A Simple Expandable Plug-in Architecture

By Ron Davis

This paper discusses issues having to do with plug-in architectures and possible solutions to those problems. It presents a Simple Expandable Plug-in Architecture(SEPA) that has many of the benefits of more complicated commercial solutions.

Introduction:

There are two kinds of plug-ins which serve two different purposes. The first is what I call a linkable plug-in. The purpose of these plug-ins is to allow others to implement some functionality of a program without explicit knowledge of the internals of the app. For example, a word processor might want the ability to edit pictures in it, but not want to implement this functionality inside the application. They could define a method of communicating the needed information to a plug-in and have someone else implement the plug-in as a separate code fragment. Then the plug-in is weak linked into the application.

The other kind of plug-in I call a dynamic plug-in, for lack of a better term. The purpose of dynamic plug-ins is to allow other people to write multiple implementations of a specific functionality. A screen saver, such as After Dark, is the ultimate example of this. People write plug-ins which implement animations. Each plug-in performs the same functionality, but in different ways. They are each opened and called at run time, and not linked into the main application.

This paper focuses on dynamic plug-ins. What a smart programmer wants in such a plug-in is something flexible, easy to implement, and compatible across revisions of the application. This paper will show a way of accomplishing this.

Through out the paper we will use as an example a screen saver like program. You will find the code for this program in the accompanying source archive.

OS implementation

SEPA uses shared libraries for its plug-ins. It ignores 68K, though it could easily be adapted to 68K code resources.

Shared libraries are in a way a plug-in, but are really intended to be linked into the application and not dynamic plug-ins. They are good at segmenting one piece of functionality and moving it out of the application, but they are poor at having multiple implementations of the same functionality.

It will be assumed the reader knows how to create a shared library project in CodeWarrior. There will also only be the barest discussion of how to load and call these libraries. The accompanying source can serve as a basic introduction to how to do this.

SEPA is also based on C++ and requires a knowledge of polymorphism and pure virtual base classes.

Plug-in Basics

Let's start with our screen saver example. For a plug-in to implement the drawing of a screen saver it needs a number of functions called. There are two kinds of information we want to pass back and forth to the plug-in. One is generic information about the plug-in itself and the other is instruction for drawing.

These are the drawing routines:

GetPluginInformation

Called when the plug-in is loaded to get information about it, such as the name of the plug-in.

Initialize()

Called by the screen saver right before it starts using the plug-in for drawing. In the screen saver case it is called when the screen is blanked after the waiting period.

DrawFrame()

Tells the plug-in to draw a frame in the animation.

Terminate()

The screen saver is going away, stop drawing and clean up.

SetDrawingRect(Rect inRect)

Called early on to tell the plug-in the bounds of where it should draw.

These are the plug-in information routines:

GetPluginName(char* outName)

Simply returns the name of the plug-in for display.

GetPluginVersion(short* outVersion)

Returns a version number for the plug-in.

Calling the functions in a code fragment.

A traditional plug-in would be a shared library which implements these routines and exports them. Then the application would use CFM to get a pointer to these routines and call them via the pointer.

There are a couple of drawbacks to this approach. First, it is cumbersome to implement. You have to plan for exporting of all the relevant functions, and you have to jump through the CFM hoops to get the functions to call. Since there are multiple plug-ins that implement the same functions you can't just link against them.

Secondly, I hate making function calls via proc pointers, and want to keep it to a minimum.

Interfaces.

What if you could implement your plug-in as a simple C++ object and call its functions just like you call any C++ function? It can be done, by implementing a simple CFM API that is 100% reusable from plug-in to plug-in.

What we are going to do is create two things, a single function to export from the CFM code fragment and some pure virtual base classes. The CFM export will be the minimum needed to get pointers to classes that implement the virtual base classes. This export will be the same one used in every plug-in using SEPA, no matter what the application.

Before we define what the exports need to be, let's discuss the C++ objects we are going to create. One of the cool things about a C++ object is you can call the functions of one object just like they were from another object. For instance, if you have an object foo:

```
class foo
{
public:
     foo();
     virtual function1() = 0;
     virtual function2() = 0;
};
```

Then you can declare a subclass of this function:

```
Class bar: public foo
{
public:
      bar();
      function1();
      function2();
}
```

You don't even need to implement the methods in foo. You can just promise that the subclasses will. This a pure virtual base class. It's pure because none of the methods are defined. An interface is a pure virtual base class.

So for our plug-in we declare interfaces in headers that are shared by the plug-in and the application. Then the application asks the plug-in for a pointer to a C++ object that implements the interface. Then the application can call the functions in the interface and it will magically call into the plug-in's implementation of those functions.

This is the core of SEPA.

So given each plug-in in SEPA needs to be able to supply interfaces on demand, what do we want each plug-in to export?

Basically we need a way to ask for a pointer to a specific C++ object that implements a certain interface from a plug-in. These pointers to objects are called Interface Pointers. Remember a plug-in might implement more than one interface. In our example it will have an interface to handle drawing and one to handle plug-in information. So we need a way to differentiate interfaces. The easiest way to do this is to give each interface a unique ID. In SEPA we'll use a simple long, defined as a constant in the header containing the interface definition.

```
const unsigned long IPluginInfo_ID
      = 'PInf';
```

```
const unsigned long IPluginDraw_ID
      = 'PIdw';
```

Now that we can identify an interface it is simple to define an export routine that gives us an interface pointer. We'll call it GetInterface

```
Void*      GetInterface( unsigned
long inIntfID );
```

The problem with this routine is to you have to use a proc pointer call every time you want an interface, and you know how I feel about doing that. So how do we get rid of proc pointer function calls? We define an interface. What I've done is create an object that returns interface pointers. I call it CInterfaceProvider, and it has one function you can call to get an interface pointer.

```
OSErr CInterfaceProvider::
GetInterfacePointer( unsigned long
inID, void** outInterfacePtr )
```

There is a simple C routine you call on each plug-in to get a pointer to this CInterfaceProvider object. It is defined in the file PluginExports.c.

```
IInterfaceProvider*
     GetInterfaceProvider()
```

It is the only routine that is exported.

Now look inside CInterfaceProvider::GetInterfacePointer. There is a simple switch statement with each implemented ID and a call to new on the implementing object to be returned.

This is the simplest interface for plug-ins. It assumes the application knows about all of the interfaces it can call on the plug-in or at least makes sure it actually gets a valid pointer back.

Backward compatibility between plug-in versions

What happens when the next version of your program comes out? Your users have screamed for the ability to handle keyboard input, but

your interfaces only handle drawing and info. Also, the plug-in developers have asked to know what bit depths they can draw in, but your API doesn't give them this information.

In a traditional plug-in you would have to define new function in the CFM API or modify the ones that were there. Afterwards none of the 1.0 plug-ins work with version 2.0 or at least there is a bunch of duplication in the API to support them. And no one even gives a thought to those people who don't upgrade to version 2.0 but download some 2.0 plug-ins. Can we make these new plug-ins work for them?

With SEPA the answer is yes. We can easily add the new keyboard interface to pass on keyboard input. If we detect a 1.0 plug-in and ask it for the IKeyboard interface, it will return null, and the app won't call those routines.

Talking both ways.

The cleanest way to talk to the application from the plug in is to use interfaces. We get all of the benefits of interfaces by doing this. Since we are now dealing with getting an interface from an application instead of a shared library, we can't use CFM to get the GetInterfaceProvider() routine.

Instead what we do is add a method to the CInterfaceProvider interface that lets the application tell the plug-in about its CInterfaceProvider object.

Conclusion

This is just the tip of the iceberg for working with plug-ins. There are a lot of things you could add to it. Hopefully this will help you create a simple, easy to implement, and easy to use plug-in architecture for your applications.