Le langage Ada : l'orientation objet

Yves Bailly

EN DEUX MOTS **Pratiquement tous les langages modernes suivent plus ou moins le paradigme objet, qu'ils soient purement objet (comme Smalltalk ou Ruby) ou autorisent la programmation procédurale (comme C++, Python, Perl...). Dès sa conception en 1979, le langage Ada possède des fondements objets sans pour autant pouvoir être qualifié de langage orienté objet. La norme Ada95 a introduit dans le langage des techniques et principes spécifiquement objets, faisant de ce langage le premier langage orienté objet à être standardisé.**

Avec l'évolution des techniques depuis 1995, il semble naturel que la norme Ada 2005 apporte une refonte importante de l'infrastructure objet du langage Ada. Tout en conservant un aspect procédural fort, nous allons voir comment Ada s'est vu enrichi des techniques objets les plus modernes, de la dérivation au polymorphisme en passant par la notion d'interface empruntée au langage Java.

Des types taggés

Une structure en Ada, type de donnée introduit par le mot-clef `record`, est très similaire en mémoire à une structure en langage C. Elle contient les données indiquées, rien de plus. Imaginons un instant un enregistrement représentant une forme géométrique générique, ne contenant que sa position :

```
type Geom is record
   pos_x: Float ;
   pos_y: Float ;
end record ;
```

Toute variable de ce type occupera très exactement `2*Float'Size` bits en mémoire (c'est-à-dire 64 bits, ou 8 octets, sur architecture 80x86 32bits).

Les habitués de la littérature consacrée à la programmation objet me voient sans doute venir : et si maintenant nous voulions par exemple un cercle ? La position désigne le centre, mais nous avons également besoin du rayon.

Le problème avec notre type `Geom` est qu'il est « figé », pas moyen de lui ajouter des informations. De plus, sa taille étant exactement égale à la somme des tailles de ses composants, on voit mal comment mettre en place un mécanisme comme les méthodes virtuelles du C++...

Depuis Ada 95, ces contraintes sont levées par l'introduction de fonctionnalités orientées objet. Modifions légèrement la déclaration de notre type :

```
type Geom is tagged record
   pos_x: Float ;
   pos_y: Float ;
end record ;
```

L'ajout du discret mot-clef `tagged` change complètement la façon dont notre type est représenté en mémoire et ce que l'on peut en faire. Si vous demandez la valeur de `Geom'Size`, vous n'obtiendrez plus 64 (pour mémoire, l'attribut `'Size` donne la taille d'un objet en bits), mais 96, soit 32 bits de plus, ce qui correspond à la taille d'un pointeur (sur architectures 32bits).

Dès lors, notre type est marqué, ce n'est plus un simple enregistrement, mais un type objet. Plein de choses deviennent alors possibles.

Opérations primitives

Contrairement aux langages objets comme C++, Python, Java et consorts, le langage Ada ne fait pas figurer les déclarations des méthodes d'un type objet au sein de la structure : elles sont simplement déclarées immédiatement après la déclaration du type, dans la même spécification de paquetage.

Plutôt que le terme « méthodes », le verbiage Ada utilise les termes « opérations primitives » pour désigner l'ensemble des sous-programmes (procédures et fonctions) qui agissent directement sur un type taggé. Pour être considéré comme une opération primitive d'un type donné, un sous-programme doit répondre à certaines conditions :

- être déclaré juste après le type, dans la même spécification de paquetage ;
- pour une procédure, prendre un paramètre de ce type, ou un pointeur (`access`) sur ce type ;
- pour une fonction, soit prendre un paramètre de ce type ou un pointeur sur ce type, soit retourner une instance du type ou un pointeur sur une instance du type.

D'autres conditions ou possibilités existent, nous les verrons plus loin.

Ajoutons donc quelques opérations primitives à notre type `Geom`, par exemple une fonction retournant l'aire de l'objet et une procédure permettant de le déplacer, dans un fichier `geom_pkg.ads` :

```
1 package Geom_Pkg is
2
3    type Geom is tagged record
```

```
 4       pos_x: Float := 0.0 ;
 5       pos_y: Float := 0.0 ;
 6    end record ;
 7    function Aire(g: in Geom) return Float ;
 8    procedure Déplacer(g : in out Geom ;
 9                       dx: in Float ;
10                       dy: in Float) ;
11 end Geom_Pkg ;
```

Tout cela ne semble rien avoir de bien particulier, si ce n'est le mot-clef `tagged` ligne 3. Le corps de ce paquetage (dans `geom_pkg.adb`) est trivial :

```
 1 package body Geom_Pkg is
 2    function Aire(g: in Geom) return Float is
 3    begin
 4       return 0.0 ;
 5    end Aire ;
 6    procedure Déplacer(g : in out Geom ;
 7                       dx: in Float ;
 8                       dy: in Float) is
 9    begin
10       g.pos_x := g.pos_x + dx ;
11       g.pos_y := g.pos_y + dy ;
12    end Déplacer ;
13 end Geom_Pkg ;
```

Voyons maintenant un petit programme d'exemple `geom_1.adb` utilisant notre paquetage :

```
 1 with Text_IO ; use Text_IO ;
 2 with Geom_Pkg ;
 3 procedure Geom_1 is
 4    g: Geom_Pkg.Geom ;
 5 begin
 6    Put_Line(Float'Image(Geom_Pkg.Aire(g))) ;
 7    Put_Line(Float'Image(g.Aire)) ;
 8    Geom_Pkg.Déplacer(g, 1.0, 0.0) ;
 9    Put_Line(Float'Image(g.pos_x)) ;
10    g.Déplacer(1.0, 0.0) ;
11    Put_Line(Float'Image(g.pos_x)) ;
12 end Geom_1 ;
```

Pour compiler ce programme, utilisez la commande :

```
$ gnatmake -Wall -gnat05 -gnatif geom_1
```

L'option `-gnatif` est nécessaire pour pouvoir utiliser des caractères non-ASCII dans les identificateurs, comme c'est le cas de la procédure `Déplacer`.

La ligne 6 affiche le résultat de la fonction `Aire`, appliquée à la variable `g` déclarée ligne 4. La ligne suivante (7) fait exactement la même chose, mais en utilisant une nouvelle possibilité apportée par Ada 2005 : la notation pointée, si commune dans l'immense majorité des autres langages objets. Celle-ci n'est utilisable que si le sous-programme invoqué est une opération primitive du type de la variable sur laquelle on l'applique. Les lignes 6 et 7 sont ainsi parfaitement synonymes.

Remarquez que la fonction `Aire` ne prenant qu'un seul paramètre (de type `Geom`, ce qui en fait une opération primitive sur ce type, étant donné qu'elle est déclarée juste après celui-ci), celui-ci lui est implicitement passé dans la notation pointée. Tout se passe alors comme si elle ne prenait pas de paramètre : les parenthèses ne doivent donc pas être utilisées, comme pour tout sous-programme ne prenant pas de paramètre.

La situation est tout à fait similaire à celle d'une méthode dans une classe C++ qui ne prendrait pas de paramètre, ou d'une méthode d'une classe Python dont le seul paramètre serait `self`.

De la même façon, les lignes 8 et 10 sont parfaitement équivalentes : la première utilise la notation traditionnelle en Ada, tandis que la seconde utilise la notation pointée. Remarquez que le paramètre de type `Geom` « disparaît » : il est implicitement passé à la procédure, un peu comme un paramètre qui serait nommé `this` pour une méthode d'une classe C++.

Pour information, la notation pointée n'est supportée par le compilateur GNAT que dans les versions récentes de la suite GCC. Si votre compilateur date un peu, il est possible qu'il renvoie une erreur si vous l'utilisez.

À titre de comparaison, voyons comment serait déclaré notre type `Geom` de façon équivalente en C++ et en Python :

Trois langages pour un type

Ada

```
type Geom is tagged record
   pos_x: Float := 0.0 ;
   pos_y: Float := 0.0 ;
end record ;
function Aire(g: in Geom) return Float ;
procedure Déplacer(g : in out Geom ;
                   dx: in Float ;
                   dy: in Float) ;
```

C++

```
struct Geom
{
   Geom():
      pos_x(0.0),
      pos_y(0.0)
   { }
   float Aire() const ;
   void Deplacer(float dx,
                 float dy) ;
   float pos_x ;
   float pos_y ;
} ;
```

Python

```
class Geom:
   def __init__(self):
      self.pos_x = 0.0
      self.pos_y = 0.0
   def Aire(self):
      return 0.0
   def Deplacer(self, dx, dy):
      self.pos_x += dx
      self.pos_y += dy
```

Il est intéressant de constater que ce langage réputé si verbeux qu'est Ada nécessite moins de lignes que les deux autres pour cette déclaration.

Opérations nulles

Il n'est pas rare en conception orientée objet de prévoir une opération sur un type de base qui ne fait rien. Cette opération est destinée à être surchargée dans les types dérivés (que nous verrons un peu plus loin), mais elle présente un sens même sur le type de base. Par exemple, une telle opération pourrait être une mise à l'échelle (homothétie), qui n'est pas en elle-même absurde sur un objet géométrique se résumant à un point. Simplement, elle ne fait rien.

On pourrait déclarer cette opération primitive ainsi :

```
procedure Homothétie(g : in out Geom ;
                     sx: in Float ;
                     sy: in Float) ;
```

Son implémentation ne contiendrait alors pas d'instruction, seulement le mot-clef `null` imposé par le langage :

```
procedure Homothétie(g : in out Geom ;
                     sx: in Float ;
                     sy: in Float) is
begin
   null ;
end Homothétie ;
```

Ce qui fait bien beaucoup de code pour si peu de chose. Mais Ada prévoit une syntaxe particulière pour ce genre de situations. Notre opération peut être avantageusement qualifiée par `is null` :

```
procedure Homothétie(g : in out Geom ;
                     sx: in Float ;
                     sy: in Float) is null ;
```

L'ajout du qualificatif `is null` stipule que cette opération ne fait rien, qu'aucun code ne lui est associé. Procéder de la sorte présente plusieurs avantages :

- L'utilisateur du paquetage est informé du fait que l'invocation de cette procédure n'aura aucun effet.
- Le compilateur peut utiliser cette information pour améliorer l'optimisation du code exécutable généré.
- Enfin, si vous décidez subitement de donner un corps à cette procédure dans le corps du paquetage, le compilateur vous signalera qu'il y a un problème : cela peut permettre d'identifier des erreurs pouvant survenir lors d'une reconception un peu hâtive.

Le dernier point est probablement le plus important, dans la perspective de la maintenance et de l'évolution du programme. La conception du langage Ada est dirigée par le constat simple (parmi d'autres) que le code est écrit une seule fois, mais relu et modifié de nombreuses fois. Le qualificatif `is null` est donc un outil destiné à améliorer la qualité de l'évolution des programmes.

Naturellement, une fonction ne peut pas recevoir ce qualificatif : toute fonction doit retourner une valeur, ce qu'il est impossible de faire sans au moins une instruction, aussi triviale soit-elle.

Opérations abstraites

Il arrive également en conception objet qu'une opération n'ait pas de sens sur un type de base, qu'elle doive donc être définie explicitement dans les types dérivés, mais qu'elle s'impose pour la cohérence générale. C'est une opération abstraite ou encore une opération virtuelle. Reconsidérons un instant l'homothétie évoquée précédemment. On peut la concevoir de deux façons : soit une mise à l'échelle à partir de la position de l'objet, soit à partir de l'origine (le point de coordonnées (0,0)). La première pourrait fort bien entrer dans la catégorie des opérations abstraites, selon une conception un peu plus stricte de notre type `Geom`.

Une opération abstraite est déclarée par le qualificatif `is abstract` :

```
procedure Homothétie_Position(g : in out Geom ;
                              sx: in Float ;
                              sy: in Float) is abstract ;
```

Définir une opération abstraite sur un type en fait de facto un type abstrait, qui doit alors impérativement être déclaré ainsi :

```
type Geom is abstract tagged record
   pos_x: Float := 0.0 ;
   pos_y: Float := 0.0 ;
end record ;
```

Le mot-clef `abstract` a été ajouté devant `tagged` : le type `Geom` devient abstrait, donc il possède (au moins) une opération abstraite. Non seulement aucun code n'est associé à une telle opération (comme pour une opération nulle), mais en plus son invocation n'a pas de sens : elle est donc interdite.

S'il est impossible d'invoquer une opération définie sur un type, alors créer une instance de ce type n'a pas de sens non plus : il est donc impossible de créer une variable d'un type abstrait. Cela implique que notre programme de test ne compilera plus, une erreur sera signalée sur la ligne 4.

On se trouve ici dans la même situation que celle des classes abstraites en C++, qui contiennent (au moins) une méthode virtuelle pure. De tels types ne sont pas utilisables directement. Par contre, ils prennent tout leur sens lorsque l'on manipule des pointeurs sur ces types, par le mécanisme du polymorphisme associé à la dérivation.

Types dérivés

Ils ont été évoqués depuis le début de cet article, les voici enfin : les types dérivés. Un type dérivé est construit à partir d'un type dit « ancêtre », auquel on ajoute des données et des opérations. Ce principe est également désigné par le terme d'« héritage ».

Créons par exemple un type représentant une ellipse à partir de notre type Geom, dans un paquetage Ellipses :

```
 1 with Geom_Pkg ;
 2 package Ellipses is
 3    type Ellipse is new Geom_Pkg.Geom with
 4    record
 5       rayon_x: Float := 1.0 ;
 6       rayon_y: Float := 1.0 ;
 7    end record ;
 8    overriding
 9    function Aire(e: in Ellipse) return Float ;
10    overriding
11    procedure Homothétie_Position(e : in out Ellipse ;
12                                  sx: in Float ;
13                                  sy: in Float) ;
14    overriding
15    procedure Homothétie_Origine(e : in out Ellipse ;
16                                 sx: in Float ;
17                                 sy: in Float) ;
18    not overriding
19    procedure Put(e: in Ellipse) ;
20 end Ellipses ;
```

Comme nous allons étendre le type Geom défini dans le paquetage Geom_Pkg, il semble naturel de commencer par indiquer son utilisation (ligne 1). La déclaration du nouveau type Ellipse commence ligne 3. La partie is new Geom_Pkg.Geom indique que Ellipse dérive du type Geom_Pkg.Geom. La partie with record..end record donne le contenu de l'extension du type, c'est-à-dire les données que nous lui ajoutons. Si vous ne voulez ajouter aucune donnée supplémentaire, il suffit d'écrire with null record.

Suivent les opérations définies sur ce type. On retrouve la fonction donnant l'aire, les deux types d'homothétie évoquées plus haut, ainsi qu'une nouvelle opération Put() dont la vocation est simplement d'afficher une chaîne représentant l'ellipse (sa position et ses rayons). Deux remarques.

D'abord, l'opération Homothétie_Position() n'est plus abstraite : nous allons lui donner un corps. Dès lors, le type Ellipse ne présentant plus aucune opération abstraite, il devient un type concret : nous pourrons déclarer des variables de ce type.

Ensuite, les déclarations des opérations sont précédées du mot-clef overriding, lui-même précédé de not dans le cas de Put(). Ces qualificatifs ne sont pas obligatoires, mais il est vivement recommandé de les utiliser. Leur signfication est simple.

overriding signale une opération surdéfinie par rapport à celle du type ancêtre. Ici, les deux homothéties existaient pour le type ancêtre Geom. Dérivant de celui-ci, le type Ellipse hérite de ces opérations primitives. Mais elles doivent parfois être adaptées pour tenir compte des spécificités du type dérivé. C'est exactement ce que nous faisons ici. Au contraire, l'opération Put() est nouvelle pour le type Ellipse, elle n'existait pas dans le type ancêtre : elle ne surdéfinie donc aucune opération, d'où la présence de la négation not devant overriding.

Encore une fois, ces informations ne sont pas obligatoires, le compilateur s'y retrouve très bien sans elles. Alors, pourquoi s'embêter à les écrire ? Par prévention, en prévision des modifications futures du code.

Imaginons que pour une raison ou une autre, vous décidiez de supprimer l'opération Homothétie_Position() au niveau du type Geom. Elle est abstraite, aussi cela ne semble-t-il pas avoir beaucoup de conséquences. Sans la présence de overriding dans la surdéfinition au niveau de Ellipse, pas de problème, le paquetage Ellipses compile toujours. Mais ce changement malgré tout assez important peu avoir des conséquences pour d'autres utilisateurs du type Geom. Si overriding a été inscrit, le compilateur vous signalera immédiatement une erreur : en effet, au niveau de Ellipse l'opération Homothétie_Position() ne surdéfinit plus rien, ce qui peut être un avertissement d'un problème de conception. C'est une sécurité contre des manipulations aux conséquences parfois difficilement prévisibles dans un grand programme.

Dans l'autre sens, supposons que vous décidiez d'ajouter une opération Put() au niveau de Geom, sous la forme d'une procédure ne prenant qu'un seul paramètre (de type Geom). Dès lors, l'opération éponyme de Ellipse devient une surdéfinition de celle-ci. Ce n'est pas très grave dans notre exemple, mais dans un programme plus vaste cela peut conduire à des incohérences et des bogues difficiles à identifier. Grâce à la présence de not overriding, là encore le compilateur signalera une erreur, vous invitant à bien vérifier ce que vous faites. Encore une sécurité, destinée à assurer la bonne qualité du programme au fil du temps.

Voici maintenant le corps du paquetage Ellipses :

```
 1 with Text_IO ; use Text_IO ;
 2 package body Ellipses is
 3    overriding
 4    function Aire(e: in Ellipse) return Float is
 5    begin
 6       return 3.14159*e.rayon_x*e.rayon_y ;
 7    end Aire ;
 8    overriding
 9    procedure Homothétie_Position(e : in out Ellipse ;
10                                  sx: in Float ;
11                                  sy: in Float) is
12    begin
13       e.rayon_x := e.rayon_x * sx ;
14       e.rayon_y := e.rayon_y * sy ;
15    end Homothétie_Position ;
16    overriding
17    procedure Homothétie_Origine(e : in out Ellipse ;
18                                 sx: in Float ;
19                                 sy: in Float) is
20    begin
21 --       Geom_Pkg.Homothétie_Origine(Geom_Pkg.Geom(e), sx, sy) ;
22       Geom_Pkg.Geom(e).Homothétie_Origine(sx, sy) ;
23       Homothétie_Position(e, sx, sy) ;
24    end Homothétie_Origine ;
25    not overriding
```

```
26    procedure Put(e: in Ellipse) is
27    begin
28       Put("Ellipse[(" & Float'Image(e.pos_x) & ", " & Float'Image(e.pos_y) &
29          "), " & Float'Image(e.rayon_x) & "x" & Float'Image(e.rayon_y) & "]") ;
30    end Put ;
31 end Ellipses ;
```

On retrouve les `(not) overriding`, scrupuleusement reproduits. Regardez le corps de `Homothétie_Origine()`, lignes 16 à 18. La première est commentée, car elle est exactement équivalente à la seconde, qui utilise la notation pointée. Le but est d'invoquer l'opération ancêtre, celle définie sur le type `Geom` du paquetage `Geom_Pkg`.

Pour cela, on effectue un transtypage du paramètre `e` : l'écriture `Geom_Pkg.Geom(e)` « transforme » `e` pour qu'il présente l'aspect du type `Geom`. Cela n'est naturellement possible que parce que le type `Ellipse` dérive de `Geom` : dans le cas contraire, la conversion serait refusée par le compilateur. Cela n'a rien à voir avec la conversion de type du langage C, qui permet de transformer des carottes en lapins sans que le compilateur n'y trouve rien à redire. On pourrait plutôt l'assimiler au `static_cast<>()` du C++.

Voici un petit programme d'exemple :

```
 1 with Text_IO ; use Text_IO ;
 2 with Geom_Pkg ;
 3 with Ellipses ;
 4 procedure Geom_2 is
 5    e: Ellipses.Ellipse := (pos_x => 0.1,
 6                            pos_y => 0.2,
 7                            rayon_x => 1.0,
 8                            rayon_y => 2.0)  ;
 9 begin
10    e.Put ; New_Line ;
11    e.Homothétie_Position(2.0, 3.0) ;
12    e.Put ; New_Line ;
13    e.Homothétie_Origine(2.0, 3.0) ;
14    e.Put ; New_Line ;
15 end Geom_2 ;
```

Rappelons que le type `Geom_Pkg.Geom` étant abstrait, nous ne pouvons pas créer d'instance de ce type. Par contre le type `Ellipse` est concret : une instance en est déclarée ligne 5, avec un agrégat d'initialisation. Remarquez que les données de `Geom` y figurent, comme s'il s'agissait de données de `Ellipse` – ce qui est plus ou moins le cas.

Voici l'affichage de ce programme :

```
$ ./geom_2
Ellipse[( 1.00000E-01,  2.00000E-01),  1.00000E+00x 2.00000E+00]
Ellipse[( 1.00000E-01,  2.00000E-01),  2.00000E+00x 6.00000E+00]
Ellipse[( 2.00000E-01,  6.00000E-01),  4.00000E+00x 1.80000E+01]
```

Tout cela est finalement assez simple, surtout si vous avez déjà pratiqué la programmation objet. Mais voyons maintenant une petite subtilité, qui aura par la suite de grandes conséquences.

La classe, le type et l'objet

Imaginons une hiérarchie de types, issus de `Geom`, un peu plus fournie. De `Geom` on tire `Triangle`, de `Ellipse` on tire `Ellipse_Inclinée` et `Rectangle`, duquel on obtient `Rectangle_Incliné`. On a donc six types taggés différents. Maintenant, interrogeons-nous un instant sur ce qui fait un type. D'un point de vue théorique, un type est défini par l'ensemble de ses valeurs et l'ensemble des opérations disponibles sur ces valeurs. Cela peut sembler assez évident quand on parle d'un type entier comme `Integer`, mais cela vaut également pour des types taggés comme `Geom` ou `Ellipse`. En Ada, à chaque type taggé `T` est associé une classe désignée par `T'Class`. La classe d'un type `T` est un type indéfini (un peu comme un tableau non contraint) ne possédant pas d'opérations primitives propres, et dont les valeurs sont l'union des valeurs du type `T` et des valeurs des types dérivés de `T`. Voyons cela sur un exemple. Le diagramme suivant est une modélisation UML de la hiérarchie de types évoquée plus haut. Les cadres noirs représentent la « portée » des classes de chaque type dans cette hiérarchie (Fig. 1).

Considérons maintenant un petit programme, contenant une procédure affichant simplement l'aire de l'objet géométrique passé en paramètre :

```
 1 with Text_IO ; use Text_IO ;
 2 with Ellipses ; use Ellipses ;
 3 with Rectangles ; use Rectangles ;
 4 procedure Geom_3 is
 5    procedure Aff_Aire(ell: in Ellipse) is
 6    begin
 7       Put_Line("Aire :" & Float'Image(ell.Aire)) ;
 8    end Aff_Aire ;
 9    e: Ellipse ;
10    r: Rectangle ;
11 begin
12    Aff_Aire(e) ;
13    Aff_Aire(r) ;
14 end Geom_3 ;
```

Tout cela paraît fort bien, surtout du point de vue d'un programmeur C++. Les lignes 12 et 13 semblent naturelles : après tout, un `Rectangle` « est » une `Ellipse`, n'est-ce pas ? Pas en Ada. Si vous l'aviez oublié, Ada est un langage au typage strict. En l'état, ce programme ne compile pas : la ligne 13 est une erreur, car on tente de passer une variable de type `Rectangle` à une procédure qui attend un paramètre de type `Ellipse`. Donc cela ne va pas.

C'est là qu'intervient la notion de « classe ». Modifiez la ligne 5 ainsi :

```
5    procedure Aff_Aire(ell: in Ellipse'Class) is
```

Maintenant la procédure `Aff_Aire()` attend un paramètre de type `Ellipse'Class`, c'est-à-dire une valeur dont le type est `Ellipse` ou l'un des types dérivés (directement ou non) de `Ellipse`. La variable `r` est de type `Rectangle`, qui dérive de `Ellipse`, donc elle peut être passée à la procédure : la ligne 13 n'est plus une erreur. Voici le résultat de l'exécution :

```
$ ./geom_3
Aire : 3.14159E+00
Aire : 4.00000E+00
```

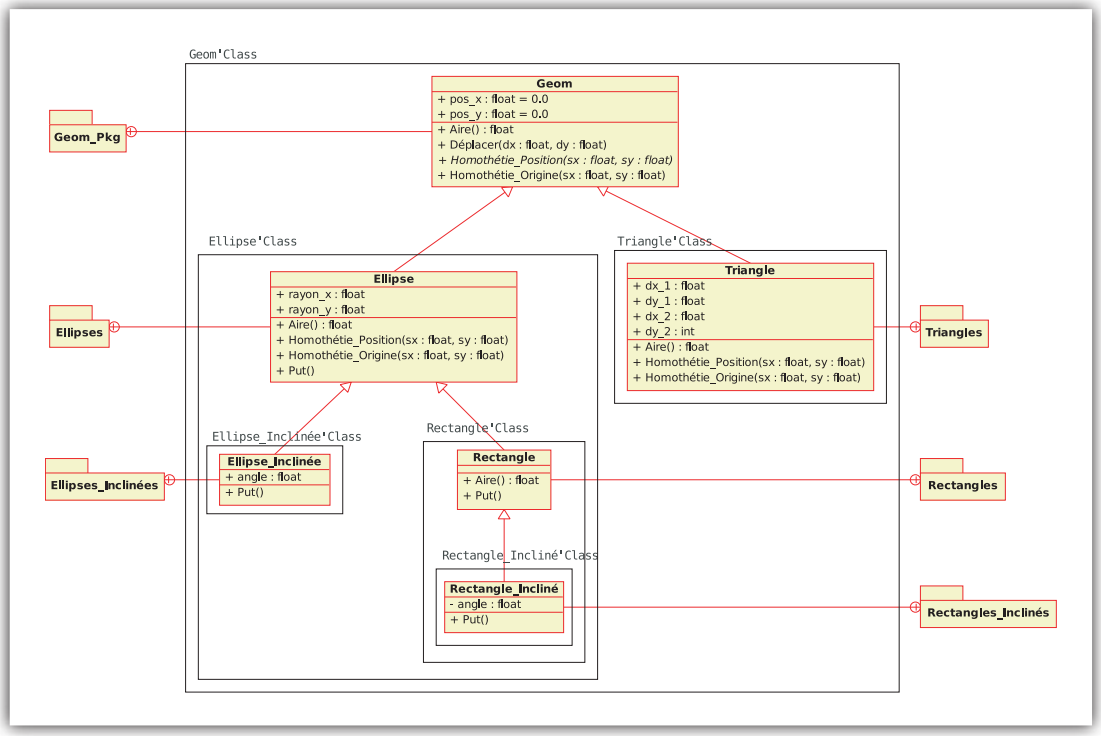

Fig. 1

Remarquez que c'est la « bonne » fonction `Aire()` qui est invoquée : celle du type `Ellipse` lorsque est passée une variable de type `Ellipse`, celle (surdéfinie) du type `Rectangle` lorsque est passée une variable de type `Rectangle`. Nous avons là un bel exemple de polymorphisme.

Le langage Ada fait donc une distinction entre un type objet et la classe de ce type, deux notions qui sont généralement confondues dans la plupart des autres langages objets. Reprenez la comparaison de la déclaration du type `Geom` entre les langages Ada, C++ et Python.

En réalité, ces déclarations ne sont pas tout à fait équivalentes. En C++ et Python, on définit à la fois un type (nommé `Geom`) et une classe de types (nommée `Geom`). En Ada, on ne définit que un type nommé `Geom`, la classe de types associée étant nommée `Geom'Class`.

Considérez les déclarations suivantes :

```
1 ell       : Ellipse ;
2 rect      : Rectangle ;
3 ell_class : Ellipse'Class := rect ;
4 rect_class: Rectangle'Class := ell ;
```

Les deux premières sont « normales ». La troisième déclare une variable de type `Ellipse'Class`, c'est-à-dire de la classe du type `Ellipse`. Il a été signalé plus haut qu'un type classe est indéfini, comme un tableau non contraint : une variable d'un tel type doit donc impérativement être initialisée pour être effectivement définie.

La ligne 3 est correcte, car on affecte à `ell_class` une valeur de type `Rectangle`, laquelle fait bien partie de l'ensemble des valeurs de `Ellipse'Class`, étant donné que `Rectangle` dérive de `Ellipse`.

Si on passait la variable `ell_class` à la procédure `Aff_Aire()` de l'exemple précédent, l'opération `Aire()` invoquée serait celle du type réel de `ell_class`, donc celle de `Rectangle`.

Par contre, la quatrième ligne est invalide. Une valeur de type `Ellipse` ne fait pas partie de l'ensemble des valeurs du type `Rectangle'Class`.

Le polymorphisme n'opère que vers le haut (la racine) de la hiérarchie de types, pas vers le bas. La ligne 4 sera donc refusée par le compilateur.

Cette distinction entre type et classe est fondamentale en Ada. Elle est à la base du mécanisme du polymorphisme tel que proposé par Ada, mécanisme que l'on désigne plus souvent sous le terme de *dispatching*.

Interfaces

Information qui va décevoir les développeurs C++ ou Python : Ada ne supporte pas l'héritage multiple. Information qui va ravir les développeurs Java : Ada supporte la notion d'« interface », depuis le standard Ada 2005.

La version annotée de celui-ci fait même explicitement référence à Java comme inspiration de cette fonctionnalité.

On désigne par interface un « pseudo-type » ne contenant aucune donnée, ne pouvant pas être instancié et n'offrant que des opérations abstraites. Il s'agit donc d'un type purement abstrait.

Ce type peut être utilisé lors de la dérivation d'un type de base, éventuellement avec d'autres interfaces. Cela permet de garantir qu'un type donné supporte au minimum un certain ensemble d'opérations.

Considérons le cas de l'opération `Aire()` de notre petite hiérarchie d'objets géométriques. Chaque type (ou presque) possède son propre principe pour calculer la surface d'une instance.

Par ailleurs, cette notion d'aire n'est pas limitée à des formes en deux dimensions, elle a également un sens pour des formes en trois dimensions – mais pas toutes... On se trouve là typiquement dans la situation où la définition d'une interface devient pertinente.

Reconsidérons donc notre hiérarchie ainsi :

- à la racine se trouve un type `Geom`, pour lequel l'aire n'a pas de sens ;
- de ce type, sont dérivés les types `Ellipse` et `Triangle`, pour lesquels l'aire a un sens ;
- de `Ellipse`, on dérive `Rectangle` ; l'aire a un sens mais est calculée différemment.

Il semble alors logique de créer une interface `Surface`, fournissant une opération `Aire()`. Une représentation UML (partielle) ressemblerait à la figure 2.

Clairement, les trois types `Ellipse`, `Triangle` et `Rectangle` dérivent de deux choses : cela ressemble à de l'héritage multiple, mais cela n'en est pas, car un type ne peut dériver que d'un seul autre type.

Par contre, il peut également « dériver » de plusieurs interfaces, une interface pouvant elle-même dériver de plusieurs interfaces.

Déclarons tout cela dans un paquetage Ada (dans la réalité, chaque type aurait son propre paquetage) :

```
 1 package Geometrie is
 2
 3    type Geom is abstract tagged null record ;
 4
 5    type Surface is interface ;
 6    function Aire(s: in Surface) return Float is abstract ;
 7
 8    type Triangle is new Geom and Surface with null record ;
 9    overriding
10    function Aire(t: in Triangle) return Float ;
11
12    type Ellipse is new Geom and Surface with null record ;
13    overriding
14    function Aire(e: in Ellipse) return Float ;
15
16    type Rectangle is new Ellipse with null record ;
17    overriding
18    function Aire(r: in Rectangle) return Float ;
19
20    procedure Aff_Aire(s: in Surface'Class) ;
21
22 end Geometrie ;
```

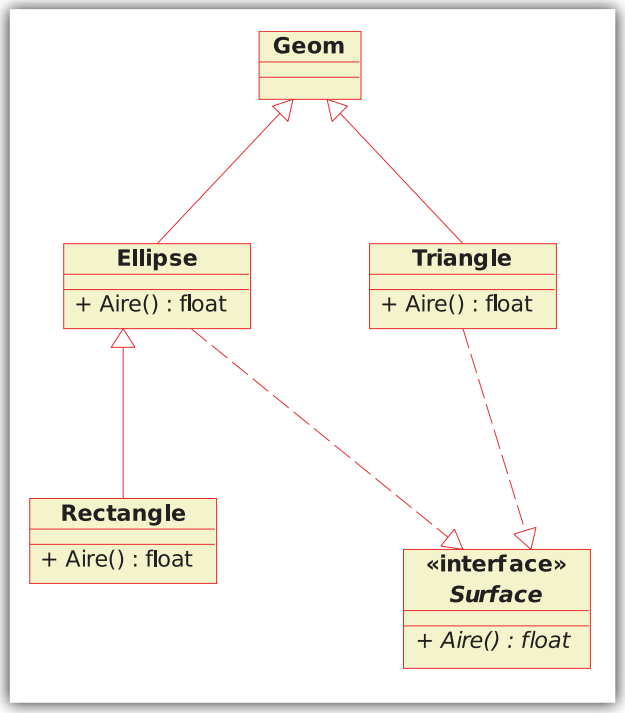

Fig. 2

Pour simplifier, aucun type ne contient de donnée : c'est pourquoi ils sont tous déclarés avec `with null record`, écriture raccourcie pour spécifier un contenu vide.

L'interface `Surface` est déclarée ligne 5, son opération `Aire()` ligne 6. Celle-ci est qualifiée `abstract` : une interface étant toujours un type abstrait, ses opérations sont forcément abstraites. Remarquez qu'on ne lui donne aucun contenu. Ce n'est pas par choix, c'est obligatoire : par définition, une interface ne contient aucune donnée.

Les types `Triangle` et `Ellipse`, qui héritent à la fois du type abstrait `Geom` et de l'interface `Surface`, surdéfinissent l'opération `Aire()`. S'ils ne le faisaient pas, ils demeureraient des types abstraits (car contenant au moins une opération abstraite) et ne pourraient donc pas être instanciés.

Le type `Rectangle` dérive de `Ellipse`. Par conséquent, il possède également l'opération `Aire()`. Elle n'est ici surdéfinie que pour tenir compte de la spécificité du type. Ce n'est pas syntaxiquement obligatoire. `Rectangle` hérite indirectement de `Surface`.

Enfin, on retrouve la procédure `Aff_Aire()` pour afficher l'aire d'une surface. Son paramètre peut être n'importe quel type dérivant (directement ou non) de l'interface `Surface` :

on garantit ainsi que ce paramètre possède au moins les opérations de Surface, sans faire aucune hypothèse sur d'autres opérations ou d'éventuelles données. On pourrait donc passer à Aff_Aire() un paramètre d'un type complètement différent, issu d'un autre paquetage, du moment qu'il dérive de Surface.

Voyons le corps de ce paquetage, volontairement simplifié :

```
 1 with Text_IO ; use Text_IO ;
 2 package body Geometrie is
 3
 4    overriding
 5    function Aire(t: in Triangle) return Float is
 6    begin
 7       return 1.0 ;
 8    end Aire ;
 9
10    overriding
11    function Aire(e: in Ellipse) return Float is
12    begin
13       return 2.0 ;
14    end Aire ;
15
16    overriding
17    function Aire(r: in Rectangle) return Float is
18    begin
19       return 3.0 ;
20    end Aire ;
21
22    procedure Aff_Aire(s: in Surface'Class) is
23    begin
24       Put_Line("Aire =" & Float'Image(s.Aire)) ;
25    end Aff_Aire ;
26
27 end Geometrie ;
```

Et un petit programme d'exemple :

```
 1 with Geometrie ;
 2 procedure Geom_4 is
 3    t: Geometrie.Triangle ;
 4    e: Geometrie.Ellipse ;
 5    r: Geometrie.Rectangle ;
 6 begin
 7    Geometrie.Aff_Aire(t) ;
 8    Geometrie.Aff_Aire(e) ;
 9    Geometrie.Aff_Aire(r) ;
10 end Geom_4 ;
```

Une instance de chaque type est créée, puis on demande l'affichage de l'aire pour chacune, ce qui donne :

```
$ ./geom_4
Aire = 1.00000E+00
Aire = 2.00000E+00
Aire = 3.00000E+00
```

C'est bien la « bonne » opération Aire() qui est invoquée pour chaque instance. Le polymorphisme a encore frappé.

Imaginons un instant un type Segment dérivant de Geom. On peut estimer que la notion d'« aire » n'a pas de sens pour un segment (cela est discutable d'un point de vue mathématique, mais passons là-dessus). Logiquement, Segment n'hériterait pas de l'interface Surface.

Tenter alors de passer une instance de Segment à Aff_Aire() résulterait immédiatement en une erreur de compilation.

Conclusion

Vous pouvez constater que le langage Ada n'a pas grand-chose à envier à d'autres langages concernant les fonctionnalités orientées objet.

Le mois prochain, nous verrons quelques applications de ce principe en relation avec l'encapsulation ultime que constituent les types limités et les types contrôlés.

Yves Bailly