On Dynamic Plug-in Loading with Ada 95 and Ada 2005

Cyrille Comar, Pat Rogers
AdaCore
{comar|rogers}@adacore.com

1. Introduction

Maintenance of high-availability systems (e.g., servers) requires the ability to modify, enhance, or correct parts of the application without having to stop and re-link the entire system. This capability is relatively straight-forward with interpreted languages or virtual-machine based languages such as Java, in which new code is loaded upon demand. In languages typically implemented with static executable images this capability can be offered though dynamically loaded/linked libraries ("DLLs"). However, in practice it is impractical to make full use of this capability because the protocol for invoking subprograms in a DLL is very low-level and unsafe. In the case of Ada, global coherency requirements and elaboration ordering constraints add an additional degree of complexity over less strict/safe languages.

Object-oriented programming makes this approach practical by using dynamic dispatching to invoke dynamically loaded functions with a more robust, high-level protocol. In an OO paradigm, a "plug-in" contains new classes that enrich the class set of the original application. Calls to subprograms in the shared library (plug-in) are done implicitly through dynamic dispatching which is much simpler, transparent to the programmer, type-safe, and more robust. This application note shows how a statically-typed, statically-built, object-oriented language such as Ada can make full use of the notion of dynamic plug-ins á la Java without relying on a comparatively inefficient virtual machine. We build an extensible application and illustrate adding new functionality at run-time, without first stopping execution, using plug-ins. We use GNAT Pro to build the plug-ins and main program on a Windows system. Tailoring to run on Linux or similar operating systems would be straight-forward. Selected new features of Ada 2005 are used in the implementation and are highlighted. In particular, we use the "distinguished receiver" syntax familiar to users of C++ and Java, as well as the current Ada 95 parameter-based notation for dynamic dispatching to primitive operations. We also use one of the new "containers" data structures package and the new interface to the underlying file system.

Section two describes the structure of the demonstration application and how plug-ins are discovered and loaded. Section three then shows the commands used to build and run the main program and the plug-ins. Section four provides closing remarks.

2. The Application

The demonstration application is a revision of one of the oldest GNAT examples provided with the compiler. The application is a very crude simulation of instruments on an automobile dashboard. Not only does it now make use of the recently implemented Ada 2005 features and new library components, but in particular, the application has also been restructured to allow instruments to be coded in separate plug-ins, independent of the main program. Instruments are loaded at run-time such that the simulation can take newly created instruments into account without having to be restarted.

2.1 The Code Base

The code base is shared by both the application and any plug-ins. All the foundation classes are declared in this part of the application and are shared between all the plug-ins and the main program. This sharing of source code and object code ensures type consistency.

In the GNAT model, this base is represented by a regular dynamic library that is elaborated along with the main program. In our example, the code base consists of the package representing the dashboard along with the abstract Instrument class. This abstract class is the base class for the extensions defined by the plug-ins, as Figure 1 illustrates.

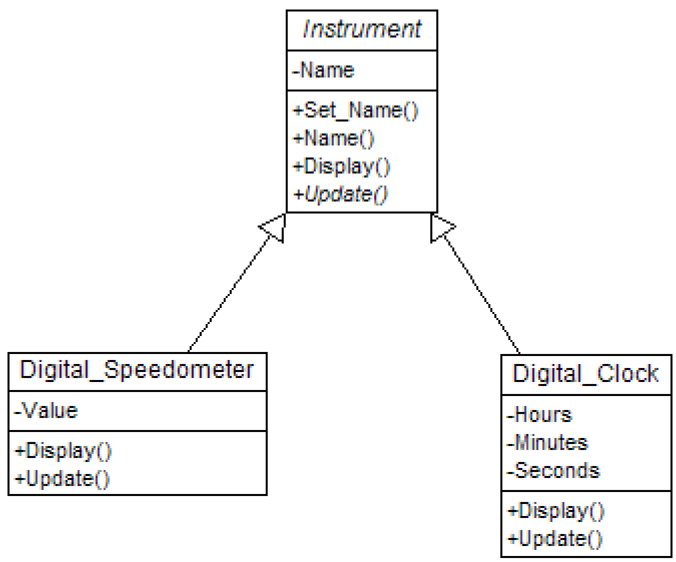

Figure 1. Instrument Class Hierarchy

The code base also includes the object representing the instruments on the dash board. Essentially this object is a registry of access values designating instrument objects created by the plug-ins during plug-in loading. For example, a fragment of the package body for package Dash_Board follows. Note the instantiation of one of the new Ada 2005 data structure packages on line 2. The package Ada.Containers.Vectors provides an extensible array abstraction, including operations to append and remove from arbitrary index locations as well as to the logical end of the vector (line 9 in the code below). We instantiate the package with Positive as the index subtype. More importantly, the element type contained within this vector is specified as Any_Instrument, an access-to-class-wide type designating type Instrument'Class so that the vector can (indirectly) contain any type of instrument, regardless of when the instrument subclass is created.

```
1. package Instruments is
2.    new Ada.Containers.Vectors (Positive, Any_Instrument);
3. use Instruments;
4.
5. Registry : Instruments.Vector;
6.
```

```
procedure Register (This : Any_Instrument) is
begin
   Append (Registry, This);
end Register;
```

Plug-ins call the Register procedure for the objects they allocate locally, within the plug-ins themselves, passing access values designating the objects. The single registry object is shared among the main program and all the loaded plug-ins. This registry is then traversed whenever the known instruments are to be displayed or updated.

2.2 The Main Program

In our example, the client main program iteratively discovers and loads any available plug-ins, displays the current dashboard, and updates the states of the instruments according to how much time has elapsed. Any new plug-ins are recognized and loaded on each iteration so the number of instruments appearing in the dashboard can increase dynamically. The main program is as follows, with the implementation of procedure Discover_Plugins elided temporarily.

```
with InDash;
with Dash_Board;
with Hashed_Strings;
with Plugins;                use Plugins;
with Ada.Text_IO;            use Ada.Text_IO;
with Ada.Integer_Text_IO;    use Ada.Integer_Text_IO;
with Ada.Directories;        use Ada.Directories;
with Ada.Strings.Unbounded;  use Ada.Strings.Unbounded;

procedure Demo is

   Increment : Integer;
   --  elapsed time in milliseconds

   Loaded_DLLs : Hashed_Strings.Set := Hashed_Strings.Empty_Set;
   --  the names of all currently discovered plug-ins

  function "+" (Source : String) return Unbounded_String
     renames To_Unbounded_String;

   procedure Discover_Plugins is ...

begin
   Put ("Give a time increment in milliseconds (0 to abandon): ");
   Get (Increment);
   if Increment <= 0 then
      return;
   end if;
   loop
      Discover_Plugins;
      Dash_Board.Display;
      Dash_Board.Update (Increment);
      delay Duration (Increment/1000);
   end loop;
end Demo;
```

As shown, the main procedure maintains a set of known DLL names (line 15) that Discover_Plugins updates once per iteration (line 30). The code in the loop calls Discover_Plugins, displays the instruments registered with the dash board, updates the states of the instruments in terms of time elapsed so that the values change, and then suspends for the time increment specified by the user at the beginning of the sequence of statements (lines 30-33). The program loops indefinitely.

2.3 The Plug-ins

Plug-ins are independent of the application in terms of dependencies and can be developed while the application is running. Each plug-in defines one or more subclasses (type extensions) of the base Instrument class, allocates objects of those subclasses, and registers them with the dashboard during elaboration. The main program discovers and loads the DLL corresponding to a given plug-in, thereby causing the allocation and registration to occur.

For example, the declaration of the speedometer plug-in is as follows:

```
1. with InDash;
2.
3. package Speedometer is
4.
5.    subtype Speed is Float range 0.0 .. 200.0; -- mph
6.
7.    type Digital_Speedometer is new InDash.Instrument with private;
8.
9.    type Digital_Speedometer_Reference is access all Digital_Speedometer;
10.
11.   procedure Display (This : access Digital_Speedometer);
12.
13.   procedure Update (This : access Digital_Speedometer;
14.                     Millisec : Integer);
15.
16.   -- this would really be in a child package
17.   function Make_Digital_Speedometer (Name : String; Value : Speed)
18.      return Digital_Speedometer_Reference;
19.
20. private
21.
22.   type Digital_Speedometer is new InDash.Instrument with record
23.      Value : Speed;
24.   end record;
25.
26. end Speedometer;
```

This is a typical abstract data type declaration using the normal information hiding techniques of the language. Note that the “constructor” function (line 17) would really be declared in a child package to avoid becoming abstract in subclasses but this approach is sufficient for our demonstration.

The package body both implements the primitive operations and also allocates and registers one or more objects of this type:

```
with Ada.Text_IO;         use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;
with Dash_Board;
```

```
4.
5. package body Speedometer is
6.
7.    procedure Display (This : access Digital_Speedometer) is
8.    begin
9.       InDash.Instrument_Reference (This).Display;
10.      Put (Integer (This.Value), 3);
11.      Put (" Miles per Hour");
12.   end Display;
13.
14.   procedure Update
15.      (This : access Digital_Speedometer;
16.       Millisec : Integer)
17.   is
18.   begin
19.      -- speed grows at 2mph per 15 seconds
20.      This.Value := This.Value + 2.0 * (Float(Millisec) / 1000.0 / 15.0);
21.   end Update;
22.
23.   function Make_Digital_Speedometer (Name : String; Value : Speed)
24.      return Digital_Speedometer_Reference
25.   is
26.      Result : Digital_Speedometer_Reference;
27.   begin
28.      Result := new Digital_Speedometer;
29.      Result.Set_Name (Name);
30.      Result.Value := Value;
31.      return Result;
32.   end Make_Digital_Speedometer;
33.
34.   use InDash;
35. begin
36.   Dash_Board.Register
37.      (Any_Instrument (Make_Digital_Speedometer ("Speed", 45.0)));
38. end Speedometer;
```

Allocation is performed by the call to Make_Digital_Speedometer (line 37). The resulting value is converted to the class-wide access type and passed to procedure Register. These operations (lines 36-37) occur during elaboration of the package body, which is itself invoked when the plug-in DLL is discovered and dynamically loaded by the main program. We are thus guaranteed that registration will occur and that it will occur only once.

2.4 Plug-in Discovery and Loading

The body for procedure Discover_Plugins is responsible for locating and loading new plug-ins. This version uses Windows DLLs to represent plug-ins so the procedure searches for files with a corresponding extension. The new Ada 2005 package Ada.Directories is used to implement this search. Ada.Directories provides an operating-system-independent interface to the underlying file system. Specifically, it presents a hierarchical file system abstraction allowing queries and manipulation of files and directories, as well as searches in the form of active iterators that traverse the directory structure. The declarations on lines one through seven are all written in terms of the types provided by Ada.Directories. Similarly, nearly all the statements use Ada.Directories operations, except for lines 22 through 27.

```
1.  procedure Discover_Plugins is
2.     S, S1      : Search_Type;
3.     D, D1      : Directory_Entry_Type;
4.     Only_Dirs  : constant Filter_Type :=
5.        (Directory => True, others => False);
6.     Only_Files : constant Filter_Type :=
7.        (Ordinary_File => True, others => False);
8.     Base       : constant String := "base.dll";
9.     use Hashed_Strings, InDash;
10. begin
11.    Start_Search (S, ".", "", Only_Dirs);
12.    while More_Entries (S) loop
13.       Get_Next_Entry (S, D);
14.       Start_Search (S1, Full_Name (D), "*.dll", Only_Files);
15.       while More_Entries (S1) loop
16.          Get_Next_Entry (S1, D1);
17.          declare
18.             P     : Plugin;
19.             Name  : constant String := Simple_Name (D1);
20.             Fname : constant String := Full_Name (D1);
21.          begin
22.             if Name (Name'Last - Base'Length + 1 .. Name'Last) /= Base
23.               and then not Contains (Loaded_DLLs, +Fname)
24.             then
25.                P := Plugins.Load (Fname);
26.                P.Call (Name (1.. Name'length-4) & "init");
27.                Insert (Loaded_DLLs, +Fname);
28.             end if;
29.          end;
30.       end loop;
31.    end loop;
32. end Discover_Plugins;
```

The name of any located DLL is compared to the name of the base DLL and to the names of those DLLs previously loaded (lines 22-23). The Load procedure loads the named DLL if no match is found (line 25). Procedure Load is defined by our package Plugins, declared as follows:

```
with Interfaces.C;

package Plugins is

   type Plugin is tagged private;

   function Load (Path : String) return Plugin;
   --  Attempts to load the plugin located at Path.
   --  Raises Not_Found with Path as the message if no plugin is
   --  located at Path.

   procedure Call (P : Plugin; Unit_Name : String);
   --  Attempts to call the function named Unit_Name within the plugin P.
   --  Raises Not_Found with the Unit_Name as the message if no such unit
   --  is present.

```

```
   function Call (P : Plugin; Unit_Name : String)
      return Interfaces.C.int;
   --  Attempts to call the function named Unit_Name within the plugin P
   --  and returns the result returned by that function.
   --  Raises Not_Found with the Unit_Name as the message if no such unit
   --  is present.

   Not_Found : exception;

   procedure Unload (P : in out Plugin);
   --  Remove the plugin from service.  Note the actual effect is
   --  operating-system dependent.  However, in all cases subsequent use
   --  of the plugin P raises Constraint_Error.

private

   type Implementation;

   type Reference is access Implementation;

   type Plugin is tagged record
      Ref : Reference;
   end record;

end Plugins;
```

Along with routines to load and unload a plug-in, there are two subprograms to call a given routine with the name specified. One subprogram returns the result code returned by the named routine; the other ignores the result. We use a "Taft Type" to represent the actual implementation so that we can support multiple operating systems with one package declaration. Thus the full declaration for type Implementation will occur in a package body that corresponds to a specific operating system. On Windows, for example, type Implementation is a derivation of type HINSTANCE ("handle to instance") defined by Windows.

Once loaded, the DLL must be manually initialized. (This is an artefact of our current Windows DLL implementation.) The initialization procedure is named by the catenation of the DLL name and "init". For example, a DLL named "clock" would be initialized by calling "clockinit". Initialization is performed on line 26 of Discover_Plugins. Note that this is an example of low-level calls that are less robust than dynamic dispatching. The implementation of the function Call illustrates the issue: only the *name* of the routine to call is specified by the user.

```
with Win32;          use Win32;
with Win32.Windef;
with Win32.Winbase;  use Win32.Winbase;
with Ada.Exceptions; use Ada.Exceptions;
with ...
package body Plugins is

   type Implementation is new Win32.Windef.HINSTANCE;

   ...

   function Call (P : Plugin; Unit_Name : String)
      return Interfaces.C.int
   is
```

```
15.       Name   : aliased String (1 .. Unit_Name'Length + 1);
16.       F      : Win32.Windef.FARPROC;
17.       Result : Interfaces.C.int;
18.       use type Win32.Windef.FARPROC;
19.    begin
20.       Name (Unit_Name'Range) := Unit_Name;
21.       Name (Name'Last) := ASCII.Nul;
22.       --  get a pointer to the function within the plugin
23.       F := Win32.Winbase.GetProcAddress
24.               (Win32.Windef.HINSTANCE (P.Ref.all),
25.                As_LPCSTR (Name'Address));
26.       if F = null then
27.          Raise_Exception (Not_Found'Identity, Unit_Name);
28.       end if;
29.       --  now we call the function via the pointer and capture the result
30.       Result := F.all;
31.       return Result;
32.    end Call;
33.
34.    ...
35. end Plugins;
```

This name is used to get an address for the corresponding routine within the DLL (line 23), which is then used to call the routine (line 30). The profile of the routine ("FARPROC") is assumed to be correct but there is no guarantee. ("FARPROC" is a Windows-defined access to subprogram type designating a function that returns a value of type Interfaces.C.int.)

In contrast, procedures Dash_Board.Display and Dash_Board.Update, called by the main program, illustrate high-level dispatching calls. Both procedures iterate over an internal registry of instrument objects maintained by package Dash_Board. The body of Display follows. Note the use of the "distinguished receiver" method invocation syntax on line 5. This is the "object.operation (parameters)" notation used by popular object-oriented languages such as Java and C++ and added to Ada 2005, although we note that the powerful OOP languages CLOS and Smalltalk have radically different notations. Use of this notation rather than that of Ada 95 for dynamic dispatching is purely a matter of taste.

```
1. procedure Display is
2.    C : Cursor := First (Registry);
3. begin
4.    while C /= No_Element loop
5.       Element (C).Display; -- dispatches
6.       Next (C);
7.    end loop;
8.    Ada.Text_IO.New_Line;
9. end Display;
```

On line 5 we call the function Element from the Ada.Containers.Vectors instantiation to get the access-to-class-wide value at the current cursor C. This access value designates an instrument object to which we can apply the Display operation, thereby dynamically dispatching to a Display routine specific to the type of the designated instrument. The call therefore dispatches from the main program to the primitive operation defined within the plug-in, and it does so in a type-safe manner because the primitive operation's signature (the parameter and result type profile) is specified by a declaration in code that is common to both the plug-in and the client main program. The fact that the object is of a type not necessarily in existence when the main program was compiled is completely transparent to both the programmer and the client main program.

3. Building and Running the Application

Now that the infrastructure is in place we can build and run the main program and then extend it with individual plug-ins while it executes.

First we build the main program. This step involves creating the "base" DLL and then creating the main program that is linked against that library. We use GNAT Project Files for this purpose. Project Files encapsulate switch settings and source file information, among other things, and make building applications very convenient. In particular, building libraries (i.e., plug-ins) is trivial with project files. Out "main" project file references a "base" project file shared by all the plug-ins and the client main program. The "base" project file is as follows:

```
1.  project Base is
2.     for Source_Files use ("dash_board.adb", "indash.adb");
3.     for Object_Dir use "obase";
4.     for Library_Dir use "lbase";
5.     for Library_Name use "base";
6.     for Library_Kind use "dynamic";
7.     package Compiler is
8.        for Default_Switches ("ada") use
9.            ("-g", "-gnat05", "-gnatwcfkmruv");
10.    end Compiler;
11. end Base;
```

The syntax is intentionally close to that of Ada packages and aspect clauses, with extensions. (The reserved words are shown in bold typeface.) This particular project is used to create a library because it contains attributes specifying library information. For example, it specifies a dynamically loadable library (line 6) named "base" (line 5) to be placed in subdirectory "lbase". It contains two source files (line 2) and compiles the resulting object files into a subdirectory named "obase" (line 3), using compilation switches that enable debugging, Ada 2005 features, and various warnings (line 9).

The project describing the client main program references both the "base" project and a project describing the Windows Ada binding:

```
with "win32ada";
with "base";

project Main is

   for Source_Files use
      ("demo.adb", "plugins.adb", "hashed_strings.ads", "h.adb");
   for Main use ("demo");

   package Compiler renames Base.Compiler;

   for Source_Dirs use (".");
   for Object_Dir use ".";

end Main;
```

Using the project file involves merely naming it as an argument to the tools. We use the "gnatmake" tool that performs all processing required to build the requested unit:

```
C:\Source\demo>gnatmake -Pmain
```

Thus, gnatmake will generate a main program named "demo" (line 8) using the sources in the current directory (line 12) and those named in particular on line 7. The compiler will use the same switches defined by project "base" (line 10) and will put the object files and executable in the current directory (line 13).

At this stage, no plug-ins have been built but the application can be launched. No instruments will be displayed because the dashboard is empty but the main program will iterate nonetheless. We first put the subdirectory containing the base DLL on the path so that it will be found by the executable and then launch the program. An arbitrary time increment of 3000 milliseconds is specified.

```
C:\Source\demo>set path=%path%;lbase
C:\Source\demo>demo
Give a time increment in milliseconds (0 to abandon): 3000
```

While the main program is printing blank lines we build the speedometers plug-in using the same command but with a dedicated project file for that specific plug-in:

```
with "base";
project speedometers is
   for Source_Files use ("speedometer.adb");
   for Object_Dir use "ospeedo";
   for Library_Dir use "lspeedo";
   for Library_Name use "speedo";
   for Library_Kind use "dynamic";
   for Library_Interface use ("speedometer");
   package Compiler renames Base.Compiler;
end speedometers;
```

Upon the next iteration the main program will discover the new plug-in and dispatch to the corresponding Display and Update routines, resulting in the following output for the one instrument currently registered (four iterations are shown):

```
Speed          :  45 Miles per Hour

Speed          :  45 Miles per Hour

Speed          :  46 Miles per Hour

Speed          :  46 Miles per Hour
```

We can build the gauges using the same approach and now three new instruments appear:

```
Speed          :  89 Miles per Hour
Fuel           :  60.00 %
Water          : <****************    >
Oil            : <********            >
```

Likewise when the third plug-in creating various clocks is built they too appear in the output:

```
Speed         : 108 Miles per Hour
Fuel          :  57.65 %
Water         : <***************     >
Oil           : <*******             >
Time in NY    : 12:15:00
Stopwatch     : <<0000>>
Time in Paris : 06:15:00:000

...

Speed         : 133 Miles per Hour
Fuel          :  54.50 %
Water         : <*************       >
Oil           : <*****               >
Time in NY    : 12:18:09
Stopwatch     : <<0189>>
Time in Paris : 06:18:09:000
```

Upon each iteration the speed increases, the clocks advance by the time increment, and the fuel, water, and oil values decrease. These changes are simply hard-coded in the instruments for the sake of the demonstration and are determined by the amount of time elapsed. Execution continues, displaying and incrementing the states of the instruments, until ranges are eventually exceeded.

4. Closing Remarks

We have demonstrated that although Ada has a static notion of types and implementations typically provide statically-linked executables, dynamic extensions are possible even while the client program continues to execute. Additionally, we have demonstrated that the concept of "plug-ins" invoked via dynamic dispatching provides a high-level, robust mechanism compared to the typical approach using only string names to identify the operation to invoke. Selected new Ada 2005 features and constructs have been demonstrated as well. In particular we used the Ada.Containers.Vectors generic and the Ada.Directories package. We also used the distinguished receiver syntax. In some cases the resulting syntax is more readable but that largely usage of this syntax is a matter of taste.

The source code for this demonstration is available upon request from the authors. Note that this source code uses features new for Ada 2005 and must, as a result, be compiled with a GNAT compiler supporting these additions. We used a pre-release of GNAT Pro 5.04 for this purpose.