Le langage Ada 95 – 6
Les paquetages

Yves Bailly

EN DEUX MOTS **Jusqu'ici, nos exemples contenaient en un seul fichier toutes les instructions nécessaires à l'exécution du programme. Pour la création de logiciels complexes de grande envergure, cette manière de faire n'est évidemment pas idéale, sans parler de l'absence totale de modularité, de compilation séparée et de possibilité de réutilisation. Comme tous les langages de haut niveau, Ada propose un mécanisme pour parvenir à ces fins : les paquetages.**

Le mécanisme de paquetages est à la base de l'encapsulation des données en Ada, ainsi que de la compilation séparée des composants d'un programme – de la même façon qu'en C/C++, un logiciel peut être décomposé en bibliothèques compilables séparément et offrant chacune une interface de programmation (API) pour accéder à ses services.

On peut voir un paquetage comme une petite bibliothèque ; on parle toutefois plutôt d'unité de compilation (en anglais *compilation unit*). Incidemment, les programmes que nous avons écrits jusqu'ici, composés d'un unique fichier ne contenant qu'une seule procédure (qui peut elle-même contenir d'autres sous-programmes) sont également désignés par les termes d'« unité de compilation ».

Contrairement à d'autre langages, Ada impose une distinction extrêmement stricte entre la partie déclarative d'un paquetage, où sont annoncés les types et sous-programmes disponibles, et la partie implémentation, qui contient les instructions des sous-programmes annoncés dans la partie déclarative (ainsi que d'autres types et sous-programmes, si besoin est).

Dans le verbiage Ada, la première partie est appelée la « spécification d'un paquetage », tandis que la deuxième est le « corps du paquetage ».

La spécification et le corps

Prenons un exemple élémentaire. Nous allons créer un paquetage qui va contenir deux fonctions, chacune effectuant l'addition de deux entiers donnés en paramètres, mais la première retournant le résultat sous forme d'un entier tandis que l'autre retourne le résultat sous la forme d'un réel. La spécification d'un tel paquetage pourrait ressembler à ceci, dans un fichier `addition.ads` (notez le `s` pour spécification de l'extension) :

```
 1 package Addition is
 2
 3    function Ajoute(int_1: in Integer ;
 4                    int_2: in Integer)
 5       return Integer ;
 6    -- --------------------------------
 7    function Ajoute(int_1: in Integer ;
 8                    int_2: in Integer)
 9       return Float ;
10
11 end Addition ;
```

Remarquez qu'une paire de déclarations équivalentes en C/C++ serait invalide : les deux fonctions ne se distinguent que par leur type de retour, ce qui est parfaitement correct en Ada.

La spécification d'un paquetage est introduite par le mot-clef `package`, suivi du nom du paquetage, suivi de `is`. Puis apparaissent les déclarations de types et de sous-programmes. Enfin la spécification se termine par la répétition du nom du paquetage précédée du mot-clef `end`.

Rien de bien extraordinaire, tout cela ressemble fort à ce que l'on trouve dans un fichier d'en-tête C/C++, à ceci près qu'il est impossible et interdit d'introduire du code (des instructions) au sein de la spécification.

Voyons maintenant le corps de notre paquetage, dans un fichier `addition.adb` (notez le `b` pour *body*, corps, de l'extension) :

```
 1 package body Addition is
 2
 3    function Ajoute(int_1: in Integer ;
 4                    int_2: in Integer)
 5       return Integer is
 6    begin
 7       return int_1 + int_2 ;
 8    end Ajoute ;
 9    -- --------------------------------
10    function Ajoute(int_1: in Integer ;
11                    int_2: in Integer)
12       return Float is
13    begin
14       return Float(int_1 + int_2) ;
15    end Ajoute ;
16
17 end Addition ;
```

Le corps est signalé par la présence du mot-clef `body` entre le mot `package` et le nom du paquetage, en première ligne. Ensuite, on reprend la spécification, en donnant les instructions de chacun des sous-programmes annoncés. Rien n'interdit de déclarer ici de nouveaux types ou de nouveaux sous-

programmes. Toutefois, ceux-ci ne seront pas accessibles par les utilisateurs du paquetage : ils ne seront visibles qu'au sein de ce corps, pas ailleurs. Enfin, on termine comme précédemment, par le mot-clef `end` suivi du nom du paquetage.

Maintenant que nous avons un paquetage, voyons comment l'utiliser.

Utilisation

Réalisons un petit programme de test, dans un fichier `test.adb` :

```
 1 with Text_IO ; use Text_IO ;
 2 with Addition ;
 3 procedure Test is
 4    entier: Integer ;
 5    reel  : Float ;
 6 begin
 7    entier := Addition.Ajoute(1, 2) ;
 8    Put_Line("entier = " & Integer'Image(entier)) ;
 9    reel   := Addition.Ajoute(3, 4) ;
10    Put_Line("reel   = " & Float'Image(reel)) ;
11 end Test ;
```

La première ligne est maintenant familière, elle nous permet d'utiliser la procédure `Put_Line()` pour afficher du texte à l'écran. La deuxième signale que nous souhaitons utiliser un paquetage nommé `Addition`, au moyen d'une clause `with`.

Les lignes 7 et 9 font justement appel aux deux fonctions que nous avons précédemment définies. Ada appliquant un typage strict, il n'y a pas de conversion implicite entre types : c'est ainsi que le type de la variable dans laquelle placer le résultat de chacune des fonctions permet de déterminer laquelle invoquer, sur la base du type de retour.

On utilise ici la notation pointée, que l'on retrouve dans de nombreux autres langages (comme Python) : on fait précéder le nom du sous-programme par un préfixe rappelant dans quel paquetage il se trouve. Cela permet de lever les ambiguïtés, si deux paquetages différents définissent deux sous-programmes parfaitement identiques dans leur déclaration. On peut toutefois éviter ce préfixe, en utilisant une clause `use` à la suite de la clause `with` : c'est ce qui est fait en première ligne pour le paquetage `Text_IO`.

Pour compiler ce petit programme à l'aide du compilateur GNAT, il suffit d'utiliser l'utilitaire gnatmake :

```
$ gnatmake test
gnatgcc -c test.adb
gnatgcc -c addition.adb
gnatbind -x test.ali
gnatlink test.ali
gnatlink: warning: executable name "test" may conflict with shell
command
```

La commande `gnatmake test` va automatiquement chercher un fichier nommé `test.adb` et le compiler, produisant l'habituel fichier `object test.o`, ainsi qu'un fichier des types déclarés privés.

Lors de cette compilation, les clauses `with` vont être interprétées pour déterminer quels paquetages sont nécessaires au fonctionnement du programme. La première ligne va provoquer la recherche d'un fichier `text_io.ads`, la deuxième d'un fichier `addition.ads`.

Le premier sera évidemment trouvé dans un emplacement standard prédéfini propre au compilateur, typiquement `/usr/lib/gcc-lib/i486-linux/2.8.1/adainclude` dans le cas du compilateur GNAT 3.15p ; le deuxième sera recherché dans le répertoire courant, ou dans tout répertoire indiqué par une option `-I` (la même que celle utilisée pour indiquer les fichiers d'en-tête d'un programme C/C++).

Les fichiers `*.ads` trouvés sont lus et interprétés à leur tour. Dans certains cas, par exemple lorsque la spécification d'un paquetage ne contient que des déclarations de types mais pas de sous-programme, cela suffit au bonheur du compilateur.

Mais dans la plupart des situations, une spécification implique un corps quelque part. gnatmake recherche alors une paire de fichiers d'extension `.o` et `.ali` ayant le nom du paquetage concerné – dans notre exemple, `addition.o` et `addition.alious_Paq`. Le contenu de celui-ci est .adb (ici, `addition.adb`) est recherché et compilé, produisant la paire voulue.

Le processus se poursuit récursivement, si un paquetage fait lui-même référence à un ou plusieurs autres paquetages.

À noter que gnatmake recherche et signale les éventuelles interdépendances qui constituent une erreur de compilation.

L'extension .ali signifie *Ada Library Information*. Ces fichiers contiennent de nombreuses informations diverses qui seront utilisées entre autres lors de l'édition des liens, ainsi que la version du compilateur, les options utilisées, des informations de références croisées, etc.

En plus d'être utilisés par gnatmake, ces fichiers sont forts utiles pour des programmes d'analyse de code afin d'examiner la structure d'un logiciel.

Enfin l'édition des liens est réalisée, pour générer un fichier exécutable. Dans notre cas, ce fichier est `test` : remarquez l'avertissement de gnatmake (en réalité, de gnatlink, le programme réalisant l'édition des liens) signalant que le nom de notre exécutable est également celui d'une commande du shell courant.

Par la suite, une nouvelle invocation de gnatmake ne provoquera la recompilation que de ce qui est nécessaire. Si vous modifiez le contenu du fichier `test.adb`, lui seul sera recompilé.

De même, si vous modifiez le contenu de `addition.adb`. Par contre, si vous modifiez le contenu de `addition.ads`, c'est-à-dire de la spécification du paquetage `Addition`, tout sera recompilé.

En effet, le corps d'un paquetage dépend de sa spécification, ainsi que toute unité de compilation (paquetage ou programme principal) référençant cette spécification par une clause `with`.

Voici l'affichage de notre petit programme :

```
$ ./test
entier =  3
reel   =  7.00000E+00
```

Ce qui est bien ce que nous attendions.

Partie privée d'un paquetage

Une spécification, telle que nous l'avons écrite plus haut, décrit la partie publique d'un paquetage, c'est-à-dire ce qui est visible et utilisable depuis l'extérieur du paquetage, par d'autres unités de compilation.

Mais souvent, on souhaite restreindre l'accès à certains éléments du paquetage, notamment aux types.

Reprenons l'exemple consacré aux Tours de Hanoï de l'article précédent. La spécification d'un paquetage qui s'appellerait « `Hanoï` » pourrait ressembler à ceci :

```
 1 package Hanoi is
 2
 3    type Disque is new Integer range 0..10 ;
 4    type Étage is new Integer range 0..10 ;
 5    type Pile_Disques is array (Étage) of Disque ;
 6
 7    type Tour is
 8    record
 9       dernier_étage: Étage := 0 ;
10       pile:          Pile_Disques := (others=>0) ;
11    end record ;
12
13    type Num_Tour is new Integer range 1..3 ;
14    type Rangée_Tours is array (Num_Tour) of Tour ;
15    type Num_Mouvements is new Integer range 0..Integer'Last ;
16    type Jeu_Hanoï is
17    record
18       tours: Rangée_Tours ;
19       nb_mouvements: Num_Mouvements := 0 ;
20    end record ;
21
22    procedure Init(jeu     : out   Jeu_Hanoï ;
23                   nb_init: in    Étage) ;
24
25    procedure Afficher(jeu: in Jeu_Hanoï) ;
26
27 end Hanoi ;
```

Je vous laisse le soin de réaliser le corps de ce paquetage. Un programme de test pourrait être :

```
1 with Hanoi ;
2 procedure Test is
3    jeu: Hanoi.Jeu_Hanoï ;
4 begin
5    Hanoi.Init(jeu, 5) ;
6    Hanoi.Afficher(jeu) ;
7 end Test ;
```

Ce n'est toutefois pas satisfaisant. Les types `Tours` et `Jeu_Hanoï` étant déclarés et définis dans la partie publique et visible du paquetage, un utilisateur pourrait accéder directement aux données contenues dans ces types.

Par exemple, il serait parfaitement possible de réaliser une affectation comme `jeu.tours(1).pile(2) := 3`, ce qui pourrait avoir des conséquences catastrophiques sur le programme : les données ne sont plus cohérentes.

Il est donc nécessaire de protéger les données pour en contrôler l'accès. Autrement dit, nous avons besoin d'encapsulation.

Cela peut se faire au niveau du paquetage au moyen d'une section privée. La spécification du paquetage devient alors (à quelques omissions près) :

```
 1 package Hanoi is
 2
 3 -- types Disque, Étage et Pile_Disques...
 4
 5    type Tour is private ;
 6
 7 -- types Num_Tour, Rangée_Tours et Num_Mouvements...
 8
 9    type Jeu_Hanoï is private ;
10
11 -- les deux procédures, comme précédemment...
12
13 private
14
15    type Tour is
16    record
17       dernier_étage: Étage := 0 ;
18       pile:          Pile_Disques := (others=>0) ;
19    end record ;
20
21    type Jeu_Hanoï is
22    record
23       tours: Rangée_Tours ;
24       nb_mouvements: Num_Mouvements := 0 ;
25    end record ;
26
27 end Hanoi ;
```

Rien ne change dans le corps du paquetage ou le programme de test. Mais désormais, le contenu des types Tour et Jeu_Hanoï ne sont plus visibles, donc accessibles, depuis l'extérieur du paquetage.

Seul le corps peut y avoir accès (ce qui est heureux). Remarquez comment les types sont déclarés, lignes 5 et 9 : ils sont simplement qualifiés de private (privés). Leur description complète n'apparaîtra que dans une section particulière de la spécification, introduite justement par le mot-clef private, ligne 13.

Cette partie privée peut contenir ce que bon vous semble : des définitions de types (qui n'auront pas forcément été annoncés dans la partie publique), des déclarations de sous-programmes, des variables globales (beurk !), etc. Tous ces éléments ne seront pleinement utilisables que dans le corps du paquetage.

Pourquoi alors ne pas se contenter de les déclarer directement dans le corps, du moins pour ceux qui ne sont pas destinés à être visibles de l'extérieur ? Les raisons d'un tel choix apparaîtront bientôt.

Naturellement, il est nécessaire de prévoir des sous-programmes permettant de manipuler les variables des types déclarés privés. Les deux procédures Init() et Afficher() auront les possibilités offertes par l'un des aspects les plus puissants.

Paquetages imbriqués

Pour des besoins d'organisation, il est parfois intéressant de déclarer un paquetage dans un autre. Voici un exemple de spécification :

```
 1 package Paq is
 2    procedure P1 ;
 3    -- ----------------
 4    package Sous_Paq is
 5       procedure SP1 ;
 6    private
 7       procedure SP2 ;
 8    end Sous_Paq ;
 9    -- ----------------
10 private
11    procedure P2 ;
12 end Paq ;
```

Le paquetage Paq définit deux procédures, dont l'une est en section privée, ainsi qu'un paquetage imbriqué nommé Sous_Paq. Ce « sous-paquetage » contient lui-même deux procédures, dont l'une également en section privée.

Depuis l'extérieur de ce paquetage, seules sont accessibles les entités publiquement visibles – ici, les procédures P1 et SP1. On pourrait les utiliser ainsi :

```
1 with Paq ;
2 procedure Test is
3 begin
4    Paq.P1 ;
5    Paq.Sous_Paq.SP1 ;
6 end Test ;
```

Le fait de déclarer l'accès au paquetage Paq (ligne 1) donne automatiquement l'accès au paquetage Sous_Paq.

Son contenu est alors référencé en faisant apparaître le chemin d'accès à ses entités au moyen de la notation pointée (ligne 5).

Mais voyons le corps de ce paquetage et les intéressantes possibilités d'accès qu'on y trouve :

```
 1 with Text_IO ; use Text_IO ;
 2 package body Paq is
 3    procedure P1 is
 4    begin
 5       Put_Line("Paq.P1") ;
 6       P2 ;
 7       Sous_Paq.SP1 ;
 8    end P1 ;
 9    -- ----------------
10    package body Sous_Paq is
11       procedure SP1 is
12       begin
13          Put_Line("Paq.Sous_Paq.SP1") ;
14          P2 ;
15          SP2 ;
16       end SP1 ;
17       procedure SP2 is
18       begin
19          Put_Line("Paq.Sous_Paq.SP2") ;
20          P2 ;
21       end SP2 ;
22    end Sous_Paq ;
23    -- ----------------
24    procedure P2 is
25    begin
26       Put_Line("Paq.P2") ;
27    end P2 ;
28 end Paq ;
```

Dans le corps d'un paquetage, les mentions private disparaissent : tout est public... ou presque.

La procédure P1(), par exemple, a naturellement accès aux entités privées du paquetage dont elle est issue (comme la procédure P2(), invoquée ligne 6), ainsi qu'aux entités publiques du paquetage imbriqué Sous_Paq... mais pas aux entités privées de ce dernier.

Les entités privées de Sous_Paq ne sont accessibles que depuis Sous_Paq. Par contre, comme le montre les corps des procédures SP1() et SP2(), Sous_Paq a accès aux entités privées du paquetage qui le contient : voyez les invocations de la procédure privée P2() depuis SP1() et SP2(), lignes 14 et 20.

D'une manière générale, un paquetage imbriqué a accès aux entités privées (et publiques, bien sûr) du paquetage qui le contient.

Si Sous_Paq contenait lui-même un troisième paquetage Sous_Sous_Paq, celui-ci aurait accès aux entités privées de Sous_Paq et de Paq.

La création de paquetages imbriqués permet de hiérarchiser clairement les idées mises en œuvre dans un programme. Mais il y a encore mieux...

Paquetages enfant

Les paquetages imbriqués présentent quelques inconvénients. Les fichiers ont tendance à devenir fort longs. Ils ne favorisent pas vraiment la compilation séparée des différents éléments du programme, au contraire.

Aussi est-il souvent préférable d'utiliser la technique des paquetages enfant. Reprenons l'exemple précédent, cette fois-ci organisé de la façon suivante :

Paquetage Paq

paq.ads

```
1 package Paq is
2    procedure P1 ;
3 private
4    procedure P2 ;
5 end Paq ;
```

paq.adb

```
1 with Text_IO ; use Text_IO ;
2 with Paq.Sous_Paq ;
3 package body Paq is
4    procedure P1 is
5    begin
6       Put_Line("Paq.P1") ;
7       P2 ;
8       Sous_Paq.SP1 ;
9    end P1 ;
10   procedure P2 is
11   begin
12      Put_Line("Paq.P2") ;
13   end P2 ;
14 end Paq ;
```

Paquetage Paq.Sous_Paq

paq-sous_paq.ads

```
1 package Paq.Sous_Paq is
2    procedure SP1 ;
3 private
4    procedure SP2 ;
5 end Paq.Sous_Paq ;
```

paq-sous_paq.adb

```
1 with Text_IO ; use Text_IO ;
2 package body Paq.Sous_Paq is
3    procedure SP1 is
4    begin
5       Put_Line("Paq.Sous_Paq.SP1") ;
6       P2 ;
7       SP2 ;
8    end SP1 ;
9    procedure SP2 is
10   begin
11      Put_Line("Paq.Sous_Paq.SP2") ;
12      P2 ;
13   end SP2 ;
14 end Paq.Sous_Paq ;
```

Exemple de paquetages enfant

Les noms des fichiers ne doivent rien au hasard. Pour que gnatmake fonctionne dans les meilleures conditions, il y a trois règles simples à respecter :

- Les fichiers contenant des spécifications ont l'extension .ads, ceux contenant des corps, l'extension .adb.
- Le nom des fichiers est le nom du paquetage, en minuscules.
- Dans le cas d'un paquetage enfant, les noms des fichiers sont composés du nom du paquetage précédé du nom du paquetage parent, séparé par un tiret.

Le nom complet du paquetage Sous_Paq est Paq.Sous_Paq, car il est enfant de Paq.

Remplacez le point par un tiret, passez tout en minuscules, et vous obtenez le nom de base des fichiers qui concernent Sous_Paq : paq-sous_paq.ads et paq-sous_paq.adb.

Incidemment, de la dernière règle découle le fait qu'un nom de paquetage ne puisse pas contenir un tiret.

Remarquez que bien que se trouvant dans des fichiers séparés, Sous_Paq a comme précédemment accès aux entités privées de Paq (lignes 6 et 12 dans paq-sous_paq.adb).

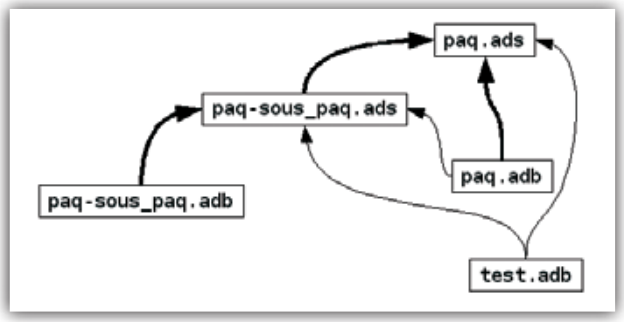
De plus, le corps d'un paquetage parent peut utiliser la spécification d'un de ses paquetages enfant (ligne 2 dans paq.adb).

Par contre, la spécification d'un parent ne peut utiliser la spécification d'un enfant : cela créerait une interdépendance. En effet, implicitement l'enfant dépend du parent.

Le programme de test se voit simplement ajouter une ligne :

```
1 with Paq ;
2 with Paq.Sous_Paq ;
3 procedure Test is
4 begin
5    Paq.P1 ;
6    Paq.Sous_Paq.SP1 ;
7 end Test ;
```

Tout cela est finalement assez naturel. On peut établir un graphe des dépendances de compilation entre les différents fichiers, les flèches se lisant « dépend de » :



Les flèches épaisses signalent les dépendances implicites, impliquées par les relations de parenté d'une part, et les relations entre corps et spécifications d'autre part.

On peut déduire de ce schéma qu'une modification dans paq.ads provoquera une recompilation totale (car tout en dépend, directement ou non). Par contre, une modification dans paq.adb ne provoquera que la recompilation de paq.adb, rien d'autre. Ce qui est tout de même une économie notable. Pour revenir sur l'encapsulation, un paquetage enfant peut être qualifié de privé : il suffit pour cela de placer le mot-clef private devant l'en-tête de la spécification. Par exemple, pour

que `Sous_Paq` devienne un enfant privé de `Paq`, sa première ligne deviendrait :

```
1 private package Paq.Sous_Paq is
```

Le contenu de `Sous_Paq` devient alors inaccessible à toute unité de compilation (programme principal ou autre paquetage), à moins que celle-ci ne soit elle-même un enfant privé de `Paq`.

Quelques paquetages standards

Comme pour la plupart des langages de programmation, le compilateur Ada vient, accompagné d'un certain nombre d'outils généraux, pour accomplir des tâches communes. Sans entrer dans trop de détails, voici quelques paquetages normalement présents avec tout compilateur Ada.

▶Le paquetage Standard contient les définitions des types fondamentaux, ainsi que les opérations disponibles. Cela peut surprendre, mais les types de base que sont `Boolean`, `Integer` ou `Float` ne sont pas définis au sein même du compilateur, mais dans ce paquetage. Il est inutile de l'inclure avec un `with` Standard, il est toujours implicitement disponible.

▶`Ada.Characters.Handling`, paquetage enfant de `Characters` lui-même enfant de `Ada`, offre des facilités pour l'utilisation des caractères comme des fonctions de classification (pour savoir si un caractère est une lettre, un chiffre...) ou de conversion depuis ou vers les caractères Unicode.

▶`Ada.Strings` concerne les chaînes de caractères et se décompose en plusieurs enfants, comme `Ada.Strings.Unbounded` pour manipuler des chaînes à longueur variable ou `Ada.Strings.Wide_Unbounded` pour des chaînes Unicode.

▶`Ada.Text_IO` (dont le nom peut se réduire à `Text_IO`) et `Ada.Wide_Text_IO` fournissent fonctions et procédures pour l'entrée-sortie de caractères, le second destiné à l'Unicode. Nous avons par exemple déjà fréquemment utilisé la procédure `Put_Line`.

▶Le paquetage `System` contient des définitions dépendantes de l'implémentation et de la plate-forme. Par exemple, `System.Max_Int` contient la valeur du plus grand entier positif signé disponible. `System.Address` est un type représentant l'adresse d'une unité de mémoire, chacune de ces unités étant composée de `System.Storage_Unit` bits. Par exemple, sur une plate-forme « classique » comme un PC, `System.Storage_Unit` vaut 8 (un octet) et `System.Address` n'est rien d'autre qu'un pointeur non typé.

Il en existe de nombreux autres que nous verrons progressivement selon nos besoins.

Conclusion

Voilà pour cette petite présentation des paquetages en Ada. Ils sont un outil essentiel pour la bonne organisation d'un logiciel et l'encapsulation de ses données. La prochaine fois, nous nous pencherons sur la généricité, technique d'une grande puissance, offerte par le langage Ada dès sa création en 1979, bien avant les *templates* du langage C++.

Yves Bailly,