Introduction to Systematic Programming
Unit 17 - Writing Larger Programs; Packages

17.1 Introduction

In our earlier approach to program design (see Unit 7) we have advocated the use of top-down design and stepwise refinement as a method which we can reliably use to produce programs of moderate size, typically of up to a few hundred lines of program. This design method results in programs which are largely self-contained, but which import small numbers of procedures, functions, etc, from other packages. A illustration of this point is the use of packages to provide the procedures and functions for performing input and output. Thus for example we have used:

package name	**facilities provided**	**comment**
`Ada.Text_IO`	`Get`	for character input
	`Put`	for character output
	`End_Of_Line`	for locating the end of a line
	`Get`	for string input
	`Put`	for string output
	`Get_Line`	for string input
	`Put_Line`	for string output
	`End_Of_File`	for locating the end of a file
	`New_Line`	
	`Skip_Line`	... and so on.

We have also used many other I/O procedures from different packages. For convenience we will refer to the procedures, functions, etc. which may be imported from a package as **package components**.

It is useful to see how a package is written, so let us examine some of the contents of the package `Ada.Text_IO`. It consists of two parts, a **package declaration** and a **package body** as follows:

```
PACKAGE Ada.Text_IO IS

     -- Exported procedures, functions, etc.

     PROCEDURE Get (Item : OUT Character);

     PROCEDURE Put (Item : IN Character);

     FUNCTION End_Of_Line RETURN Boolean;

     .................

END Ada.Text_IO;


PACKAGE BODY Ada.Text_IO IS

     -- Private definitions of procedures, functions, etc.

     PROCEDURE Get (Item : OUT Character) IS
     BEGIN -- Steps of the body of the procedure Get
          .................
     END Get;

     PROCEDURE Put (Item : IN Character) IS
     BEGIN -- Steps of the body of the procedure Put
          .................
     END Put;

     FUNCTION End_Of_Line RETURN Boolean IS
     BEGIN -- Steps of the body of the function End_Of_Line
          .................
          RETURN ..........
     END End_Of_Line;

     .................

END Ada.Text_IO;
```

In this case the package declaration consists of incomplete procedure and function definitions, ie. the headings of the procedures and functions `Get`, `Put`, `End_Of_Line`, etc. The package body, on the other hand, contains of the full definitions of these procedures and functions.

In our previous use of procedures and functions we have defined them by using the form shown in the package body, however we see that this definition can be subdivided into two. Firstly, a **procedure** or **function declaration** (the incomplete form shown in the package declaration), and secondly a **procedure** or **function body** (the complete form shown in the package body). This separation is useful here, because the package declaration includes the definition of those components which are exported from the package, and the package body the definition of those parts of the package which are necessary for its internal workings and functionality, but which are not exported, ie. they are private to the package, and are not visible to a client program which imports components from the package.

Thus:	**package declaration**	**package body**
	is the 'public view' of the package, containing those parts of the package which are necessary for a client program to be able to use its contents, often referred to as the package interface.	is the 'private view' of the package, containing the detailed working of its contents, which the client program has no need to see.

For example, a client program which imports and uses the procedure `Get` to input a data value has no need to 'see' or understand how this procedure works internally. Indeed *you have been using versions of this procedure* since it was introduced in Unit 1 without 'seeing' its complete definition. In the case of procedures and functions exported by a package, all that a client program needs to know is:

i) The name of the procedure or function.

ii) How it is called, ie. what are the names, modes and types of its formal parameters (and in the case of a function, what type of value is returned).

iii) What computation or processing is performed by the procedure or function. Notice this is *what* computation is to be performed, not *how* it is to be achieved.

We see that a package declaration, containing procedure declarations (ie. just the procedure headings) contains the information needed for (i) and (ii) above. The documentation which accompanies a package normally provides the information needed for (iii). This documentation is sometimes included in the package declaration, which then contains comments describing the purpose of each procedure.

This separation of a package into 'declaration' and 'body' parts has many advantages. For example the detailed implementation of the package components (contained in the package body) may be 'hidden' from the client program (which has no need to 'see' their workings) - thus ensuring that the package components cannot be accidentally corrupted by an incorrect client program. Another example would be that the detailed implementation of the package components may be changed by its programmer (possibly to correct errors or to implement a more efficient algorithm) without affecting the way in which client programs import and use its components. We see that there are advantages to be gained if the package declaration and package body are stored in separate files, and compiled separately, and this is the normal arrangement for defining a package.

17.2 What components can be exported from a package?

The most common components exported from a package are:

i) Constants.
So far we have seen no examples of this.

ii) Types and subtypes.
We have seen one illustration of this in the example program at the end of Unit 16, where the package `Enumeration_IO` (and hence the package `SeatType_IO`) has provided an enumeration type with the two literal values `Upper_Case` and `Lower_Case`.

iii) Procedures and functions.
We have seen many examples where procedures and functions have been imported from `Ada.Text_IO`, `CS_Int_IO`, `CS_Flt_IO` and `CS_File_IO` for performing input and output of values of different types.

17.3 Using the components of a package in a client program

We are already familiar with the two ways in which the components of a package can be referred to in a program. In the first approach, the client program includes the context clauses:

```
WITH Package_name;
USE  Package_name;
```

and then an imported component can be used in the client program simply by stating its name. Throughout these units we have consistently adopted this approach when importing from `Ada.Text_IO`, `CS_Int_IO`, `CS_Flt_IO` and `CS_File_IO` components (ie. procedures and functions) for performing I/O, and when importing from `Ada.Numerics.Elementary_Functions` the mathematical functions described in Unit 9. This approach has been adopted since most Ada programs will make extensive use of procedures and functions from these packages, and it is therefore more convenient to allow for their use in the most straightforward manner possible, ie. by just stating the name of the component.

However as the number of packages used in a client program increases, a number of problems arise with this approach, namely:

a) The program becomes difficult to read and understand since it is not immediately obvious whether any given identifier referred to in a client program is defined in that program or is imported from a package (or indeed if it is imported from a package, it is not obvious from which package it may have been imported).

b) We need to take care to avoid the possibility that we may define identifiers in the client program which are the same identifiers as those of components imported from packages.

b) The possibility that different packages will contain components with the same identifiers increases. This clearly causes some difficulties.

The answer to these problems is to use the second approach which requires that the client program includes the single context clause (for each package used):

```
WITH Package_name;
```

Then to use an imported component in the client program we state both the package and component names using:

```
Package_name.Component_name
```

This form clarifies any ambiguity there may previously have been in (a) about the origin of an identifier. For the reasons outlined above we will continue to use the first approach for components imported from `Ada.Text_IO`, `CS_Int_IO`, `CS_Flt_IO` and `CS_File_IO` for performing I/O, and for functions imported from the `Ada.Numerics.Elementary_Functions` package, and the second approach for all other imported package components.

17.4 Why should we want to write our own packages?

There are a number of reasons why we might wish to write our own packages. These are:

a) As a repository for commonly used components which have been developed in the solution of one problem and are of sufficiently general applicability to be likely to be re-used in the solution of other problems.

b) As a means for decomposing very large programs into smaller, more manageable program units.

c) As a convenient mechanism for implementing data structures.

We will now examine the rationale, and look at some examples of (a) and (b). The rationale for using packages for (c) is beyond the scope of this course - but it will be covered in later courses.

We will see that the process of division of a large program into packages (sometimes referred to as modules), which is known as **modularisation**, is a very important one in large-scale program design. We only have time to examine a few of the issues relating to using packages to achieve program modularisation in this unit, and the full impact of modularisation on the complete program design, development and maintenance processes is beyond the scope of this course. However, it will be dealt with more fully in later courses.

17.4.1 Reusing commonly used components

In Unit 13 we introduced the enumeration type `DayOfWeek` to represent the days of the week. In Unit 15 we introduced the enumeration type `MonthOfYear` to represent the months of a year. We also introduced in Unit 16 a record type `ClockTime` to represent a time of the day. Together these provide mechanisms for representing a precise instant in time, ie. year, month, day and time of day, which we might find useful in several different programs. Thus having written these type definitions (and (say) some procedures and functions for processing such values), possibly as part of the development of a program for a particular application, how might we use them in other programs? We could place their text into files and edit them into our new programs as required, but:

a) This is a time-consuming and tedious process.

b) It creates duplicate copies within each program that they are used.

c) It requires that they be recompiled within each program that they are used.

A better approach would be to form a new package which contains the type definitions, procedures and functions, and allow them to be used by importing them into the programs where they are required. Hence in this example we might produce the package:

```
PACKAGE Date_Pack IS
   TYPE DayOfWeek   IS (Mon, Tues, Wed, Thurs, Fri, Sat, Sun);
   TYPE MonthOfYear IS (Jan, Feb, Mar, Apr, May, Jun,
                        Jul, Aug, Sep, Oct, Nov, Dec);
   SUBTYPE DayOfMonth IS Positive RANGE 1..31;
   TYPE DateType IS RECORD
                       Day       : DayOfWeek;
                       DayNumber : DayOfMonth;
                       Month     : MonthOfYear;
                       Year      : Positive;
                    END RECORD;
   SUBTYPE HourRange IS Natural RANGE 0..23;
   SUBTYPE MinRange  IS Natural RANGE 0..59;
   TYPE ClockTime IS RECORD
                        Hours   : HourRange;
                        Minutes : MinRange;
                     END RECORD;

   FUNCTION IsLeap (Year : Positive) RETURN Boolean;
      -- To deliver True if Year is a leap year, and False otherwise
   FUNCTION MonthLength (Month : MonthOfYear;
                         Year  : Positive) RETURN DayOfMonth;
      -- To deliver the number of days in the Month of this Year
END Date_Pack;
```

```
PACKAGE BODY Date_Pack IS
   FUNCTION IsLeap (Year : Positive) RETURN Boolean IS
   BEGIN
      RETURN (Year REM 4 = 0 AND Year REM 100 /= 0)
                OR Year REM 400 = 0;
   END IsLeap;

   FUNCTION MonthLength (Month : MonthOfYear;
                         Year  : Positive) RETURN DayOfMonth IS
   BEGIN
      CASE Month IS
         WHEN Jan|Mar|May|Jul|Aug|Oct|Dec => RETURN 31;
         WHEN Apr|Jun|Sep|Nov             => RETURN 30;
         WHEN Feb => IF IsLeap(Year) THEN
                        RETURN 29;
                     ELSE
                        RETURN 28;
                     END IF;
      END CASE;
   END MonthLength;
END Date_Pack;
```

Notice that constants, types, subtypes, etc. included in a package declaration are available for use within the package body (just as if they had been declared within the package body).

In our approach to procedurisation, we have advocated the use of parameters and local variables in order to write procedures and functions which are self-contained; hence allowing them to be used to perform similar computations in several programs, and thus making them more readily available for re-use. An example of this principle would be the function `Sum` from Unit 15:

```
FUNCTION Sum(A : IN Vector) RETURN Float IS
    -- To deliver the total of all the elements in the array A
    TotalSoFar : Float := 0.0;
BEGIN
    FOR I IN A'Range LOOP
        TotalSoFar := TotalSoFar + A(I);
    END LOOP;
    RETURN TotalSoFar;
END Sum;
```

which could be used to form the total of the values stored in an array of any constrained subtype of the unconstrained type:

```
TYPE Vector IS ARRAY (Integer RANGE <>) OF Float;
```

In the case of a package containing the `Sum` function above, the question arises as to where the type `Vector` should be declared. Since the definition of the `Sum` function makes reference to `Vector`, the type should also be defined in the package. It would also be logical to collect together any other procedures and functions we had written for processing arrays of type `Vector` into the package. We would thus obtain the package `Vector_Pack` (say), defined by:

```
PACKAGE Vector_Pack IS
     TYPE Vector IS ARRAY (Integer RANGE <>) OF Float;
     FUNCTION Sum(A : IN Vector) RETURN Float;
     -- Plus other procedure and function declarations
     -- for manipulating arrays of type Vector
     ................
END Vector_Pack;
```

and:

```
PACKAGE BODY Vector_Pack IS
     FUNCTION Sum(A : IN Vector) RETURN Float IS
          TotalSoFar : Float := 0.0;
     BEGIN
          FOR I IN A'Range LOOP
               TotalSoFar := TotalSoFar + A(I);
          END LOOP;
          RETURN TotalSoFar;
     END Sum;
     -- Plus other procedure and function bodies
     -- for manipulating arrays of type Vector
     .................
END Vector_Pack;
```

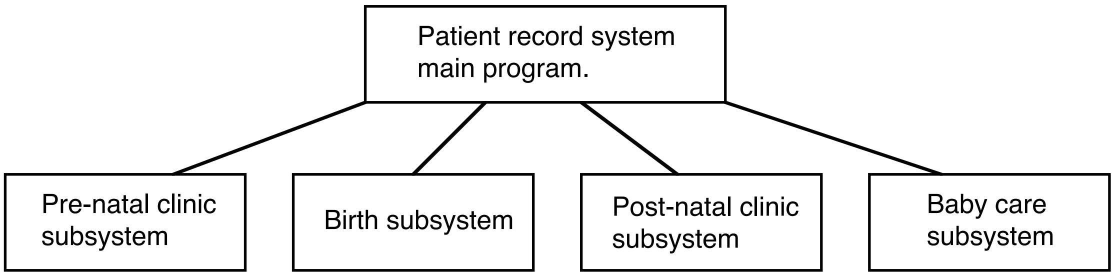
Then we could write a program which needed to declare and process variables of the type `Vector`, by including steps of the form:

```
WITH Vector_Pack;
...............
PROCEDURE ExampleProg IS
     SUBTYPE BigVector IS Vector_Pack.Vector(1..100);
     X, Y  : BigVector;
     Total : Float;
BEGIN
     ...............
     Total := Vector_Pack.Sum(X) + Vector_Pack.Sum(Y);
     ...............
END ExampleProg;
```

to form in `Total` the sum of the contents of two 100 element arrays `X` and `Y`.

17.4.2 Decomposing large programs into packages

In the design of very large programs, consisting of many hundreds or thousands of lines of program code, we will often find that the program will be composed of several quite distinct subsections, each subsection interacting with the others. For example, suppose we were to design a software system for recording and managing mother and baby patient records in a maternity hospital. The program may have several 'primary' subsystems, represented diagramatically by:


Patient record system main program.
Pre-natal clinic subsystem
Birth subsystem
Post-natal clinic subsystem
Baby care subsystem

Where the user will typically be using the program to enter and manipulate records when (potential) mothers attend pre-natal clinics, or when the baby is born, or when the mother and baby attend post-natal clinics, or when clinics are held to monitor the development of the baby into childhood. Thus over a period of time more and more information will be gathered about the progress of mothers and their babies. However at any particular instant in time the user of the system will be largely concerned with interacting with one of the subsystems, rather than with the system as a whole. Hence the main program might consist simply of some initialisation of the system together with the presentation of a menu, so that the user can select an appropriate subsystem. It might thus consist of the outline steps:

```
Input patient data from file ................................ A
LOOP
    Display menu for user ................................... B
    Get UserChoice .......................................... C
    CASE UserChoice IS
        WHEN PreNatal  => Enter the pre-natal subsystem ....... D
        WHEN Birth     => Enter the birth subsystem ........... E
        WHEN PostNatal => Enter the post-natal subsystem ...... F
        WHEN BabyCare  => Enter the baby care subsystem ....... G
        WHEN Quit      => EXIT
    END CASE
END LOOP
Save patient data to file ................................... H
```

where we have made explicit reference to the enumeration variable:

```
TYPE MenuOption IS (PreNatal, Birth, PostNatal, BabyCare, Quit);
UserChoice : MenuOption;
```

and implicit reference to a data structure to hold the list of patient records.

The 'primary' subsystems represented in the diagram above may also make reference to other 'utility' subsystems. For example we may have a subsystem to define constants, types and subtypes used throughout the rest of the system, another to access and manipulate the list of patient records and another to arrange the user input and output.

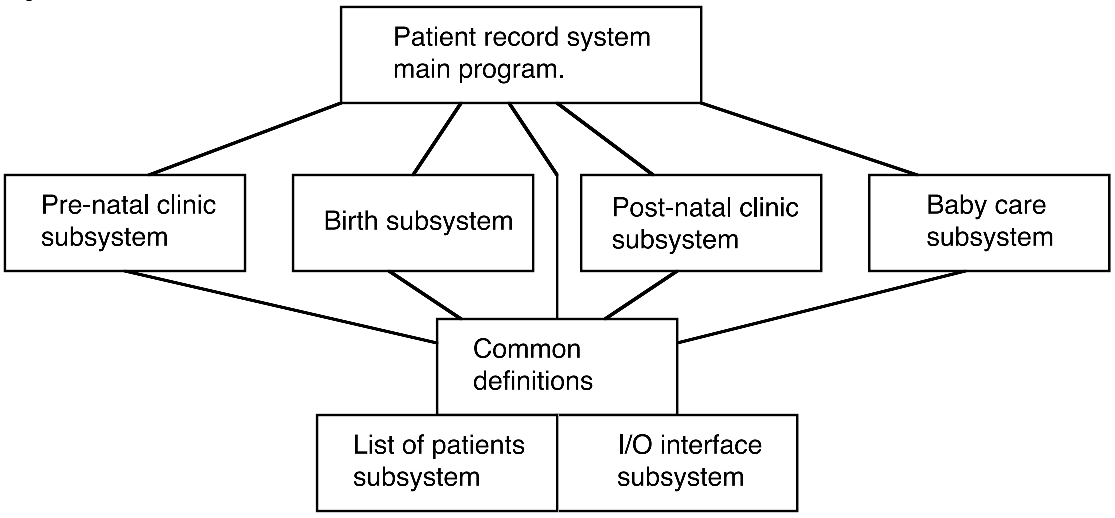
Normally the subsystem to define common constants, types and subtypes would be written as a package, for example:

```
WITH Date_Pack;
PACKAGE Common_Pack IS
    TYPE MenuOption IS (PreNatal, Birth, PostNatal, BabyCare, Quit);
    TYPE GenderType IS (Male, Female);
    MaxNameLen : CONSTANT Positive := 10;
    SUBTYPE NameString IS String(1..MaxNameLen);
    TYPE FullName IS RECORD
                         ForeName   : NameString;
                         SecondName : NameString;
                         Surname    : NameString;
                     END RECORD;
    TYPE Status IS (BeforeBirth, AfterBirth);
```

```
        TYPE PatientType IS RECORD
                               ProgressState : Status;
                               MothersName   : FullName;
                               BabysName     : FullName;
                               BabysGender   : GenderType;
                               TimeOfBirth   : Date_Pack.ClockTime;
                               DateOfBirth   : Date_Pack.DateType;
                            END RECORD;
    END Common_Pack;
```

Notice that since `Common_Pack` only contains definitions of constants, types and subtypes, and does not contain any procedures or functions (whose complete definitions need to be provided in a package body), there is no need for a `Common_Pack` package body.

Typically each of the remaining subsystems will be quite a large programming task, and of such a complexity that they would normally be given to separate programmers or programmer teams for independent development. The final system being reconstructed from these 'building blocks' when they are complete and have been independently tested. This development process can of course be performed using many target programming languages. However, Ada is designed to allow such program decomposition to be undertaken easily and systematically, by implementing the subsystems as packages. Once the `Common_Pack` package, and the package declarations for the other subsystems have been written, the separate programmers or programmer teams can develop and test their corresponding subsystem package bodies largely independently. The diagram below illustrates the interdependence of all the subsystems, ie. which packages import components from other packages.


Patient record system main program.
Pre-natal clinic subsystem
Birth subsystem
Post-natal clinic subsystem
Baby care subsystem
Common definitions
List of patients subsystem
I/O interface subsystem

We have some idea how the 'primary' subsystems may be written as packages. For example the post-natal clinic subsystem will comprise a package declaration containing a single procedure (which is imported into and called from the main program), ie. a package declaration of the form:

```
        PACKAGE Post_Natal_Pack IS
            PROCEDURE Process_Clinic (.....);
        END Post_Natal_Pack;
```

and the step **F** of the main program will then consist of the procedure call:

```
        Post_Natal_Pack.Process_Clinic (.....);
```

The corresponding package body will consist of the full definition of the exported procedure, together with any additional constants, types, subtypes, variables, procedures and functions necessary to complete this exported procedure, ie. a package body of the form:

```
        WITH Date_Pack;
        WITH Common_Pack;
        PACKAGE BODY Post_Natal_Pack IS
            ...................
            PROCEDURE Process_Clinic (.....) IS
            BEGIN
                ...............
            END Process_Clinic;
            ...................
        END Post_Natal_Pack;
```

We can see an obvious reason for the existence of the procedures corresponding to each of the 'primary' subsystems. They are just the procedures we might have identified by applying our top-down design method to the problem, except they are going to be defined in separate packages rather than in the main program so that their independent development is easier to achieve. It is less obvious why we might wish to choose (and implement as packages) the two remaining 'utility' subsystems mentioned above. We can get some idea of the reason for the list of patient records subsystem since the list of patient records will need to be accessed by all of the 'primary' subsystems and by steps **A** and **H** of the main program. But as was stated earlier its complete rationale and method of implementation is beyond the scope of the course and will therefore not be considered further. However, we will consider the I/O interface subsystem. Suppose we were to define this package using a package declaration something like:

```
WITH Common_Pack;
PACKAGE IO_Pack IS
    PROCEDURE Clear_Screen;
    PROCEDURE Display_Menu;
    PROCEDURE Get_Menu_Choice (Item : OUT Common_Pack.MenuOption);
    ...........
END IO_Pack;
```

where it is our intention that the package should contain all the procedures and functions needed by the rest of the entire system for performing its interactive inputs and outputs, ie. no interactive input or output should be performed by our system by any means other than through the facilities provided by this package. Thus the precise mechanism of interactive communication between the system and the user is localised to the package body of `IO_Pack`. The completed procedures in the package body then have to be written to take account of:

a) The intended hardware, eg. VDU and normal keyboard, or touch sensitive VDU screen, or VDU and specially tailored keyboard, etc.

b) The precise wording and layout of the menu to be output.

c) The precise form of permitted input, eg. whether the user is required to type the number of the item in the menu, or some abbreviated name, or the first letter of the menu description, or is allowed to enter all of these as valid inputs.

Once appropriate decisions had been made concerning the intended interactive user interface, the completed package body could be produced as something like:

```
WITH Common_Pack;
PACKAGE BODY IO_Pack IS

    PROCEDURE Clear_Screen IS
    BEGIN
        ...........
    END Clear_Screen;

    PROCEDURE Display_Menu IS
    BEGIN
        ...........
    END Display_Menu;

    PROCEDURE Get_Menu_Choice (Item : OUT Common_Pack.MenuOption) IS
    BEGIN
        ...........
    END Get_Menu_Choice;
    ...........
  END IO_Pack;
```

Notice that the above details, which implement the entire interactive user interface, are independent of the rest of the software system. Thus it would be possible for example to redesign the user interface from English to French (say) so that the system could be used in a French maternity hospital, or to redesign the user interface from a command based form (say) to a WIMP based form (say), or to redesign the hardware from a normal VDU and keyboard (say), to a touch sensitive screen (say), by rewriting only the contents of the `IO_Pack` package body. No other part of the software system would need to be changed.

Thus we see from the above use of packages that we have a system which is considerably easier to maintain, eg. to alter to take account of changes in the hardware or changes in the requirements of the users, than a system where the definitions of all the I/O activities were distributed throughout the entire software system .