la programmation orientée objet, il est possible de définir des classes distantes, et donc d'effectuer des appels dynamiques à travers le réseau. Il est également possible de définir des classes d'objet s'identifiant par un nom logique à un serveur de noms, et d'utiliser de tels objets en ignorant totalement leur localisation physique sur le réseau. Ce mécanisme est de plus très efficace, car seule l'obtention d'un objet requiert un appel au serveur de noms; l'appel de ses méthodes s'effectue directement.

Le modèle Ada d'exécution distribuée est compatible avec les standards existant ou en cours de développement (CORBA, RPC). Un standard de mapping CORBA vers Ada 95 ainsi que des traducteurs d'IDL vers Ada 95 sont actuellement en cours de développement et devraient apparaître dans le cours de l'année 1995.

IV Conclusion

Ada fournit désormais, avec les types étiquetés, l'outil de base qui lui manquait pour les méthodes par classification. Les objets à facettes multiples sont réalisés par un moyen original, l'utilisation de la généricité plutôt que par un mécanisme spécial du langage comme l'héritage multiple. Cette différence d'approche par rapport aux autres langages permet une meilleure sécurité et un contrôle plus précis du typage et des relations entre classes, au prix d'une plus grand lourdeur d'écriture. Ceci correspond à la philosophie du langage, qui fait primer la facilité de maintenance et la fiabilité sur la facilité de conception initiale.

Nous nous permettrons d'insister sur ce point, car la question de l'héritage multiple fait l'objet d'un vif débat jusque dans la communauté POO traditionnelle, certains y voyant la panacée, d'autres un danger permanent. Il est certain que si l'on demande s'il est possible, en Ada 95, de dériver un type directement de plusieurs autres à la fois, la réponse est "non". Mais en fait, l'héritage multiple, comme tout autre outil fourni par un langage d'ailleurs, n'est qu'un *moyen* au service de *besoins*. Si l'on demande si Ada 95 répond aux besoins que satisfait l'héritage multiple dans d'autres langages, la réponse est "oui", par ses moyens propres, qui conservent au langage ses points forts que sont la sécurité, la portabilité, l'efficacité et l'indépendance vis-à-vis des représentations machines.

Une autre originalité d'Ada pour la classification est que le mécanisme du langage ne repose absolument pas sur l'utilisation de pointeurs (au contraire, notamment, de C++ et d'Eiffel). L'utilisateur reste donc libre de définir des types à sémantique de valeur aussi bien qu'à sémantique d'objet. Les classes ne souffrent d'aucune des limitations d'imbrication connues dans les autres langages.

De nouveaux outils tels que les auto-pointeurs et les pointeurs généralisés permettent de définir des relations subtiles entre classes. Il est possible de définir des objets actifs et même des objets distribués, de façon indépendante de tout système d'exploitation et de tout mécanisme propriétaire.

On peut donc dire qu'Ada 95 non seulement supporte entièrement les méthodes par classification, mais que les nouveaux outils qu'il apporte sont de nature à faire progresser l'état de l'art en programmation orientée objet.

V Bibliographie

[Bek94] D. Bekele et J-M. Rigaud, "Ada et les systèmes répartis", TSI Vol 13 n°5, HERMES Paris, 1994.

[Bar94] S. Barbey, M. Kempe et A. Strohmeier, "La programmation par objets avec Ada 9X", TSI Vol 13 n°5, HERMES Paris, 1994.

[Ket93] N. Kettani, "Ada 9X: Object Oriented Programming with Inherant Qualities of Ada", Actes des journées "Génie Logiciel et ses Applications", EC2, Nanterre 1993.

[Ros92] J-P. Rosen, "Ada 9X : une évolution, pas une révolution", Techniques et science informatiques, Volume 11 n°5/1992, HERMES, Paris.

[Ros92] J-P. Rosen, "What Orientation Should Ada Objects Take?", Communications of the ACM, Volume 35 n° 11, ACM, New-York.

[Ros95] J-P. Rosen, Méthodes de génie logiciel avec Ada 95, InterEditions, Paris.