Conception modulaire : les paquetages

Chapitre 12

MCours.com

Les paquetages : intérêt

Les applications informatiques devenant de plus en plus complexes, il y a nécessité de :

- travailler en équipe
- réutiliser des composants existants
- faciliter la maintenance

Les paquetages pour le travail en équipe

Après l'étape de conception, l'application est décomposée en unités indépendantes.

Chaque unité peut être développée indépendamment :

- Codage
- Compilation
- Tests

Compilation séparée

Chaque unité doit pouvoir être facilement connectée aux autres

Couplage faible

Paquetages : réutilisabilité

Chaque unité doit traduire une entité du domaine de l'application

`Cohérence forte`

Un travail de généralisation permet à l'unité de s'abstraire du contexte de l'application

`Généricité`

La maintenance est facilitée par une bonne structure du code et une bonne documentation. La modification du code d'une unité n'impose aucune modification sur le reste de l'application. Une simple recompilation de l'ensemble est suffisante.

Paquetages : conception

Le monde réel peut être vu comme un ensemble de composants reliés entre eux et interagissant.

Par analogie, le domaine d'une application informatique peut être vu de la même manière

L'application est alors un ensemble de composants informatiques possédant des caractéristiques communes, offrant et utilisant des services (fonctions, procédures, types, …)

Caractéristiques communes (1)

Les objets du monde réel (exemple : une montre, une platine, un téléphone, …) possèdent des caractéristiques qui sont partagées par les composants d'une application informatique :

- fournissent un ensemble de services
- l'utilisateur n'a pas besoin de savoir comment ils sont réalisés
- l'utilisateur n'a pas le droit de le modifier sans risque de perdre la garantie
- la réalisation n'est pas toujours identique alors que les services rendus le sont
- tous les exemplaires d'un même modèle fournissent les mêmes services
- ils peuvent s'inter-connecter (montre avec un micro-onde) pour former des systèmes plus complexes

Caractéristiques communes (2)

Un langage informatique doit supporter une entité qui reflète cette notion de composant.

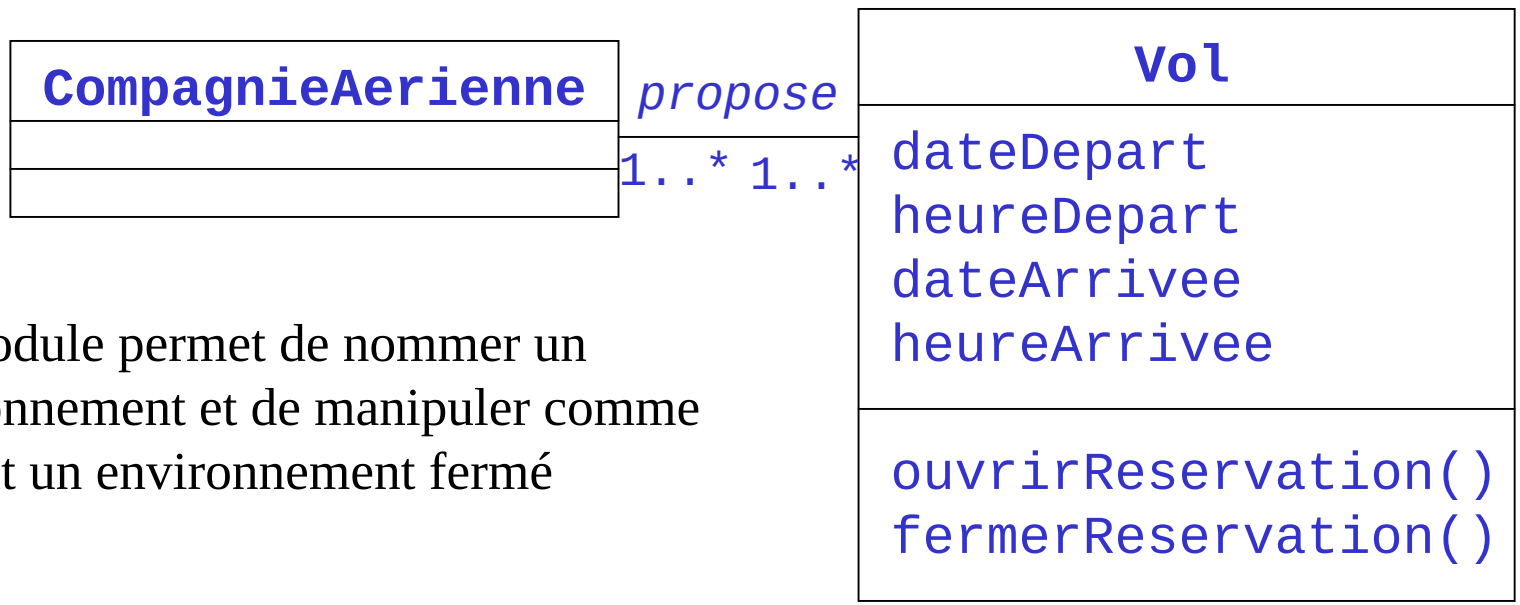
Elle doit donc satisfaire les caractéristiques suivantes :

- séparation en 2 points de vue, client et serveur
- une vue présente l'interface et l'autre sa réalisation
- réunir ces 2 points de vue en une seule entité (encapsulation)
- la réalisation doit être cachée au client (masquage de l'information)
- toute modification de la réalisation ne doit pas impliquer une modification de l'interface

Conception modulaire

Spécification, conception, documentation s'exprime en UML (Unified Modelling Language)

L'application est découpée en entités distinctes



Un module permet de nommer un environnement et de manipuler comme un tout un environnement fermé

Abstraire des données (1)

Le concept de procédure permet d'abstraire (d'un contexte particulier) une suite d'instructions et donc d'enrichir le langage d'une nouvelle instruction

Le concept de module (paquetage) permet au programmeur d'enrichir le langage avec de nouveaux types de données

Abstraire des données (2)

Un module rassemble :

- la déclaration d'un type de données
- les opérations de manipulation des valeurs de ce type

Un module est divisé en 2 parties :

- La première définit l'interface (point de vue utilisateur du type de données) et regroupe les déclarations des opérations sur ce type
- La seconde définit l'implantation du type (point de vue développeur). Elle regroupe les déclarations des corps des fonctions et procédures qui manipulent les valeurs de ce type

Encapsulation et masquage de l'information

Ces deux mécanismes du langage réalisent l'abstraction de données.

L'`encapsulation` permet la réunion des points de vue client et serveur

Le `masquage de l'information` (*data hiding*) empêche l'utilisateur d'avoir accès à la réalisation des opérations du module

L'utilisateur a accès au ¨*quoi*¨ d'un type de données sans avoir à se préoccuper du ¨*comment*¨.

Paquetage Ada : interface (1/3)

Syntaxe

<déclaration_interface_paquetage>::=

```
    package <ident_paquetage> is
       <liste_déclarations>
    end <ident_module>
```

Exemple

```
package Vecteur is
  type Vect_int is array (Positive range <>) of
   Integer;
  function ¨+¨(U,V:Vect_int) return Vect_int;
  function ¨*¨(U:Vect_int;N:Integer) return Vect_int;
  procedure put(X:in Vect_int);
end Vecteur;
```

Paquetage Ada : interface (2/3)

Sémantique

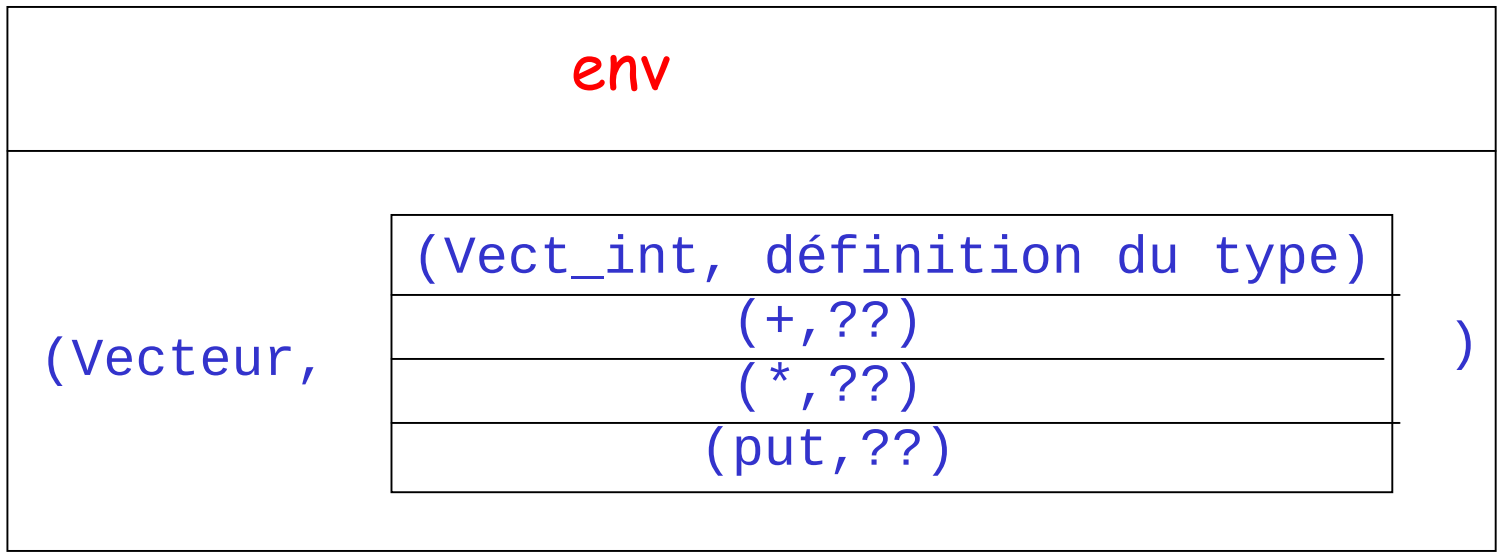
- L'évaluation de la déclaration d'un paquetage a pour effet de lier l'identificateur du paquetage à l'environnement qu'il construit
- L'environnement est construit à partir des déclarations contenues dans l'interface
- Evaluation dans `env` :

  Ajout de la liaison `(ident,??)` à `env`

  Évaluation de la valeur d'environnement => évaluation de l'ensemble des déclarations du paquetage et création des liaisons correspondantes

  Remplacement de la valeur indéfinie `(??)` par la valeur de l'environnement

Paquetage Ada : interface (3/3)



Paquetage Ada : implantation

Syntaxe

```
<déclaration_corps_paquetage>::=
  package body <ident_paquetage> is
    <liste_déclarations>
  begin
    <suite_instructions>
  exception
    <traitement_exceptions>
  end <ident_paquetage>
```

Implantation : exemple (1/4)

```
package body Vecteur is
 function "+"(U,V:Vect_int) return Vect_int is
  W:Vect_int(V'range);
 begin
  for i in U'range loop
    W(i):=U(i)+V(i);
  end loop;
  return W;
 end "+";
```

Implantation : exemple (2/4)

```
function "*"(U:Vect_int;N:Integer) return Vect_int is
 W:Vect_int(U'range);
begin
 for i in U'range loop
    W(i):=U(i)*N;
 end loop;
 return W;
end "*";
procedure put(X:in Vect_int) is
begin
 for i in X'range loop
   put(X(i));put(",");
 end loop;
end put;
end Vecteur;
```

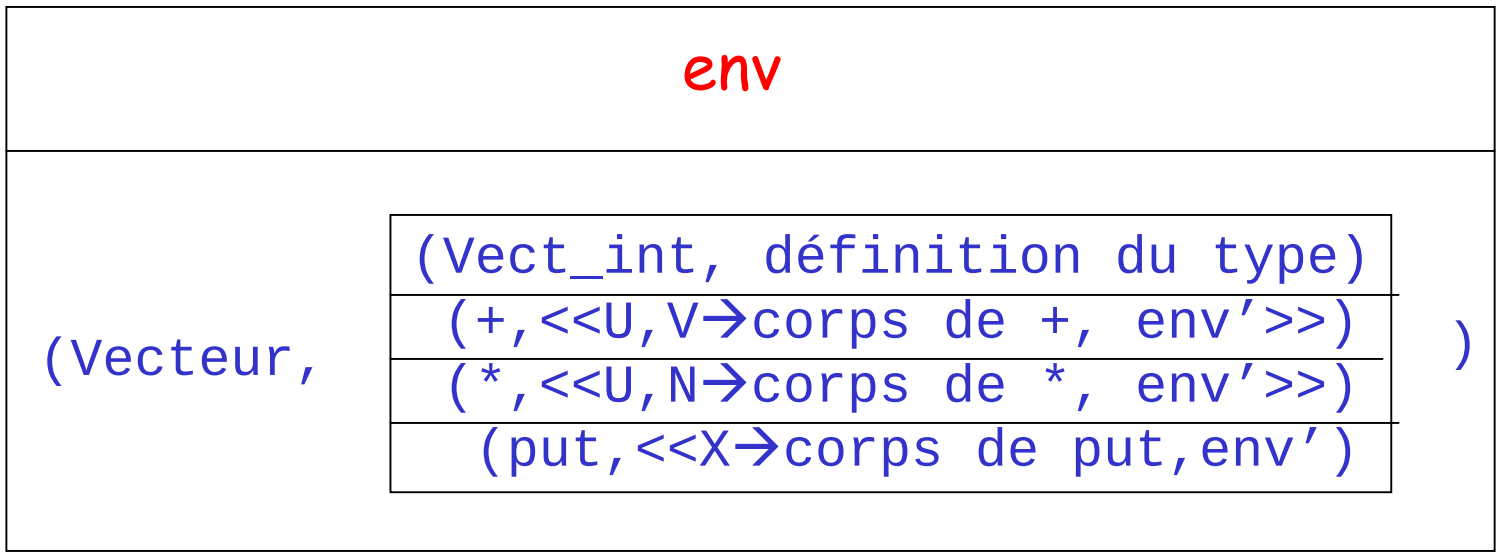
Implantation : exemple (3/4)

Sémantique (évaluation dans env')

1. Détermination des fermetures des fonctions et procédures déclarées dans l'interface

1. Modification des liaisons correspondantes dans la valeur d'environnement

Implantation : exemple (4/4)



Opérations sur les paquetages

Pour rendre visible un objet déclaré dans un paquetage, on utilise l'opérateur `.` :

`<op_acces_objet>::=<ident_paquetage>.<ident_objet>`

Evaluation

1. Recherche de la liaison `(ident_paquetage,valeur d'env)` dans l'environnement courant
2. Soit env la valeur d'environnement trouvée, recherche dans env de la liaison `(ident objet, valeur d'objet)`

Exemple d'accès à un paquetage

Accès à un type

```
Vect:Vecteur.Vect_int(1..4):=(3,8,8,6);
```

Accès à une opération

```
vect:Vecteur."+"(vect,vect);
```

ou

```
use Vecteur;
vect:=vect+vect;
```

Partie privée
Partie publique

Une interface de paquetage possède une partie privée et une partie publique :

```
<déclaration_interface>::=
  package <ident_paquetage> is
    <liste_déclarations>
  private
    <liste_déclarations>
  end <ident_paquetage>;
```

La partie publique est visible de l'extérieur. Elle décrit l'ensemble des objets disponibles pour un client.

La partie privée contient les déclarations des objets privés à usage exclusif du paquetage

Types privés (1/2)

Intérêt : la déclaration du type est publique, sa définition est cachée aux utilisateurs extérieurs

On peut ainsi modifier la définition d'un type sans avoir à modifier les programmes qui utilisent ce type

La définition d'un type privé est donnée dans la partie **`private`** de l'interface du paquetage. Elle est donc visible mais inutilisable de l'extérieur du paquetage

Types privés (2/2)

Syntaxe

```
<déclaration_type_privé>::=
                        type <ident_type> is private;
```

Opérations disponibles pour un utilisateur extérieur :

1. Les opérations sur ce type déclarées dans la partie publique
2. L'affectation (:=), l'égalité (=), l'inégalité (/=)

Types privés et intégrité des données

A un type (ensemble de valeurs) est associé un ensemble de sous-programmes (procédures et fonctions) qui opèrent sur lui.

Aucun autre sous-programme ne doit pouvoir agir sur ce type pour porter atteinte à l'intégrité des valeurs de ce type.

```
package dates is
   subtype Jour is Integer range 1..31;
   subtype Mois is Integer range 1..12;
   type Date is
      record
         unJour:Jour;
         unMois:Mois;
         unAn:Positive;
      end record;
   procedure changeJour(d:in out Date;j:in Jour);
   procedure changeMois(d:in out Date;m:in Mois);
   procedure changeAn(d:in out Date;a:in Positive);
   function quelJour(d:Date) return Jour;
   function quelMois(d:Date) return Mois;
   function quelAn(d:Date) return Positive;
end dates;
```

```
package body dates is
   procedure changeJour(d:in out Date;j:in Jour) is
   begin d.unJour:=j; end changeJour;

   procedure changeMois(d:in out Date;m:in Mois) is
   begin d.unMois:=m; end changeMois;

   procedure changeAn(d:in out Date;a:in Positive) is
   begin d.unAn:=a; end changeAn;

   function quelJour(d:Date) return Jour is
   begin return d.unJour; end quelJour;

   function quelMois(d:Date) return Mois is
   begin return d.unMois; end quelMois;

   function quelAn(d:Date) return Positive is
   begin return d.unAn; end quelAn;
end dates;
```

```
with Ada.Text_io,dates;use Ada.Text_io,dates;
with Ada.Integer_Text_io;use Ada.Integer_Text_io;
procedure manipdates is
   uneDate:Date;
   leJour:Jour:=5;
   leMois:Mois:=12;
   lAn:Positive:=2002;
begin
   changeJour(uneDate,leJour);
   changeMois(uneDate,leMois);
   changeAn(uneDate,lAn);
   put( "Jour=" );put(quelJour(uneDate));new_line;
   uneDate.unJour:=30;
   put( "Jour=" );put(quelJour(uneDate));new_line;
end manipdates;
```

Paquetage `Date` sans intégrité des données

Le résultat d'exécution du programme précédent donne :

```
Jour=    5
Jour=    30
```

L'utilisateur (`manipdates`) a donc la possibilité de changer directement une valeur du type `Date` sans passer par une opération du type `Date`.

Paquetage `Date` avec intégrité des données

```
package dates is
   subtype Jour is Integer range 1..31;
    subtype Mois is Integer range 1..12;
   type Date is private;
   procedure changeJour(d:in out Date;j:in Jour);
    procedure changeMois(d:in out Date;m:in Mois);
    procedure changeAn(d:in out Date;a:in Positive);
    function quelJour(d:Date) return Jour;
    function quelMois(d:Date) return Mois;
    function quelAn(d:Date) return Positive;
private
      type Date is
       record
          unJour:Jour;unMois:Mois;unAn:Positive;
       end record;
end dates;
```

Paquetage `Date` avec intégrité des données : résultat

Le résultat d'exécution du nouveau programme donne :

```
manipdates:14:04: invalid prefix in selected component
   "uneDate "
```

L'utilisateur (`manipdates`) n'a plus la possibilité de changer directement une valeur du type `Date` sans passer par une opération du type `Date`.

Types limités privés

Syntaxe

```
<déclaration_type_limité_privé>::=
  type <ident_type> is limited private;
```

Opérations

- Les opérations suivantes sont **indisponibles** pour un utilisateur extérieur au paquetage :
  - affectation (:=)
  - égalité (=)
  - inégalité (/=)

Types abstraits de données et paquetages (1/2)

Dans l'esprit de la conception par objets, on spécifie un paquetage sous la forme d'un *type abstrait de données* (TAD).

Par exemple, si après analyse, on fait apparaître la notion de compteur, alors on le définit par un ensemble de valeurs et un ensemble d'opérations sur ces valeurs

Types abstraits de données et paquetages (2/2)

```
type Compteur
 ensemble de valeurs
       N (ensemble des entiers naturels)
 opérations
     initialiser : vide → N
     incrementer : N → N
     decrementer : N → N
     zero : N → booléen
```

Pour être utilisable, ce TAD doit être implanté sous la forme d'un paquetage Ada

Interface du paquetage **compteurs**

```
package compteurs is
 type Compteur is limited private;
 procedure initialiser(X:out Compteur);
 procedure incrementer(X:in out Compteur);
 procedure decrementer(X:in out Compteur);
 function zero(X: Compteur) return Boolean;
private
 type Compteur is new Integer;
end compteurs;
```

Remarques :

- `limited private` => affectation impossible sur le type `Compteur`
- Pour modifier la valeur d'un compteur, il faut obligatoirement utiliser `incrementer` et/ou `decrementer`

Implantation du paquetage **compteurs**

```
package body compteurs is
 procedure initialiser(X:out Compteur) is
  begin X:=0; end initialiser;

 procedure incrementer(X:in out Compteur) is
  begin X:=X+1; end initialiser;

 procedure decrementer(X:in out Compteur) is
  begin X:=X-1; end initialiser;

 function zero(X: Compteur) return Boolean is
  begin return X=0; end zero;
end compteurs;
```

Exemple (1/10)

On souhaite écrire un programme, qui, étant donnée une suite de températures, cherche s'il y un nombre égal de températures positives et négatives.

2 solutions sont proposées

Exemple (2/10)

```
with Ada.Text_io, compteurs; use Ada.Text_io;
procedure solution_1 is
 TEMP_MAX:constant Float:=+60.0;
 TEMP_MIN:constant Float:=-60.0;
 type Temperature is Float range TEMP_MAX..TEMP_MIN;
 type Reponse is (oui,non);
 package temperature_io is new Float_io(Temperature);
 package reponse_io is new Enumeration_io(Reponse);
 tmp:Temperature;
 posNeg:compteurs.Compteur;
 encore:Reponse:=oui;
begin
```

Exemple (3/10)

```
 compteurs.initialiser(posNeg);
  put( "Tapez une température : " );
  while encore=oui loop
   temperature_io.get(temp);
   if temp>0.0 then compteurs.incrementer(posNeg);
   else compteurs.decrementer(posNeg); end if;
   put( "Nouvelle température (oui/non): " );
   reponse_io.get(encore);
  end loop;
  if compteurs.zero(posNeg)
  then put( "températures positives=températures
    négatives" );
  else put( "températures positives/=températures
    négatives" );
  end if;
  new_line;
end solution_1;
```

Exemple (4/10)

Le programme est indépendant de la manière dont sont implantées les opérations du type `Compteur`

Il n'est pas indépendant de la manière dont sont implantées les opérations du type `Temperature`.

Les opérations sur le type `Température` sont dissociées de la définition de ce type.

Exemple (5/10)

La seconde solution propose de définir le type `Temperature` comme un type de données abstrait et en l'implantant dans un paquetage `temperatures`.

Exemple (6/10)

Le Type de Données Abstrait `Temperature`

```
type Temperature
 ensemble de valeurs : [-60.0..+60.0]
 opérations
  get : vide → Temperature
  put : Temperature → vide
  supZero : Temperature → Boolean
```

Exemple (7/10)

```
package temperatures is
 type Temperature is private;
 procedure get(X:out Temperature);
 procedure put(X:in Temperature);
 function supZero(X: Temperature) return Boolean;
private
 TEMP_MAX:constant Float:=+60.0;
 TEMP_MIN:constant Float:=-60.0;
 type Temperature is new
              Float range TEMP_MIN..TEMP_MAX;
end temperatures;
```

Exemple (8/10)

```
with Ada.Text_io;use Ada.Text_io;
package body temperatures is
 package temperature_io is new Float_io(Temperature);
 procedure get(X:out Temperature) is
 begin
  put( "Taper une température : " );
  temperature_io.get(X);
 end get;
 procedure put(X:in Temperature) is
 begin temperature_io.put(X); end get;
 function supZero(X:Temperature) return Boolean is
 begin
  if X>0.0 then return true;
  else return false; end if;
 end supZero;
end temperatures;
```

Exemple (9/10)

```
with Ada.Text_io,compteurs,temperatures;
use Ada.Text_io,compteurs,temperatures;
procedure solution_2 is
 type Reponse is (oui,non);
 package reponse_io is new Enumeration_io(Reponse);
 tmp:Temperature;
 posNeg: Compteur;
 encore:Reponse:=oui;
begin
```

Exemple (10/10)

```
 initialiser(posNeg);
 while encore=oui loop
  get(temp);
  if supZero(temp) then incrementer(posNeg);
  else decrementer(posNeg); end if;
  put( "Nouvelle température (oui/non): " );
  reponse_io.get(encore);
 end loop;
 if zero(posNeg)
 then
  put( "températures positives=températures négatives" );
 else
 put( "températures positives/=températures négatives" );
 end if;
 new_line;
end solution_2;
```

Le paquetage : `listeEntiers` (1/2)

```
package listeEntiers is
 ListeVide:exception;
 type Liste is private;
 function cons(elt:Integer;l:Liste) return Liste;
 function vide return Liste;
 function estVide(l:Liste) return Boolean;
 function tete(l:Liste) return Integer;
 function queue(l:Liste) return Liste;
private
 type Cellule;
 type Liste is access Cellule;
 type Cellule is
  record
   valeur:Integer;
   suivant:Liste;
  end record;
end listeEntiers;
```

Le paquetage : `listeEntiers` (2/2)

```
package body listeEntiers is
 function cons(elt:Integer;l:Liste) return Liste is
 begin return new Cellule'(elt,l); end cons;
 function vide return Liste is
 begin return NULL; end vide;
 function estVide(l:Liste) return Boolean is
 begin return l=NULL; end estVide;
 function tete(l:Liste) return Integer is
 begin
  if l=NULL then raise listeVide;
  else return l.valeur; end if;
 end tete;
 function queue(l:Liste) return Liste is
 begin
 if l=NULL then raise listeVide;
  else return l.suivant; end if;
 end queue;
end listeEntiers;
```

Utilisation de `listeEntiers`

Ecrire un programme qui :

1. Saisit un nombre indéterminé d'entiers jusqu'à ce qu'un caractère non numérique soit saisi
2. Calcul la moyenne des valeurs saisies
3. Affiche la liste constituée
4. Affiche la moyenne

```
with listeEntiers;use listeEntiers;
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;use Ada.Integer_Text_io;
procedure calculMoyenne is
   function somme(l:Liste) return Integer is
   begin
   if estVide(l) then raise listeVide;
      elsif estVide(queue(l)) then return tete(l);
         else return tete(l)+somme(queue(l));
      end if;
   end somme;
   function longueur(l:Liste) return Natural is
   begin
      if estVide(l) then return 0;
      else return 1+longueur(queue(l));
      end if;
   end longueur;
```

```
procedure put(l:in Liste) is
      x:Liste:=l;
begin
       put( "(" );
       loop
       exit when estvide(x);
       put(tete(x),WIDTH=>2);
       put( "," );
       x:=queue(x);
       end loop;
       put_line( ")" );
end put;

laListe:Liste;
taille:Positive;
```

```
begin
 while true loop
       put( "tapez un entier : " );
       get(taille);
       laListe:=cons(taille,laListe);
 end loop;

exception
   when DATA_ERROR=>put(laListe);new_line;
    put( "Moyenne=" );
    put(somme(laListe)/longueur(laListe),WIDTH=>2);
   when listeVide=>put_line( "liste vide!" );
end calculMoyenne;
```

Modularité

Une unité de programme, pour être modulaire, doit être indépendante et autonome :

- Autonome : forte cohérence interne. Elle doit être le reflet fidèle d'un objet de l'espace du problème

- Indépendante : faiblement couplée. Elle entretient un ensemble de relations minimum et explicitement identifié avec les autres unités du programme

Identification des modules

L’étape de conception réalise un découpage qui fait apparaître les modules constituant son architecture.

Deux approches possibles :

- Approche descendante
- Approche ascendante

Nature des modules

Les modules se traduisent par des paquetages Ada.

Ils remplissent des fonctions de nature différente dans le programme :

- Type abstrait de données
- Machine abstraite
- Regroupement de ressources

Approche descendante

Elle procède par la décomposition d'un problème en sous-problème et ainsi de suite jusqu'à un sous-problème trivial.

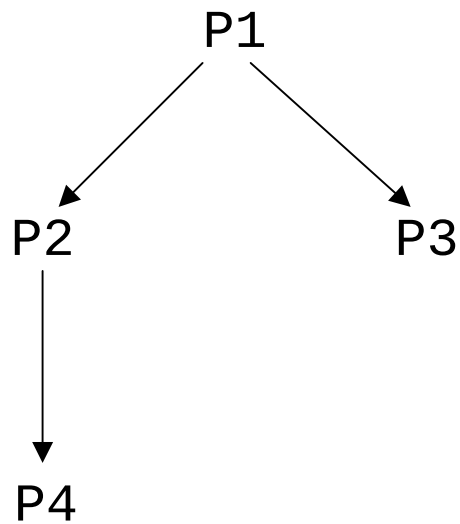
Elle n'est valide que si, à chaque niveau de raffinement, la solution des sous-problèmes identifiés est supposée connue.

Traduction Ada :

- S'appuie sur la notion d'unité (paquetage, procédure)
- Chaque unité contient la spécification du niveau inférieur
- Chaque unité peut être compilée séparément. Pour cela, le corps des unités internes est remplacé par le mot clé **`separate`**

Exemple d'architecture Ada

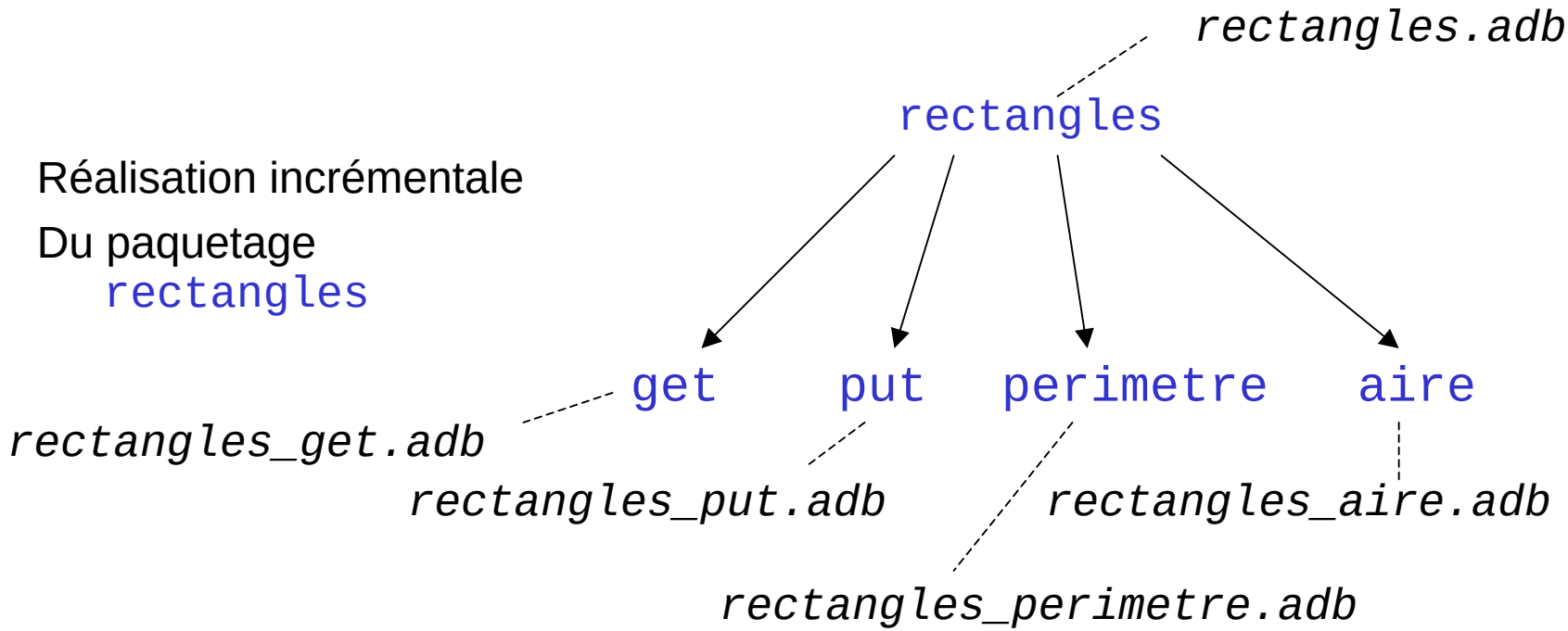
```
procedure P1 is
 ..
 procedure P2(…) is separate;
 procedure P3(…) is separate;
 ..
end P1;
separate (P1)
procedure P2 is
 ..
 procedure P4(…) is separate;
 ..
end P2;
separate (P1)
procedure P3 is
 ..
end P3;
separate (P1.P2)
procedure P4 is
 ..
end P4;
```



Exemple : le paquetage **rectangles**

Réalisation incrémentale

Du paquetage
`rectangles`



Exemple : interface de `rectangles`

```
package rectangles is
   type Rectangle is private;
   procedure get(r:out Rectangle);
   procedure put(r:in Rectangle);
   function perimetre(r:Rectangle) return Positive;
   function aire(r:Rectangle) return Positive;
private
   type Rectangle is
      record
         long:Positive;
         larg:Positive;
      end record;
end rectangles;
```

Exemple : implantation de **rectangles**

```
package body rectangles is
   procedure get(r:out Rectangle) is separate;
   procedure put(r:in Rectangle) is separate;
   function perimetre(r:Rectangle)
                              return Positive is separate;
   function aire(r:Rectangle)
                              return Positive is separate;
end rectangles;
```

Exemple : implantation séparée de l'unité `get`

```
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;use Ada.Integer_Text_io;

separate (rectangles)
procedure get(r:out Rectangle) is
begin
  put( "Entrez la largeur : " );
  get(r.larg);
  put( "Entrez la longueur : " );
  get(r.long);
end get;
```

Exemple : implantation séparée de l'unité **put**

```
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;use Ada.Integer_Text_io;

separate (rectangles)
procedure put(r:in Rectangle) is
begin
  put_line( "--- Affichage d'un rectangle ---" );
  put( "Largeur=" );put(r.larg,WIDTH=>2);new_line;
  put( "Longueur=" );put(r.long,WIDTH=>2);new_line;
end put;
```

Exemple : implantation séparée de `perimetre` et `aire`

```
separate (rectangles)
function perimetre(r:Rectangle)
                        return Positive is
begin
  return 2*(r.long+r.larg);
end perimetre;
```

```
separate (rectangles)
function aire(r:Rectangle) return Positive is
begin
  return r.long*r.larg;
end aire;
```

Exemple : utilisation de `rectangles`

```
with rectangles;use rectangles;
with Ada.Integer_Text_io;use Ada.Integer_Text_io;
with Ada.Text_io;use Ada.Text_io;

procedure maniprectangles is
  r:Rectangle;
begin
  get(r);put(r);
  put( "périmètre =" );put(perimetre(r),WIDTH=>2);
  new_line;
  put( "aire = " );put(aire(r),WIDTH=>2);
  new_line;
end maniprectangles;
```

Nature des modules : Type Abstrait de Données

Identification des données manipulées et des opérations associées => identification d'un type de données.

On parle *d'abstraction de données* car l'implantation physique des opérations et le représentation physique des données est laissée de coté.

Le *Type de Données Abstrait* est implanté en Ada par un paquetage qui *exporte* le *type concret.*

L'interface du paquetage décrit le *contrat* entre l'utilisateur du paquetage (du type) et son développeur

Nature des modules : regroupement de ressources

Le paquetage regroupe des constantes, exceptions, sous-programmes d'intérêt général

```
with Ada.Text_io;use Ada.Text_io;
package e_s is
package positif_io is new Integer_io(Positive);
package naturel_io is new Integer_io(Natural);
package bool_io is new Enumeration_io(Boolean);
package reel_io is new Float_io(Float);
end e_s;
```

Regroupement de ressources

```
with e_s; use e_s;
with Ada.Text_io; use Ada.Text_io;

procedure client_e_s is
   a:Positive := 89;
   b:Natural := 23;
   c:Boolean := false;
   d:Float := 56.8;
begin
   positif_io.put( a, WIDTH=>3 ); new_line;
   naturel_io.put( b, WIDTH=>3 ); new_line;
   bool_io.put( c ); new_line;
   reel_io.put( d ); new_line;
end client_e_s;
```

Nature des modules : machine abstraite

Le paquetage encapsule la donnée

Pas de type exporté, seulement des opérations

La donnée (un objet) est déclarée dans le corps

Le paquetage contient un objet dont *l'état est modifié* au cours de l'exécution par les opérations du paquetage

Exemple : interfaces des paquetage `livres` et `bibliothèque` [G.Booch]

```
package livres is
   subtype Livre is String(1..21);
end livres;

with livres;use livres;
package bibliotheque is
   procedure ajouter(l:in Livre);
   procedure retirer(l:in Livre);
   procedure put;
end bibliotheque;
```

Exemple : corps du paquetage
bibliotheque (1/3)

```
with livres;use livres;
with Ada.Text_io;use Ada.Text_io;

package body bibliotheque is
   biblio:array(1..5) of Livre;
   indice:Positive range 1..biblio'last:=1;
   procedure ajouter(l:in Livre) is
   begin
      biblio(indice):=l;
      indice:=indice+1;
   end ajouter;
```

Exemple : corps du paquetage
bibliotheque (2/3)

```
procedure retirer(l:in Livre) is
   begin
      for i in 1..biblio'last-1 loop
         if biblio(i)=l
         then
            indice:=indice-1;
            for j in i..biblio'last-1 loop
               biblio(j):=biblio(j+1);
            end loop;
         end if;
      end loop;
   end retirer;
```

Exemple : corps du paquetage bibliotheque (3/3)

```
procedure put is
   begin
      for i in biblio'range loop
         put_line(biblio(i));
      end loop;
   end put;
end bibliotheque;
```