Back to sections 8.0 through 8.3.4

8.3.5 Using Generic Units for Data Abstraction

guideline

- Consider using abstract data types (not to be confused with Ada's abstract types) in preference to abstract data objects.
- Consider using generic units to implement abstract data types independently of their component data type.

example

This example presents a series of different techniques that can be used to generate abstract data types and objects. A discussion of the merits of each follows in the rationale section below. The first is an abstract data object (ADO), which can be used to encapsulate an abstract state machine. It encapsulates one stack of integers:

```
------------------------------------------------------------------------
package Bounded_Stack is
   subtype Element is Integer;
   Maximum_Stack_Size : constant := 100;
   procedure Push (New_Element : in     Element);
   procedure Pop  (Top_Element :    out Element);
   Overflow  : exception;
   Underflow : exception;
   ...
end Bounded_Stack;
------------------------------------------------------------------------
```

The second example is an abstract data type (ADT). It differs from the ADO by exporting the `Stack` type, which allows the user to declare any number of stacks of integers. Because multiple stacks may now exist, it is necessary to specify a `Stack` argument on calls to `Push` and `Pop`:

```
------------------------------------------------------------------------
package Bounded_Stack is
   subtype Element is Integer;
   type    Stack   is limited private;
   Maximum_Stack_Size : constant := 100;
   procedure Push (On_Top      : in out Stack;
                   New_Element : in     Element);
   procedure Pop  (From_Top    : in out Stack;
                   Top_Element :    out Element);
   Overflow  : exception;
   Underflow : exception;
   ...
private
   type Stack_Information;
   type Stack is access Stack_Information;
end Bounded_Stack;
------------------------------------------------------------------------
```

The third example is a parameterless generic abstract data object (GADO). It differs from the ADO (the first example) simply by being generic, so that the user can instantiate it multiple times to obtain multiple stacks of integers:

```
------------------------------------------------------------------------
generic
```

```
package Bounded_Stack is
   subtype Element is Integer;
   Maximum_Stack_Size : constant := 100;
   procedure Push (New_Element : in     Element);
   procedure Pop  (Top_Element :    out Element);
   Overflow  : exception;
   Underflow : exception;
   ...
end Bounded_Stack;
------------------------------------------------------------------------
```

The fourth example is a slight variant on the third, still a GADO but with parameters. It differs from the third example by making the data type of the stack a generic parameter so that stacks of data types other than `Integer` can be created. Also, `Maximum_Stack_Size` has been made a generic parameter that defaults to 100 but can be specified by the user, rather than a constant defined by the package:

```
------------------------------------------------------------------------
generic
   type Element is private;
   Maximum_Stack_Size : in Natural := 100;
package Bounded_Stack is
   procedure Push (New_Element : in     Element);
   procedure Pop  (Top_Element :    out Element);
   Overflow  : exception;
   Underflow : exception;
   ...
end Bounded_Stack;
------------------------------------------------------------------------
```

The fifth example is a generic abstract data type (GADT). It differs from the GADO in the fourth example in the same way that the ADT in the second example differed from the ADO in the first example; it exports the `Stack` type, which allows the user to declare any number of stacks:

```
------------------------------------------------------------------------
generic
   type Element is private;
   Maximum_Stack_Size : in Natural := 100;
package Bounded_Stack is
   type Stack is private;
   procedure Push (On_Top      : in out Stack;
                   New_Element : in     Element);
   procedure Pop  (From_Top    : in out Stack;
                   Top_Element :    out Element);
   Overflow  : exception;
   Underflow : exception;
   ...
private
   type Stack_Information;
   type Stack is access Stack_Information;
end Bounded_Stack;
------------------------------------------------------------------------
```

rationale

The biggest advantage of an ADT over an ADO (or a GADT over a GADO) is that the user of the package can declare as many objects as desired with an ADT. These objects can be declared as standalone variables or as components of arrays and records. They can also be passed as parameters. None of this is possible with an ADO, where the single data object is encapsulated inside of the package. Furthermore, an ADO provides no more protection of the data structure than an ADT. When a private type is exported by the ADT package, as in the example above, then for both the ADO and ADT, the only legal operations that can modify the data are those defined explicitly by the package (in this case, `Push` and `Pop`). For these reasons, an ADT or GADT is almost always preferable to an ADO or GADO, respectively.

A GADO is similar to an ADT in one way: it allows multiple objects to be created by the user. With an ADT, multiple objects can be declared using the type defined by the ADT package. With a GADO (even a GADO with no generic formal parameters, as shown in the third example), the package can be instantiated multiple times to produce multiple objects. However, the similarity ends there. The multiple objects produced by the instantiations suffer from all restrictions described above for ADOs; they cannot be used in arrays or records or passed as parameters. Furthermore, the objects are each of a different type, and no operations are defined to operate on more than one of them at a time. For example, there cannot be an operation to compare two such objects or to assign one to another. The multiple objects declared using the type defined by an ADT package suffer from no such restrictions; they can be used in arrays and records and can be passed as parameters. Also, they are all declared to be of the same type, so that it is possible for the ADT package to provide operations to assign, compare, copy, etc. For these reasons, an ADT is almost always preferable to a parameterless GADO.

The biggest advantage of a GADT or GADO over an ADT or ADO, respectively, is that the GADT and GADO are generic and can thus be parameterized with types, subprograms, and other configuration information. Thus, as shown above, a single generic package can support bounded stacks of any data type and any stack size, while the ADT and ADO above are restricted to stacks of `Integer`, no more than 100 in size. For this reason, a GADO or GADT is almost always preferable to an ADO or ADT.

The list of examples above is given in order of increasing power and flexibility, starting with an ADO and ending with a GADT. These advantages are not expensive in terms of complexity or development time. The specification of the GADT above is not significantly harder to write or understand than the specification of the ADO. The bodies are also nearly identical.

Compare the body for the simplest version, the ADO:

```
------------------------------------------------------------------------
package body Bounded_Stack is
   type Stack_Slots is array (Natural range <>) of Element;
   type Stack_Information is
      record
         Slots : Stack_Slots (1 .. Maximum_Stack_Size);
         Index : Natural := 0;
      end record;
   Stack : Stack_Information;
   ---------------------------------------------------------------------
   procedure Push (New_Element : in     Element) is
   begin
      if Stack.Index >= Maximum_Stack_Size then
         raise Overflow;
      end if;
      Stack.Index := Stack.Index + 1;
      Stack.Slots(Stack.Index) := New_Element;
   end Push;
   ---------------------------------------------------------------------
   procedure Pop (Top_Element :    out Element) is
   begin
      if Stack.Index <= 0 then
         raise Underflow;
      end if;
      Top_Element := Stack.Slots(Stack.Index);
      Stack.Index := Stack.Index - 1;
   end Pop;
   ---------------------------------------------------------------------
   ...
end Bounded_Stack;
------------------------------------------------------------------------
```

with the body for the most powerful and flexible version, the GADT:

```
------------------------------------------------------------------------
package body Bounded_Stack is
   type Stack_Slots is array (Natural range <>) of Element;
   type Stack_Information is
      record
         Slots : Stack_Slots (1 .. Maximum_Stack_Size);
         Index : Natural := 0;
      end record;
   ---------------------------------------------------------------------
   procedure Push (On_Top      : in out Stack;
                   New_Element : in     Element) is
   begin
      if On_Top.Index >= Maximum_Stack_Size then
         raise Overflow;
      end if;
      On_Top.Index := On_Top.Index + 1;
      On_Top.Slots(On_Top.Index) := New_Element;
   end Push;
   ---------------------------------------------------------------------
   procedure Pop (From_Top    : in out Stack;
                  Top_Element :    out Element) is
   begin
      if From_Top.Index <= 0 then
         raise Underflow;
      end if;
      Top_Element := From_Top.Slots(From_Top.Index);

      From_Top.Index := From_Top.Index - 1;
   end Pop;
   ---------------------------------------------------------------------
   ...
end Bounded_Stack;
------------------------------------------------------------------------
```

There is only one difference. The ADO declares a local object called `Stack`, while the GADT has one additional parameter (called `Stack`) on each of the exported procedures `Push` and `Pop`.

8.3.6 Iterators

guideline

- Provide iterators for traversing complex data structures within reusable parts.
- Consider providing both active and passive iterators.
- Protect the iterators from errors due to modification of the data structure during iteration.
- Document the behavior of the iterators when the data structure is modified during traversal.

example

Ada provides several mechanisms for building reusable iterators. The following examples discuss the alternatives of "simple" generics, access discriminants, and type extension. The terms active and passive are used to differentiate whether the iteration mechanism (i.e., the way in which the complex data structure is traversed) is exposed or hidden. A passive iterator hides the traversal (e.g., looping mechanism) and consists of a single operation, iterate, that is parameterized by the processing you do on each element of the data structure. By contrast, an active iterator exposes the primitive operations by which you traverse the data structure (Booch 1987).

The first example shows a generic package that defines an abstract list data type, with both active and passive iterators for traversing a list:

```
------------------------------------------------------------------------
generic
   type Element is limited private;
   ...
package Unbounded_List is
   type List is limited private;
   procedure Insert (New_Element : in     Element;
                     Into        : in out List);
   -- Passive (generic) iterator.
   generic
      with procedure Process (Each : in out Element);
   procedure Iterate (Over : in     List);
   -- Active iterator
   type Iterator is limited private;

   procedure Initialize (Index         : in out Iterator;
                         Existing_List : in     List);

   function  More       (Index         : in     Iterator)
     return Boolean;

   -- The procedure Get_Next combines an "Advance" and "Current" function
   procedure Get_Next   (Index           : in out Iterator;
                         Current_Element :    out Element);
   ...
private
   ...
end Unbounded_List;
------------------------------------------------------------------------
```

After instantiating the generic package and declaring a list as:

```
------------------------------------------------------------------------
with Unbounded_List;
procedure List_User is
   type Employee is ...;
   package Roster is
      new Unbounded_List (Element => Employee, ...);
   Employee_List : Roster.List;
```

the passive iterator is instantiated, specifying the name of the routine that should be called for each list element when the iterator is called.

```
   ---------------------------------------------------------------------
   procedure Process_Employee (Each : in out Employee) is
   begin
      ...
      -- Perform the required action for EMPLOYEE here.
   end Process_Employee;
   ---------------------------------------------------------------------
   procedure Process_All is
      new Roster.Iterate (Process => Process_Employee);
```

The passive iterator can then be called as:

```
begin  -- List_User
   Process_All (Employee_List);
end List_User;
------------------------------------------------------------------------
```

Alternatively, the active iterator can be used without the second instantiation required by the passive iterator:

```
   Iterator         : Roster.Iterator;
   Current_Employee : Employee;
   procedure Process_Employee (Each : in     Employee) is separate;
begin  -- List_User
   Roster.Initialize (Index         => Iterator,
                      Existing_List => Employee_List);

   while Roster.More (Iterator) loop

      Roster.Get_Next (Index           => Iterator,
                       Current_Element => Current_Employee);

      Process_Employee (Current_Employee);
   end loop;
end List_User;
------------------------------------------------------------------------
```

The second example shows a code excerpt from Rationale (1995, §3.7.1) on how to construct iterators using access discriminants:

```
generic
   type Element is private;
package Sets is
   type Set is limited private;
   ... -- various set operations
   type Iterator (S : access Set) is limited private;
   procedure Start (I : Iterator);
   function Done (I : Iterator) return Boolean;
   procedure Next (I : in out Iterator);
   ...  -- other iterator operations
private
   type Node;
   type Ptr is access Node;
   type Node is
      record
         E    : Element;
         Next : Ptr;
      end record;
   type Set is new Ptr;
   type Iterator (S : access Set) is
      record
         This : Ptr;
      end record;
end Sets;
package body Sets is
   ...  -- bodies of the various set operations
   procedure Start (I : in out Iterator) is
   begin
      I.This := Ptr(I.S.all);
   end Start;
   function Done (I : Iterator) return Boolean is
   begin
      return I.This = null;
   end Done;
   procedure Next (I : in out Iterator) is
   begin
      I.This := I.This.Next;
   end Next;
   ...
end Sets;
```

The iterator operations allow you to iterate over the elements of the set with the component `This` of the iterator object accessing the current element. The access discriminant always points to the enclosing set to which the current element belongs.

The third example uses code fragments from Rationale (1995, §4.4.4) to show an iterator using type extension and dispatching:

```
type Element is ...
package Sets is
   type Set is limited private;
   -- various set operations
   type Iterator is abstract tagged null record;
   procedure Iterate (S : in Set; IC : in out Iterator'Class);
   procedure Action (E : in out Element;
                     I : in out Iterator) is abstract;
private
   -- definition of Node, Ptr (to Node), and Set
end Sets;
package body Sets is
   ...
   procedure Iterate (S : in Set; IC : in out Iterator'Class) is
      This : Ptr := Ptr (S);
   begin
      while This /= null loop
         Action (This.E, IC);  -- dispatch
         This := This.Next;
      end loop;
   end Iterate;
end Sets;
```

The general purpose iterator looks like this:

```
package Sets.Something is
   procedure Do_Something (S : Set; P : Parameters);
end Sets.Something;
package body Sets.Something is
   type My_Iterator is new Iterator with
      record
         -- components for parameters and workspace
      end record;
   procedure Action (E : in out Element;
                     I : in out My_Iterator) is
   begin
      -- do something to element E using data from iterator I
   end Action;
   procedure Do_Something (S : Set; P : Parameters) is
      I : My_Iterator;
   begin  -- Do_Something
      ...  -- copy parameters into iterator
      Iterate (S, I);
      ... copy any results from iterator back to parameters
   end Do_Something;

end Sets.Something;
```

rationale

Iteration over complex data structures is often required and, if not provided by the part itself, can be difficult to implement without violating information hiding principles.

Active and passive iterators each have their advantages, but neither is appropriate in all situations. Therefore, it is recommended that both be provided to give the user a choice of which to use in each situation.

Passive iterators are simpler and less error-prone than active iterators, in the same way that the `for` loop is simpler and less error-prone than the `while` loop. There are fewer mistakes that the user can make in using a passive iterator. Simply instantiate it with the routine to be executed for each list element, and call the instantiation for the desired list. Active iterators require more care by the user. Care must be taken to invoke the iterator operations in the proper sequence and to associate the proper iterator variable with the proper list variable. It is possible for a change made to the software during maintenance to introduce an error, perhaps an infinite loop.

On the other hand, active iterators are more flexible than passive iterators. With a passive iterator, it is difficult to perform multiple, concurrent, synchronized iterations. For example, it is much easier to use active iterators to iterate over two sorted lists, merging them into a third sorted list. Also, for multidimensional data structures, a small number of active iterator routines may be able to replace a large number of passive iterators, each of which implements one combination of the active iterators. Finally, active iterators can be passed as generic formal parameters while passive iterators cannot because passive iterators are themselves generic, and generic units cannot be passed as parameters to other generic units.

For either type of iterator, semantic questions can arise about what happens when the data structure is modified as it is being iterated. When writing an iterator, be sure to consider this possibility, and indicate with comments the behavior that occurs in such a case. It is not always obvious to the user what to expect. For example, to determine the *"closure"* of a mathematical "set" with respect to some operation, a common algorithm is to iterate over the members of the set, generating new elements and adding them to the set. In such a case, it is important that elements added to the set during the iteration be encountered subsequently during the iteration. On the other hand, for other algorithms, it may be important that the iterated set is the same set that existed at the beginning of the iteration. In the case of a prioritized list data structure, if the list is iterated in priority order, it may be important that elements inserted at lower priority than the current element during iteration not be encountered subsequently during the iteration but that elements inserted at a higher priority should be encountered. In any case, make a conscious decision about how the iterator should operate, and document that behavior in the package specification.

Deletions from the data structure also pose a problem for iterators. It is a common mistake for a user to iterate over a data structure, deleting it piece by piece during the iteration. If the iterator is not prepared for such a situation, it is possible to end up dereferencing a null pointer or committing a similar error. Such situations can be prevented by storing extra information with each data structure, which indicates whether it is currently being iterated, and using this information to disallow any modifications to the data structure during iteration. When the data structure is declared as a `limited private` type, as should usually be the case when iterators are involved, the only operations defined on the type are declared explicitly in the package that declares the type, making it possible to add such tests to all modification operations.

The Rationale (1995, §4.4.4) notes that the access discriminant and type extension techniques are inversions of each other. In the access discriminant approach, you have to write out the looping mechanism for each action. In the type extension approach, you write one loop and dispatch to the desired action. Thus, an iterator that uses the access discriminant technique would be considered active, while an iterator that uses the type extension technique would be considered passive.

notes

You can use an access to subprogram type as an alternative to generic instantiation, using a nongeneric parameter as a pointer to subprogram. You would then apply the referenced subprogram to every element in a collection (Rationale 1995, §3.7.2). There are drawbacks to this approach, however, because you cannot use it to create a general purpose iterator. Anonymous access to subprogram parameters is not allowed in Ada; thus, the following fragment is illegal:

```
procedure Iterate (C      : Collection;
                   Action : access procedure (E : in out Element));
```

The formal parameter `Action` must be of a named access subtype, as in:

```
type Action_Type is access procedure (E : in out Element);
procedure Iterate (C      : Collection;
                   Action : Action_Type);
```

In order for this to work, you must make sure that the action subprogram is in scope and not defined internal to another subprogram. If it is defined as a nested procedure, it would be illegal to access it. See the Rationale (1995, §4.4.4) for a more complete example.

For further discussion of passive and active iterators, see the Rationale (1995, §3.7.1 and §4.4.4), Ross (1989), and Booch (1987).

8.3.7 Decimal Type Output and Information Systems Annex

guideline

- Localize the currency symbol, digits separator, radix mark, and fill character in picture output.

- Consider using the # character in picture layouts so that the edited numeric output lengths are invariant across currency symbols of different lengths.

example

```
with Ada.Text_IO.Editing;
package Currency is

   type Dollars is delta 0.01 digits 10;
   type Marks   is delta 0.01 digits 10;

   package Dollar_Output is
      new Ada.Text_IO.Editing.Decimal_Output
             (Num                => Dollars,
              Default_Currency   => "$",
              Default_Fill       => '*',
              Default_Separator  => ',',
              Default_Radix_Mark => '.');

   package Mark_Output is
      new Ada.Text_IO.Editing.Decimal_Output
             (Num                => Marks,
              Default_Currency   => "DM",
              Default_Fill       => '*',
              Default_Separator  => '.',
              Default_Radix_Mark => ',');

end Currency;
with Ada.Text_IO.Editing;
with Currency;  use Currency;
procedure Test_Picture_Editing is

   DM_Amount     : Marks;
   Dollar_Amount : Dollars;

   Amount_Picture : constant Ada.Text_IO.Editing.Picture
      := Ada.Text_IO.Editing.To_Picture ("##ZZ_ZZZ_ZZ9.99");

begin   -- Test_Picture_Editing

   DM_Amount     := 1_234_567.89;
   Dollar_Amount := 1_234_567.89;

   DM_Output.Put (Item => DM_Amount,
                  Pic  => Amount_Picture);

   Dollar_Output.Put (Item => Dollar_Amount,
                      Pic  => Amount_Picture);

end Test_Picture_Editing;
```

rationale

Currencies differ in how they are displayed in a report. Currencies use different symbols of different lengths (e.g., the American $, the German DM, and the Austrian ÖS). They use different symbols to separate digits. The United States and the United Kingdom use the comma to separate groups of thousands, whereas Continental Europe uses the period. The United States and the United Kingdom use a period as a decimal point; Continental Europe uses the comma. For a program involving financial calculations that is to be reused across countries, you need to take these differences into account. By encapsulating them, you limit the impact of change in adapting the financial package.

8.3.8 Implementing Mixins

guideline

- Consider using abstract tagged types and generics to define reusable units of functionality that can be "mixed into" core abstractions (also known as mixins).

example

Note the use of an abstract tagged type as a generic formal parameter and as the exported extended type in the pattern that follows, excerpted from the Rationale (1995, §4.6.2):

```
generic
   type S is abstract tagged private;
package P is
   type T is abstract new S with private;
   -- operations on T
private
   type T is abstract new S with
      record
         -- additional components
      end record;
end P;
```

The following code shows how the generic might be instantiated to "mixin" the desired features in the final type extension. See also Guideline 9.5.1 for a related example of code.

```
-- Assume that packages P1, P2, P3, and P4 are generic packages which take a tagged
-- type as generic formal type parameter and which export a tagged type T
package Q is
   type My_T is new Basic_T with private;
   ... -- exported operations
private
   package Feature_1 is new P1 (Basic_T);
   package Feature_2 is new P2 (Feature_1.T);
   package Feature 3 is new P3 (Feature_2.T);
   package Feature_4 is new P4 (Feature_3.T);
   -- etc.
   type My_T is new Feature_4.T with null record;
end Q;
```

rationale

The Rationale (1995, §4.6.2) discusses the use of a generic template to define the properties to be mixed in to your abstraction:

The generic template defines the mixin. The type supplied as generic actual parameter determines the parent . . . the body provides the operations and the specification exports the extended type.

If you have defined a series of generic mixin packages, you would then serialize the instantiations. The actual parameter to the next instantiation is the exported tagged type from the previous instantiation. This is shown in the second code segment in the example. Each extension is derived from a previous extension, so you have a linearized succession of overriding subprograms. Because they are linearized, you have a derivation order you can use to resolve any conflicts.

You should encapsulate one extension (and related operations) per generic package. This provides a better separation of concerns and more maintainable, reusable components.

See Guideline 9.5.1 for a full discussion of the use of mixins.

8.4 INDEPENDENCE

A reusable part should be as independent as possible from other reusable parts. A potential user is less inclined to reuse a part if that part requires the use of other parts that seem unnecessary. The "extra baggage" of the other parts wastes time and space. A user would like to be able to reuse only that part that is perceived as useful.

The concept of a *"part"* is intentionally vague here. A single package does not need to be independent of each other package in a reuse library if the *"parts"* from that library that are typically reused are entire subsystems. If the entire subsystem is perceived as providing a useful function, the entire subsystem is reused. However, the subsystem should not be tightly coupled to all the other subsystems in the reuse library so that it is difficult or impossible to reuse the subsystem without reusing the entire library. Coupling between reusable parts should only occur when it provides a strong benefit perceptible to the user.

8.4.1 Subsystem Design

guideline

- Consider structuring subsystems so that operations that are only used in a particular context are in different child packages than operations used in a different context.
- Consider declaring context-independent functionality in the parent package and context-dependent functionality in child packages.

rationale

The generic unit is a basic building block. Generic parameterization can be used to break dependencies between program units so that they can be reused separately. However, it is often the case that a set of units, particularly a set of packages, are to be reused together as a subsystem. In this case, the packages can be collected into a hierarchy of child packages, with private packages to hide internal details. The hierarchy may or may not be generic. Using the child packages allows subsystems to be reused without incorporating too many extraneous operations because the unused child packages can be discarded in the new environment.

See also Guidelines 4.1.6 and 8.3.1.

8.4.2 Using Generic Parameters to Reduce Coupling

guideline

- Minimize `with` clauses on reusable parts, especially on their specifications.
- Consider using generic parameters instead of `with` statements to reduce the number of context clauses on a reusable part.
- Consider using generic formal package parameters to import directly all the types and operations defined in an instance of a preexisting generic.

example

A procedure like the following:

```
------------------------------------------------------------------------
with Package_A;
procedure Produce_And_Store_A is
   ...
begin  -- Produce_And_Store_A
   ...
   Package_A.Produce (...);
   ...
   Package_A.Store (...);
   ...
end Produce_And_Store_A;
------------------------------------------------------------------------
```

can be rewritten as a generic unit:

```
------------------------------------------------------------------------
generic
   with procedure Produce (...);
   with procedure Store   (...);
procedure Produce_And_Store;
------------------------------------------------------------------------
procedure Produce_And_Store is
   ...
begin  -- Produce_And_Store
   ...
   Produce (...);
   ...
   Store   (...);
   ...
end Produce_And_Store;
------------------------------------------------------------------------
```

and then instantiated:

```
------------------------------------------------------------------------
with Package_A;
with Produce_And_Store;
procedure Produce_And_Store_A is
   new Produce_And_Store (Produce => Package_A.Produce,
                          Store   => Package_A.Store);
------------------------------------------------------------------------
```

rationale

Context (with) clauses specify the names of other units upon which this unit depends. Such dependencies cannot and should not be entirely avoided, but it is a good idea to minimize the number of them that occur in the specification of a unit. Try to move them to the body, leaving the specification independent of other units so that it is easier to understand in isolation. Also, organize your reusable parts in such a way that the bodies of the units do not contain large numbers of dependencies on each other. Partitioning your library into independent functional areas with no dependencies spanning the boundaries of the areas is a good way to start. Finally, reduce dependencies by using generic formal parameters instead of with statements, as shown in the example above. If the units in a library are too tightly coupled, then no single part can be reused without reusing most or all of the library.

The first (nongeneric) version of Produce_And_Store_A above is difficult to reuse because it depends on Package_A that may not be general purpose or generally available. If the operation Produce_And_Store has reuse potential that is reduced by this dependency, a generic unit and an instantiation should be produced as shown above. The with clause for Package_A has been moved from the Produce_And_Store generic procedure, which encapsulates the reusable algorithm to the Produce_And_Store_A instantiation. Instead of naming the package that provides the required operations, the generic unit simply lists the required operations themselves. This increases the independence and reusability of the generic unit.

This use of generic formal parameters in place of with clauses also allows visibility at a finer granularity. The with clause on the nongeneric version of Produce_And_Store_A makes all of the contents of Package_A visible to Produce_And_Store_A, while the generic parameters on the generic version make only the Produce and Store operations available to the generic instantiation.

Generic formal packages allow for "safer and simpler composition of generic abstractions" (Rationale 1995, §12.6). The generic formal package allows you to group a set of related types and their operations into a single unit, avoiding having to list each type and operation as an individual generic formal parameter. This technique allows you to show clearly that you are extending the functionality of one generic with another generic, effectively parameterizing one abstraction with another.

8.4.3 Coupling Due to Pragmas

guideline

- In the specification of a generic library unit, use pragma `Elaborate_Body`.

example

```
---------------------------------------------------------------------------
generic
   ...
package Stack is

   pragma Elaborate_Body (Stack); -- in case the body is not yet elaborated

   ...
end Stack;
---------------------------------------------------------------------------
with Stack;
package My_Stack is
   new Stack (...);
---------------------------------------------------------------------------
package body Stack is
begin
   ...
end Stack;
---------------------------------------------------------------------------
```

rationale

The elaboration order of compilation units is only constrained to follow the compilation order. Furthermore, any time you have an instantiation as a library unit or an instantiation in a library package, Ada requires that you elaborate the body of the generic being instantiated before elaborating the instantiation itself. Because a generic library unit body may be compiled after an instantiation of that generic, the body may not necessarily be elaborated at the time of the instantiation, causing a `Program_Error`. Using pragma `Elaborate_Body` avoids this by requiring that the generic unit body be elaborated immediately after the specification, whatever the compilation order.

When there is clear requirement for a recursive dependency, you should use pragma `Elaborate_Body`. This situation arises, for example, when you have a recursive dependency (i.e., package A's body depends on package B's specification and package B's body depends on package A's specification).

notes

Pragma `Elaborate_All` controls the order of elaboration of one unit with respect to another. This is another way of coupling units and should be avoided when possible in reusable parts because it restricts the number of configurations in which the reusable parts can be combined. Recognize, however, that pragma `Elaborate_All` provides a better guarantee of elaboration order because if using this pragma uncovers elaboration problems, they will be reported at link time (as opposed to a run-time execution error).

Any time you call a subprogram (typically a function) during the elaboration of a library unit, the body of the subprogram must have been elaborated before the library unit. You can ensure this elaboration happens by adding a pragma `Elaborate_Body` for the unit containing the function. If, however, that function calls other functions, then it is safer to put a pragma `Elaborate_All` on the unit containing the function.

For a discussion of the pragmas `Pure` and `Preelaborate`, see also the Ada Reference Manual (1995, §10.2.1) and the Rationale (1995, §10.3). If you use either pragma `Pure` or `Preelaborate`, you will not need the pragma `Elaborate_Body`.

The idea of a registry is fundamental to many object-oriented programming frameworks. Because other library units will need to call it during their elaboration, you need to make sure that the registry itself is elaborated early. Note that the registry should only depend on the root types of the type hierarchies and that the registry should only hold "class-wide" pointers to the objects, not more specific pointers. The root types should not themselves depend on the registry. See Chapter 9 for a more complete discussion of the use of object-oriented features.

8.4.4 Part Families

guideline

- Create families of generic or other parts with similar specifications.

example

The Booch parts (Booch 1987) are an example of the application of this guideline.

rationale

Different versions of similar parts (e.g., bounded versus unbounded stacks) may be needed for different applications or to change the properties of a given application. Often, the different behaviors required by these versions cannot be obtained using generic parameters. Providing a family of parts with similar specifications makes it easy for the programmer to select the appropriate one for the current application or to substitute a different one if the needs of the application change.

notes

A reusable part that is structured from subparts that are members of part families is particularly easy to tailor to the needs of a given application by substitution of family members.

Guideline 9.2.4 discusses the use of tagged types in building different versions of similar parts (i.e., common interface, multiple implementations).

8.4.5 Conditional Compilation

guideline

- Structure reusable code to take advantage of dead code removal by the compiler.

example

```
------------------------------------------------------------------------
package Matrix_Math is
   ...
   type Algorithm is (Gaussian, Pivoting, Choleski, Tri_Diagonal);
   generic
      Which_Algorithm : in     Algorithm := Gaussian;
   procedure Invert ( ... );
end Matrix_Math;
------------------------------------------------------------------------
package body Matrix_Math is
   ...
   ---------------------------------------------------------------------
   procedure Invert ( ... ) is
      ...
   begin  -- Invert
      case Which_Algorithm is
         when Gaussian     => ... ;
         when Pivoting     => ... ;
         when Choleski     => ... ;
         when Tri_Diagonal => ... ;
      end case;
   end Invert;
   ---------------------------------------------------------------------
end Matrix_Math;
------------------------------------------------------------------------
```

rationale

Some compilers omit object code corresponding to parts of the program that they detect can never be executed. Constant expressions in conditional statements take advantage of this feature where it is available, providing a limited form of conditional compilation. When a part is reused in an implementation that does not support this form of conditional compilation, this practice produces a clean structure that is easy to adapt by deleting or commenting out redundant code where it creates an unacceptable overhead.

This feature should be used when other factors prevent the code from being separated into separate program units. In the above example, it would be preferable to have a different procedure for each algorithm. But the algorithms may differ in slight but complex ways to make separate procedures difficult to maintain.

caution

Be aware of whether your implementation supports dead code removal, and be prepared to take other steps to eliminate the overhead of redundant code if necessary.

8.4.6 Table-Driven Programming

guideline

- Write table-driven reusable parts wherever possible and appropriate.

example

The epitome of table-driven reusable software is a parser generation system. A specification of the form of the input data and of its output, along with some specialization code, is converted to tables that are to be "walked" by preexisting code using predetermined algorithms in the parser produced. Other forms of "application generators" work similarly.

rationale

Table-driven (sometimes known as data-driven) programs have behavior that depends on data `with`'ed at compile time or read from a file at run-time. In appropriate circumstances, table-driven programming provides a very powerful way of creating general-purpose, easily tailorable, reusable parts.

See Guideline 5.3.4 for a short discussion of using access-to-subprogram types in implementing table-driven programs.

notes

Consider whether differences in the behavior of a general-purpose part could be defined by some data structure at compile- or run-time, and if so, structure the part to be table-driven. The approach is most likely to be applicable when a part is designed for use in a particular application domain but needs to be specialized for use in a specific application within the domain. Take particular care in commenting the structure of the data needed to drive the part.

Table-driven programs are often more efficient and easier to read than the corresponding `case` or `if-elsif-else` networks to compute the item being sought or looked up.

8.4.7 String Handling

guideline

- Use the predefined packages for string handling.

example

Writing code such as the following is no longer necessary in Ada 95:

```
function Upper_Case (S : String) return String is

   subtype Lower_Case_Range is Character range 'a'..'z';

   Temp : String := S;
   Offset : constant := Character'Pos('A') - Character'Pos('a');

begin
   for Index in Temp'Range loop
      if Temp(Index) in Lower_Case_Range then
         Temp(Index) := Character'Val (Character'Pos(Temp(Index)) + Offset);
      end if;
   end loop;
   return Temp;
end Upper_Case;


with Ada.Characters.Latin_1;
function Trim (S : String) return String is
   Left_Index  : Positive := S'First;
   Right_Index : Positive := S'Last;
   Space : constant Character := Ada.Characters.Latin_1.Space;
begin
   while (Left_Index < S'Last) and then (S(Left_Index) = Space) loop
      Left_Index := Positive'Succ(Left_Index);
   end loop;

   while (Right_Index > S'First) and then (S(Right_Index) = Space) loop
      Right_Index := Positive'Pred(Right_Index);
   end loop;

   return S(Left_Index..Right_Index);
end Trim;
```

Assuming a variable `S` of type `String`, the following expression:

```
Upper_Case(Trim(S))
```

can now be replaced by more portable and preexisting language-defined operations such as:

```
with Ada.Characters.Handling;  use Ada.Characters.Handling;
with Ada.Strings;              use Ada.Strings;
with Ada.Strings.Fixed;        use Ada.Strings.Fixed;

...
To_Upper (Trim (Source => S, Side => Both))
```

rationale

The predefined Ada language environment includes string handling packages to encourage portability. They support different categories of strings: fixed length, bounded length, and unbounded length. They also support subprograms for string construction, concatenation, copying, selection, ordering, searching, pattern matching, and string transformation. You no longer need to define your own string handling packages.

8.4.8 Tagged Type Hierarchies

guideline

- Consider using hierarchies of tagged types to promote generalization of software for reuse.
- Consider using a tagged type hierarchy to decouple a generalized algorithm from the details of dependency on specific types.

example

```
with Wage_Info;
package Personnel is
   type Employee is abstract tagged limited private;
   type Employee_Ptr is access all Employee'Class;
   ...
   procedure Compute_Wage (E : Employee) is abstract;
private
   type Employee is tagged limited record
      Name  : ...;
      SSN   : ... ;
      Rates : Wage_Info.Tax_Info;
      ...
   end record;
end Personnel;
package Personnel.Part_Time is
   type Part_Timer is new Employee with private;
   ...
   procedure Compute_Wage (E : Part_Timer);
private
   ...
end Personnel.Part_Time;
package Personnel.Full_Time is
   type Full_Timer is new Employee with private;
   ...
   procedure Compute_Wage (E : Full_Timer);
private
   ...
end Personnel.Full_Time;
```

Given the following array declaration:

```
type Employee_List is array (Positive range <>) of Personnel.Employee_Ptr;
```

you can write a procedure that computes the wage of each employee, regardless of the different types of employees that you create. The `Employee_List` consists of an array of pointers to the various kinds of employees, each of which has an individual `Compute_Wage` procedure. (The primitive `Compute_Wage` is declared as an abstract procedure and, therefore, must be overridden by all descendants.) You will not need to modify the payroll code as you specialize the kinds of employees:

```
procedure Compute_Payroll (Who : Employee_List) is
begin -- Compute_Payroll
   for E in Who'Range loop
      Compute_Wage (Who(E).all);
   end loop;
end Compute_Payroll;
```

rationale

The general algorithm can depend polymorphically on objects of the class-wide type of the root tagged type without caring what specialized types are derived from the root type. The generalized algorithm does not need to be changed if additional types are added to the type hierarchy. See also Guideline 5.4.2. Furthermore, the child package hierarchy then mirrors the inheritance hierarchy.

A general root tagged type can define the common properties and have common operations for a hierarchy of more specific types. Software that depends only on this root type will be general, in that it can be used with objects of any of the more specific types. Further, the general algorithms of clients of the root type do not have to be changed as more specific types are added to the type hierarchy. This is a particularly effective way to organize object-oriented software for reuse.

Separating the hierarchy of derived tagged types into individual packages enhances reusability by reducing the number of items in package interfaces. It also allows you to `with` only the capabilities needed.

See also Guidelines 9.2, 9.3.1, 9.3.5, and 9.4.1.

8.5 SUMMARY

understanding and clarity

- Select the least restrictive names possible for reusable parts and their identifiers.
- Select the generic name to avoid conflicting with the naming conventions of instantiations of the generic.
- Use names that indicate the behavioral characteristics of the reusable part, as well as its abstraction .
- Do not use abbreviations in identifier or unit names.
- Document the expected behavior of generic formal parameters just as you document any package specification.

robustness

- Use named numbers and static expressions to allow multiple dependencies to be linked to a small number of symbols.
- Use unconstrained array types for array formal parameters and array return values.
- Make the size of local variables depend on actual parameter size, where appropriate.
- Minimize the number of assumptions made by a unit.
- For assumptions that cannot be avoided, use subtypes or constraints to automatically enforce conformance.
- For assumptions that cannot be automatically enforced by subtypes, add explicit checks to the code.
- Document all assumptions.
- If the code depends upon the implementation of a specific Special Needs Annex for proper operation, document this assumption in the code.
- Use first subtypes when declaring generic formal objects of mode `in out`.
- Beware of using subtypes as subtype marks when declaring parameters or return values of generic formal subprograms.
- Use attributes rather than literal values.
- Be careful about overloading the names of subprograms exported by the same generic package.
- Within a specification, document any tasks that would be activated by `with`'ing the specification and by using any part of the specification.
- Document which generic formal parameters are accessed from a task hidden inside the generic unit.
- Document any multithreaded components.
- Propagate exceptions out of reusable parts. Handle exceptions within reusable parts only when you are certain that the handling is appropriate in all circumstances.

- Propagate exceptions raised by generic formal subprograms after performing any cleanup necessary to the correct operation of future invocations of the generic instantiation.
- Leave state variables in a valid state when raising an exception.
- Leave parameters unmodified when raising an exception.

adaptability

- Provide core functionality in a reusable part or set of parts so that the functionality in this abstraction can be meaningfully extended by its reusers.
- More specifically, provide initialization and finalization procedures for every data structure that may contain dynamic data.
- For data structures needing initialization and finalization, consider deriving them, when possible, from the types `Ada.Finalization.Controlled` or `Ada.Finalization.Limited_Controlled`.
- Use generic units to avoid code duplication.
- Parameterize generic units for maximum adaptability.
- Reuse common instantiations of generic units, as well as the generic units themselves.
- Consider using a limited private type for a generic formal type when you do not need assignment on objects of the type inside the generic body.
- Consider using a nonlimited private type for a generic formal type when you need normal assignment on objects of the type inside the body of the generic.
- Consider using a formal tagged type derived from `Ada.Finalization.Controlled` when you need to enforce special assignment semantics on objects of the type in the body of the generic.
- Export the least restrictive type that maintains the integrity of the data and abstraction while allowing alternate implementations.
- Consider using a limited private abstract type for generic formal types of a generic that extends a formal private tagged type.
- Use generic units to encapsulate algorithms independently of data type.
- Consider using abstract data types (not to be confused with Ada's abstract types) in preference to abstract data objects.
- Consider using generic units to implement abstract data types independently of their component data type.
- Provide iterators for traversing complex data structures within reusable parts.
- Consider providing both active and passive iterators.
- Protect the iterators from errors due to modification of the data structure during iteration.
- Document the behavior of the iterators when the data structure is modified during traversal.
- Localize the currency symbol, digits separator, radix mark, and fill character in picture output.
- Consider using the `#` character in picture layouts so that the edited numeric output lengths are invariant across currency symbols of different lengths.
- Consider using abstract tagged types and generics to define reusable units of functionality that can be "mixed into" core abstractions (also known as mixins).
- Consider structuring subsystems so that operations that are only used in a particular context are in different child packages than operations used in a different context.
- Consider declaring context-independent functionality in the parent package and context-dependent functionality in child packages.

independence

- Minimize `with` clauses on reusable parts, especially on their specifications.
- Consider using generic parameters instead of `with` statements to reduce the number of context clauses on a reusable part.
- Consider using generic formal package parameters to import directly all the types and operations defined in an instance of a preexisting generic.
- In the specification of a generic library unit, use pragma `Elaborate_Body`.
- Create families of generic or other parts with similar specifications.

- Structure reusable code to take advantage of dead code removal by the compiler.
- Write table-driven reusable parts wherever possible and appropriate.
- Use the predefined packages for string handling.
- Consider using hierarchies of tagged types to promote generalization of software for reuse.
- Consider using a tagged type hierarchy to decouple a generalized algorithm from the details of dependency on specific types.
