3

Programming in the Large

DeRemer and Kron (1975) distinguished the activities of writing large programs from that of writing small programs. They considered large programs to be systems built from many small programs (modules), usually written by different people. It is common today to separate the features of a programming language along the same lines. In Chapter 2, we presented the aspects of Ada required to write the most basic programs. In this chapter, we discuss some of Ada's features that support the development of large programs.

To facilitate the construction of large programs, Ada makes use of programming units. An Ada program consists of a main subprogram that uses services provided by library units. A *library unit* is a unit of Ada code that we may compile separately. Library units are often called *compilation units*. We have already made use of many predefined library units in our examples. The *with clause* provides access to a library unit. The *use clause* provides direct visibility to the public declarations within a library unit so we do not have to prefix them with the name of the library unit.

A library unit is a subprogram (a procedure or function), package, or generic unit. The main subprogram is itself a library unit. Subprograms, packages, and generic units that are nested within another programming unit are not library units; they must be compiled with the programming unit in which they are nested. Generally, we use a compiler and linker to create an executable from a collection of library units. Library units also play a role in mixing SPARK and non-SPARK code in a single program – a topic we discuss in Chapter 7. In the following sections, we will introduce you to the package and to generic units.

Encapsulation and information hiding are the cornerstones of programming in the large. Both concepts deal with the handling complexity. There are two aspects of encapsulation: the combining of related resources and the separation of specification from implementation. In object-oriented design and

programming, we use encapsulation to combine data and methods into a single entity called a *class*. Encapsulation also allows us to separate what methods a class supplies for manipulating the data without revealing how those methods are implemented.

The *package* is Ada's construct for encapsulation. The package supports abstract data types, separate compilation, and reuse. We write packages in two parts: the package declaration and the package body. The declaration specifies the resources the package can supply to the rest of the program. These resources may include types, subtypes, constants, variables, and subprograms. The package body provides the implementation of the subprograms defined in the package declaration.

Information hiding is related to but different than encapsulation. Encapsulation puts things into a box. Whether that box is opaque or clear determines whether the information is hidden or not.

Information hiding is what we do in the design process when we hide the decisions that are most likely to change. We hide information to protect the other portions of our design from changes to that decision. Modern programming languages provide mechanisms to ensure that details of a design are not accessible to portions of our program that do not need those details.

Information hiding ensures that the users of a class are not affected when we make a change to the implementation of that class. Suppose, for example, our program makes use of a sorted list. If we were to change the implementation of that list from one based on linked lists to one based on arrays, information hiding ensures that this change has no affect on the parts of our program that use a sorted list.

The major Ada construct for information hiding is the *private type*, which is introduced in Section 3.3.2. Private subprograms and private child packages are additional Ada constructs for restricting access to design details.

Although there are many different ways to define and use packages, we can usually place packages into one of four categories: definition packages, utility packages, type packages, and variable packages. This classification scheme is neither strict nor inclusive. In the following sections we will look at an example from each category.

3.1 Definition Packages

A definition package groups together related constants and types. Such packages are useful when the same types must be used in several different programs

or by different programmers working on different parts of one large program. Here is an example of a definition package:

```
with Ada.Numerics;
package Common_Units is

   type Degrees is digits 18 range 0.0 .. 360.0;
   type Radians is digits 18 range 0.0 .. 2.0 * Ada.Numerics.Pi;

   type Volts is delta 1.0 / 2.0**12 range -45_000.0 .. 45_000.0;
   type Amps is delta 1.0 / 2.0**16 range -1_000.0 .. 1_000.0;
   type Ohms is delta 0.125 range 0.0 .. 1.0E8;

   type Light_Years is digits 12 range 0.0 .. 20.0E9;

   subtype Percent is Integer range 0 .. 100;
end Common_Units;
```

This package defines six types and one subtype. It uses the value of π from the Ada library definition package Ada.Numerics. Because definition packages have no subprograms, there is nothing to implement. In fact, the compiler will give us an error should we try to compile a body for it. Here is a short program that uses our definition package:

```
with Common_Units; use type Common_Units.Ohms;
with Ada.Text_IO; use Ada.Text_IO;
procedure Ohms_Law is

   package Ohm_IO is new Fixed_IO (Common_Units.Ohms);
   package Amp_IO is new Fixed_IO (Common_Units.Amps);
   package Volt_IO is new Fixed_IO (Common_Units.Volts);

   A : Common_Units.Amps;
   R1 : Common_Units.Ohms;
   R2 : Common_Units.Ohms;
   V : Common_Units.Volts;
begin
   Put_Line ("Enter current and two resistances");
   Amp_IO.Get (A);
   Ohm_IO.Get (R1);
   Ohm_IO.Get (R2);
   V := A * (R1 + R2);
   Put ("The voltage drop over the two resistors is ");
```

```
   Volt_IO.Put (Item => V,
                Fore => 1,
                Aft  => 2,
                Exp  => 0);
   Put_Line (" volts");
end Ohms_Law;
```

This program also illustrates the *use type clause*. When we declare a type, we define its domain and a set of operations. As type Ohms is a fixed point type, the operations include all of the standard arithmetic operators. To add two resistance values, we use the plus operator. However, because this operator is defined in package Common_Units, we must either prefix the plus operator with the package name or include a use clause to access the operator directly. A use clause makes all of the resources in the named package available without prefixing. A use type clause is more specific; it allows us to use operators[1] of the given type without prefixing.

3.2 Utility Packages

A utility package groups together the constants, types, subtypes, and subprograms necessary to provide some particular service. The library package Ada.Numerics.Elementary_Functions is a utility package that includes twenty-nine mathematical functions such as square root, trigonometric functions, and logarithms for Float values. There is also a generic version of this mathematical package that may be instantiated for any floating point type. Here is the declaration of a utility package that provides three operations for control over output displayed on a screen:

```
package Display_Control is

   procedure Bold_On;
   -- Everything sent to the screen after this procedure
   -- is called will be displayed in bold characters

   procedure Blink_On;
   -- Everything sent to the screen after this procedure
   -- is called will be blinking

   procedure Normal;
   -- Everything sent to the screen after this procedure
   -- is called will be displayed normally

end Display_Control;
```

The implementation of these three operations depends on the display hardware. Having placed this dependency in a package body allows us to use the operations without knowledge of that hardware. Here is a body for this package with the implementation for a display that supports ANSI escape sequences. It includes one procedure body for each procedure declared in the package specification.

```
with Ada.Text_IO;
with Ada.Characters.Latin_1;   -- Characters in the
                               -- ISO 8859-1 character set
package body Display_Control is

-- Assumes that the display accepts and processes American
-- National Standards Institute (ANSI) escape sequences.

   -- Code to start an ANSI control string (the Escape
   -- control character and the left bracket character)
   ANSI_Start : constant String :=
                        Ada.Characters.Latin_1.ESC & '[';

   procedure Bold_On is
   begin -- "ESC[1m" turns on Bold
      Ada.Text_IO.Put (ANSI_Start & "1m");
      -- Send any buffered characters to the display
      Ada.Text_IO.Flush;
   end Bold_On;

   procedure Blink_On is
   begin -- "ESC[5m" turns on Blink
      Ada.Text_IO.Put (ANSI_Start & "5m");
      Ada.Text_IO.Flush;
   end Blink_On;

   procedure Normal is
   begin -- "ESC[0m" turns off all attributes
      Ada.Text_IO.Put (ANSI_Start & "0m");
      Ada.Text_IO.Flush;
   end Normal;

end Display_Control;
```

This package body uses resources from two library packages. Although not necessary in this package body, bodies may include additional subprograms.

These helper subprograms are local to the package body; they may not be called from outside.

3.3 Type Packages

We use the type package to create abstract data types (ADTs). An *abstract data type* consists of a set of data values and associated operations that are specified independent of any particular implementation. The abstract data type is a fundamental concept of Ada (Dale and McCormick, 2007; Barnes, 2014; Ben-Ari, 2009). Our example for an abstract data type is a bounded queue. Here is the package declaration for a bounded queue whose elements are integers:

```
package Bounded_Queue_V1 is
-- Version 1, details of the queue type are not hidden

   subtype Element_Type is Integer;

   type Queue_Array is array (Positive range <>) of Element_Type;
   type Queue_Type (Max_Size : Positive) is
      record
         Count : Natural;      -- Number of items
         Front : Positive;     -- Index of first item
         Rear  : Positive;     -- Index of last item
         Items : Queue_Array (1 .. Max_Size);  -- The element array
      end record;

   function Full (Queue : in Queue_Type) return Boolean;

   function Empty (Queue : in Queue_Type) return Boolean;

   function Size (Queue : in Queue_Type) return Natural;

   function First_Element (Queue : in Queue_Type) return Element_Type
      with
         Pre => not Empty (Queue);

   function Last_Element (Queue : in Queue_Type) return Element_Type
      with
         Pre => not Empty (Queue);

   procedure Clear (Queue : in out Queue_Type)
      with
         Post => Empty (Queue) and then Size (Queue) = 0;
```

```
   procedure Enqueue (Queue : in out Queue_Type;
                      Item  : in     Element_Type)
      with
         Pre  => not Full (Queue),
         Post => not Empty (Queue) and then
                 Size  (Queue) = Size (Queue'Old) + 1 and then
                 Last_Element (Queue) = Item;

   procedure Dequeue (Queue :    in out Queue_Type;
                      Item  :    out    Element_Type)
      with
         Pre  => not Empty (Queue),
         Post => Item = First_Element (Queue'Old) and then
                 Size  (Queue) = Size (Queue'Old) - 1;

end Bounded_Queue_V1;
```

This package defines a queue type as a record with five components (a discriminant and four fields) and eight queue operations. Five of the queue operations include contracts for preconditions and postconditions. We will get to those shortly. But first, let us look at a short program that uses the abstract queue type defined in the package specification.

```
with Bounded_Queue_V1; use Bounded_Queue_V1;
with Ada.Text_IO;      use Ada.Text_IO;
procedure Bounded_Queue_Example_V1 is
-- Uses the first version of the bounded queue package

   My_Queue : Bounded_Queue_V1.Queue_Type (Max_Size => 100);
   Value    : Integer;

begin
   Clear (My_Queue); -- Initialize queue
   for Count in Integer range 17 .. 52 loop
      Enqueue (Queue => My_Queue, Item => Count);
   end loop;
   for Count in Integer range 1 .. 5 loop
      Dequeue (Queue => My_Queue, Item => Value);
      Put_Line (Integer'Image (Value));
   end loop;
   Clear (My_Queue);
   Value := Size (My_Queue);
   Put_Line ("Size of cleared queue is " & Integer'Image (Value));
end Bounded_Queue_Example_V1;
```

The first declaration in this example program defines a bounded queue with a maximum size of 100. For clarity, we chose to prefix the type Queue_Type in our declaration of the variable My_Queue even though the use clause allows us to omit the package name. After the loop enqueues thirty-six values, the program dequeues five values and displays them. After clearing the queue, it displays the size of the queue.

All that remains is to complete the body of our queue package where we implement the queue operations defined in the package specification. Here is the body of our bounded queue package:

```
package body Bounded_Queue_V1 is

   function Full (Queue : in Queue_Type) return Boolean is
   begin
      return Queue.Count = Queue.Max_Size;
   end Full;

   function Empty (Queue : in Queue_Type) return Boolean is
   begin
      return Queue.Count = 0;
   end Empty;

   function Size (Queue : in Queue_Type) return Natural is
   begin
      return Queue.Count;
   end Size;

   function First_Element (Queue : in Queue_Type) return Element_Type is
   begin
      return Queue.Items (Queue.Front);
   end First_Element;

   function Last_Element (Queue : in Queue_Type) return Element_Type is
   begin
      return Queue.Items (Queue.Rear);
   end Last_Element;

   procedure Clear (Queue : in out Queue_Type) is
   begin
      Queue.Count := 0;
      Queue.Front := 1;
      Queue.Rear  := Queue.Max_Size;
   end Clear;
```

```
   procedure Enqueue (Queue : in out Queue_Type;
                      Item  : in     Element_Type) is
   begin
      Queue.Rear := Queue.Rear rem Queue.Max_Size + 1;
      Queue.Items (Queue.Rear) := Item;
      Queue.Count := Queue.Count + 1;
   end Enqueue;

   procedure Dequeue (Queue : in out Queue_Type;
                      Item  :    out Element_Type) is
   begin

      Item := Queue.Items (Queue.Front);
      Queue.Front := Queue.Front rem Queue.Max_Size + 1;
      Queue.Count := Queue.Count - 1;
   end Dequeue;

end Bounded_Queue_V1;
```

3.3.1 *Introduction to Contracts*

The contracts in our queue package specification (page 73) are given in the form of aspects. An *aspect* describes a property of an entity. Ada 2012 defines nearly seventy different aspects that we may use in our programs. In this chapter, we will look at a few of these standard aspects. An implementation of Ada may provide additional aspects. We will begin our look at SPARK specific aspects in Chapter 4.

The specification of a typical aspect consists of a name, an arrow (=>), and a definition. Take, for example, the postcondition aspect of procedure Clear:

```
   procedure Clear  (Queue : in out Queue_Type)
      with
         Post => Empty (Queue) and then Size (Queue) = 0;
```

The name Post indicates that this aspect is a postcondition for the procedure. This aspect requires a Boolean definition after the arrow. Our Boolean definition is an expression that includes calls to the queue functions Empty and Size. Postconditions are expected to be true after completion of the operation. In this case, after Clear completes its execution we expect that the queue will be empty and have a size of zero. Of course, there could be an error in the implementation of procedure Clear. We need to verify that Clear does indeed meet its postcondition.

Testing is the obvious way to verify this procedure. We could write a test program that enqueues items into a queue, clears the queue, and finally calls and displays the values returned by functions Empty and Size. Ada 2012 provides a quicker way of testing postconditions. By setting a compiler option, we can have the Ada compiler generate code to check contracts at runtime. Should any postcondition in the program not be true on completion of a subprogram call, the program will halt with a runtime error stating which postcondition was violated.

Executing the postcondition definition at the end of every subprogram call increases the running time of our programs. And, of course, just because our test runs never find a violation of a postcondition does not mean that the subprogram is correct for all possible executions. SPARK provides another approach to verifying postconditions. We can use the GNATprove tool to formally verify the postcondition without executing the code. We will discuss this static verification approach in Chapter 6.

Now let us look at the contract for procedure Enqueue.

```
procedure Enqueue (Queue : in out Queue_Type;
                   Item  : in     Element_Type)
   with
      Pre  => not Full (Queue),
      Post => not Empty (Queue) and then
              Size (Queue) = Size (Queue'Old) + 1 and then
              Last_Element (Queue) = Item;
```

The aspect name Pre indicates a precondition for a subprogram. This Boolean expression should be true each time the subprogram is called. As with postconditions, the Boolean expression may include function calls. In this example, the precondition tells us that the procedure Enqueue should not be called when the queue is full. As with postconditions, setting a compiler option will generate code to check the precondition each time the subprogram is called. Should a precondition in the program not be true for a subprogram call, the program will halt with a runtime error stating which precondition was violated. We may also use the GNATprove tool to statically verify that all subprogram calls in our program meet the given preconditions.

The postcondition for procedure Enqueue states that, after calling the procedure, the queue is not empty, its size has been increased by one, and the item is the last element of the queue. The logic of the size expression illustrates the use of the 'Old attribute to refer to the original value of an **in out** mode parameter. In our example, after calling procedure Enqueue, the size of the resulting queue (Queue) is one greater than the size of the queue before the call (Queue'Old).

As preconditions refer to the state of the parameters before the call, the 'Old attribute may not be used in them. The precondition **not** Full (Queue) refers to the queue that is passed in to be modified by the procedure.

We use the 'Result attribute to refer to the result of a function in its postcondition. Here, for example, is a function that returns the square root of a natural number:

```
function Sqrt (Item : in Natural) return Natural
   with
      Post => Sqrt'Result ** 2 <= Item and then
              (Sqrt'Result + 1) ** 2 > Item;
```

The postcondition states that the result of the function is the largest whole number whose square is less than or equal to the parameter Item.

3.3.2 *Information Hiding*

The details of the type Queue_Type defined in our queue package specification on page 73 are public. Programmers using this package may ignore the operations defined in the package and access the components of the record directly to manipulate a queue. They may set the fields defining a queue with inconsistent or invalid states. Should we later change the record defining the queue type, the parts of the program that accessed the original components would fail. This approach is at odds with the concept of information hiding. By enforcing information hiding we can eliminate the possibilities for inconsistency while ensuring that changes to the implementation details have no affect on other parts of the program.

Ada uses the *private type* for information hiding. Here is a second version of our queue package that uses a private type to protect the details that comprise our queue type:

```
package Bounded_Queue_V2 is
-- Version 2, details of the queue type are hidden

   subtype Element_Type is Integer;

   type Queue_Type (Max_Size : Positive) is private;

   function Full (Queue : in Queue_Type) return Boolean;

   function Empty (Queue : in Queue_Type) return Boolean;

   function Size (Queue : in Queue_Type) return Natural;
```

```
   function First_Element (Queue : in Queue_Type) return Element_Type
      with
         Pre => not Empty (Queue);

   function Last_Element (Queue : in Queue_Type) return Element_Type
      with
         Pre => not Empty (Queue);

   procedure Clear (Queue : in out Queue_Type)
      with
         Post => Empty (Queue) and then Size (Queue) = 0;

   procedure Enqueue (Queue : in out Queue_Type;
                      Item  : in     Element_Type)
      with
         Pre  => not Full (Queue),
         Post => not Empty (Queue) and then
                 Size (Queue) = Size (Queue'Old) + 1 and then
                 Last_Element (Queue) = Item;

   procedure Dequeue (Queue : in out Queue_Type;
                      Item  :    out Element_Type)
      with
        Pre  => not Empty (Queue),
        Post => Item = First_Element (Queue'Old) and then
                Size (Queue) = Size (Queue'Old) - 1;

private

   type Queue_Array is array (Positive range <>) of Element_Type;
   type Queue_Type (Max_Size : Positive) is
      record
         Count : Natural  := 0;    -- Number of items
         Front : Positive := 1;    -- Index of first item
         Rear  : Positive := Max_Size; -- Index of last item
         Items : Queue_Array (1 .. Max_Size); -- The element array
      end record;

end Bounded_Queue_V2;
```

The first change to notice in this version is that the definition of Queue_Type has been changed from a record type to **private**. An application programmer may use this type to declare queue variables. As with the public record

implementation in our first version, the discriminant Max_Size is used to give a maximum size to each queue object. The operations on a private type object available to a programmer are limited to those defined in the package specification (Full, Empty, Clear, Enqueue, and Dequeue in our example), assignment, and equality testing.

The second change is the division of the specification into two parts by the keyword **private** written just after the definition of procedure Dequeue. Everything above the word **private** may be freely used by an application programmer. This part of the package specification is called the *visible part*. Everything below the word **private** is hidden. This part of the package is called the *private part*. A client programmer using this package may see these details when they read the specification, but they may not reference them in their programs. In our example, the details of the array type used to store the elements of a queue and the record type that actually defines the queue type are *private*. Application programmers may not manipulate the fields of the queue record as they could with our first version. They must call the public operations in the package to manipulate a queue.

The package body for our hidden version that implements the operations is identical to that of the public version. A copy is available on http://www.cambridge.org/us/academic/subjects/computer-science/programming-languages-and-applied-logic/building-high-integrity-applications-spark. The sample application we gave on page 74 that made use of our queue type may be used with the hidden version simply by changing the imported package name.

3.3.3 *Generic Packages*

Both versions of our queue package define a queue type whose elements are integers. To create a type for a queue of characters, we could copy the integer queue package specification and body, change the type of the element from integer to character, and compile our new files. Ada's generic packages provide a simpler and safer approach. In Chapter 2 we used generic packages from the Ada library to create packages for the input and output of our own scalar types. Now we look at writing such a package. We use generic parameters to supply the information needed by the compiler to customize our package. With our queue, we need only supply the type of the element we wish to store in our queues. Table 2.2 lists the most commonly used generic formal types. As the queue package body makes use of element assignment and the contracts in the queue specification make use of element equality testing, the appropriate generic formal type is **private**. Here is the specification of a generic queue package that can be instantiated for any element type that has assignment and equality testing operations:

```
generic
   type Element_Type is private;
package Bounded_Queue is
-- Final version, generic with hidden details

   type Queue_Type (Max_Size : Positive) is private;

   function Full  (Queue : in Queue_Type) return Boolean;

   function Empty (Queue : in Queue_Type) return Boolean;

   function Size  (Queue : in Queue_Type) return Natural;

   function First_Element (Queue : in Queue_Type) return Element_Type
      with
         Pre => not Empty (Queue);

   function Last_Element (Queue : in Queue_Type) return Element_Type
      with
         Pre => not Empty (Queue);

   procedure Clear (Queue : in out Queue_Type)
      with
         Post => Empty (Queue) and then Size (Queue) = 0;

   procedure Enqueue (Queue : in out Queue_Type;
                      Item  : in     Element_Type)
      with
         Pre  => not Full (Queue),
         Post => not Empty (Queue) and then
                 Size  (Queue) = Size (Queue'Old) + 1 and then
                 Last_Element (Queue) = Item;

   procedure Dequeue (Queue : in out Queue_Type;
                      Item  :    out Element_Type)
      with
        Pre  => not Empty (Queue),
        Post => Item = First_Element (Queue'Old) and then
                Size  (Queue) = Size (Queue'Old) - 1;

private

   type Queue_Array is array (Positive range <>) of Element_Type;
```

```
   type Queue_Type (Max_Size : Positive) is
      record
         Count : Natural  := 0;    -- Number of items
         Front : Positive := 1;    -- Index of first item
         Rear  : Positive := Max_Size; -- Index of last item
         Items : Queue_Array (1 .. Max_Size); -- The element array
      end record;

end Bounded_Queue;
```

The reserved word **private** is used for two different purposes in this package. It is used in the definition of the generic formal parameter Element_Type to specify that the actual parameter can be any type that has assignment and equality testing. The generic package body may only use those operations with values of this type. The second use of **private** in our generic package specification is in the definition of Queue_Type. As before, this type restricts the operations that can be used in an application to those defined in the package. In both situations, private restricts access to details. In a generic formal parameter, it restricts the writer of the package body. In a type declaration, it restricts the application using the type. The third time the word **private** appears in this package (eleven lines up from the bottom on a line by itself) relates to the second usage. As we discussed in the previous section, the details of our private Queue_Type are given below this line. Everything above this line is public and accessible to other program units.

When we instantiate a queue package from this generic package, we supply an actual parameter that is the type of the desired queue component. Here is a version of our queue application revised to use the generic queue package to instantiate a queue type with character components:

```
with Bounded_Queue;
with Ada.Text_IO;    use Ada.Text_IO;
procedure Bounded_Queue_Example is
   -- Uses the generic version of the bounded queue package

   -- Instantiate a queue package with character elements
   package Char_Queue is new Bounded_Queue (Element_Type => Character);
   use Char_Queue;

   My_Queue : Char_Queue.Queue_Type (Max_Size => 100);
   Value    : Character;

begin
   Clear (My_Queue); -- Initialize queue
```

```
   for Char in Character range 'f' .. 'p' loop
      Enqueue (Queue => My_Queue, Item => Char);
   end loop;
   for Count in Integer range 1 .. 5 loop
      Dequeue (Queue => My_Queue, Item => Value);
      Put (Value);
      New_Line;
   end loop;
   Clear (My_Queue);
   Put_Line ("Size of cleared queue is " & Integer'Image (Size (My_Queue)));
end Bounded_Queue_Example;
```

3.4 Variable Packages

A variable package is used to encapsulate a single object. This concept is sometimes called a *singleton* or *singleton class*. As an example, we look at some packages that might be used in a simulation of the game of Bingo. Players of this game manage a number of different Bingo cards. As there are many cards involved, we would use a type package with appropriate operations to model them. When the game is played, numbers are randomly drawn from a single source. We can use a variable package to model this source. We begin with a definition package that describes the numbers used in Bingo.

```
package Bingo_Numbers is

-- This package defines BINGO numbers and their associated letters

   -- The range of numbers on a Bingo Card
   type Bingo_Number is range 0 .. 75;

   -- 0 can't be called, it is only for the Free Play square
   subtype Callable_Number is Bingo_Number range 1 .. 75;

   -- Associations between Bingo numbers and letters
   subtype B_Range is Bingo_Number range 1 .. 15;
   subtype I_Range is Bingo_Number range 16 .. 30;
   subtype N_Range is Bingo_Number range 31 .. 45;
   subtype G_Range is Bingo_Number range 46 .. 60;
   subtype O_Range is Bingo_Number range 61 .. 75;

   -- The 5 Bingo letters
   type Bingo_Letter is (B, I, N, G, O);

end Bingo_Numbers;
```

Here is the specification of a variable package that models the basket from which Bingo numbers are drawn:

```
with Bingo_Numbers; use Bingo_Numbers;
package Bingo_Basket is

   function Empty return Boolean;

   procedure Load   -- Load all the Bingo numbers into the basket
      with
         Post => not Empty;

   procedure Draw (Letter : out Bingo_Letter;
                   Number : out Callable_Number)
   -- Draw a random number from the basket
      with
         Pre => not Empty;

end Bingo_Basket;
```

Notice that there is no type for a Bingo basket and no basket parameter for any of the operations. The package body hides a single basket. All three operations act on this hidden basket object. Here is that body:

```
with Ada.Numerics.Discrete_Random;
package body Bingo_Basket is

   type Number_Array is array (Callable_Number) of Callable_Number;
   The_Basket : Number_Array; -- A sequence of numbers in the basket
   The_Count : Bingo_Number; -- The count of numbers in the basket

   package Random_Bingo is new Ada.Numerics.Discrete_Random
                                  (Result_Subtype => Callable_Number);
   use Random_Bingo;
   -- The following object holds the state of a random Bingo number generator
   Bingo_Gen : Random_Bingo.Generator;

   procedure Swap (X : in out Callable_Number;
                   Y : in out Callable_Number) is
      Temp : Callable_Number;
   begin
      Temp := X; X := Y;  Y := Temp;
   end Swap;
```

```
   function Empty return Boolean is (The_Count = 0);
   -- Example of an expression function

   procedure Load is
      Random_Index : Callable_Number;
   begin
      -- Put all numbers into the basket (in order)
      for Number in Callable_Number loop
         The_Basket (Number) := Number;
      end loop;
      -- Randomize the array of numbers
      Reset (Bingo_Gen);  -- Seed random generator from clock
      for Index in Callable_Number loop
         Random_Index := Random (Bingo_Gen);
         Swap (X => The_Basket (Index),
               Y => The_Basket (Random_Index));
      end loop;
      The_Count := Callable_Number'Last;  -- all numbers now in the basket
   end Load;

   procedure Draw (Letter : out Bingo_Letter;
                   Number : out Callable_Number) is
   begin
      Number := The_Basket (The_Count);
      The_Count := The_Count - 1;

      -- Determine the letter using the subtypes in Bingo_Definitions
      case Number is
         when B_Range => Letter := B;
         when I_Range => Letter := I;
         when N_Range => Letter := N;
         when G_Range => Letter := G;
         when O_Range => Letter := O;
      end case;
   end Draw;
end Bingo_Basket;
```

The variable The_Basket, an array containing all of the Bingo numbers, is global to all of the operations in this package body. The global variable The_Count keeps track of how many numbers in this array have not yet been drawn. Function Empty returns True when The_Count is zero. Together, these two variables maintain the state of the bingo basket. The variable Bingo_Gen is a random Bingo number generator used globally by the Load operation. Procedure

Swap was not defined in the package specification. It is not a basket operation but a local subprogram called by procedure Load.

3.4.1 *Package Initialization*

A program using our Bingo_Basket package should call the Load operation to initialize the global package variable The_Basket before drawing numbers from it. This explicit initialization is appropriate for this particular variable package as we will likely use this package to play multiple Bingo games. However, some situations require that initialization occur only once. For example, consider the following package that implements a serial number generator:

```
package Serial_Numbers is
   type Serial_Number is range 1000 .. Integer'Last;
   procedure Get_Next (Number : out Serial_Number);
end Serial_Numbers;
```

Calls to procedure Get_Next return the next available serial number. Because we do not want to repeat numbers, we have not included an operation to initialize the sequence. Instead, we set the initial serial number in the package body:

```
package body Serial_Numbers is

   Next_Number : Serial_Number := Serial_Number'First;

   procedure Get_Next (Number : out Serial_Number) is
   begin
      Number := Next_Number;
      Next_Number := Next_Number + 1;
   end Get_Next;

end Serial_Numbers;
```

The declaration of the global package variable Next_Number includes an assignment of an initial value. This value is assigned to the variable during elaboration of this package body. *Elaboration* is the runtime processing of declarations. Elaboration brings the item being declared into existence and then, if the declaration includes an initial value, assigns that value to the item.

When a variable package encapsulates a nontrivial data structure, initializing that data structure may require more than can be accomplished with assignments of initial values to its variables. We may, for example, need a loop to create an initial linked list. Package bodies may include an optional sequence of statements that are executed when the body is elaborated. This initialization

code is placed between a **begin** and the **end** of the package. Here is how we might use package initialization code to give Next_Number its initial value:

```
package body Serial_Numbers2 is

   Next_Number : Serial_Number;

   procedure Get_Next (Number : out Serial_Number) is
   begin
      Number := Next_Number;
      Next_Number := Next_Number + 1;
   end Get_Next;

begin -- package initialization code
   Next_Number := Serial_Number'First;
end Serial_Numbers2;
```

There are no restrictions on what may be included in package initialization code. We can call subprograms defined there or in any package. It is, however, wise to keep this initialization code simple to minimize the possibility of an exception being raised. Although package initialization code is primarily used to initialize the state of a variable package (by initializing its global variables), there are no rules against using it in any package that has a body.

When does elaboration occur? Each time we call a subprogram, all of its local declarations are elaborated prior to the execution of the code after the word **begin**. In this case, storage for the local variables is allocated on the stack and any initial values are assigned.

Each package specification and body is elaborated once after the program is loaded. The order in which packages are elaborated must follow the basic rule that a unit must be elaborated before another unit can use a resource within it. For most projects, the Ada compiler can determine a legal elaboration order. However, using packages with mutual dependencies may make it difficult or impossible for the compiler to determine an elaboration order. Ada provides pragmas that we may use to give the compiler hints on a correct order. In some cases, however, it may be necessary to remove mutual dependencies by restructuring the design of the program. Section 3.6 provides additional discussion and examples of elaboration.

3.5 Child Packages

Ada provides a hierarchical naming scheme for library units. A package named Apple may have a child package with the name Apple.McIntosh. Apple is the

parent of McIntosh. This naming scheme overcomes problems with uniqueness of names just as a hierarchical file system allows us to distinguish between two files with the same name by keeping them in two different directories. We have seen examples of this hierarchy of package names in the Ada library. Ada is the parent package of all predefined units in the library. Text_IO is a child of Ada containing resources for the input and output of text.

In addition to providing a hierarchy for organizing names, child units provide important information hiding properties. We have seen how private types are used to hide the details of an abstract data type. In our bounded queue type package examples, the record and array defining the implementation are not accessible to program units that use our queue type. These clients must use the operations, such as Enqueue, that we defined in the public portion of the package specification to manipulate queue objects. However, the details of the private type are available in the package body where the operations are implemented. These private details are also available to the private part and body of a child package. This access allows a group of units to share private information while keeping that information hidden from external clients.[2]

This sharing allows us to easily extend an abstract data type. In fact, child packages are the key construct for object-oriented programming features such as inheritance and dynamic dispatching that are beyond the scope of this book. Let us look at an example. Here is the specification of a simple stack type package:

```
package Stacks is
-- Implements a simple stack of integers (with no safety features)
   type Stack_Type (Max_Size : Positive) is private;

   procedure Clear (Stack : out Stack_Type);
   function Empty (Stack : in Stack_Type) return Boolean;
   procedure Push (Item  : in     Integer;
                   Stack : in out Stack_Type);
   procedure Pop (Item  :    out Integer;
                  Stack : in out Stack_Type);
private
   type Stack_Array is array (Positive range <>) of Integer;
   type Stack_Type (Max_Size : Positive) is
      record
         Top   : Natural := 0;  -- Initialize all stacks to empty
         Items : Stack_Array (1 .. Max_Size);
      end record;
end Stacks;
```

And here is a child package specification with a new operation for our stack type:

```
package Stacks.More is
   function Peek (Stack : in Stack_Type) return Integer;
   -- Returns a copy of the top element on the Stack
end Stacks.More;
```

We certainly could have added function Peek to our original stack package. However, that change to the original package would require us to recompile all clients that used it. And, of course, we wanted to illustrate a use of child packages. To that end, here is the body of our child package:

```
package body Stacks.More is
   function Peek (Stack : in Stack_Type) return Integer is
   begin
      return Stack.Items (Stack.Top);
   end Peek;
end Stacks.More;
```

You can see that this body makes use of the details of the stack type given in the private part of the stack package specification.

3.5.1 *Private Children*

Package Stacks.More is an example of a *public child package*. Public child packages allow extension and continued privacy of their private types to provide additional resources for clients.

When developing a subsystem, there are times that we would like to decompose it into pieces without giving clients of that subsystem direct access to those pieces. Ada's *private child package* provides the mechanism for hiding these pieces. The use of resources from a private child package is restricted to the hierarchy of packages rooted at its parent.

To illustrate this role of private child packages, consider a flight management system (FMS). An FMS is a component of the cockpit software that automates a wide variety of in-flight tasks. A primary function of an FMS is in-flight management of the flight plan. Here is an outline of a package that supplies clients with operations on flight plans:

```
package FMS is
   type Flight_Plan is private;
   . . .
   -- Many operations on flight plans
   . . .
```

```
private
   -- Hidden details of the flight plan type
   type Flight_Plan is ....
end FMS;
```

Navigational databases and position determination are two of the many components making up an FMS. The client of our flight management subsystem has no need to access these two components directly. They are accessed by operations in the FMS that are called by the clients. Therefore, we place these two subsystems into private child packages. Here are the outlines of these two private children:

```
private package FMS.Navigation_Database is
   . . .
end FMS.Navigation_Database;

private package FMS.Positioning is
   . . .
end FMS.Positioning;
```

The body of package FMS will with these private packages to use their resources. Client units cannot with these packages.

Let us look at one more level of this FMS design. Position information is obtained through multiple subsystems including the global positioning system, inertial reference systems (IRS), and VHF omnidirectional radio range (VOR). As these three subsystems are components of the positioning system, we make them private child packages of that system.

```
private package FMS.Positioning.GPS is
   . . .
end FMS.Positioning.GPS;

private package FMS.Positioning.IRS is
   . . .
end FMS.Positioning.IRS;

private package FMS.Positioning.VOR is
   . . .
end FMS.Positioning.VOR;
```

In this book we make use of private child packages in the development of hierarchical state abstractions (Sections 4.3.3 and 7.3.3), isolating non-SPARK code from legacy Ada software (Section 8.2), and partitioning unproved SPARK code (Section 9.3.5).

3.5.2 *Visibility and the Child Hierarchy*

Each of the packages in our hierarchy may have both visible and private parts and a body. These parts and bodies may access parts of other packages in the hierarchy by default or by withing them. Of course, we can with any public unit outside the hierarchy. Here are the visibility rules:

- A child specification never needs to *with* its parent; a specification may *with* a sibling except that a public child specification may not *with* a private sibling; a specification may not *with* its own child.
- A body never needs to *with* its parent.
- The entities of a parent are accessible by simple name within its descendants (children, grandchildren, etc.); **use** clauses are not required.
- A **with** clause given in the specification of a parent also applies to its body and its descendants.
- A private child is never visible outside the tree rooted at its parent. And within that tree, it is not visible to the visible parts of public siblings.
- The private part and body of any child can access the private parts of its ancestors (parent, grandparent, etc.).
- The visible part of a private child can access the private parts of its ancestors.
- A **with** clause for a child automatically implies **with** clauses for all its ancestors.
- A **use** clause for a unit makes the resources in its descendants accessible by simple name. Those descendants must be *withed*.
- A **private with** clause allows the private part but not the visible part of a package to access resources in the named package. A public child may *private with* a private sibling.

Figure 3.1 illustrates the direct visibility between children and parent packages. This visibility is automatic; it does not require any **with** or **use** context clauses. The arrows show that every private part of a specification has direct visibility of the visible part of that specification. Every body has access to everything in its specification and the specification of its parent. Both the visible part and the private part of a private child have direct access to everything in their parent's specification. The access of a public child is more limited. The visible part of a public child can only access resources in its parent's visible part. However, the private part of a public child has direct access to everything in its parent' specification.

Figure 3.2 shows options for obtaining access to other related units via **with** clauses placed at the beginning of a package specification or body. In all cases, a **with** clause provides access to only the resources in the visible part of

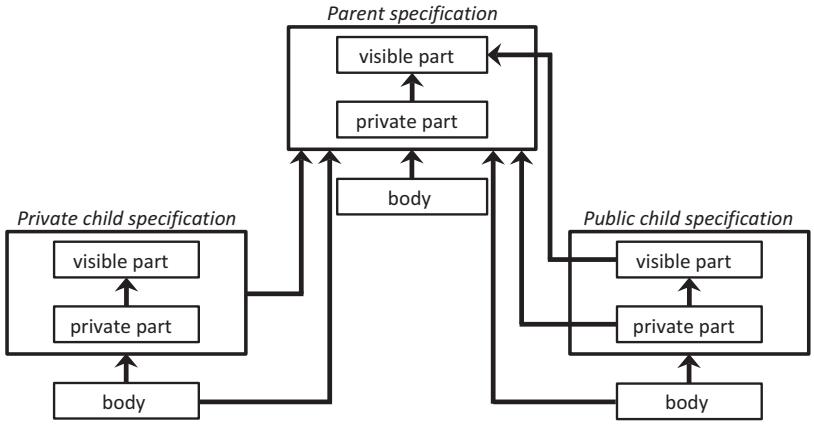

Figure 3.1. Direct visibility with child packages.

the package being withed. The body of a parent package may with any of its descendants. The body of any descendant in the hierarchy, whether private or public, can with any other package in the hierarchy. But as shown in Figure 3.1, no with is needed to access ancestor packages. Whereas the specification of a private child package can with any package in the hierarchy, the specification of a public child may only with other public children.

There are two special forms of the **with** clause. Including a **private with** clause at the beginning of a package's specification allows access to the named package from only the private part of that specification. The dotted arrow in Figure 3.2 representing a **private with** clause shows that resources in a private

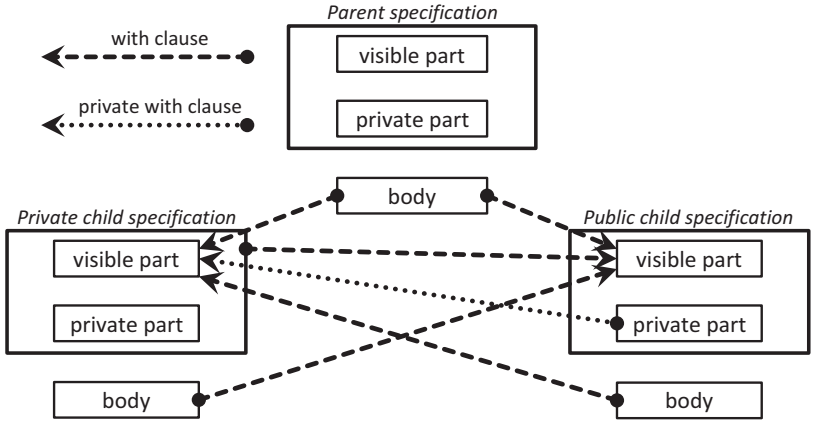

Figure 3.2. Visibility via **with** clauses in child packages.

child's visible part may be used in the private part of a public child. In later chapters, we make use of **private with** clauses to exempt private parts from our SPARK analyses. A **limited with** provides a mechanism for mutually dependent units. We have not used any **limited with** clauses in this book.

Both public and private children can themselves have children of both kinds. A private child of a public child begins a new hierarchy of visibility beginning at its public parent. A private child of a private child also begins a new hierarchy of visibility beginning at its private parent. A public child of a private child extends the specification of the private child. This public child is visible to the bodies of its public uncles (packages sharing the same grandparent as the new public child) and to all parts of its private uncles.

Child packages provide a very flexible mechanism for implementing complex architectures. As usual, it is important to keep the relations between modules as understandable as possible.

3.6 Elaboration

We conclude this chapter with another look at elaboration, a concept introduced in Section 3.4.1. Elaboration is the runtime processing of a declaration, declarative part, or program unit body. This processing occurs during the execution of a program and consists of activities such as allocating space and providing initial values to objects. As shown in Section 3.4.1, the elaboration of a package body may include the execution of the sequence of statements at the end of the body.

Here is a simple example of elaboration in a procedure:

```
procedure Elaboration_Demo (Size  : in  Positive;
                            Count : out Natural) is
 Line  : String (1 .. Size) := (others => ' ');
 Guess : Float := (1.0 + Ada.Numerics.Elementary_Functions.Sqrt (5.0)) / 2.0;
begin
   . . . .
end Elaboration_Demo;
```

When procedure Elaboration_Demo is called, its declarations are elaborated before the execution of the statements between its **begin** and **end**. Space for the array variable Line with Size components is allocated on the system stack and all its components are initialized to blank. Similarly, space is allocated for Guess and it is initialized to $\frac{1+\sqrt{5}}{2}$, the Golden Ratio.

Subprogram calls are possible during elaboration as demonstrated by the call to the square root function in the initialization of Guess. Such calls allow any arbitrary part of the program to be executed as part of elaboration.

Elaboration can be more complicated with packages as the order in which packages are elaborated is important. Take, for example, the following two skeleton package specifications and bodies:

```
with Pack_B;
package Pack_A is
   Var_1 : Integer := 2 * Pack_B.Var_3 / 3;
   . . . .
end Pack_A;
----------------
package body Pack_A is
   Var_2 : Integer := 5 * Pack_B.Var_3 / 6;
   . . . .
end Pack_A;

-----------------------------------
package Pack_B is
   Var_3 : Integer  := 17;
      . . . .
end Pack_B;
----------------
with Pack_A;
package body Pack_B is
   Var_4 : Integer := 3 * Pack_A.Var_1 / 4;
      . . . .
end Pack_B;
```

During the elaboration of the specification of Pack_A, the value of Pack_B.Var_3 is used in the initialization of Var_1. So it is important that the elaboration of the specification of Pack_B be completed before the elaboration of the specification of Pack_A. Similarly, the elaboration of the specification of Pack_B must be completed before the elaboration of the body of Pack_A. Finally, the body of Pack_B uses the value of Pack_A.Var_1 in the initialization of Var_4. So the specification of Pack_A must be elaborated before the body of Pack_B.

The determination of legal orders of elaboration in a program with many packages is a problem that must be solved prior to linking the object code of the packages into an executable file.[3] The GNAT Ada compiler uses a tool called GNATbind that, among other tasks, checks that an acceptable order of elaboration exists for the program and generates a main program incorporating a valid elaboration order.

A program with circular elaboration dependencies has no valid order of elaboration. In such cases, GNATbind issues an error message. It is then up to the programmer to reorganize the packages into a set that does have a valid order of elaboration or provide the binder with additional information so it can find a valid order. Decomposing a package into child packages is one way to remove circular elaboration dependencies.

Appendix C of the *GNAT User's Guide* (GNAT, 2015b) provides an excellent discussion of the elaboration process, elaboration problems, and solutions. These solutions include a number of pragmas that a programmer may use to give additional information to the binder that might allow it to determine a valid order that it could not determine on its own.

Summary

- A library unit is a separately compiled program unit.
- In Ada, library units may be subprograms, packages, or generic units.
- The with clause provides access to the public declarations in a library unit.
- The use clause provides direct visibility of the public declarations in a library unit so we do not have to prefix them with the library unit name.
- Packages provide the logical structure to large software applications.
- Packages are the primary means in Ada for abstraction, encapsulation, and information hiding.
- A package consists of a specification and a body that implements the specification.
- We define a definition package as a package that groups together related constants and types.
- As there are no operations to implement, a definition package has no body.
- We define a utility package as a package that groups together related constants, types, and subprograms that provide some service.
- The type package is used to create abstract data types.
- Private types are used to encapsulate the details of an abstract data type.
- We use aspects to specify preconditions and postconditions for operations in a package.
- We may use an optional compiler switch to generate code to check each precondition and postcondition when our program executes.
- Generic packages are templates that may be instantiated for a specific purpose.
- Generic formal parameters provide the mechanism for customizing generic packages.
- A variable package is used to create singletons – hidden single objects.

- A package may contain initialization code that is executed when the package is elaborated.
- Initialization code is commonly used to initialize the state of a variable package that encapsulates a nontrivial data structure.
- Public and private child packages allow us to decompose subsystems in a structured manner.
- Public children enable the decomposition of the view of a subsystem to the user of the subsystem.
- Private children enable the decomposition of the implementation of a subsystem.
- Private with clauses allow access to the named package's visible part from only the private part of that specification.
- Elaboration is the runtime processing of a declaration, declarative part, or program unit body.

Exercises

3.1 What are the purposes of the *with clause* and *use clause*?

3.2 Define the following terms (some require knowledge from sources outside of this book).

a. Encapsulation
b. Information hiding
c. Definition package
d. Utility package
e. Abstract data type
f. Type package
g. Aspect
h. Attribute
i. Precondition
j. Postcondition
k. Variable package
l. Elaboration

3.3 Look at the specification of the package Ada.Text_IO in section A.10.1 of the Ada Reference Manual. Explain how this package could be classified as a utility package. Explain how this package could be classified as a type package.

3.4 Obtain a copy of the specification and body of the Bounded_Queue_V2 from the http://www.cambridge.org/us/academic/subjects/computer-science/programming-languages-and-applied-logic/building-high-integrity-applications-spark. Also obtain a copy of the sample application Bounded_Queue_Example_V1.

a. Make the necessary changes in names so the application will use this second version of the queue package. Build and run the application.
b. Corrupt the implementation of the dequeue procedure by incrementing the length of the queue rather than decrementing it. Build and run

the application. Was the error detected or were the results simply incorrect?

c. Look up the appropriate switch for the compile command that adds code to check the contracts at runtime. Build and run the application with this switch. Was the error detected or were the results simply incorrect?

3.5 Ada uses the reserved word **private** to restrict access to details. Who has complete access (application programmer or package programmer) to the details of a private type? Who has complete access to the details of a private generic parameter?

3.6 Using version 2 of our queue package as a guide, write a type package for a stack whose components are characters. Include appropriate preconditions and postconditions. Write, build, and run a simple test program that uses your stack package.

3.7 Convert the character stack package you wrote in Exercise 3.6 into a generic package that may be instantiated for any component type that supports assignment. Modify your test program, build, and run it.

3.8 Why can we not use the 'Old attribute in preconditions?

3.9 Why does the square root function, Sqrt, defined on page 78 not require a precondition stating that Item is not negative?

3.10 Write the specification of a generic type package for a mathematical set with operations Union, Intersection, Is_Member, Add_Value, Remove_Value, and Make_Empty. Encapsulate the set in a private type. Use an array of Booleans indexed by the set component type to implement the set type. Select an appropriate generic formal parameter type for the set component type. (Hint: What restrictions must we place on the kind of elements we might use in this simple array implementation?) Check the syntax of your package specification.

3.11 Write the body of the set package from the Exercise 3.10. A set is made empty by setting all components in the array to False. An element is added to or removed from a set by setting its component to True or False. The Union operation may be implemented by **or**'ing the corresponding Boolean components in the two arrays to create a resulting array of Booleans. Similarly, the Intersection may be implemented by **and**'ing the corresponding components.

3.12 Write a simple application that uses the generic set package developed in Exercises 3.10 and 3.11. Instantiate a set package whose components are the twenty-six uppercase letters of the alphabet. Write a procedure

that displays the letters in a set. (Hint: Write a loop that goes through all twenty-six letters and displays those that are members of the set.) Include code to construct sets and perform operations on them.

3.13 What is *elaboration?* When does elaboration occur for a subprogram?

3.14 Procedure Load in the package body of Bingo_Basket resets the random number generator each time it is called. We need only reset the generator once. Use package initialization to make this change to the body of Bingo_Basket. Also set the initial basket to empty.

3.15 What is the major use of public children in the decomposition of a system? Of private children?

3.16 What restriction does a private with clause impose that is not imposed by a with clause?