The TMS Plugin Framework
Advanced Technical Information

Copyright © 2010 - 2016 by tmssoftware.com bvba
Web: http://www.tmssoftware.com
Email : info@tmssoftware.com

Introduction

The TMS Plugin Framework (TPF) allows you to extend your applications using Plugins. Plugins are packages that are loaded by your application and can add additional functionality to it, without recompiling. This allows you to release additional features at a later date, post small interim updates and allow third-parties to develop add-ons into your product.

This release of the TPF has been designed with two key ideas in mind: First, that the system should be easy to use. Second, it should be able to expand to suit any future needs of your application.

The first goal is provided by a powerful plugin manager, which loads and maintains any loaded plugins. It also supplies the plugins with a robust application interface allowing for easy menu, toolbar, form and database integration. It is also provided with a number of plugin wizards, which allow for both code-based and visual plugin development.

The second goal is provided by the interface-based architecture. Since the whole system is designed around interfaces, it is very easy for an application developer to create a new, extended application or plugin interface and register it with the application. This is covered in the section Advanced Plugin Design.

This document is not for the casual user of the TPF. It is for the developer who wants to know how the system is designed, why it was designed this way, and how to add new capabilities to both the application and the plugins.

Interfaces

Before getting into the details of the plugin system, a brief discussion about interfaces is in order. When a class descends from an interface, it is obligated to provide the methods that the interface defines. For instance, given the interface:

```
type
  IMyInterface = interface
  ['{E160C67A-B7F8-4AEF-8651-89D97D957704}']
     procedure DoSomething;
  end;
  TMyClass = class(TInterfacedObject, IMyInterface)
     procedure DoSomething;  // this method MUST be implemented
  end;
```

In this case, the TMyClass declaration is obliged to provide the method DoSomething, since the Interface has it. The interface acts (conceptually) like a virtual abstract method. There is no stubbed procedure, no default code, just the requirement that all methods in the interface are implemented.

A quick side note to this: when you implement an interface, you can't just stub the procedures. Doing this defeats one of the key purposes of interfaces, and will invoke the wrath of Danny Thorpe!

Now we are able to do things like:

```
procedure SomeProc(SomeClass : TInterfacedObject);
begin
  if  SomeClass is IMyInterface then
     (SomeClass as IMyInterface).DoSomething;
end;
```

Take a second to let this sink in. What we have here is the ability to inject desirable functionality into a class from the side, instead of having to directly descend from an ancestor. If the variable SomeClass supports IMyInterface, then we can safely call any methods defined by IMyInterface.

Before moving on, just a quick note about the GUID. While people associate GUIDs with COM objects, they are not exclusively for COM, and it does not mean that the plugin system is COM compliant. It's merely a mostly-guaranteed unique identifier for the interface. Without it, you can't do things like IS and AS on interfaces.

There are a couple of downsides to interfaces, specifically code size and speed. I won't go into the details of this, but simplified, virtual abstract methods have one VMT and interfaces have two. This slows things down a bit (but not noticeably for most applications). Also, because the linker isn't aware of all specifically used interfaces at link-time, it needs to link in all possible code, thus defeating smart linking. The net result is larger executables. Both of these are things you should be aware of, but will probably not affect you adversely.

What is in the Plugin System?

Plugin systems have a number of responsibilities, including loading and unloading the plugins. However, the real responsibility of a plugin system is to provide plugins with hooks into the host application. This hook is called the Application Services, and is represented in the TPF as a variable called ApplicationServices. This variable is instantiated by the plugin manager, and accessed by the plugins. It is contained in the pfCoreX package (with X being the Delphi version number).

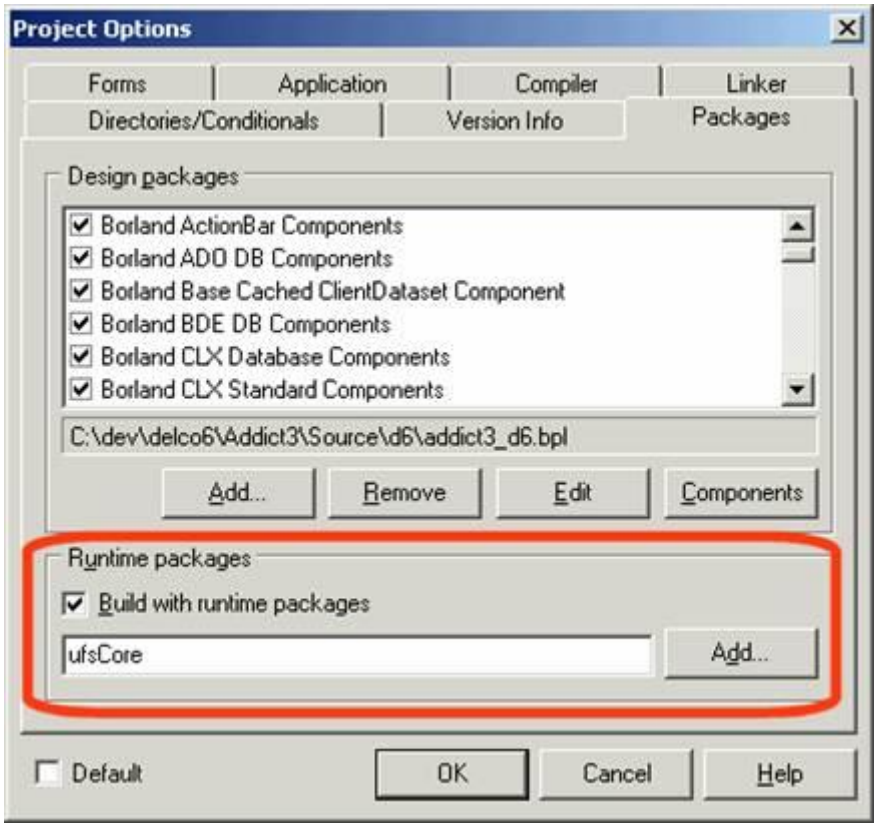
The TPF comes with a default Application Services provider, called TUPFStandardAppServices, which implements the IufsApplicationServices and IufsStandardAppServices interfaces. Plugins, since they know about IufsApplicationServices and IufsStandardAppServices, can interact with the host application.

The Myth of Packages

Most people assume that compiling with packages means you need to deploy all the packages you've installed for every third-party component, as well as the VCL packages. This is simply not true.

Packages are a major technical achievement in Delphi. They allow for a program to dynamically load what are essentially DLL files and treat them as if they're a part of your program, including RTTI information. This magic is not only very difficult to duplicate with standard DLLs, it's also crazy to try, given that Delphi has already done all the work for us. We just need to use it.

So, we want to build our application using packages. To do this, click on the Project menu, and select Options. That brings up the dialog box below. Click on the Packages tab, and at the bottom, click the Build with runtime packages checkbox. Clear out the items in the edit box below it, and add pfCoreX (X being the Delphi version number you're using). This is the only runtime package that you must distribute, since it's what the plugins are going to communicate with.

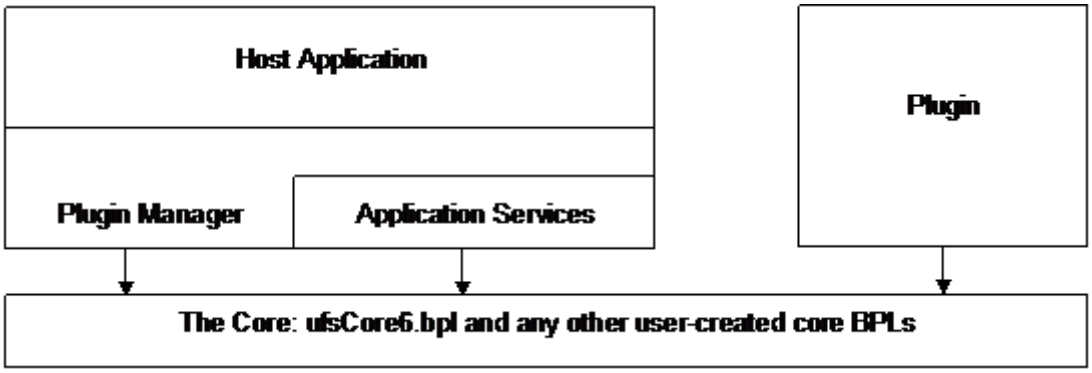


When you distribute the application, make sure that pfCoreX.bpl is in the same folder as your application, and it will be loaded by the application at run time.

There are many benefits to distributing other packages for other components that you use. However, it's up to you to decide what packages to use and what packages not to use.

The TPF Architecture

The TPF was designed as a drop-in solution for most applications. It works by supplying the application with an entry point for the plugin to communicate with the application. There are a number of items involved in this communication.



The Plugin Manager

The TUPFPluginManager component is the key component on the host application side. It creates and manages the Application Services for the plugin to communicate with, as well as handles the loading and unloading of plugins.

The plugin manager can instantiate any class that implements IUPFApplicationServices as the application service and is registered into the design-time environment. You can select the class you want to instantiate through the plugin manager's ApplicationServicesClass property.

The Application Services

There are a number of classes and interfaces that declare and implement the application services. First, we have IUPFApplicationServices, which is the key interface. Any class that is to be an application services provider must implement this interface. There is also a default implementation of the interface that, barring any special needs by the application, is automatically created by the plugin manager.

The application developer can decide to instantiate a different class that implements IUPFApplicationServices, or a descendent of it. In addition, the different class could also implement additional interfaces that the plugin could be aware of.

There is a default class, TUPFApplicationServices that implements the required methods of IUPFApplicationServices, but does not use IUPFApplicationServices. The default class created by the plugin manager (TUPFStandardAppServices) is a descendent of TUPFApplicationServices and IUPFApplicationServices.

The default way of handling plugins is that the plugin is responsible for creating and registering an instance of a class that supports IUPFPlugin. This is not always the desirable behaviour. In some cases you want to register the class with the host application, and allow it to create instances of this class. To facilitate this, an interaface called IUPFClassServices is available.

The Core

The core BPL file is the piece that both the plugin and the application communicate with. It contains the ApplicationServices variable, and knows the interfaces that are used by the application and the plugins. If you wish to create a new application services interface, you need to include the interface in the core. The core supplied with the TPF provides a number of common interfaces and classes that can handle most situations. If you require additional interfaces not defined in the core, you should put them in their own package. This package will be required by both your application and by any plugin developers, in addition to the pfCoreX file. Changing the contents of the pfCoreX file should be avoided at all costs.

The Plugin

Plugins implement at minimum the IUPFPlugin interface, and optionally implement additional interfaces, such as IUPFCommandPlugin (for plugins that add menu items and toolbar buttons) or IUPFComponentPlugin (for plugins that are components, such as forms, data modules or tab sheets). They can also implement any of the interfaces that the application has defined in the core. So, if you want to create a new plugin type that does something unique for your application, simply create the new interface, include it in your core, and handle it in your Application Services descendant or by handling the OnPluginRegistered event.

Helper Classes

The TPF comes with a number of helper classes that make it easy to get started, both from the host application developer's point of view, and from the plugin developer's point of view.

Visual Plugin Development Classes

The first is the TUPFVisualPlugin. This descends from TDataModule and there is a wizard to create it automatically.

What this amounts to is that from the plugin developer's point of view, developing plugins is done by: click on File, New, Project, TMS Plugin, you get a datamodule, and can set the properties at design time.

However, in the background, a lot is happening. First, the TUPFVisualPlugin class descends from TDataModule, but it also implements the IUPFPlugin and IUPFCommandPlugin interfaces. The IUPFCommandPlugin has a property, Commands, of type IUPFCommands. IUPFCommands (and the related interface, IUPFCommand) are designed around the TCollection and TCollectionItem classes, so implementing them with a TCollection is pretty easy.

Application Services Classes

The Plugin Manager, as mentioned before in the TPF Architecture section, by default creates an Application Services class, TUPFStandardAppServices, that can handle a number of the default plugin interfaces, including the IUPFPlugin, IUPFComponentPlugin and IUPFCommandPlugin. This means that an applications programmer simply needs to drop the plugin manager component onto a form, and they can handle the majority of plugins that are developed.

The idea behind this is that someone (either the host application developer, TMS, or a third party) can develop descendants of the TUPFStandardAppServices class (or a replacement for it, descending from TUPFApplicationServices and implementing IUPFApplicationServices) that can support additional types of interfaces. For example, an interface could be created that could load in old-style DLL plugins.

The TPF Plugin Wizard

In order to make developing plugins very easy, we've included the TPF Plugin Wizard. By default it creates plugins using both the TUPFStandardPlugin and TUPFVisualPlugin models. This allows developers to quickly create plugins.

The plugin wizard is itself a host to a plugin manager, and uses plugins to do the actual wizard code. This means that as additional application services come out, and additional plugins types are developed, this wizard can be expanded to include easy ways to develop them!