TD n° 16 (type Tagged)

Objectif : Mise en œuvre des notions d'héritage, de polymorphisme et d'objet. Parmi les concepts de la P.O.O. (Programmation Orientée Objet cours 10), on a la possibilité d'**étendre** un type en le dérivant et en ajoutant de nouveaux composants (**objet**) et la possibilité d'identifier un type particulier à l'**exécution** et de sélectionner une opération particulière en fonction d'un type particulier (**liaison dynamique**). Ce sont ces notions qu'il faut appliquer ici (tout au moins le concept d'objet). Le TP est long (3 heures) mais il faut commencer par quelques rappels de cours (1 heure).

Le type étiqueté : Le type Ada permettant une extension est issu d'un type article **tagged**. Une extension peut être considérée **comme un ajout de nouveaux composants**. D'autre part si nous voulons distinguer ce type à l'exécution, il devra contenir une indication de ce type. Ceci est réalisé par un composant **caché** appelé étiquette (ou Tag). Le constructeur **`Class`** cher au C++ et à Java n'est pas proposé par le langage Ada pour des raisons historique mais c'est bien **du même concept dont il s'agit** !!!

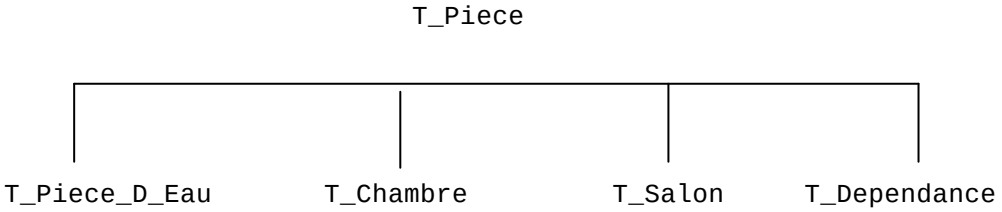
En Ada (cours 10) l'extension de type est réalisé en utilisant des types articles étiquetés.

Soit l'exemple :

```
type  T_Piece is tagged
record
     Surface         :  Float ;
     Nb_Fenetres     : Natural ;
end record ;
```

Nous avons introduit le mot réservé « **tagged** ».

D'autres types seront dérivés de ce type (avec extension) :



Exemple :

```
type T_Poste_Eau is (Lavabo, Baignoire, Bidet, Douche, Evier, Robinet);

type T_Piece_D_Eau is new T_Piece with
record
    Point_D_Eau : T_Poste_Eau ;
end record ;
```

Notez le **`new`** et surtout le **`with`**

Un tel type peut servir à définir par exemple une cuisine ou une autre pièce comme une salle de bain (**après d'autres améliorations** à voir plus loin).

Le type `T_Piece_D_Eau` dérivé du type `T_Piece` a **trois** composants. Il **possède** les deux composants, originels, de `T_Piece` (`Surface` et `Nb_Fenetres`) et le troisième composant `Point_D_Eau` **ajouté** à la dérivation. De plus il **hérite** des fonctionnalités de `T_Piece` mais uniquement sur la restriction à ses deux premiers composants (of course !).

Autre dérivation :

```
type  T_Dependance is new T_Piece with null record ;
```

nous avons, là aussi, un autre type dérivé mais, cette fois, **sans ajouter** de nouveau composant.

Masquage (**`private`**) : un type privé peut être étiqueté :

```
type T_Piece  is tagged private ;
```

avec la déclaration complète repoussée (comme d'habitude) en partie privée :

```
private
        type T_Piece is tagged
        record
            ……..
```

Si `T_Piece_D_Eau` est dérivé de `T_Piece` :

```
type T_Piece_D_Eau is new T_Piece with private ;
```

avec la déclaration complète :

```
private
 type T_Piece_D_Eau is new T_Piece with
        record
            ……
```

Les opérations **primitives d'un type** sont celles qui sont déclarées **dans la même spécification** (donc en même temps) que le type ; elles possèdent des paramètres ou des résultats de ce type.

Par **dérivation** le nouveau type **hérite** de ces opérations. Les opérations peuvent être remplacées (surcharge) par de nouvelles versions et de nouvelles opérations peuvent être introduites.

Résumé : La spécification est toujours héritée. L'implémentation peut être héritée mais aussi être remplacée. **A noter qu'on ne peut ni supprimer un composant ni éliminer une opération**.

Conversion de valeur : Il est possible de convertir les objets.

```
Piece         : T_Piece      := (3.25, 2) ;
Piece_D_Eau   : T_Piece_D_Eau := (46.78, 1, Lavabo) ;
```

On peut écrire :

```
Piece         :=  T_Piece (Piece_D_Eau) ;  --  restriction
Piece_D_Eau   := (Piece with Lavabo) ;      -- agrégat d'extension
```

Type de classe : Lorsqu'un type (tagged ou issu de tagged) est dérivé on parle de hiérarchie de type. L'ensemble des types (racine et dérivés) est appelé une **classe**. A chaque classe est associé un type appelé **type à échelle de classe** pour lequel l'ensemble ayant comme racine (ex : `T_Piece`) est dénoté : `T_Piece'Class`.

Polymorphisme : Chaque type étiqueté `T` possède un type associé dénoté par `T'Class`. Ce type englobe l'union de tous les types de l'arbre des types dérivés enracinés à `T`. Les valeurs de `T'Class` sont donc toutes les valeurs de `T` et des ses dérivés. Une valeur d'un type dérivé de `T` peut être convertie implicitement dans le type `T'Class`. `T'Class` est indéfini (comme l'est un tableau non contraint). Un paramètre formel peut être à échelle de classe et le paramètre effectif peut être de n'importe quel type spécifique de la classe.

Supposons la fonction :

```
function  Affiche_Surface (P : T_Piece'Class) return Float is
begin
      …..
```

Cette fonction possède un paramètre formel à échelle de classe. Cela signifie qu'elle peut être appelée avec un **paramètre effectif** dont le type est **un type quelconque (mais concret !) de la classe**.
Si cette fonction fait appel à une **opération remplacée** pour chaque type, le choix de l'opération sera déterminé par l'étiquette de l'objet passé en paramètre. Cet aiguillage (aspect essentiel du comportement dynamique du polymorphisme appelé liaison dynamique) se produit uniquement quand le paramètre est d'un type à échelle de classe.

Remarque : La fonction Affiche_Surface **n'est pas une opération primitive** et ceci pour aucun type.

L'aiguillage est rendu possible grâce à l'étiquette de l'objet passé en paramètre. Un objet étiqueté est auto-identifiable car il contient l'indication de l'identité de son type. Il est possible de tester l'étiquette d'un objet grâce à l'attribut « TAG ».

```
P : T_Piece'Class :=  …… ;
if   P'Tag  =  T_Piece_D_Eau'Tag
```

`Tag` est un type privé déclaré dans le paquetage `Ada.Tags`.

Type abstrait : Si nous voulons déclarer un type comme fondement d'une classe de types avec certaines propriétés communes, mais sans permettre la déclaration d'objets de ce type d'origine on utilise le mot clé **abstract** :

```
type T_Piece is abstract tagged
      record
…..
      end record ;

function Surface ( T : in T_Piece) return float is abstract ;
```

Il est impossible de déclarer un objet de type abstrait. La fonction abstraite n'a pas de corps. Par contre en dérivant un type concret d'un type abstrait, toutes les opérations héritées abstraites doivent être redéfinies par des opérations concrètes. Si toutes les opérations héritées ne sont pas redéfinies, le nouveau type sera abstrait et déclaré comme tel.

D'autre part si nous faisons une dérivation à partir d'un type concret, nous pouvons fournir de nouvelles opérations abstraites et en conséquence le nouveau type doit être déclaré comme abstrait.

Résumé à connaître:

- Les types articles (et privés) peuvent être étiquetés. Les instances des types étiquetés transportent leur étiquette avec elles. Un type étiqueté peut être étendu par dérivation avec des composants supplémentaires. L'étiquette d'un objet ne peut jamais être modifiée.

- Les opérations primitives d'un type étiqueté sont celles qui sont implicitement déclarées avec en plus tous les sous-programmes ayant un paramètre ou résultat de ce type, déclarés dans cette spécification de paquetage.

- Les opérations primitives sont héritées par dérivation mais peuvent être redéfinies. Si la dérivation figure dans une spécification de paquetage, on peut ajouter ensuite d'autres opérations primitives.

- Les types et les sous-programmes peuvent être déclarés comme abstraits. Un sous-programme abstrait n'a pas de corps. Seul un type étiqueté abstrait peut avoir des sous-programmes abstraits.

- La conversion de type va toujours vers la racine de l'arbre des types étiquetés, jamais en descendant (évident).

- `T'Class` dénote le type à échelle de classe de racine `T`. C'est un type indéfini. Un type accès peut désigner n'importe quelle valeur de `T'Class`.

- Appeler une opération primitive avec un paramètre effectif d'un type à échelle de classe provoque l'aiguillage. La sélection suivant l'étiquette peut être retardée jusqu'à l'exécution (dite liaison dynamique).

- Les paramètres des types étiquetés sont toujours passés par référence. Ils sont considérés comme aliasés et de ce fait on peut leur appliquer l'attribut `Access`.

- Les opérations héritées de type étiqueté peuvent être utilisées par l'attribut `Access`.

Types contrôlés : Les types contrôlés permettent à l'utilisateur un contrôle complet sur l'initialisation et la terminaison des objets, tout en permettant une affectation définie par l'utilisateur (cf. cours 10).

Trois activités primitives distinctes concernent le contrôle des objets :

- L'initialisation après la création.
- La finalisation avant la destruction .
- L'ajustage après affectation.

L'utilisateur a la possibilité de fournir des procédures appropriées pour réaliser tout ce qui est nécessaire aux différents moments de la vie d'un objet.

```
Initialize,  Finalize,  Adjust.
```

Exemple :

```
declare
      A : T ;   --  création  de A  ,  Initialize(A)
      begin

      A := E ;  -- Finalize(A)  copier une valeur,  Adjust  (A)

      end ;   -- Finalize(A)  -- fin de A
```

Après la déclaration de A et les initialisations par défaut, la procédure `Initialize` est appelée. Pour l'affectation `Finalize` est appelée pour nettoyer l'ancien objet qui sera détruit et réécrit. La copie physique est ensuite réalisée et enfin `Adjust` réalise tout ce qui peut être requis pour la nouvelle copie.
A la fin du bloc, `Finalize` sera à nouveau appelé avant la destruction de l'objet.

Les trois procédures sont appelées automatiquement par le code compilé.

Exemple : Nous voulons déclarer un type et garder une trace du nombre d'objets instanciés du type. Nous voulons aussi enregistrer dans chaque objet un numéro d'identification.

```
with Ada.Finalization ;
use   Ada.Finalization ;
package Choses_Tracees is

      type T_Chose is new Limited_Controlled with
      record
            Numero_Id : Natural ;
      end record ;

     procedure Initialize (Objet : in out T_Chose) ;
     procedure Finalize  (Objet : in out  T_Chose) ;

end Choses_Tracees ;
```

```
package body Choses_Tracees is

      Compteur : Natural := 0 ;  -- variables globales au corps
      Suivant  :  Positive := 1 ;

      procedure  Initialize  (Objet : in out T_Chose) is
      begin
            Compteur := Compteur + 1 ;
            Objet.Numero_Id := Suivant ;
            Suivant := Suivant + 1 ;
      end Initialize ;

      procedure Finalize (Objet : in out T_Chose) is
      begin
            Compteur := Compteur –1 ;
      end   Finalize ;

end   Choses_Tracees;
```

Dans le paquetage `Ada.Finalization`, on trouve le type abstrait contrôlé (`Controlled`) mais aussi le type limité contrôlé (`Limited_Controlled`). Pour le premier type, le paquetage exporte les primitives `Initialize`, `Adjust` et `Finalize` mais pour le second il n'exporte que `Initialize` et `Finalize`. (Pas d'affectation pour les types limités privés).

Cet exemple sera repris en TP. Nous commencerons par présenter ce TP pendant encore une heure avant de passer sur machine.

TP n° 16 Tag (3 heures)

Objectif : Mise en œuvre de l'héritage et du polymorphisme à partir de l'exemple donné dans le TD sur le type étiqueté. Utilisation d'un type limité contrôlé (`Limited_Controlled`).

Méthodes: A partir du type `T_Piece` défini dans le paquetage `P_Piece` (fichiers **p_piece.ads** et **p_piece.adb**) on dérivera les types `T_Piece_D_Eau`, `T_Chambre`, `T_Salon` et `T_Dependance`. Ces différents types seront définis **chacun dans un paquetage fils** ayant pour père le paquetage `P_Piece`. A partir de ces types dérivés on réalisera un type `T_Appartement` dérivé du type `Limited_Controlled`.

Détails du TP :

- Créez le répertoire tp16. Copiez les fichiers accessibles.

- Voyez les fichiers `p_piece.ads` et `p_piece.adb` (deux pages suivantes). A **commenter** en préparation du TP.

- Copiez les fichiers `p_piece-eau.ads`, `p_piece-chambre.ads`, `p_piece-salon.ads` et `p_piece-dependance.ads`. Le type `T_Piece` (cf. page suivante) contient trois composants. Les types `T_Piece_D_Eau` (à imaginer avec l'enseignant sur le modèle de `T_Chambre`), `T_Chambre` (cf. page suivante) et `T_Salon` (donné en TP) contiennent tous les trois un composant supplémentaire. Par contre `T_Dependance` (cf. page suivante) ne contient pas de champ supplémentaire. Tous les types sont déclarés privés dans des paquetages fils.

- Après avoir étudié les différentes spécifications **écrivez complètement** le corps du paquetage fils `P_Piece.Eau` (fichier `p_piece-eau.adb`), les autres réalisations seront données.

- Copiez le fichier `p_appartement.ads` (2 pages plus loin). Ce paquetage contient la définition du type `T_Appartement`. Il est dérivé du type `Limited_Controlled`. A commenter en TD.

- Après avoir étudié les spécifications de ce paquetage, réalisez en le corps (on complètera le squelette donné (fichier `p_appartement.adb.squ` à renommer). Finir les sous programmes : `Nb_Objet`, `Initialize`, `Finalize`, `Num_Id`, `Put`).

- Réalisez un bon test avec un fichier `ts_appartement.adb`. Compilez, exécutez (fichier d'entrée redirigé). Voyez les fichiers .in et .ora !

```
-- fichier   p_piece.ads (un aperçu)

package P_Piece is
   type T_Piece  is tagged private;
   procedure Get (P : out T_Piece);
   procedure Put (P : in  T_Piece);

private
   subtype T_Surface is Float range 0.0..60.0;
   type T_Nature_Sol is (Carrelage,Moquette,Parquet,Marbre,Ciment);
   type T_Piece  is tagged
   record
      Surface      : T_Surface;
      Nb_Fenetre   : Natural;
      Nature_Sol   : T_Nature_Sol;
   end record;
end  P_Piece;
```

Un peu plus compliqué qu'en TD.
Body page suivante.

```
-- fichier p_piece-dependance.ads

package P_Piece.Dependance is
   type T_Dependance is  new T_Piece with private;
   -- function PUT héritée
   -- procedure GET héritée

private
   type T_Dependance is new T_Piece with null record;
end P_Piece.Dependance;
```

Pas de body !
Pourquoi ?

```
-- fichier p_piece-chambre.ads

package P_Piece.Chambre is
   type T_Meuble is (Lit,Table_De_Nuit,Armoire,Commode);
   subtype T_Nb_Meuble is Natural range 0..5;
   type T_Piece_Chambre is new T_Piece  with private;
   function  Get_Nb (P_Chambre :in T_Piece_Chambre)return T_Nb_Meuble;
   procedure Get (P_Chambre : out T_Piece_Chambre);
   procedure Put (P_Chambre : in  T_Piece_Chambre);

private
   type T_Vect_Meuble is array (Natural range <>) of T_Meuble;
   type T_Meuble_Chambre (Nb_Elem : T_Nb_Meuble:=0) is
   record
      V:T_Vect_Meuble(1..Nb_Elem);
   end record;
   type T_Piece_Chambre is new T_Piece  with
      record
      Meuble: T_Meuble_Chambre;
   end record;
end P_Piece.Chambre;
```

A commentez, puis

Imaginez avec votre enseignant le type `T_Piece_D_Eau` (spécifications) puis réalisez.

```
-- fichier p_piece.adb
package body P_Piece is
   procedure Get (P: out T_Piece) is
      S   : T_Surface;
      N   : Natural;
      Sol : T_Nature_Sol;
   begin
      Put(Standard_Output,"Entrez la surface : ");
      Get (S); New_Line(Standard_Output); Skip_Line;
      P.Surface := S;
      Put(Standard_Output,"Entrez le nombre de fenêtres : ");
      Get(N); New_Line(Standard_Output); Skip_Line;
      P.Nb_Fenetre := N;
      Put(Standard_Output,"Entrez la nature du sol : ");
      Get(Sol); New_Line(Standard_Output); Skip_Line;
      P.Nature_Sol := Sol;
   end Get;
   procedure Put(P : in T_Piece) is
   begin
      Put("Surface  de la piece : ");
      Put(P.Surface,5,2,0); New_Line;
      Put("Nombre de fenetres : ");
      Put(P.Nb_Fenetre,4); New_Line;
      Put("Nature du sol : ");
      Put(P.Nature_Sol,12); New_Line;
   end Put;
end P_Piece;
```

A commenter

```
-- fichier p_appartement.ads

with Ada.Finalization;
use  Ada.Finalization;
with P_Piece,P_Piece.Eau,P_Piece.Salon,P_Piece.Chambre,P_Piece.Dependance;
use  P_Piece,P_Piece.Eau,P_Piece.Salon,P_Piece.Chambre,P_Piece.Dependance;

package P_Appartement is
   type T_Appartement is new Limited_Controlled  with
      record
      Cuisine       : T_Piece_D_Eau;
      Salle_De_Bain : T_Piece_D_Eau;
      Salon         : T_Piece_Salon;
      Chambre1      : T_Piece_Chambre;
      Chambre2      : T_Piece_Chambre;
      Cave          : T_Dependance;
      Numero_Id     : Positive;
   end record;

   procedure Get (Appart : out T_Appartement);
   procedure Put (Appart : in  T_Appartement);
   procedure Initialize (Objet : in out T_Appartement);
   procedure Finalize   (Objet : in out T_Appartement);
   function  Num_Id (Objet  : in T_Appartement) return  Positive ;
   function  Nb_Objet return  Natural;

end P_Appartement;
```

A commenter