Ada for MS-Windows

Paul R. Pukite
DAINA
4111 Central Ave. NE, Suite 212
Columbia Heights, MN, 55421
pukite@daina.com

The debate over languages and tools for programming reminds me of a mythical story concerning a marathon bicycle race. In the race, the breakaway leader of the field notices a light to his rear as he enters the twilight hours. Fearing that the competition has closed in, he races harder. Ultimately winning the race well into the night, he gets a better view of the light and realizes the moon has been his adversary all along.

One can draw a parallel between this tale and language selection. As a guiding principle, we must realize that whichever language leads the pack, any real or imagined competition will result in progress. Thus, providing a *domestique* to the more widely used Windows languages, this paper demonstrates the use of Ada as a Windows design and development language. As a reader, you can decide whether to ride with it or view it as a competitor and nagging presence. Remember that like Cobol and Fortran (and the moon), Ada will be around for years to come.

Ada

Because of the sprawling nature of the Windows environment, developers continually demand alternative languages and tools to handle complicated and often large projects. Clearly, Ada manages big projects well, as experience in large defense and civilian embedded software systems has shown, but its usefulness for Windows applications remains little known.

That large, well-integrated, reliable, maintainable, and reusable Ada applications *can* be developed for Windows should come as no surprise. The development process becomes more manageable and less obscure by using standard Ada constructs such as package specifications, generics, tasking, and exceptions. In particular, if you want to design components, Ada provides elegant support. Furthermore, the use of automatic code generation techniques through standards such as ASIS (Ada Semantic Interface Specification) provide advantages for many of the mundane interface coding tasks.

For those interested, the presently available commercial Windows Ada offerings include Rational, R&R, Alsys, and IntegrAda. And you might want to consider the free Ada 9X GNAT (GNU) compiler for Windows NT and OS/2, as it promises even better support for large projects and incorporates several object-oriented extensions. For the best way to get information on Ada, try the World Wide Web server on the Internet at "http://lglwww.epfl.ch/Ada/".

Hello World

Of course, one can use Ada for developing Windows programs in a manner similar to C. But given the powerful constructs available, why use this approach?

The code in Figure 1 provides a simple example:

```
with Skeleton, Display;
procedure Hello is
   procedure Run is
   begin
      Display.Prompt ( "Hello world" );
   end Run;
   package App is new Skeleton ( Test => Run );
begin
   App.Main;
end Hello;
```

Figure 1: Ada "Hello world" program

This program will create a true Windows application, which includes a menu. The generic Ada package called `App` serves as a template or placeholder for the component procedure called `Run` (note that from the code's appearance, you cannot tell that it is targeted for Windows—it looks portable). If compiled and bound as the main procedure, the executable will be called `hello.exe`. When you select the first menu item, the program will display a message box

prompt with the now familiar saying.

Right about here someone should exclaim "That's not fair, you're hiding stuff!". In turn, I will usually reply that of course *reuse* and *implementation hiding* are not entirely fair, especially by developers that feel the need to get under the hood and see the details. However, one must admit that reuse does provide a productivity advantage over those not using the concept. The same can be said for implementation hiding leading to ease of maintenance. Thus, the Ada approach encourages abstractions which promote reuse and maintainability.

That implies the next question: "OK, but what *is* under the hood?". To answer this, consider that Ada packages play the role of specifying the application programming interfaces (API) to the user. A separately compiled Ada package *body* serves to implement the actual code. It suffices to say that conventional imperative methods, similar to C or Pascal idioms, can be used whenever necessary in the body of code (the how-to of coding the package bodies is the subject over 60 available Ada textbooks).

To reiterate the essential theme, the code not shown in the "Hello world" example has been tucked away in a package library that any program can include or "with" to obtain the components and API services. Thus, packages and their encapsulated subprograms can be considered in a similar fashion to the "::" notation used to call static member functions from C++ classes.

Packaging Components

As the previous example suggests, the design of an Ada program or component should stress the concepts of separating specification from implementation. Once again, the key is to hide the Windows complexities under a wrapper. By showing as little code as possible, it is easier to point to the salient features and demonstrate additional powerful capabilities of Ada packages.

DLL Encapsulation

A useful application of Ada packages is in providing a solid component-like interface to an existing dynamic link library (DLL). This has more than passing importance, as specific DLL bindings provide the interface between an Ada program and the Windows API itself.

Figure 2 is a specification binding to a rather simple DLL called Console. The only service this DLL provides is a separate console window for writing text.

```
with Wintypes;
package Console is
   procedure Output ( Name : in STRING );
   function Link ( Window : in Wintypes.HWND )
                 return BOOLEAN;
private
   pragma INTERFACE ( WINDOWS, Link,
                      "Console_Link" );
end Console;
with System;
package body Console is
   procedure Put ( Name : in System.Address );
   pragma INTERFACE ( WINDOWS, Put,
                      "Console_Output" );

   procedure Output ( Name : in STRING ) is
      Str : constant STRING := Name & ASCII.NUL;
   begin
      Put ( Str(Str'FIRST)'ADDRESS );
   end Output;
end Console;
```

Figure 2: Ada binding to Console

The package body consists of a string conversion routine which adds a NUL termination to an Ada string, effectively creating a C-style pointer. Otherwise, the DLL interface is defined through an import library via the `pragma INTERFACE` directives. A side-effect of the representation is that, by replacing the DLL defined names (which occupy a global name space) with the package specification names, we achieve a better encapsulation of the available component subprograms. Therefore, the pragmas essentially serve to connect the Ada package components to the DLL names.

The actual DLL for Console was created by using the Meridian/Rational Ada compiler for Windows (in fact, any language that supports DLLs could have been used to implement Console, given the same specification). An example of how to invoke the DLL is shown in Figure 3.

```
-- Example test code
with Console;
....
function Test_Window_Call ( Window  : HWND;
                            Message : UINT;
                            wParam  : UINT;
                            lParam  : LONG )
                 return LONG is
begin
   case Message is
      -- window creation
      when Windows.WM_CREATE =>
         if Console.Link ( Window ) then
            null;  -- Links OK
         else
            raise Program_Error; -- No link
         end if;
      -- command from application menu
      when Windows.WM_COMMAND =>
         case wParam is
            when 101 =>
               Console.Output ( "test1" );
            when 102 =>
               Console.Output ( "test2" );
            ....
```

Figure 3: Test code for calling Console

In the test code, the `raise Program_Error` signifies an Ada exception, a handy way of developing software for robustness and keeping code free of excessive return values. So, instead of checking for return codes on functions, error conditions can be caught by exception handlers, leading to more clearly coded programs. A good example involves escape handling for dialog boxes, which brings up the next item of business...

Dialog Box Interface

A dialog box is the ubiquitous layout form in the Windows environment. Among other uses, dialog boxes typically serve to select options for an application function.

Ordinarily, the developer creates the boxes and corresponding interface code. Tools for creating the boxes and their associated controls (buttons, text, *etc.*) make the layout task easier, but little exists for interfacing to Ada code. To remedy this situation, both automatic code generation via a specification mapping, or the use of generics can substantially reduce or eliminate the remaining manual interface coding. In the former case, tools such as a standard ASIS extension allow one to extract semantic information from an Ada library and do elegant transformations.

Example - Dialog Box Interface Code Generator: One entirely practical scheme relies on Ada package specifications. In this method, the specifications provide a foundation for dialog boxes; whereby, we construct a one-to-one mapping between Ada types and dialog controls.

To briefly outline the method, a procedure specification contains a set of arguments corresponding to each of the dialog box controls (check box, radio button, edit text, *etc.*) and the desired direction of data transfer. Thus, the specification becomes a declarative interface. The utility of this approach becomes apparent when we use a code generator to create the body, which contains the interface code and the required callback function. Together, Ada's strong typing, specification constructs, and object attribute mechanisms play a key role in allowing code to be generated with a minimum of additional annotations necessary.

Consider, for example, interactively setting parameters for a graph display as in Figure 4 (this is a dialog box from a code tracing and timing analysis tool called *TrAda*).

In this box, three options are available:

1. Set the time scale.
2. Set the number of intervals on the time scale.
3. Clean Edges places “nice” values on the time scale endpoints.

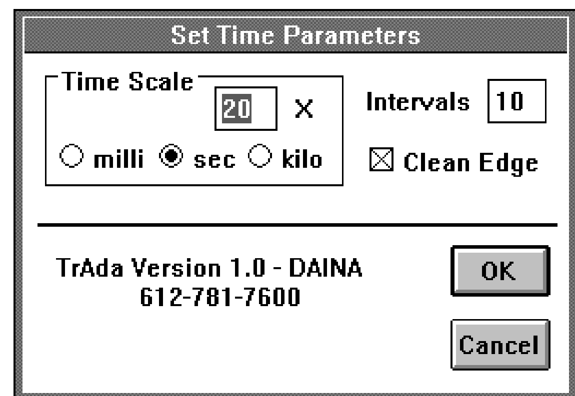

Figure 4: Dialog box for setting graphing options

The box was drawn through a standard visually-oriented Windows resource construction tool (Resource

Workshop in this case). However, the standard tool does not provide the Ada glue or interface code; this is where the specification-to-body code generator plays a role.

Figure 5 shows the Ada specification for the dialog box in Figure 4. Note the mapping of integer, enumeration type, and boolean arguments to the corresponding dialog box controls.

```
with Wintypes;
package Set_Time is

   type Scale is ( Milli, Sec, Kilo );

   procedure Box
           ( Window     : in     Wintypes.HWND;
             Max_Time   : in out INTEGER;
             Multiplier : in out Scale;
             Intervals  : in out INTEGER;
             Clean_Edge : in out BOOLEAN );

   Escape_Error : exception; -- If user cancels

end Set_Time;
```

Figure 5: Ada specification of dialog box

Thus given a set of mapping rules, the Ada body (details not shown) can be automatically generated from the Ada specification (via ASIS or YACC) to provide the glue code to the previously constructed dialog box.

Example - Generic Dialog Boxes: For dialog boxes that are used frequently, a generic specification can replace the automatically generated code. Figure 6 shows a dialog box that asks for a string input. Figure 7 is a generic package specification which features an abstract data type for the string. An instantiation of the generic (Figure 8) would require matching the generic formal objects to the dialog box resource name (`Name`) and dialog box control identifiers (`Prompt_ID` and `Input_ID`).

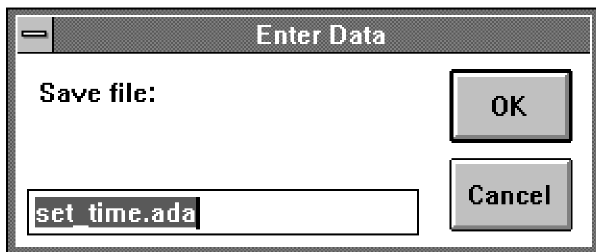

Figure 6: A generic data input dialog box

```
with Wintypes; use Wintypes;
generic
   Name      : in STRING;
   Prompt_ID : in INTEGER;
   Input_ID  : in INTEGER;
   Max_Input : in INTEGER := 128;
package Data_Handler is
   subtype Index is INTEGER range 0 .. Max_Input;
   type Edit_String is private;
   procedure Set ( Object : in out Edit_String;
                   Value  : in     STRING );
   function Value ( Object : Edit_String )
                     return STRING;
   function Length ( Object : Edit_String )
                      return Index;
   procedure Get_String
         ( Window    : in     HWND;
           Str       : in out Edit_String;
           Prompt    : in     STRING;
           Highlight : in     BOOLEAN := TRUE );
   procedure Get_Integer
         ( Window    : in     HWND;
           Number    : in out INTEGER;
           Low, High : in     INTEGER;
           Prompt    : in     STRING := "" );
private
   subtype Max_String is STRING(1..Max_Input);
   type Edit_String is
      record
         Val : Max_String;
         Len : Index;
      end record;
end Data_Handler;
```

Figure 7: A generic specification for dialog box

Overall, this method more closely follows the conventional way of creating reusable objects through Ada language constructs, but does not allow as much flexibility as a code generator.

```
with Data_Handler;
with Control_IDs;
package Data_Interface is new Data_Handler
     ( Name      => "Edit_Input",
       Prompt_ID => Control_IDs.Prompt,
       Input_ID  => Control_IDs.Input,
       Max_Input => 256 );
```

Figure 8: Instantiation of a dialog box

Tasking and DDE

Typically, each operating system provides its own specific method for inter-program data communication. In the Windows environment, applications generally support dynamic data exchange (DDE). Protocols of this kind actually conform nicely to Ada tasking models.

In one form of DDE, the client may send out a request to the server and wait for a reply. No problem understanding this concept. However, since most languages do not support concurrency primitives, one most likely will have to adopt a busy waiting scheme to signal the reply message. On the other hand, Ada provides *built-in* primitives that handle the problem cleanly. In particular, starting a separate task thread to wait on an incoming signal event results in an ideal solution to a problem that often agonizes Windows developers.

Figure 9 shows the message flow corresponding to an Ada DDE tasking design. In addition to the main thread of control A (which handles the windows events), a task B synchronizes to the DDE reply message. The actual blocking or waiting is handled by the Ada/Windows runtime system.

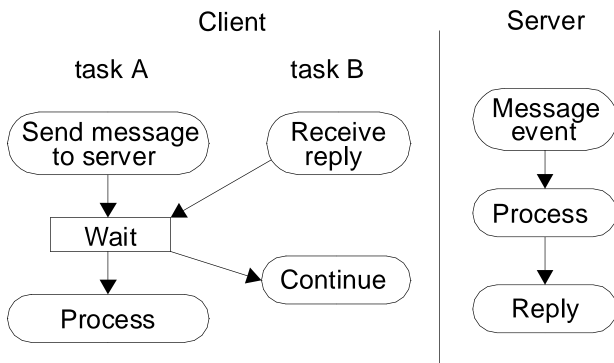

Figure 9: Task threading for a client/server

In the rest of the code, generic client and server package bodies encapsulate the details of the DDE protocol. Thus, the internal processing involves setting up channels for the communication and either posting request messages or waiting for data or acknowledgments via Ada tasking constructs and rendezvous semantics.

As another example of data exchange in action, consider a client-server tool. In a typical case, a consumer wishes to receive data in the form of tokens from a producer application. This is an on-demand service, whereby the producer does not use processing power unless needed. In this case, separate tasks can be invoked on the server side to wait in mid-processing for a request.

As a rule, Ada tasking enhances the logical flow of the program, eliminating the need for Windows constructions, such as busy waiting on message loops, the use of timer library functions, or saving the state of a computation.

Example - TrAda DLL: In a perfect world, all programs would be in Ada. In the real Windows world, we have to make concessions. One of these is to provide interfaces to other languages. The utility of the DLL approach described earlier is that a single library can be used with other DLL-compliant languages, leading to a uniform interface notation.

Unfortunately, other languages cannot use the Ada tasking runtime, so to implement DDE in a DLL we use only procedural Ada constructs. No problem, just hide the implementation details. A standard method wraps a DLL client layer around a server application's DDE channels. This provides several benefits:

1. The client linked-code import library stays small.
2. The server tool remains executable as a standalone program and thus one can use conventional development methods (tasking, *etc.*).
3. Any DLL-compliant languages can call the tool's library.

As an example, *TrAda* is a tracing and timing analysis DLL-wrapped tool for use during Windows program development. In a typical situation, one can use *TrAda* as a debugging tool for monitoring Ada tasking or sequential code flow. *TrAda* works by supplying a DDE message link via a DLL to a client application. By inserting a monitoring symbol string to the Ada code, one can follow the timing behavior of the executing program. In other words, *TrAda* acts as a Windowed oscilloscope trace of program execution.

During execution, *TrAda* acts as the server and the targeted Ada application is the client. Figure 10 shows an Ada binding to the DLL. (Note that the comment line indicated by a `--$STR`, should actually be read as a Meridian-specific compiler directive/ shortcut to indicate an Ada `STRING` is being converted to a C-style pointer with a NUL termination.)

```
-- Meridian OpenAda 2.0 binding for TrAda
with Wintypes;
package TrAda is
  procedure Trace_Name ( Name : in STRING ); --$ST
  pragma INTERFACE ( WINDOWS, Trace_Name,
                     "TrAda_Trace_Name" );
  function Link ( Window : in Wintypes.HWND )
                   return BOOLEAN;
  pragma INTERFACE ( WINDOWS, Link, "TrAda_Link" )
end TrAda; -- body is empty
```

Figure 10: DLL specification for tracing tool

To use *TrAda* the following steps are invoked:

1. The procedure `TrAda.Link(Window)` initializes the link client to *TrAda* and starts the program if it is not already loaded. `Window` is the client window's handle.
2. The `TrAda.Trace_Name("Name")` procedure sends the monitoring symbol via DDE to the concurrently executing *TrAda* program.

Figure 11 is an example of test code to be traced. When menu commands with menu identifiers numbered 101 and 102 are selected, corresponding symbols are sent to *TrAda* via the DDE connection. Figure 12 is a screen-shot of the *TrAda* executable with a typical output trace. The `Reset` button resets the time and trace. The `Time` button brings up the *Set Time Parameters* dialog box (see Figure 4).

```
-- Example test code
with TrAda;
...
function Test_Window_Call ( Window  : HWND;
                            Message : UINT;
                            wParam  : UINT;
                            lParam  : LONG )
                 return LONG is
begin
   case Message is
      when Windows.WM_CREATE =>
         if TrAda.Link ( Window ) then
            Con_IO.Put_Line ( "Linked OK" );
         end if;
      when Windows.WM_COMMAND =>
         case wParam is
            when 101 =>
               TrAda.Trace_Name ( "test 101" );
            when 102 =>
               TrAda.Trace_Name ( "test 102" );
            ...
```

Figure 11: Test code for tracing flow

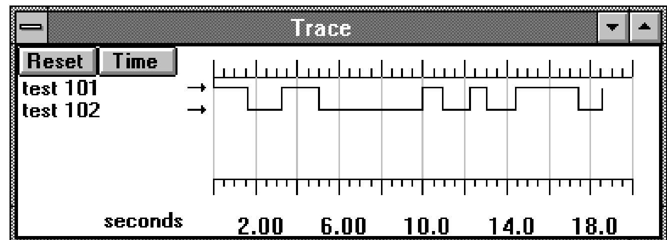

Figure 12: Example trace output

Summary

Windows possesses a strongly C-oriented application programming interface. However, with proper Ada abstractions, many potential problems associated with language interoperability can be minimized. In addition, tools to convert representations are very useful (such as converting between Ada constant declarations and C `#define` header files). In the end, Ada encapsulation, generics, and tasking can hide much of the Windows-specific details. Plus, you don't have to throw out legacy DLLs.

Ada is a good choice as a base language for Windows development, particularly for developing large-scale programs. I have found that the Ada program development was not hindered by the lack of a native code debugger (thanks to the strong type checking, exception handling, *etc.*). However, availability of a more extensive development environment, with source browsers, *etc.*, would be desirable.

Because of the overall complexity of Windows and the effort that typically goes into coding for a GUI, the Ada developer's goal should be to avoid manually generating Windows-specific code. Instead, reuse code through generics or automatic code generators whenever possible. For many programmers, the popularity of a language results from how few lines need to be written to run a "Hello World" program. In this regard, the use of high-level Ada constructs such as packages, generics, tasking, and exceptions provide for a productive programming environment.

The described tools (Console and TrAda, as well as Ada source browser BrAda) are public domain and can be obtained from the AdaNET repository at `RBSE.mountain.net` or from the author.