Safe and Secure Software

An Invitation to **Ada 2005**

4

Safe Architecture

Courtesy of

AdaCore

The GNAT Pro Company

John Barnes

When speaking of buildings, a good architecture is one whose design gives the required strength in a natural and unobtrusive manner and thereby provides a safe environment for the people within. An elegant example is the Pantheon in Rome whose spherical shape has enormous strength and provides an uncluttered space. Many ancient cathedrals are not so successful, and need buttresses tacked on the outside to prop up the walls. In 1624, Sir Henry Wooton summed the matter up in his book, *The Elements of Architecture*, by saying "Well building hath three conditions – commoditie, firmenes & delight". In modern terms, it should work, be strong and be beautiful as well.

A good architecture in a program should similarly provide unobtrusive safety for the detailed workings of the inner parts within a clean framework. It should permit interaction where appropriate and prevent unrelated activities from accidentally interfering with each other. And a good language should enable the writing of programs with a good architecture.

There is perhaps an analogy with the architecture of office spaces. An arrangement where everyone has an individual office can inhibit communication and the flow of ideas. On the other hand, an open plan office often causes problems because noise and other distractions interfere with productivity.

The structure of an Ada program is based primarily around the concept of a package, which groups related entities together and provides a natural framework for hiding implementation details from its clients.

Package specifications and bodies

Early languages such as Fortran have a flat structure with everything essentially at the same level. As a consequence all data (other than that local to a subroutine) is visible everywhere. This can be considered as rather like an open plan office. The same flat structure appears in C, although C does provide a degree of encapsulation by allowing programmer control over the external visibility of functions and file-scope variables.

Other languages such as Algol and Pascal have a simple block structure, rather like nested Russian dolls. This is a bit better but really is no more than having an open plan office subdivided into more such offices. There are still big problems of communication.

Consider the simple problem of a stack of numbers. The protocol we want to have is that an item can be added to the stack by calling a procedure Push and that the top item can be removed from the stack by calling a function Pop – and perhaps also a procedure Clear to set the stack to an empty state. We do not want any other means of manipulating the stack since we want this protocol to be independent of the way we implement it.

Now consider the following implementation of a stack written in Pascal. The stack is represented by an array of reals and there are three operations, Push and Pop to add items and remove items respectively, and Clear to set it empty. We also declare a constant max and give it a suitable value such as 100. This avoids writing 100 in several places, which would be bad if we changed our minds later on about the required size of the stack.

```
const   max = 100;

var     top : 0 .. max;
        a : array[1..max] of real;

procedure Clear;
begin
        top := 0
end;

procedure Push(x : real);
begin
        top := top + 1;
        a[top] := x
end;

function Pop : real;
begin
        top := top – 1;
        Pop:= a[top + 1]
end
```

The main trouble with this is that max, top and a have to be declared outside Push, Pop and Clear so that they can all be accessed. And from any part of the program from which we can call Push, Pop and Clear we can also change a and top directly and so bypass the protocol and create an inconsistent stack.

This is a source of danger. If we want to monitor how many times the stack is changed then adding monitoring statements to count the calls of Push, Pop and Clear to do this is not adequate. Similarly, if we are reviewing a large program and are looking for all places where the stack is changed then we have to track all references to top and a as well as the calls of Push, Pop and Clear.

This problem applies to C as well as to Fortran and Pascal. These languages to some extent overcome the problem by adding some form of separate compilation facility. Those entities which are to be visible to other separately compiled units can then be marked by special statements such as **extern** or by using a header file. However, by its very nature separate compilation is itself flat and unstructured. Furthermore, type checking in these languages is weaker across compilation units than within a single file.

The technique in Ada is to use a package to encapsulate and hide the data shared by Push, Pop and Clear so that only those subprograms can access it. A package comes in two parts – its specification which describes its interface to other units and its body, which describes how it is implemented. We can paraphrase this by saying that the specification says what it does and the body says how it does it. The specification would simply be

```
package Stack is
  procedure Clear;
  procedure Push(X: Float);
  function Pop return Float;
end Stack;
```

This just describes the interface to the outside world. So outside the package all that is available are the three subprograms. The specification gives just enough information for the external client to write calls to the subprograms and for the compiler to compile the calls. The body could then be written as

```
package body Stack is

  Max: constant := 100;
  Top: Integer range 0 .. Max := 0;
  A: array (1 .. Max) of Float;

  procedure Clear is
  begin
    Top := 0;
  end Clear;

  procedure Push(X: Float) is
  begin
    Top := Top + 1;
    A(Top) := X;
  end Push;

  function Pop return Float is
  begin
    Top := Top – 1;
    return A(Top + 1);
  end Pop;

end Stack;
```

The body gives the full details of the subprograms and also declares the hidden objects Max, Top and A. Note the initial value of zero for Top.

In order to make use of the entities declared in a package, the client code must mention the package by means of a *with clause* thus

```
with Stack;
procedure Some_Client is
  F: Float;
begin
  Stack.Clear;
  Stack.Push(37.4);
  …
  F := Stack.Pop;
...
  Stack.Top := 5;            -- illegal!
end Some_Client;
```

So now we know that the required protocol is enforced. The client cannot accidentally or purposely interfere with the inner workings of the stack. Note in particular that the direct assignment to Stack.Top is prevented since Top is not visible to the client (it is not mentioned in the specification of the stack).

Observe carefully that there are three entities to consider: the specification of the package, its body, and of course the client.

There are important rules concerning their compilation. The client cannot be compiled without the specification being available and the body also cannot be compiled without the specification being available. But there are no similar constraints relating to the client and the body. If we decide to change the details of the implementation and this does not require the specification to be changed then the client does not have to be recompiled.

Packages and subprograms at the top level (that is, not nested inside other packages or subprograms) can always be and usually are compiled separately. They are often known as library units and said to be at the library level.

Note that the package Stack is mentioned each time an entity in it is used. This ensures that the client code is very clear as to what it is doing. Sometimes repeating the package name is tedious and so we can add a *use clause* thus

```
with Stack;  use Stack;
procedure Client is
begin
  Clear;
  Push(37.4);
  ...
end Client;
```

Of course if there were two packages Stack1 and Stack2, both declaring a procedure called Clear, and we try to "with" and "use" both of them then the code would be ambiguous and the compiler would reject it. In such a case the solution is to supply the desired package name explicitly, for example Stack2.Clear.

In conclusion, the specification defines a contract between the client and the package. The body promises to implement the specification and the client promises to use the package as described by the specification. Finally the compiler ensures that both sides stick to the contract. We will come back to these thoughts in the last chapter when we look into the ideas behind the SPARK toolset.

A vital point about Ada is that the strong type matching is enforced across compilation unit boundaries. Exactly the same checking applies, whether the program is just one compilation unit or consists of several units distributed across various files.

Private types

Another feature of a package is that part of the specification can be hidden from the client. This is done using a so-called private part. The above package Stack only implements a single stack. It might be more useful to declare a package that enabled us to declare many stacks – to do this we need to introduce the concept of a stack type.

We might write

```
package Stacks is                                    -- visible part
  type Stack is private;                             -- private type
  procedure Clear(S: out Stack);
  procedure Push(S: in out Stack; X: in Float);
  procedure Pop(S: in out Stack; X: out Float);

private                                              -- private part
  Max: constant := 100;
  type Vector is array (1 .. Max) of Float;
  type Stack is                                      -- full type
    record
      A: Vector;
      Top: Integer range 0 .. Max := 0;
    end record;
end Stacks;
```

The body would then be

```
package body Stacks is

  procedure Clear(S: out Stack) is
  begin
    S.Top := 0;
  end Clear;
```

```
   procedure Push(S: in out Stack; X: in Float) is
   begin
     S.Top := S.Top + 1;
     S.A(Top) := X;
   end Push;

   -- procedure Pop similarly

 end Stacks;
```

The user can now declare lots of stacks and act on them individually thus

```
with Stacks; use Stacks;
procedure Main is
  This_One: Stack;
  That_One: Stack;
begin
  Clear(This_One);  Clear(That_One);
  Push(This_One, 37.4);
  ...
```

The detailed information about the type Stack is given in the private part of the package and, although visible to the human reader, is not directly accessible to the code written by the client. So the specification is logically split into two parts, the visible part (everything up to the keyword **private**) and the private part.

If the private part alone is changed then the text of the client will not need changing but the client code will need recompiling because the object code might change even though the source code does not.

Any necessary recompilation is ensured by the compilation system and can be performed automatically if desired. Note carefully that this is required by the Ada language and is not simply a property of a particular implementation. It is never left to the user to decide when recompilation is necessary and so there is no risk of attempting to link together a set of inconsistent units – a big hazard in languages that do not specify precisely the interaction between compiling, binding and linking.

Finally, note the modes **in**, **out** and **in out** on the parameters. These refer to the flow of information and are explained in Chapter 6 on Safe Object Construction.

Generic contract model

Templates are an important feature of languages such as C++ (and now Java). These correspond to generics in Ada and in fact C++ based its templates partly on Ada generics. Ada generics are type-safe because of the so-called contract model.

We can extend the stack example to enable us to declare stacks of any type and any size (we can do the latter other ways as well). Consider

```
generic
  Max: Integer;                          -- formal generic parameters
  type Item is private;
package Generic_Stacks is
  type Stack is private;
  procedure Clear(S: out Stack);
  procedure Push(S: in out Stack; X: in Item);
  procedure Pop(S: in out Stack; X: out Item);

private                                                  -- private part
  type Vector is array (1 .. Max) of Item;
  type Stack is
    record
      A: Vector;
      Top: Integer range 0 .. Max := 0;
    end record;
end Generic_Stacks;
```

with an appropriate body obtained simply by replacing Float by Item.

The generic package is just a template and in order to be used in a program it has to be instantiated with appropriate actual parameters corresponding to the two generic formal parameters Max and Item. The result of instantiating a generic package is the declaration of an actual package. For example if we want stacks of integers with maximum size 50, we write

```
package Integer_Stacks is
                  new Generic_Stacks(Max => 50, Item => Integer);
```

This declares a package called Integer_Stacks which we can then use in the normal way. The essence of the contract model is that if we provide parameters that correctly match the generic specification then the package obtained from the instantiation will compile and execute correctly.

Other languages do not have this desirable property. In C++, for instance, some mismatches are caught by the linker rather than the compiler and others are even left until execution and throw an exception.

There are extensive forms of generic parameters in Ada. Writing: **type** Item **is private**; permits the actual type to be almost any type at all. Writing: **type** Item **is** (<>); permits the actual type to be any integer type (such as Integer or Long_Integer) or an enumeration type (such as Signal). Within the generic we can then use all the properties common to all integer and enumeration types with the certainty that the actual type will indeed provide these properties.

The generic contract model is very important. It enables the development of flexible but safe general-purpose libraries. An important goal is that the Ada user should not ever need to pore over the code of the generic body in order to puzzle out what went wrong.

Child units

The overall architecture of an Ada system can have a hierarchical (tree-like) structure of units, which provides both flexible information hiding and ease of modification. Child units can be public or private. Given a package called Parent we can declare a public child thus

```
package Parent.Child is ...
```

and a private child thus

```
private package Parent.Slave ...
```

Both have bodies and can have private parts as usual. The key difference is that a public child essentially extends the specification of the parent (and is thus visible to clients) whereas a private child extends the private part and body of the parent (and thus is not visible to clients). The structure permits grandchildren etc to any depth.

There are various rules concerning visibility. Children do not need an explicit with clause for their parent (visibility is automatic). However, the parent body can have a with clause for a child if it needs to use the functionality defined in the child. But since the specification of the parent must be available before the children are compiled (since the children share the name of the parent), the parent specification cannot have a normal with clause for a child. More of this later.

Another rule is that the visible part of a private child has visibility of the private part of its parent (just as the body of the parent does). But for a public child only its private part and its body (and not its visible part) has such visibility of the parent.

A special form of with clause (the **private with** clause) is permitted on a package specification; it only allows the private part to have visibility of the unit concerned. This is useful, for example, where the private part of a public child needs information provided by a private child. Thus we might have an application package App and two children App.User_View and App.Secret_Details thus

```
private package App.Secret_Details is
  type Inner is ...
  ...      -- various operations on Inner etc
end App.Secret_Details;

private with App.Secret_Details;
package App.User_View is

  type Outer is private;
  ...      -- various operations on Outer visible to the user

           -- type Inner is not visible here
private
           -- type Inner is visible here

  type Outer is
    record
      X: Secret_Details.Inner;
      ...
    end record;
  ...
end App.User_View;
```

A normal with clause for Secret_Details is not permitted on User_View because this would allow the client to see information in the package Secret_Details via the visible part of User_View. Ada carefully blocks all attempts to bypass the strict visibility control.

Unit testing

One of the problems that confronts the testing of code is to ensure that the testing does not upset the software being tested. There is an echo here of Quantum Mechanics whereby when we make an observation of a particle such as an electron, the very observation itself disturbs the state of the particle.

One problem with good software design is that we strive to hide detailed information in order to produce good abstractions – by the use of private types for example. But then when we test the system we often want to observe the detailed behavior of this hidden material.

To take a trivial example we might want to know the value of Top for a particular stack declared using the package Stacks (the one where Stack is a private type). We have not provided a means of doing this. We could add a function Size to the package Stacks but this would disturb the package and require its recompilation and that of all the client code. And possibly we might introduce errors into the package we were testing or (worse) might make errors when we later removed the testing code.

Child units provide a convenient way of overcoming this difficulty. We can write

```
package Stacks.Monitor is
  function Size(S: Stack) return Integer;
end Stacks.Monitor;

package body Stacks.Monitor is
  function Size(S: Stack) return Integer is
  begin
    return S.Top;
  end Size;
end Stacks.Monitor;
```

This works because the body of a child has visibility of the private part of its parent. So we can now call the function Size at will for test purposes and when we are satisfied that the software is correct we can delete the child package and the parent package Stacks did not have to be disturbed at all.

Mutually dependent types

Many languages have the equivalent of private types especially in connection with object-oriented programming. Basically, the intrinsic operations (methods) belonging to a type are those declared in a package (or a class) along with the type. Thus the intrinsic operations of the type Stack are Clear, Push and Pop. The same structure in C++ would be written as

```
class Stack {
...                 /*  details of stack structure  */
public:
  void   Clear();
  void   Push(float);
  float  Pop();
};
```

The C++ approach is convenient in that it only has one level of naming Stack whereas in Ada we have both package name and type name, thus Stacks.Stack. However, in practice the Ada style is not a burden especially if we apply use clauses. (Moreover, Ada users have the option of using a different style by giving the type some neutral name such as Object or Data so that they can then write Stacks.Object or Stacks.Data.)

On the other hand if we have two types that wish to share private information, it is very easy to write this in Ada. We can write

```
package Twins is
  type Dum is private;
  type Dee is private;
  ...
private
  ...              -- shared private part
end Twins;
```

and the private part defines both Dum and Dee and so they have mutual access to anything in the private part.

This is not so easy in other languages and involves constructs such as the much-discussed friend mechanism in C++. In Ada there is no possibility of getting it wrong or of breaking privacy in unexpected ways and the mechanism is symmetric.

Other examples exhibit mutual recursion. Suppose we wish to study patterns of points and lines where each point has three lines through it and each line has three points on it. (This is not an arbitrary example. Two of the most fundamental theorems of projective geometry, those of the geometers Pappus and Desargues concern such structures.) We use access types. A simple approach is a single package

```
package Points_and_Lines is
  type Point is private;
  type Line is private;
  ...
private
  type Point is
    record
      L, M, N: access Line;
    end record;
  type Line is
    record
      P, Q, R: access Point;
    end record;
end Points_and_Lines;
```

If we decided that each type deserved its own package then we could still define their mutually recursive structure using a *limited with clause*. (Two packages cannot have normal with clauses referring to each other because that creates a circularity that makes their initialization impossible.) We can write

```
limited with Lines;
package Points is
  type Point is private;
  ...
private
  type Point is
```

```
      record
         L, N, N: access Lines.Line;
      end record;
   end Points;
```

and similarly for the package Lines. A limited with clause gives a so-called incomplete view of the types in the package concerned, which means roughly that they can only be used to form access types.