La programmation par objets avec Ada 9X

Stéphane Barbey — Magnus Kempe — Alfred Strohmeier

Ecole Polytechnique Fédérale de Lausanne
Laboratoire de génie logiciel
EPFL–DI–LGL
CH–1015 Lausanne, Suisse
email : {Stephane.Barbey, Magnus.Kempe, Alfred.Strohmeier} @ di.epfl.ch

RÉSUMÉ. Ada 9X, la définition révisée du langage de programmation Ada, intègre la programmation par objets. Cet article examine les nouvelles caractéristiques du langage propres à la programmation par objets, leur utilisation et leur intégration avec le typage fort de Ada. Nous couvrons les mécanismes fondamentaux liés aux objets, tels que l'héritage et le polymorphisme. Nous montrons ensuite comment combiner ces mécanismes en de précieuses pratiques de programmation, comme la programmation par extension, les structures de données hétérogènes, et l'héritage par «mixin».

ABSTRACT. Ada 9X, the revised definition of the Ada programming language, supports object-oriented programming. This paper examines the new, object-oriented features of the language, their use, and how they fit into Ada's strong type system. Basic object-oriented mechanisms are covered, such as inheritance and polymorphism. We then show how to combine these mechanisms into valuable programming techniques; topics include programming by extension, heterogeneous data structures, and mixin inheritance.

MOTS-CLÉS : programmation par objets, Ada, langages de programmation, réutilisation, héritage, polymorphisme, types abstraits, contrôle de visibilité, composition d'abstractions, héritage multiple, «mixins»

KEY WORDS : object-oriented programming, Ada, programming languages, reuse, inheritance, polymorphism, abstract types, control of visibility, composition of abstractions, multiple inheritance, mixins

Cet article a publié dans la revue «Technique et science informatiques» pour un numéro spécial sur le langage Ada, TSI Volume 13 - no 5/1994.

1. Introduction

Dans la tour de Babel des langages de programmation, Ada est connu pour la qualité de sa conception. Plusieurs langages lui ont emprunté des mécanismes, tels les exceptions et la généricité. A l'âge de dix ans, Ada émerge de son premier processus de révision ISO, pendant lequel il a acquis la tournure «objets». Il est naturel de s'attendre à ce que la qualité, la maturité et la puissance accrues du langage attirent beaucoup d'attention.

Bien que Booch [BOO 83] ait montré comment effectuer une conception par objets avec Ada comme langage d'implémentation, Ada 83 lui-même n'est pas qualifié de langage à objets, mais selon la terminologie de Wegner [WEG 87] de langage «basé sur les objets», car il n'offre que des formes restreintes d'héritage et de polymorphisme. Aujourd'hui, tant l'héritage que le polymorphisme sont tenus pour essentiels à la réutilisation du logiciel.

Ces lacunes ont été reconnues par la communauté Ada. En conséquence, le cahier des charges de la révision [ADA 90] contient une section sur la programmation par spécialisation/extension : Ada 9X doit permettre la définition de *«nouvelles entités déclarées dont les propriétés soient adaptées de celles d'entités existantes par l'ajout ou la modification de propriétés ou d'opérations.»*

Après trois ans de travail intensif, l'équipe de révision de Ada 9X a soumis la définition révisée de Ada aux premiers vote ISO et scrutin ANSI. Cette définition inclut entre autres des mécanismes pour la programmation par objets. La version préliminaire du nouveau manuel de référence du langage Ada a été soumise à l'examen public en septembre 1993. Elle a passé avec succès le premier tour de vote, et après quelques corrections mineures une version définitive [ADA 94a] a été publiée en juin 1994 pour le vote d'acceptation final.

Comme Ada 9X est une extension de Ada 83, les programmeurs familiers avec Ada 83 n'ont pas à se soucier du futur de leurs applications présentes. En effet, un soin particulier a présidé à la révision pour assurer la compatibilité ascendante avec Ada 83. En particulier, les mécanismes de Ada 83 ont été exploités au maximum pour satisfaire au cahier des charges de Ada 9X. On a ainsi évité de rendre quantité de mécanismes redondants par l'introduction de nouveaux mécanismes.

Le présent article est structuré en deux parties. Dans un premier temps, nous étudions les mécanismes et la terminologie «par objets» de Ada 9X. Nous expliquons d'abord les principaux mécanismes repris de Ada 83 (dérivation de type, paquetages), puis nous introduisons les solutions que Ada 9X proposent aux lacunes de Ada 83 dans le domaine de l'héritage (les types étiquetés), du polymorphisme (la programmation à l'échelle de classe) et de la visibilité (les unités enfants).

Dans une seconde partie, nous étudions comment Ada 9X répond aux plus importantes techniques de la programmation par objets, à savoir la construction de sous-systèmes à l'aide de bibliothèques hiérarchisées, la construction de types à implémentations multiples par les types abstraits, la programmation au niveau de hiérarchie avec les collections hétérogènes, et la classification multiple avec l'héritage par mixin et l'héritage confraternel.

2. La programmation par objets de Ada 83 à Ada 9X

La programmation par objets (PPO) peut être décrite par une équation symbolique comme une somme de mécanismes de soutien :

PPO = objets + opérations + encapsulation + héritage + polymorphisme

Ada 83 offre déjà les mécanismes relatifs aux objets, leurs opérations et l'encapsulation dans des paquetages. Ces mécanismes sont suffisants pour les types de données abstraits et pour le concept de «classe» tel que défini par Wegner in [WEG 87]. Ada 83 ignore complètement le polymorphisme à l'exécution et ne possède qu'une forme limitée d'héritage—avec raffinement et extension du comportement, mais sans extension de l'état. Cette forme faible de l'héritage est obtenue par *dérivation de type (type derivation)*. Bien que la dérivation de type ne soit pas souvent utilisée en Ada 83, elle constitue la fondation du mécanisme d'héritage en Ada 9X.

Ada 9X tire profit des caractéristiques existant en Ada 83 et les étend avec des mécanismes bien intégrés. Le processus de révision n'a donc pas abouti à un langage hybride : seul un nombre minimal de blocs de construction ont été ajoutés ; de plus, ces adjonctions restent fidèles à la philosophie de Ada 83. Le pouvoir expressif de Ada n'a rien à envier à celui d'autres langages à objets. Il existe cependant quelques différences fondamentales entre le modèle «objet» de Ada 9X et celui d'autres langages à objets bien connus, qu'il s'agisse de la syntaxe ou de la sémantique. Par exemple, la classe n'est pas utilisée simultanément pour les concepts de type et de module, contrairement à C++, Eiffel et Simula.

3. L'héritage

L'héritage est un moyen de construire de façon incrémentale de nouveaux types à partir de types existants, en conservant, en modifiant et en ajoutant des propriétés. En Ada 9X, l'héritage est basé sur trois mécanismes :

Héritage = dérivation + redéfinition d'opération + extension de type

3.1. *La dérivation de type et la redéfinition d'opération*

La dérivation est un mécanisme utilisé pour qu'un nouveau type, le *type dérivé (derived type)*, également appelé le *descendant (descendant)*, hérite de la structure de données et des opérations d'un autre type, appelé le *type père (parent type)*. La structure de données est héritée telle quelle et ne peut pas être changée, sauf pour les types étiquetés, comme nous le verrons. Les opérations héritables d'un type, appelées *opérations primitives (primitive operations)* du type, sont les sous-programmes qui ont un paramètre formel du type, un résultat du type ou un paramètre accès désignant le type, et qui sont déclarés près de la déclaration du type.[1]

Le type dérivé hérite des opérations primitives du type père en remplaçant systématiquement dans leurs profils le type père par le type dérivé.

Par exemple, considérons la déclaration suivante :

```
type Entier is range -1000 .. 1000;
function "+" (Gauche, Droite: Entier) return Entier;
```

1. Plus précisément: avant que la représentation du type ne doive être complètement déterminée.

Chaque type dérivé de Entier hérite automatiquement de la fonction "+", avec la même implémentation que celle du type père :

```
type Longueur is new Entier;
-- function "+" (Gauche, Droite: Longueur) return Longueur;
```

Il faut noter le remplacement systématique du type Entier par le type Longueur, ainsi que le symbole commentaire «--» montrant que l'opération "+" est héritée automatiquement et que le programmeur n'a donc pas à fournir une déclaration explicite.

Un type dérivé peut, à la place de «partager» l'implémentation de l'opération du père, la *redéfinir (override)*, c'est-à-dire fournir une nouvelle implémentation.

```
type Angle is new Entier;
function "+" (Gauche, Droite: Angle) return Angle;
```

Dans cet exemple, une nouvelle implémentation sera fournie pour la fonction "+", pour prendre en compte la nature modulaire de l'addition d'angles.

De nouvelles opérations peuvent être ajoutées à un type dérivé, en les déclarant comme n'importe quelle opération primitive. Il va de soi que les opérations ajoutées ne sont pas disponibles pour le type père.

Aucune opération du type père ne peut être retranchée du type dérivé. L'héritage sélectif, également appelé héritage restrictif, n'est donc pas permis.

3.2. *L'extension de type*

Même si la dérivation de type et la redéfinition d'opération sont en soi des mécanismes puissants, ils ne sont pas suffisants pour mettre en oeuvre complètement l'héritage. Ils pourvoient à la spécialisation du comportement d'un type, mais ne permettent pas au programmeur de spécialiser la structure de données du type. Dans ce but, Ada 9X introduit l'extension de type. *L'extension de type (type extension)* est le mécanisme utilisé pour ajouter des composants à une structure de données. Ce mécanisme est lié à la dérivation de type : le *type étendu (extended type)*, résultat de l'extension, est un nouveau type créé par dérivation d'un type existant et par l'adjonction de composants.

Les types extensibles sont appelés *types étiquetés (tagged types)*. Chaque objet d'un tel type porte une *étiquette (tag)* qui identifie son type. Un type étiqueté, ou bien, a la forme d'un type article avec le mot réservé **tagged** dans sa déclaration, ou bien, est un type dérivé d'un type étiqueté. Seuls les types étiquetés peuvent être étendus par l'adjonction de nouveaux composants.

Voici quelques déclarations de types étiquetés :

```
type Personne is tagged
    record
        Prénom: String (1..4);
    end record;
type Homme is new Personne with
    record
        Barbu: Boolean := False;
    end record;
type Femme is new Personne with
    null record;
```

Dans cet exemple, les types Homme et Femme sont tous deux dérivés de Personne. Les deux héritent des propriétés de Personne. Homme étend Personne en ajoutant un nouveau composant, Barbu. Femme n'ajoute pas de nouveau composant à son type père ; noter la syntaxe «**with null record**» utilisée à cet effet. Personne, Homme et Femme sont des types étiquetés.

Dans une spécification de paquetage, il est possible de déclarer un type étiqueté privé, afin de cacher sa structure de données :

```
type Personne is tagged private;
```

Une extension privée ajoute des composants privés à un type :

```
type Homme is new Personne with private;
```

Dans les deux cas, la déclaration de type complète est cachée dans la partie privée de la spécification de paquetage.

3.3. *La construction d'objets*

En Ada, les objets ne sont pas créés par une construction spéciale, mais simplement déclarés comme appartenant à un type. Leurs composants peuvent être initialisés en donnant une valeur à chacun séparément ou en utilisant un agrégat. Voici quelques exemples :

```
declare
    P: Personne;
    H: Homme;
    F: Femme;
begin
    P.Prénom :=  "Jean";                          -- 1. composant séparé
    P := (Prénom => "Jean");                      -- 2. notation agrégat
    H := (Prénom => "Jean", Barbu => False);
    H := (P with Barbu => False);                 -- 3. agrégat d'extension
    F := (P with null record);
end;
```

Figure 1. *Création d'objets de types étiquetés*

L'agrégat d'extension (extension aggregate) est une forme spéciale d'agrégat pour construire un objet à partir d'un autre objet appartenant au type père. Dans les deux dernières affectations, les objets H et F sont construits à partir de l'objet P.

3.4. *Les opérations de types étiquetés*

Les opérations primitives d'un type étiqueté sont déclarées comme les opérations de n'importe quel type, à la différence notable près qu'un sous-programme ne peut pas être une opération primitive de plus d'un type étiqueté : il n'y a pas de multi-méthodes [CHA 92]. Les paramètres de types étiquetés sont toujours passés par référence. Dans l'exemple suivant, on associe des opérations primitives avec les types Personne, Homme and Femme.

```
package Humanité is
    subtype Nom is String (1..4);

    type Personne is tagged private;
    procedure Nommer (P: in out Personne; N: in Nom); -- nomme une Personne
    function Nom_de (P: Personne) return Nom;     -- rend le nom d'une Personne
    function Titre_de (P: Personne) return String;   -- rend une chaîne vide

    type Homme is new Personne with private;
    function Est_Barbu (H: Homme) return Boolean;
    -- rend True si H est barbu et False dans le cas contraire
    procedure Raser (H: in out Homme);            -- rase un Homme
    function Titre_de (H: Homme) return String;   -- rend "M."

    type Femme is new Personne with private;
    function Titre_de (F: Femme) return String;   -- rend "Mme"
private
    ... -- Personne, Homme et Femme comme définis dans § 3.2
end Humanité;
```

Figure 2. *Opérations primitives de types étiquetés*

Dans la figure 2, les sous-programmes Nommer, Nom_de et Titre_de sont des opérations primitives du type Personne. Par conséquent, elles sont héritées par ses descendants, Homme et Femme. Homme et Femme redéfinissent tous deux l'opération Titre_de et conservent les implémentations de Nommer et Nom_de. Homme ajoute ses propres opérations primitives Est_Barbu et Raser.

Dans la figure 2, les déclarations de type sont groupées dans un paquetage unique par commodité. Les extensions de type n'ont pas besoin de figurer dans le paquetage où le type père est déclaré, mais un type dérivé doit être déclaré de façon à ne pas disparaître avant son type père. Cette propriété est satisfaite pour des types déclarés dans des spécifications de paquetage de bibliothèque. Elle est vitale pour les structures de données hétérogènes.

Une opération primitive d'un type étiqueté est appelée comme n'importe quel autre sous-programme :

```
declare
    P: Personne := ...;
    H: Homme := ...;
    F: Femme := ...;
begin
    -- opérations héritées
```

```
        Nommer (H, "Jean");              -- appels aux implémentations
        ... Nom_de (H) ...               -- définies pour le type Personne

        -- opérations redéfinies
        ... Titre_de (P) ...             -- rend une chaîne vide
        ... Titre_de (H) ...             -- rend “M.”
        ... Titre_de (F) ...             -- rend “Mme”

        -- opération ajoutée
        ... Est_Barbu (H) ...            -- rend True ou False
    end;
```

Figure 3. *Appels d'opérations primitives de types étiquetés*

Il faut noter qu'il n'y a pas de différence syntaxique dans un appel d'opération entre l'objet pour lequel l'opération est primitive et les autres paramètres. D'autre langages à objets recourent souvent à une notation préfixée, par exemple «H.Est_Barbu».

Les opérations ajoutées à un type dérivé ne sont évidemment pas disponibles pour le type père. L'appel «Raser (P)» résulterait en une erreur à la compilation, car cette opération est seulement définie pour Homme et ses descendants potentiels.

3.5. *La conversion de type*

Ada 9X, comme Ada 83, a seulement des *conversions de type (type conversions)* explicites, et exclut donc, pour des raisons de sécurité, les conversions de type implicites. Pour des types étiquetés, un objet peut être converti à un autre type si le type cible est un ancêtre du type de l'objet. Il existe deux genres de conversions de type : les conversions de valeur et de vue.

Une *conversion de valeur (value conversion)* est effectuée dans une affectation :

```
    declare
        P: Personne := ...;
        H: Homme := ...;
        F: Femme := ...;
    begin
        P := H;                  -- illégal (pas de conversion implicite de type)
        P := Personne (H);       -- légal (conversion de valeur)

        H := Homme (P);          -- illégal (Homme n'est pas un ancêtre de Personne)
        H := Homme (F);          -- illégal (Homme n'est pas un ancêtre de Femme)
    end;
```

Figure 4. *Conversions de valeur*

Le résultat d'une conversion de valeur est un objet dont les valeurs des composants sont reprises de l'objet converti et dont l'étiquette est celle du type cible. Les composants additionnels sont perdus lors de la conversion. Par exemple, en convertissant de Homme vers Personne, le composant Barbu est perdu.

Une *conversion de vue (view conversion)* fournit une vue différente d'un objet quand il est paramètre effectif dans un appel de sous-programme, sans affecter son contenu : les composants supplémentaires deviennent invisibles, mais ne sont pas supprimés. Une conversion de vue préserve donc l'étiquette de l'objet, contrairement à une conversion de valeur. Comme nous le montrerons plus tard, le but principal d'une conversion de vue est d'invoquer l'opération du type père si elle a été redéfinie pour le type dérivé. Ainsi, «Operation (Personne (H))» appelle l'opération implémentée pour Personne, alors que «Operation (H)» appelle celle implémentée pour Homme.

4. Les classes et le polymorphisme

Tous les types dérivés et étendus (directement ou indirectement) d'un type étiqueté particulier appartiennent à une *classe de dérivation (derivation class)*, dont ce *type* est la *racine (root type)* ; une classe ne désigne donc pas une collection d'instances, mais une hiérarchie ouverte de types, comprenant l'union de leurs ensembles de valeurs et opérations. Cette définition de *classe (class)* est spécifique à Ada, et déjà présente dans Ada 83.

De façon générale, on peut diviser les langages de programmation en deux groupes, les langages typés et les non typés. Parmi les premiers, le typage statique fort est le plus proche du but originel du typage, à savoir la détection à la compilation d'erreurs de programmation. Une variable typée statiquement ne peut dénoter des objets que d'un seul type spécifique. Cependant, comme tous les objets d'une classe (de dérivation) partagent certaines propriétés —à savoir la structure de données et le comportement du type à la racine de la classe de dérivation— il est parfois commode de les manipuler de façon uniforme. Il y a donc un besoin de disposer de variables qui peuvent dénoter dynamiquement des objets de différents types, avec un besoin concomitant de trouver les implémentations des opérations associées à chaque type ; en bref, il existe un besoin pour le *polymorphisme (polymorphism).*

La réponse à ce besoin comporte trois ingrédients : les entités à l'échelle de classe, la surcharge et l'aiguillage (la liaison dynamique).

4.1. *Les entités à l'échelle de classe*

Un type à l'échelle de classe est associé implicitement à chaque type étiqueté ; il dénote la classe de dérivation, dont le type étiqueté est la racine. Un type à l'échelle de classe ne possède pas un véritable nom, mais il peut être désigné par l'attribut '**Class** appliqué au nom d'un type étiqueté. Les types déclarés explicitement sont aussi appelés des *types spécifiques (specific types)* pour souligner la différence avec les types à l'échelle de classe.

Dans la figure 2, Personne, Homme et Femme sont des types spécifiques définissant trois classes de dérivation, dénotées par Personne'Class, Homme'Class et Femme'Class, respectivement. Les types spécifiques Personne, Homme et Femme appartiennent à la classe de dérivation Personne'Class. Homme est la racine de la classe de dérivation Homme'Class ; Homme'Class est une sous-classe de Personne'Class. Tous les descendants potentiels de Homme appartiennent à la classe Homme'Class, mais pas les ancêtres de Homme, tel Personne, ni ses frères et soeurs, telle Femme.

Les variables et les paramètres formels, à la place d'être d'un type spécifique, peuvent être d'un type à l'échelle de classe ou dénoter des objets d'un tel type ; tous ces cas correspondent à des *entités à l'échelle de classe (class-wide entities).*

Les *variables à l'échelle de classe (class-wide variables)* fournissent la sémantique de valeur pour les classes. Un type à l'échelle de classe ne véhicule pas assez d'information pour créer un objet ; par conséquent, en déclarant une variable d'un type à l'échelle de classe, il faut toujours fournir une valeur initiale explicite. Cette valeur initiale fixera le type spécifique de la variable et donc son étiquette, pour toute la durée de vie de la variable :

E : Personne'Class := ...; -- *E, comme être humain*

E est une variable du type à l'échelle de classe Personne'Class, ou, en bref, une variable de la classe Personne, ce qui signifie que E est une Personne spécifique, un Homme spécifique, une Femme spécifique, ou un autre descendant spécifique. Son type spécifique est déterminé par sa valeur initiale obligatoire, qui peut être une expression calculée dynamiquement.

Un paramètre formel d'un sous-programme peut aussi être d'un type à l'échelle de classe. Tout paramètre effectif appartenant à la classe se conforme à un tel paramètre formel. Un tel sous-programme n'est pas attaché à un type spécifique, mais à tous les types dans une classe donnée, et c'est pourquoi nous l'appelons *sous-programme à l'échelle de classe (class-wide subprogram).* Voici un exemple :

procedure Afficher (E: **in** Personne'Class);

Tout paramètre effectif appartenant au type à l'échelle de classe Personne'Class, au type spécifique Personne ou à n'importe lequel de ses descendants (dans notre cas Homme et Femme) se conforme au paramètre formel.

4.2. *La surcharge*

La *surcharge (overloading)* est une forme statique de polymorphisme ad-hoc : un seul nom d'opération correspond à plusieurs implémentations différentes de sous-programmes, non nécessairement apparentés. Bien qu'ils aient tous le même nom, leurs profils sont différents. Quand une opération surchargée est appelée, l'implémentation de sous-programme correcte est choisie à la compilation, ce qui veut dire que la surcharge est résolue par une *liaison statique (static binding)* d'une implémentation à un appel d'opération.

L'héritage d'une opération est une forme subtile de surcharge. En dérivant un type, les opérations héritées surchargent les opérations primitives du type père avec des opérations ayant le même profil, excepté le remplacement systématique du type père par le type dérivé.

Sur la figure 3, les appels Titre_de (P), Titre_de (H) et Titre_de (F) sont liés aux implémentations de sous-programme pour les types Personne, Homme et Femme. On peut remarquer que Nom_de est surchargé, alors même que l'implémentation en est unique.

4.3. *L'aiguillage*

A l'appel d'une opération aiguillante avec un paramètre effectif à l'échelle de classe, il est en général impossible de sélectionner à la compilation l'implémentation de l'opération. Mais à l'exécution, il est possible d'*aiguiller (dispatch)* vers l'implémentation correcte selon l'étiquette du paramètre effectif ; la *liaison* du code approprié à un appel d'opération est donc *dynamique (dynamic binding)* et peut varier pour chaque exécution d'une seule et même expression d'appel.

En Ada 9X, les opérations primitives des types étiquetés sont potentiellement aiguillantes et sont donc appelées des *opérations aiguillantes (dispatching operations)*. S'il est impossible de lier un appel à une telle opération à la compilation, il s'agit d'un *appel aiguillant (dispatching call)* ; un appel aiguillant a au moins un paramètre effectif à l'échelle de classe, ou il en va ainsi du type de son résultat. Le fait que l'aiguillage se produise n'est donc pas une propriété de l'opération, mais est lié à la présence d'au moins une entité à l'échelle de classe, appelée un *opérande de contrôle (controlling operand)*. Cette distinction permet un réglage fin, pour chaque appel individuel, entre liaison statique et dynamique.

L'aiguillage se rencontre typiquement dans le corps d'un sous-programme avec un paramètre à l'échelle de classe :

```
procedure Afficher (E: in Personne'Class) is
begin
    ... Titre_de (E) ...   -- Appel aiguillant : l'implémentation est choisie
end Afficher;              -- selon l'étiquette du paramètre effectif désigné par E.
```

Un autre cas d'aiguillage est l'utilisation d'une variable à l'échelle de classe, et ce cas est une occasion de mettre en valeur la différence avec un appel «ordinaire» :

```
declare
    P1: Personne;
    P2: Personne'Class := ...;
begin
    ... Titre_de (P1) ...  -- Liaison statique : le type spécifique est connu à la compilation
    ... Titre_de (P2) ...  -- Liaison dynamique : le type spécifique n'est pas connu à la
                           -- compilation ; l'implémentation appropriée est choisie pendant
end;                       -- l'exécution
```

4.4. *L'opérateur d'appartenance*

Il arrive que le code varie quelque peu selon le type spécifique d'une certaine valeur. L'*opérateur d'appartenance (membership operator)* **in** est utile dans ce cas :

```
procedure Afficher (E: in Personne'Class) is
begin
    Put (Titre_de (E) & Nom_de (E));
    if E in Homme'Class then    -- test d'appartenance
        Put (", masculin");
        if Est_Barbu (Homme'Class (E)) then
            Put (" est barbu");
        end if;
    elsif E in Femme'Class then
        Put (", féminin");
    end if;
end Afficher;
```

Figure 5. *Test d'appartenance à un type à l'échelle de classe*

L'opération Est_Barbu peut être appliquée uniquement à un paramètre effectif E dont le type spécifique est dans la classe Homme, ce qui est la raison du test d'appartenance. Notons encore que l'appartenance à la sous-classe Homme est utilisée : «E **in** Homme'Class», plutôt que l'appartenance au type spécifique Homme : «E **in** Homme». On prend ainsi en compte les descendants potentiels de Homme.

4.5. *L'invocation d'implémentations obscurcies*

Très souvent, l'implémentation d'une opération primitive d'un type dérivé a besoin d'appeler l'implémentation du type père, qu'elle a redéfinie. Comme elle est surchargée, l'implémentation pour le père est dite *obscurcie (shadowed)* par l'implémentation pour le descendant. En Smalltalk, un appel à une implémentation (méthode) obscurcie est accompli par l'envoi d'un message à «super».

Supposons qu'une procédure Afficher (P: Personne) ait été ajoutée au type Personne. L'implémentation de la version redéfinie pour le type dérivé Homme pourrait être la suivante :

```
procedure Afficher (H: Homme) is
begin
    Afficher (Personne (H));  -- appel d'une implémentation obscurcie
    Put (", masculin");
    if Est_Barbu (H) then
        Put (" est barbu");
    end if;
end Afficher;
```

Une conversion de vue est utilisée pour invoquer l'implémentation obscurcie ; ainsi un appel à Afficher n'aboutit pas à un appel récursif infini.

4.6. *Le réaiguillage*

Un type dérivé peut toujours redéfinir une opération primitive de son type père, c'est-à-dire fournir sa propre implémentation. En présence de classes, il faut veiller à distinguer entre un appel à l'implémentation du père et un appel à l'implémentation fournie par le type dérivé.

Considérons l'implémentation suivante de Afficher (P: Personne) :

```
procedure Afficher (P: Personne) is
begin
    Put (Titre_de (P) & ...); -- attention, liaison statique !
end Afficher;
```

Si l'implémentation de Afficher pour Homme (§ 4.5) est appelée, par exemple Afficher (H), l'implémentation (obscurcie) du père est appelée dans la première instruction : Afficher (Personne (H)). A l'intérieur de l'implémentation obscurcie, Titre_de est appelé avec un opérande de contrôle ayant le type spécifique Personne, et cette opération est donc liée statiquement à l'implémentation fournie par Personne. Comme résultat, un Homme recevrait le même titre qu'une Personne, à savoir une chaîne vide, alors qu'on voudrait «M.».

Il faut donc veiller à ne pas appeler les opérations primitives de Personne, mais à appeler les opérations du type spécifique actuellement utilisé :

```
procedure Afficher (P: Personne) is
begin
    Put (Titre_de (Personne'Class (P)) & ... ;       -- réaiguillage (redispatching)
end Afficher;
```

Dans le corps de l'implémentation pour Personne, la conversion du paramètre P vers le type Personne'Class fournit le réaiguillage vers l'implémentation correcte de Titre_de, en fonction de l'étiquette de P.

5. Les hiérarchies d'unités : extension séparée et contrôle de visibilité

La programmation par objets offre de grands avantages dans le cadre de la maintenance des systèmes et de la *programmation à grande échelle (megaprogramming)*. Ada 9X apporte des soutiens additionnels dans ce cadre, avec les sous-systèmes, qui sont des hiérarchies d'unités de bibliothèques, avec l'extension de module séparée, et avec le contrôle de visibilité, qui détermine ce qu'un module de services montre d'une part à ses clients et d'autre part à des modules de services intimement liés.

Une nouvelle construction, les *unités filles (child units)*, permet de créer des hiérarchies, génériques ou non, d'unités de bibliothèque et de partager des déclarations privées à l'intérieur d'une hiérarchie d'unités. Une unité fille est liée à une *unité mère (parent unit)*, et transitivement aux mères des unités mères, les *unités ancêtres (ancestor unit)*. Les déclarations publiques et privées d'une unité mère sont visibles par ses filles, mais les déclarations du corps de l'unité mère ne le sont pas. Cette règle de visibilité s'applique par transitivité aux déclarations publiques et privées des unités ancêtres, et à celles contenues dans leurs corps. Les déclarations privées d'une unité mère sont disponibles dans la partie privée d'une fille, et donc dans le corps de la fille, mais pas dans l'interface publique de la fille ; sans cette dernière règle, les déclarations privées d'une unité pourraient devenir publiques par l'intermédiaire d'unités filles, ce qui casserait l'abstraction et l'encapsulation de l'unité mère, violant ainsi un principe du génie logiciel, le cachement de l'information.

Ada 9X offre donc un espace de noms hiérarchisé pour la bibliothèque. Par contraste, en Ada 83 la bibliothèque offrait un espace de noms plat pour les spécifications d'unités de bibliothèque.[2]

5.1. *L'extension séparée*

Grâce à l'extension par des unités filles, il est possible d'ajouter des entités à une unité de bibliothèque, sans toucher au code source de l'original. Par exemple un tel mode d'extension peut être rendu nécessaire, soit par un gestionnaire de configuration qui interdit de modifier un paquetage existant, soit pour éviter des tests de régression étendus, soit parce que l'on n'est pas propriétaire du code source.

L'extension séparée de paquetage satisfait un élément du cahier des charges de Ada 9X, qui demandait qu'on donne au programmeur un contrôle accru dans la gestion de la bibliothèque, dans le but de faciliter le développement en équipe et de diminuer les besoins en recompilation. D'autres avantages de l'extension séparée sont qu'une telle approche peut réduire la taille du code objet, puisque l'éditeur de liens n'inclura pas du code situé dans des unités filles non référencées, et que l'ordre d'élaboration peut être simplifié.

5.1.1. *Illustration : Extension d'interface*

Etant donné un type de données abstrait (abstract data type, ADT) pour la gestion de queues de messages, un parcoureur générique peut être ajouté à cet ADT, sans toucher au code original. Le parcoureur doit être dans une unité fille, puisqu'il a besoin de voir la représentation privée de l'ADT afin de le parcourir efficacement.

```
with Messages; use Messages;
package Queues_de_Messages is
    type Queue_Type is limited private;
    procedure Vider (Queue : in out Queue_Type);
    procedure Insérer (Queue : in out Queue_Type; Message : in Message_Type);
    procedure Retirer (Queue : in out Queue_Type; Message : out Message_Type);
private
    ...
end Queues_de_Messages;
```

Figure 6. *Spécification d'un ADT pour queues de messages*

```
-- note : la clause de contexte de l'unité mère est «héritée»
generic
    with procedure Action (Message : in Message_Type);
procedure Queues_de_Messages.Parcourir_G (Queue : in Queue_Type);
```

Figure 7. *Spécification d'une unité fille pour le parcours d'une queue de messages*

Si la déclaration de type complète de Queue_Type était un type accès à un type incomplet, dont la déclaration complète se trouve dans le corps du paquetage, la représentation ultime du type queue serait en fait inaccessible depuis l'unité fille. Il serait alors impossible de créer des unités filles dépendant de cette représentation, puisqu'une unité fille ne peut accéder au corps de sa mère. Alternativement, si un type privé est déclaré dans un paquetage imbriqué, sa représentation est inaccessible car une unité fille n'a pas visibilité sur la partie privée d'un paquetage imbriqué.

Ainsi, il est possible d'empêcher autrui d'accéder par l'entremise d'unités filles à la représentation d'un ADT exporté par un type privé ; il suffit pour cela d'utiliser soit un type accès vers un type incomplet (mode de représentation que l'on nommera peut-être type «opaque» ou «obscur») soit un type privé déclaré dans un paquetage imbriqué. On observe dans le premier cas qu'il devient important de distinguer les déclarations de la partie privée et celles du corps d'un paquetage, pour des raisons de visibilité sur lesquelles nous revenons plus loin.

5.1.2. *Illustration : Réduction des besoins en recompilation*

Etant donné un ADT de transactions (disposant des opérations usuelles, telles que «Start», «Commit» et «Rollback»), des services additionnels peuvent être ultérieurement fournis par une unité fille afin de produire un audit des transactions et un journal. Ainsi les services d'audit voient les déclarations privées de l'ADT afin de journaliser les étapes d'une transaction, mais l'ADT original n'est pas touché.

De cette manière, les clients de l'ADT qui ne dépendent pas des services d'audit fournis par l'unité fille n'ont pas besoin d'être recompilés, n'ont pas besoin de repasser par l'édition des liens, et n'ont pas même besoin de subir des tests de régression.

```
package Transactions is
    type Transaction_Type is limited private;
    procedure Start (T : in out Transaction_Type);
    procedure Commit (T : in out Transaction_Type);
    procedure Rollback (T : in out Transaction_Type);
```

2. On peut considérer que les unités séparées offrent une forme restreinte de hiérarchie ; par exemple, des unités séparées peuvent être développées en une hiérarchie d'unités de compilation, mais elles sont confinées aux corps des unités de bibliothèque.

```
private
    ...
end Transactions;
```

Figure 8. *Squelette d'un ADT de transactions*

```
package Transactions.Auditing is
    type Audit_Type is limited private;
    procedure Create (A : in out Audit_Type; T : in Transaction_Type);
    procedure Register (A : in out Audit_Type; T : in Transaction_Type);
    procedure Log (A : in out Audit_Type; T : in Transaction_Type);
private
    ...
end Transactions.Auditing;
```

Figure 9. *Squelette d'un ADT pour l'audit de transactions*

Une clause de contexte **with** portant sur une unité fille inclut implicitement dans le contexte tous les ancêtres de cette unité. De plus, une unité fille n'a pas besoin d'inclure sa mère dans une clause de contexte, car les règles de visibilité s'appliquent comme si la spécification de l'unité fille apparaissait textuellement dans la spécification de l'unité mère, en séparant les parties publiques et privées.

5.2. *Le contrôle de visibilité*

Le problème général de la visibilité peut être exprimé comme suit : dans de grands systèmes modulaires, il est nécessaire de décider et contrôler ce qui dans un module donné peut être vu et ne peut pas être vu. Des niveaux intermédiaires de visibilité sont possibles, par exemple en autorisant un groupe particulier d'unités du système à voir quelques composants ou opérations d'un module.

5.2.1. *Les niveaux de visibilité*

Le problème est similaire que l'on considère des types, des paquetages, ou des classes de dérivations. Des propriétés peuvent être, soit *publiques*, c'est-à-dire visibles partout ; soit *privées*, c'est-à-dire visibles uniquement dans un cercle d'amis (on aura reconnu les «friends» de C++) ou parmi les descendants, c'est-à-dire les unités filles ; soit *internes*, c'est-à-dire visibles nulle part ailleurs, même pas dans les unités filles. Par analogie, on peut considérer que ce qui se trouve devant notre maison est public (tout passant peut le voir), ce qui est dans la maison est privé (seuls ceux qui y vivent, soit la famille, peuvent le voir), et ce qui est dans notre corps est interne (notre coeur ne peut être vu, même par notre famille).

5.2.2. *Les moyens de contrôle*

Il y a donc trois niveaux de contrôle : les propriétés publiques sont dans l'interface (la spécification de paquetage), les propriétés privées sont dans la partie cachée de l'interface (la partie privée de la spécification de paquetage), et les propriétés internes sont dans l'implémentation (le corps de paquetage). La distinction entre privé et interne n'était pas nécessaire avec Ada 83 ; elle apparaît avec Ada 9X à cause du nouveau concept d'unité fille et du besoin concomitant de contrôler les niveaux de visibilité.

L'unité fille est un mécanisme qui se marie souvent avec l'usage de l'héritage. Ainsi, Ada 9X offre des mécanismes différents pour deux besoins différents :

- l'héritage est en règle générale une relation client-fournisseur à couplage faible et il n'est donc pas souhaitable que le client accède à la partie privée du fournisseur ; une clause de contexte établit la relation client-fournisseur.
- l'héritage est parfois aussi un outil du fournisseur pour une construction à forte cohésion interne où il est nécessaire d'accéder à la partie privée avec une unité fille ; la hiérarchie unité mère-unité fille établit explicitement la dépendance de représentation entre un fournisseur et l'autre.

Ada se distingue ainsi fortement de Eiffel, qui fusionne les classes avec les modules, et mélange par là le couplage faible entre un client et son fournisseur avec la cohésion forte entre deux fournisseurs.

6. Les types abstraits

On aborde ici le dernier mécanisme de Ada 9X que nous introduisons dans cet article avant d'examiner comment appliquer ces mécanismes dans des techniques de programmation adéquates.

Un *type abstrait* (*abstract type*, à ne pas confondre avec un *type de données abstrait*, c'est-à-dire un ADT) fournit la spécification d'une classe de dérivation, sans définir une implémentation complète d'un type dans cette classe ; des implémentations complètes devront être fournies ultérieurement. Un type abstrait est une interface sans représentation de données complète ni algorithmes complets. Une telle interface peut être partagée par de nombreuses et différentes implémentations de la même abstraction. De telles implémentations sont fournies par des types dérivés dans la classe de dérivation dont le type abstrait est la racine.

Par analogie avec une spécification de paquetage, un type abstrait ressemble à une spécification de type. La différence est que l'on ne possède soit pas la connaissance nécessaire soit pas le désir pour écrire le «corps» du type abstrait, puisqu'il y a potentiellement de nombreuses et différentes implémentations satisfaisant à la spécification. Un type abstrait fournit une spécification générale sous la forme d'un nom de type, des noms et profils d'opérations associées, et peut-être une représentation partielle de l'état et une implémentation partielle du comportement.

Un type abstrait a le mot réservé **abstract** dans sa déclaration et doit être étiqueté. Aucun objet ne peut être déclaré comme appartenant à un tel type, puisqu'il ne s'agit pas d'un *type concret (concrete type)*, au même sens que Integer, Character, et Boolean le sont. Un type abstrait est une sorte «d'esquisse» pour d'autres types, et des types concrets doivent être fournis ultérieurement—par l'intermédiaire d'extensions de type—afin d'utiliser les propriétés définies par le type abstrait.

```
type A is abstract tagged null record;
type C is new A
    with record
        I : Integer;
    end record;
```

Figure 10. *Déclaration d'un type abstrait et d'un type concret dérivé du type abstrait*

6.1. *Les sous-programmes abstraits*

Les *sous-programmes abstraits (abstract subprograms)* sont liés aux types abstraits : ils n'ont pas d'implémentation. Quelques opérations primitives, mais pas nécessairement toutes, associées à un type abstrait peuvent être abstraites, c'est-à-dire peuvent avoir une spécification sans implémentation. Un sous-programme abstrait ne peut pas être invoqué autrement que par un appel aiguillant (les paramètres effectifs appartenant ainsi nécessairement à des types concrets avec redéfinitions concrètes). Un type abstrait n'a pas forcément d'opération abstraite ; dans ce cas, son propos principal est simplement d'être la racine d'une classe de dérivation.

```
type A is
    abstract tagged null record;
procedure Déplacer (X : A) is abstract;
```

Figure 11. *Déclaration d'un type abstrait avec une opération abstraite*

Les types étiquetés concrets ne peuvent pas avoir d'opération primitive abstraite ; les opérations abstraites d'un type abstrait doivent donc être redéfinies pour tout type concret dérivé de ce type abstrait. Un sous-programme abstrait est essentiellement un modèle qui doit être redéfini pour les types dérivés concrets.

Seul un type abstrait peut hériter d'une opération abstraite sans la redéfinir. Une fonction opération primitive d'un type étiqueté et renvoyant un objet de ce type devient une opération abstraite lorsqu'elle est héritée sans redéfinition.

```
type C is new A
    with record
        I : Integer;
    end record;
procedure Déplacer (X : C);
```

Figure 12. *Type dérivé concret avec redéfinition d'une opération abstraite*

On peut considérer un type abstrait comme le résultat d'une factorisation d'éléments communs entre un groupe de types, les éléments factorisés étant trop abstraits pour que la création d'objets ne possédant que ces propriétés ait un sens. Par la suite, le type abstrait doit être étendu afin d'avoir un type concret et d'être directement utilisable pour des déclarations d'objets.

6.2. *Exemple : Pile abstraite*

Considérons une pile abstraite que l'on peut définir comme un type abstrait avec trois opérations, Empiler, Dépiler et Taille, sans présumer d'aucune implémentation particulière de la structure de données. Cette spécification peut être implémentée par divers types de piles concrets (bornées ou non, avec liste libre ou non, concurrentes ou séquentielles, etc.).

6.2.1. *Spécification abstraite de piles*

La spécification d'une pile abstraite est similaire au composant «pile» habituel que l'on connaît bien en Ada 83, sauf que le type exporté est abstrait, et que Empiler et Dépiler sont des sous-programmes abstraits. Dans cet exemple (figure 13), la représentation partielle d'une pile stocke sa cardinalité (le nombre d'éléments présentement empilés), et le sous-programme Taille peut donc être implémenté. Le corps du paquetage (pas montré ici) implémente uniquement l'opération Taille.

```
generic
    type Elément_Type is
        private;
package Piles_G is
    type Pile_Type is
        abstract tagged limited private;

    procedure Empiler (P : in out Pile_Type; Item : in Elément_Type) is abstract;
    procedure Dépiler (P : in out Pile_Type; Item : out Elément_Type) is abstract;
    function Taille (P : Pile_Type) return Natural;

    Erreur_Structure_Vide : exception;
private
    type Pile_Type is
        abstract tagged limited record
            Card : Natural := 0;
        end record;
end Piles_G;
```

Figure 13. *Spécification d'un type de piles abstrait*

6.3. *Les types abstraits ressemblent aux unités génériques*

Un type concret dérivé d'un type abstrait doit fournir toutes les opérations aiguillantes définies pour ce type abstrait. Il y a là un parallèle entre les génériques et les types abstraits. Une unité générique doit être instanciée avec des paramètres effectifs pour avoir une unité directement utilisable, c'est-à-dire un paquetage ou un sous-programme. Un type abstrait doit être dérivé avec redéfinitions des opérations primitives abstraites par des sous-programmes concrets afin d'avoir un type directement utilisable, c'est-à-dire un type concret avec lequel déclarer des objets.

6.3.1. *Spécification concrète de piles bornées*

Un type concret de piles bornées est dérivé dans une unité fille, car les déclarations privées de Piles_G doivent être visibles pour l'implémentation. Tant Empiler que Dépiler doivent être redéfinis pour le type concret. La représentation est exactement la même que ce que l'on écrivait hier en Ada 83 pour représenter une pile bornée avec un tableau.

```
generic
package Piles_G.Bornées_G is
    type Pile_Type (Capacité : Positive) is      -- note : discriminant ajouté par
        new Piles_G.Pile_Type                    -- l'extension de type
        with private;

    procedure Empiler (P : in out Pile_Type; Elément : in Elément_Type);
    procedure Dépiler (P : in out Pile_Type; Elément : out Elément_Type);
    -- la fonction Taille est héritée ; elle est concrète et n'a pas besoin d'être redéfinie

    Erreur_Structure_Vide : exception renames Piles_G.Erreur_Structure_Vide;
private
    type Table_Type is
        array (Positive range <>) of Elément_Type;
    type Pile_Type (Capacité : Positive) is
        new Piles_G.Pile_Type
        with record
            Elements : Table_Type (1 .. Capacité);
            Sommet : Natural := 0;
        end record;
end Piles_G.Bornées_G;
```

Figure 14. *Spécification d'un type de piles bornées concret dérivé d'un type abstrait*

6.3.2. *Utilisation des piles bornées*

Nous montrons ci-dessous comment instancier Piles_G avec des entiers, produisant ainsi un type abstrait «piles d'entiers». Ensuite, Piles_G.Bornées_G est instancié afin de produire un type concret «piles bornées d'entiers».

```
with Piles_G;
package Piles_d_Entiers is              -- spécification abstraite de piles d'entiers
    new Piles_G (Integer);

with Piles_G.Bornées_G;
package Piles_d_Entiers.Bornées is      -- définition de piles bornées d'entiers concrètes
    new Piles_G.Bornées_G;

with Piles_d_Entiers.Bornées;
procedure Client is
    type Pile_Type is
        new Piles_d_Entiers.Bornées.Pile_Type (100)
        with null record;
    Une_Pile : Pile_Type;
begin
    Empiler (Une_Pile, 3);
end Client;
```

Figure 15. *Instanciation de piles bornées d'entiers, et leur usage*

7. La sémantique de référence et les collections hétérogènes

Au contraire d'Eiffel, de Smalltalk, et de nombreux autres langages de programmation par objets, Ada 9X, comme Ada 83, encourage l'utilisation de la sémantique de valeur. Sa philosophie est de travailler directement avec les objets plutôt qu'avec des références aux objets. Néanmoins, le développeur peut au besoin obtenir une sémantique de référence au moyen des types accès.

7.1. *La sémantique de référence*

Alors que, en Ada 83, les valeurs d'un type accès (appelé pointeur dans d'autres langages) ne peuvent désigner que des objets créés dynamiquement au moyen de l'opérateur **new**, Ada 9X introduit les types accès généralisés, dont les valeurs peuvent également désigner des objets déclarés statiquement, des variables et des constantes, pour autant qu'ils soient surnommables, i.e. que le mot réservé **aliased** apparaisse dans leur déclaration.

La syntaxe de la déclaration des types accès généralisés est semblable à celle des types accès de Ada 83, hormis la présence du modificateur **all** ou **constant** au côté de **access**. Le mot réservé **all** indique que l'objet désigné est une variable, et peut être lu et modifié, alors que **constant** indique que l'objet désigné est une variable ou une constante, et ne peut qu'être lu. La valeur d'accès à un objet est obtenue au moyen de l'attribut **'Access**.

```
type Femme_Ptr is access all Femme;
F: Femme_Ptr;
Anne: aliased Femme := ...;
...
F := new Femme' (Prénom => "Emma");    -- F désigne un objet créé dynamiquement
F := Anne'Access;                      -- F désigne un objet déclaré statiquement
```

Les paramètres formels appartenant à un type étiqueté sont intrinsèquement surnommables, i.e. l'attribut **'Access** peut leur être appliqué sans que le mot réservé **aliased** figure dans leur déclaration.

7.2. *Les collections hétérogènes*

Les *types accès-à-l'échelle-de-classe (access-to-class-wide types)* désignent des objets d'une classe plutôt que des objets d'un type spécifique. Alors que, dans l'exemple précédent (§ 7.1), les valeurs du type Femme_Ptr ne peuvent désigner que des objets du type spécifique Femme, excluant ses descendants potentiels, dans l'exemple suivant P peut désigner tout objet de la classe Personne (des types spécifiques Personne, Homme ou Femme).

```
type Personne_Ref is access all Personne'Class;
P: Personne_Ref := new Homme' (Prénom => "Jean", Barbu =>False);
                                       -- P désigne un Homme
P := Anne'Access;                      -- P désigne une Femme (§ 7.1)
```

Au contraire des variables d'un type à l'échelle de classe, qui ne peuvent pas changer d'étiquette, les variables d'un type accès-à-l'échelle-de-classe peuvent désigner n'importe quel objet dans la classe spécifiée, et peuvent désigner différents objets pendant l'exécution du programme.

Cette propriété facilite la construction de collections hétérogènes, i.e. de collections d'objets de types spécifiques différents, mais qui appartiennent à une même classe. Par exemple, une collection de personnes peut facilement être implémentée à l'aide d'un tableau dont les composants sont des valeurs d'un type accès-à-l'échelle-de-classe :

```
declare
    type Personne_Ref is access all Personne'Class;

    Carole, Catherine: aliased Femme := ...;
    Alfred, Gabriel, Magnus: aliased Homme := ...;

    Labo : array (1..5) of Personne_Ref :=
        (Alfred'Access, Carole'Access, Catherine'Access,
         Gabriel'Access, Magnus'Access);
begin
    -- affiche le nom de tous les membres de Labo et
    -- rase tous ses membres masculins (qu'ils soient barbus ou non).
    for Membre in Labo'Range loop
        Put (Titre_de (Labo (Membre).all) & Nom_de (Labo (Membre).all));
        if Labo (Membre).all in Homme'Class then
            Raser (Homme'Class (Labo (Membre).all));
        end if;
    end loop;
end;
```

Figure 16. *Tableau hétérogène contenant des objets de la classe Personne*

Toutes les opérations primitives définies pour le type racine de la classe de dérivation sont évidemment directement disponibles pour tous les éléments placés dans de telles collections hétérogènes (dans cet exemple, Titre_de et Nom_de). Les appels à ces opérations sont aiguillants pour des éléments de la collection hétérogène. Les opérations spécifiques à une sous-classe (par exemple Raser, qui ne s'applique qu'à Homme et ses descendants) peuvent être appelées en utilisant la conversion de vue appropriée (§ 3.5).

8. La combinaison d'abstractions et l'héritage multiple

L'héritage multiple est un outil de combinaison d'abstractions permettant de créer une abstraction en héritant simultanément de deux ou plusieurs abstractions. Bien que, comme l'héritage simple, l'héritage multiple soit voué à la classification de concepts, son utilisation varie d'un langage à l'autre. Il est souvent utilisé pour pallier différentes lacunes d'un langage, plutôt que pour créer des hiérarchies complexes de types. En pratique, les utilisations de bon aloi de l'héritage multiple sont plutôt rares.

De plus, l'héritage multiple augmente généralement la complexité d'un langage, car de nombreuses règles doivent être introduites pour traiter des cas pathologiques, tel que l'héritage à répétition («en forme de diamant»), qui se produit lorsqu'une classe hérite de plusieurs classes qui ont un ancêtre commun.

Pour ces raisons, et surtout parce que de nombreuses techniques sont déjà disponibles en Ada pour résoudre les problèmes qui nécessitent l'héritage multiple dans d'autres langages, Ada 9X n'offre pas de construction syntaxique directe pour l'héritage multiple.

Dans les sections suivantes, nous allons d'abord voir comment traiter les utilisations de l'héritage multiple qui sont sans rapport avec la classification. Ensuite, nous montrerons des techniques qui permettent d'obtenir de la classification multiple avec Ada 9X.

8.1. *L'héritage multiple pour des intentions autres que la classification*

Ada 9X offre une variété de constructions en dehors de la dérivation qui font que l'héritage (simple ou multiple) n'est pas nécessaire pour des intentions autre que la classification, au contraire des langages où les classes sont les seules formes de module et l'héritage le mécanisme principal pour combiner des abstractions ou gérer l'espace des noms.

Par exemple, la composition est une meilleure forme d'agrégation d'objets que l'héritage multiple [LIF 93]. Le manuel de référence d'un langage de programmation par objets bien connu suggère de définir un type dérivé Tarte_aux_Pommes par héritage des types Pomme et Cannelle. Cet exemple montre un usage impropre typique de l'héritage multiple, car le type dérivé hérite indifféremment de toutes les propriétés de ses parents, que ces propriétés lui soient applicables ou non.

Comme les tartes aux pommes ne poussent pas sur les arbres et ne peuvent pas être pelées, dériver Tarte_aux_Pommes de Pomme est dénué de sens. Une tarte aux pommes contient des pommes mais n'est pas une pomme. Ada peut représenter la contenance avec des composants de type article.

De même, Ada 9X n'a pas besoin de l'héritage multiple pour gérer l'espace des noms. Ce contrôle est indépendant des classes, et géré par des paquetages et des clauses de contexte **with**. Par exemple, l'usage de l'héritage n'est pas requis pour accéder aux entités définies dans un module (tel qu'un ensemble de constantes utiles au système de fenêtres X).

Souvent, et de façon impropre, la dépendance d'implémentation, qui consiste à hériter la spécification d'un type et l'implémentation d'un autre type, est exprimée en recourant à l'héritage multiple. Par exemple, un type dérivé Pile_Bornée peut être construit en héritant l'interface d'un type abstrait Pile, qui définit les opérations Empiler et Dépiler, et l'implémentation d'un type concret Tableau. Cependant, cette forme d'héritage multiple n'est adéquate que si toutes les opérations héritées du type fournissant l'implémentation ont une signification pour le type dérivé. Ainsi, l'opération d'indexation de Tableau doit être cachée des utilisateurs de Pile_Bornée, puisque seul l'élément au sommet de la pile doit être accessible. Cet exemple est facile à implémenter sans recourir à l'héritage multiple (§ 6.3.1). La nouvelle abstraction utilise l'héritage pour définir sa spécification et la composition pour son implémentation.

8.2. *L'héritage par mixin*

L'héritage par mixin [BRA 90] est une forme d'héritage multiple dans laquelle une des deux abstractions combinées est un sous-motif. Un *sous-motif (subpattern)*, également appelé *motif de sous-classification*, est un ensemble de propriétés (composants et opérations) qui est en lui-même insuffisant pour définir autre chose qu'un type abstrait et qui se retrouve dans de nombreux types non apparentés. En d'autres termes, un sous-motif représente une valeur ajoutée, mais n'a pas d'existence propre.

Les sous-motifs peuvent être programmés sans héritage multiple explicite, en utilisant l'héritage simple et les nouvelles possibilités offertes par la généricité :

- le sous-motif est isolé dans un *dérivatif abstrait* en rassemblant l'ensemble de ses propriétés dans une extension de type, le *type mixin*, qui pourra être utilisé pour spécialiser le comportement d'une classe quelconque ;
- un paquetage générique, le *paquetage mixin*, encapsule le type mixin. Ce paquetage a un paramètre générique formel étiqueté qui joue le rôle du type père, auquel les propriétés du type mixin seront ajoutées.

Le mixage est accompli par l'extension du paramètre effectif avec le dérivatif abstrait lors de l'instanciation du paquetage mixin. Le dérivatif abstrait devient ainsi, dans un renversement de la hiérarchie d'héritage, un type père (abstrait) duquel hériter. Cette forme d'héritage est appelé *héritage par mixin (mixin inheritance)*.

Par exemple, bien que l'ensemble des propriétés qui définissent un Docteur soit clairement définissable, «Docteur» n'existe pas en tant que tel. C'est un sous-motif qui peut être appliqué aux personnes qui ont obtenu une thèse : il existe des personnes, des hommes et des femmes, qui ont obtenu un doctorat, mais pas de «Docteur» *per se*. Les propriétés qui définissent la détention d'une thèse peuvent être rassemblées dans le paquetage mixin suivant, appelé Docteurs_G :

```
with Thèses; use Thèses;-- exporte le type Thèse
generic
    type Etre_Humain is tagged private;
package Docteurs_G is
    -- Un Docteur possède une Thèse
    type Docteur is new Etre_Humain with private;

    procedure Conférer (D: in out Docteur; T: in Thèse);
    function Thèse_de (D: Docteur) return Thèse;
private
    type Docteur is new Etre_Humain with
        record
            Thèse_Présentée: Thèse;
        end record;
end Docteurs_G;
```

Figure 17. *Spécification du paquetage mixin Docteurs_G*

Lors de l'instanciation de ce paquetage avec un type étiqueté, un nouveau type est créé en étendant le type donné en paramètre effectif avec la valeur ajoutée du type mixin.

```
package Femmes_Docteur is new Docteurs_G (Femme);
Anne: Femmes_Docteur.Docteur;           -- une femme docteur
T: Thèses.Thèse :=...
```

```
    Nommer (Anne, "Anne");          -- une opération de Femme
    Conférer (Anne, T);             -- une opération de Docteur
```

Figure 18. *Mixage du type Femme et du type Docteur*

Le nouveau type (ici Femme_Docteur.Docteur) dispose des propriétés du type père (ici Femme) et du type mixin (Docteur).

8.2.1. *Mixins contraints*

Le type père du mixin (le type formel générique) peut aussi être contraint à appartenir à une classe de dérivation, pour garantir qu'il dispose de certaines propriétés qui sont requises pour implémenter les opérations du type mixin, ou pour garantir la cohérence des types mixés. Par exemple, conférer un doctorat à une pile serait dénué de sens.

La contrainte sur le type père peut être exprimée en spécifiant la classe de dérivation à laquelle le type père doit appartenir à l'aide d'un type dérivé formel (à la place d'un type étiqueté formel). Ainsi, dans l'exemple suivant, seul Personne et ses descendants pourront être utilisés pour instancier le paquetage mixin.

```
with Thèses; use Thèses;
generic
    type Etre_Humain is new Personne with private;
package Docteurs_G is
    -- Un Docteur est une Personne qui possède une thèse
    type Docteur is new Etre_Humain with private;
    ... -- Définitions de Conférer et Thèse_de comme dans la figure 17

    function Titre_de (D: Docteur) return String;
    -- redéfinition de la fonction Titre_de en tenant compte du titre de docteur
private
    ...
end Docteurs_G;
```

Figure 19. *Contrainte d'appartenance du type père Etre_Humain*

Comme les propriétés du type père sont connues, le mixin contraint peut éventuellement les redéfinir. Dans notre exemple, le sous-programme Titre_de est redéfini pour fournir une implémentation appropriée.

8.3. *La classification multiple par l'héritage confraternel*

En plus des techniques sus-mentionnées, une technique d'implémentation de l'héritage multiple, l'*héritage confraternel (sibling inheritance)* [TEM 93], est possible pour traiter le cas où un type est réellement un dérivatif de plus d'un type père.

Pour citer Templ, l'idée de base cachée derrière l'héritage confraternel est «*d'exprimer l'effet de l'héritage multiple en se reposant sur l'héritage simple seulement [...]. Il consiste essentiellement dans l'idée de représenter les instance d'une classe qui hérite de plusieurs super-classes par un ensemble d'objets liés.*».

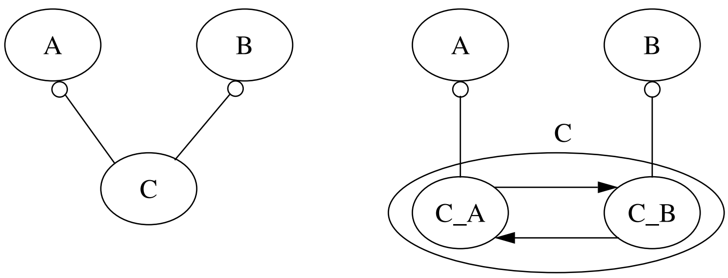

Figure 20. *Héritage multiple conceptuel et héritage confraternel*

En terme de types, l'idée est de considérer un type conceptuel C, dérivé de deux types A et B, comme un ensemble de types C_A et C_B. Ces types sont appelés types frères, et constituent une fratrie. Un objet conceptuel est réalisé à l'aide de deux objets frères liés, appartenant à ces deux types frères et constituant à leur tour une fratrie. Un tel objet conceptuel peut être manipulé comme s'il était un objet unique, et chacun des deux frères peut accéder à toutes les propriétés de la fratrie. Chaque frère est également accessible individuellement, de sorte qu'une vue partielle de l'objet conceptuel qui correspond à un type père spécifique est aisément disponible.

Comme illustration, nous allons montrer comment créer un type conceptuel «Personne Contrôlée», dérivé du type prédéfini Limited_Controlled et d'un type Personne légèrement modifié. Le composant Prénom est une valeur de type accès vers une chaîne de caractères plutôt qu'une chaîne de longueur fixe. Prénom peut dès lors désigner des chaînes de taille variable, créées dynamiquement[3]. Le type prédéfini Limited_Controlled est un type étiqueté pour lequel sont définies les opérations Initialize et Finalize, qui sont automatiquement appelées lors de la création et de la destruction d'un objet, respectivement. Nous utilisons les capacités de Limited_Controlled pour libérer automatiquement la mémoire associée à Prénom à chaque fois qu'un objet de type Personne est détruit.

```
package Personnes_Dynamiques is
    type String_Ptr is access String;
    type Personne is
        tagged limited record
            Prénom: String_Ptr;
        end record;
    procedure Nommer (E: in out Personne; N: access String);
    -- donne un nom à une Personne
    ... -- suite voir figure 2
end Personnes_Dynamiques;
```

Figure 21. *Spécification de Personnes dynamiques*

Pour réaliser la combinaison de ces deux types, deux types frères, Frère_Personne et Frère_Contrôlé sont définis, par dérivation de Personne et de Limited_Controlled, respectivement. Ces deux types forment ensemble le type conceptuel «Personne Contrôlée».

```
with Personnes_Dynamiques, Ada.Finalization;
package Personnes_Contrôlées is
    type Frère_Personne;
        -- déclaration de type incomplète, requise à cause des dépendances mutuelles
    type Frère_Contrôlé (Vers_Frère_Personne: access Frère_Personne) is
         new Ada.Finalization.Limited_Controlled with null record;
            -- Vers_Frère_Personne est le lien vers Frère_Personne
    procedure Finalize (C: in out Frère_Contrôlé);

    type Frère_Personne is new Personnes_Dynamiques.Personne with
        record
            Vers_Frère_Contrôlé: Frère_Contrôlé (Frère_Personne'Access);
                -- Vers_Frère_Contrôlé est le lien vers Frère_Contrôlé. Le
                -- discriminant de type accès désigne automatiquement
                -- l'instance courante de Frère_Personne
        end record;
    -- les opérations primitives de Personne peuvent éventuellement être redéfinies

    subtype Personne_Contrôlée is Frère_Personne;
end Personnes_Contrôlées;
```

Figure 22. *Spécification du type conceptuel Personne_Contrôlée*

Les types frères sont liés à l'aide de composants qui servent de liens pour que chaque frère puisse accéder à toutes les propriétés de la fratrie :

- Frère_Personne a un composant Vers_Frère_Contrôlé qui appartient au type Frère_Contrôlé.
- Frère_Contrôlé a un composant Vers_Frère_Personne qui désigne un Frère_Personne.

Ces liens sont de natures différentes. Vers_Frère_Contrôlé est un «lien de composition» : un objet du type Frère_Contrôlé est inclus dans chaque objet du type Frère_Personne. L'autre lien, Vers_Frère_Personne, est un lien de référence : Vers_Frère_Personne désigne un Frère_Personne.Il est toujours possible d'obtenir une vue de l'autre objet frère à travers ces liens.

Bien que cette technique de programmation nécessite du travail supplémentaire (tel que la déréférenciation), elle fournit toutes les fonctions requises de l'héritage multiple à usage de classification multiple. De plus, elle est facilement extensible pour gérer l'héritage multiple de plus de deux types.

3. Noter que l'intérêt de cet exemple est principalement académique : il eut été plus simple de déclarer Personnes_Dynamiques.Personne comme descendant de Finalization.Controlled, ou de déclarer Prénom comme appartenant à Ada.Strings.Unbounded_String, un type prédéfini de chaînes de caractère de taille variable.

8.3.1. *La génération d'un objet conceptuel*

Puisque Vers_Frère_Contrôlé est un composant de Frère_Personne, un objet du type Frère_Contrôlé est automatiquement créé à chaque fois qu'un objet du type Frère_Personne est créé. Le discriminant du lien Vers_Frère_Contrôlé est automatiquement initialisé pour référencer l'objet englobant, parce que l'attribut **'Access** appliqué au nom d'un type article à l'intérieur de sa déclaration désigne automatiquement l'occurrence courante du type. En conséquence, la déclaration «PC: Personne_Contrôlée;» déclare automatiquement un objet du type conceptuel «Personne Contrôlée» et initialise les liens.

8.3.2. *La redéfinition d'opérations*

Les opération primitives peuvent être redéfinies par le type frère qui en hérite. Comme illustration, pour se prémunir d'une pénurie de mémoire, nous allons redéfinir l'opération Finalize afin de libérer automatiquement la mémoire allouée à Prénom lorsque Personne devient inaccessible.

```
package body Personnes_Contrôlées is
    procedure Libérer is new Unchecked_Deallocation (String, String_Ptr);

    procedure Finalize (C: in out Frère_Contrôlé) is
        -- redéfinition de l'opération Finalize héritée de Limited_Controlled
    begin
        Libérer (C.Vers_Frère_Personne.Prénom);
    end Finalize;
end Personnes_Contrôlées;
```

Figure 23. *Redéfinition de l'opération primitive Finalize*

8.3.3. *L'accès aux propriétés de la fratrie*

Les composants de chacun des objets frères peuvent être sélectionnés et utilisés par toute la fratrie, en utilisant les liens pour y accéder. Par exemple, l'opération Finalize (définie pour Frère_Contrôlé) fait usage du composant Prénom (défini pour Frère_Personne). Les opérations primitives des objets frères peuvent être appelées de la même façon.

8.3.4. *L'ajout de propriétés*

De nouvelles propriétés, composants et opérations, peuvent être facilement ajoutées au type conceptuel en étendant un des types frères. Pour préserver l'encapsulation, les types frères peuvent aussi être définis comme des extensions privées.

8.3.5. *La dérivation du type conceptuel*

Le type conceptuel peut à son tour servir de père pour un autre type. Le type frère Frère_Personne, qui a un composant de l'autre type frère, peut être étendu, de sorte que les propriétés de l'autre frère seront aussi disponibles pour le type dérivé à travers le lien (hérité) Vers_Frère_Personne.

9. Conclusion

«Ada a été conçu à l'origine avec trois soucis majeurs : la fiabilité et la maintenance des programmes, la programmation en tant qu'activité humaine, et l'efficacité. La révision du langage est destinée à offrir une plus grande flexibilité et extensibilité, plus de contrôle sur la gestion de la mémoire et la synchronisation, et des paquetages normalisés voués à des domaines d'application importants, tout en préservant l'accent original placé sur la fiabilité, la maintenabilité, et l'efficacité.» [ADA 94a]

Nous avons décrit la programmation par objets avec Ada 9X à partir de l'étude des mécanismes de base : objets, opérations, encapsulation, héritage, polymorphisme. Ada 9X tire parti des caractéristiques de Ada 83 et les étend avec des mécanismes bien intégrés liés aux objets ; ces extensions restent fidèles à la philosophie de Ada. La puissance expressive de Ada est maintenant égale à celle d'autres langages à objets, comme on a pu le voir à l'aide de mécanismes et techniques avancés (hiérarchies d'unités de bibliothèque, contrôle de visibilité, types abstraits, structures de données hétérogènes, héritage par «mixin» et confraternel).

De plus, ces extensions n'ont pas pour effet de diminuer la sécurité de Ada. Nous avons également décrit comment elles peuvent être utilisées pour programmer de façon sûre grâce à une approche du polymorphisme maintenant un typage fort.

Finalement, il convient de ne pas oublier que la puissance additionnelle de Ada 9X ne se limite pas à la programmation par objets : d'autres éléments tels que synchronisation par les données, ordonnancement flexible, chaînes de longueur variable, fonctions numériques élémentaires, interfaces à d'autres langages, etc. contribueront à son succès.

«Nous avons innové dans la conception des mécanismes de programmation par objets pour Ada 9X. Nous pensons que ces innovations [...] feront la différence pour que Ada 9X ne soit pas simplement un langage à objets supplémentaire. Nous pensons que Ada 9X sera un langage à objets particulièrement sûr et maintenable, tout en offrant toute la puissance et flexibilité nécessaires.»
— S. Tucker Taft (1993), principal architecte de la révision

10. Bibliographie

[ADA 83] *Reference Manual for the Ada Programming Language*. Ada Joint Program Office, 1983.

[ADA 87] *Ada : Manuel de référence du langage de programmation*. Traduction normalisée par l'ISO et l'AFNOR, Presses polytechniques romandes, 1987.

[ADA 90] *Ada 9X Requirements* (Ada 9X Project Report). Office of the Under Secretary of Defense for Acquisition, décembre 1990.

[ADA 94a] *Programming Language Ada: Language and Standard Libraries* (Draft Version 5.0). ISO/IEC DIS 8652. Ada 9X Mapping/Revision Team, Intermetrics, june 1994.

[ADA 94b] *Rationale for the Programming Language Ada: Language and Standard Libraries* (Draft Version 5.0). ISO/IEC JTC1/SC22 WG9 N 207. Ada 9X Mapping/Revision Team, Intermetrics, june 1994.

[BAN 92] BANNER B. et SCHÖNBERG E., Assessing Ada 9X OOP: Building a Reusable Components Library. *In* C.B. Engle, Jr. (ed.), *TRI-Ada '92 Conference Proceedings, Orlando, Floride*, p. 79-90. ACM, 1992.

[BOO 83] BOOCH G., *Software Engineering with Ada*. Benjamin/Cummings, 1983 (1ère éd.), 1987 (2ème éd.), 1993 (3ème éd.).

[BOO 87] BOOCH G., *Software Components with Ada: Structures, Tools, and Subsystems*. Benjamin/Cummings, 1987.

[BOO 88] BOOCH G., *Ingénierie du logiciel avec Ada*. InterEditions, 1988.

[BOO 91] BOOCH G., *Object-Oriented Design with Applications*. Benjamin/Cummings, 1991.

[BRA 90] BRACHA G. et COOK W., Mixin-Based Inheritance. *In* N. Meyrowitz (ed.), *OOPSLA/ECOOP'90 Conference Proceedings, Ottawa, Canada*, *ACM SIGPLAN Notices*, 25(10):303–311, octobre 1990.

[CAR 85] CARDELLI L. et WEGNER P., On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471-522, décembre 1985.

[CHA 92] CHAMBERS C., Object-Oriented Multi-Methods in Cecil. *In* O. Lehrmann Madsen (ed.), *ECOOP'92 Conference Proceedings, Utrecht, Pays-Bas*, LNCS 615, p. 33–56. Springer Verlag, juin 1992.

[DIS 92] DISMUKES G.J. et ROME M.A., Implementing Tagged Types and Type Extensions for Ada 9X. *In* Charles B. Engle, Jr. (ed.), *TRI-Ada '92 Conference Proceedings, Orlando, Floride*, p. 62-67. ACM, 1992.

[GOL 83] GOLDBERG A. et ROBSON D., *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[GUI 91] GUIMARÃES N., Building Generic User Interface Tools: an Experience with Multiple Inheritance. *In* N. Meyrowitz (ed.), *OOPSLA'91 Conference Proceedings, Phoenix, Arizona*, *ACM SIGPLAN Notices*, 26(11):89-96, novembre 1991.

[KIC 91] KICZALES G., DES RIVIERS J. and BOBROW D.G., *The Art of the Metaobject Protocol*. MIT Press, 1991.

[LIF 93] LIFFERS R.H., Inheritance versus containment. *ACM Sigplan Notices*, 28(9):36–38, septembre 1993.

[MEY 90] MEYER B., *Conception et programmation par objets*. InterEditions, 1990.

[ROS 92] ROSEN J.-P., What Orientation Should Ada Objects Take? *Communications of the ACM*, 35(11):71-76, novembre 1992.

[SCH 92] SCHÖNBERG E., *CONTRASTS: Ada 9X and C++*.
URL ftp://ajpo.sei.cmu.edu/public/ada9x/reports/misc/9x-cplus.txt, avril 1992.

[SEI 91] SEIDEWITZ E., Object-Oriented Programming Through Type Extension in Ada 9X. *ACM Ada Letters*, Vol. XI, No 2, p. 86-97, mars/avril 1991.

[STR 90] STROUSTRUP B. et ELLIS M.A., *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[TAF 92a] TAFT S.T., *Multiple Inheritance in Ada 9X* (Language Study Note).
URL ftp://ajpo.sei.cmu.edu/public/ada9x/reports/misc/9xm-inh.txt, septembre 1992.

[TAF 92b] TAFT S.T., Ada 9X: A Technical Summary. *Communications of the ACM*, 35(11):77-82, novembre 1992.

[TEM 93] TEMPL J., A Systematic Approach to Multiple Inheritance Implementation. *ACM SIGPLAN Notices*, 28(4):61–66, avril 1993.

[WEG 87] WEGNER P., Dimensions of Object-Based Language Design. *In* N. Meyrowitz (ed.), *OOPSLA'87 Conference Proceedings, Orlando, Floride*, *ACM SIGPLAN Notices*, 22(12):168–182, décembre 1987.

[WEG 90] WEGNER P., Concepts and Paradigms of Object-Oriented Programming. *ACM SIGPLAN OOPS Messenger*, 1(1):7-87, août 1990.

11. Biographies

Stéphane Barbey *est assistant de recherche au Laboratoire de génie logiciel de l'EPFL depuis 1992. Il est membre actif de la délégation suisse au groupe de travail Ada de l'ISO. Il a obtenu un diplôme d'ingénieur informaticien EPFL en 1992. Ses domaines de recherches couvrent la technologie des objets, et plus particulièrement les phases de codage et test dans le cycle de vie du logiciel.*

Magnus Kempe *est chargé de cours et doctorant, travaillant comme assistant au Laboratoire de génie logiciel de l'EPFL depuis 1991. Il est membre actif de la délégation suisse au groupe de travail Ada de l'ISO. Il a obtenu un diplôme d'ingénieur informaticien EPFL en 1989. Son domaine de recherche se situe dans les fondements des méthodes du génie logiciel.*

Alfred Strohmeier *est professeur d'informatique à l'EPFL, où il enseigne le génie logiciel, les méthodes de développement de logiciel et la programmation. Il dirige un groupe de recherche en méthodes par objets couvrant le cycle de vie complet du génie logiciel. Il a enseigné la conception par objets et Ada (depuis 1981) en milieux académiques et industriels. Il est «Distinguished Reviewer» de Ada 9X et dirige la délégation suisse dans ISO IEC/JTC1/SC22/WG9. Il était membre de l'équipe qui a traduit le manuel de référence du langage Ada 83 d'anglais en français.*