Plug-in Architecture using Packed Project Libraries

Michael Lacasse
mlacasse@lowell.edu
November 2, 2011

Revision 1.1

Plug-in Architecture with Packed Project Libraries

Features:

Plug-ins:

1. Abide by a specified interface.
2. Are independent of each other.
3. Don’t result in name-conflicts when used in a built application.
4. Installation is independent of main application installation.

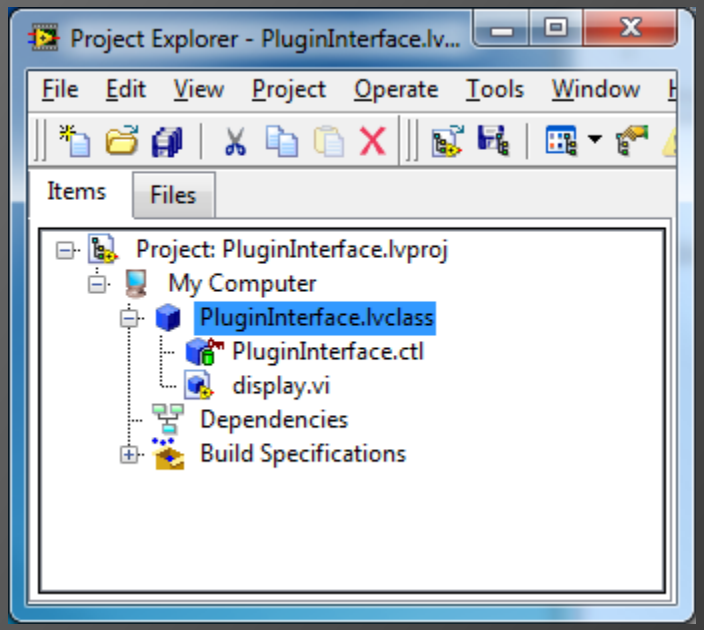
Main Application:

1. Is extensible without code modification to main application.
2. Depends only on plug-in interface, not on specific plug-ins.
3. Loads plugins dynamically.
4. Utilizes plug-ins as UI elements or OO Design Pattern such as Strategy or Command.
5. Requires no modification to successfully run in development or run-time environment.
6. Installation is independent of plug-in installation.

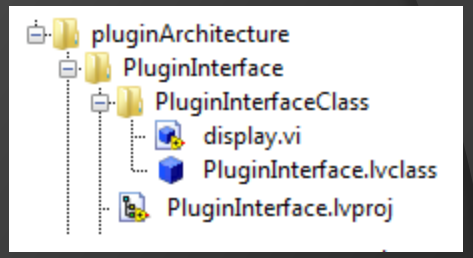
Plug-in Architecture with Packed Project Libraries

Step 1 : Create the Plugin Interface Class

In LabVIEW



On Disk



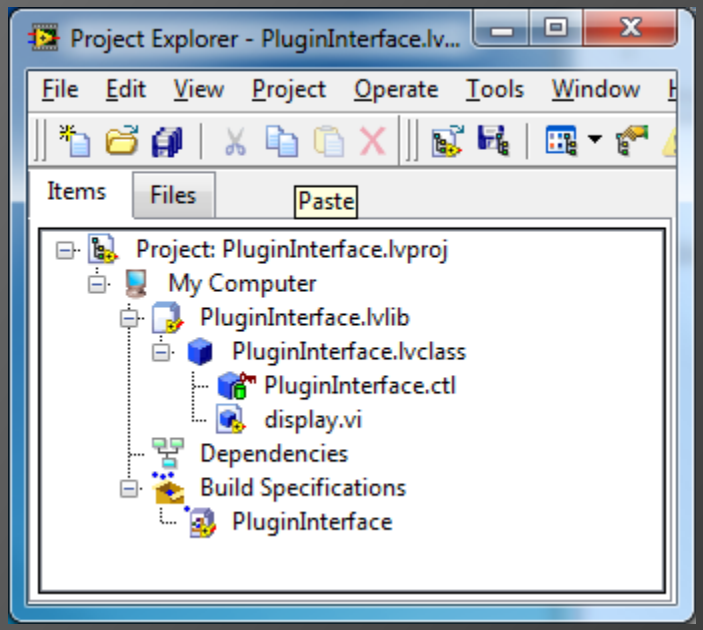
In this example, the interface and therefore the plugins will have only one method, 'display', which will simply display a pop-up dialog showing the path to 'Current VI'.

display.vi in the PluginInterface class contains no implementation.

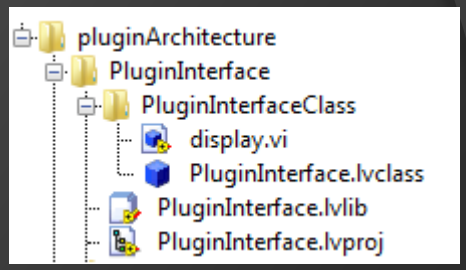
Plug-in Architecture with Packed Project Libraries

Step 2 : Create .lvlib to contain the Plugin Interface class
Create a build spec for a packed project library

In LabVIEW



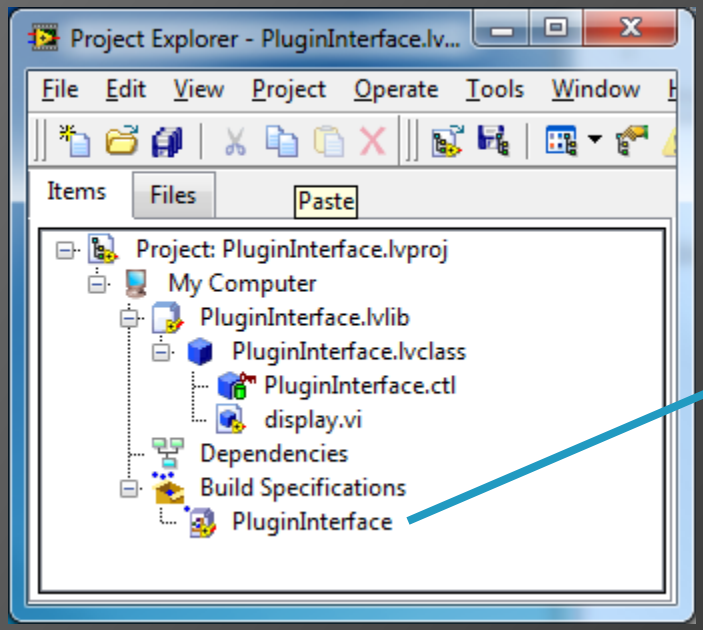
On Disk



Plug-in Architecture with Packed Project Libraries

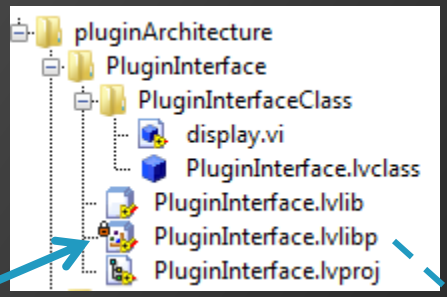
Step 3 : Set the Destination for the build to the Source directory, not the Builds directory.

In LabVIEW



Destination

On Disk



This built interface can now be used as a parent class for Concrete PlugIns.

Plug-in Architecture with Packed Project Libraries

Step 4 : Create the application or the plug-ins.

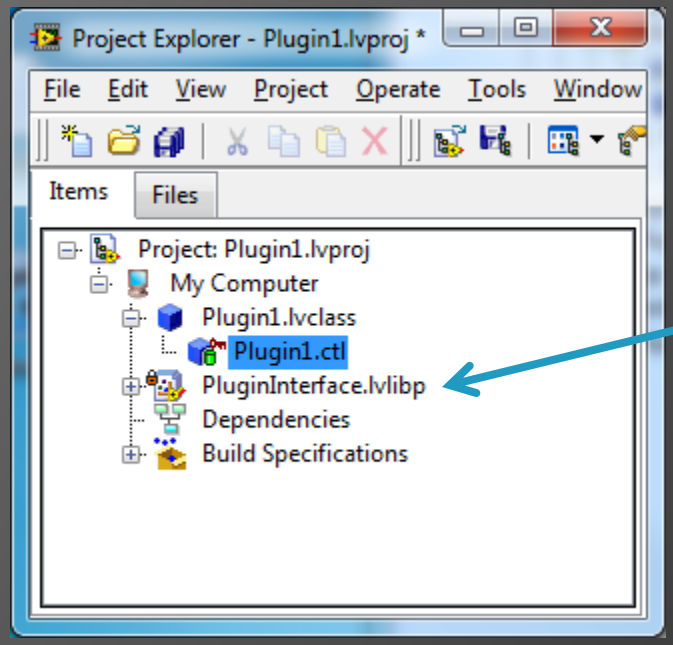
It doesn’t matter which you do first since the app will only depend on the interface.

Let’s do the plug-ins first.

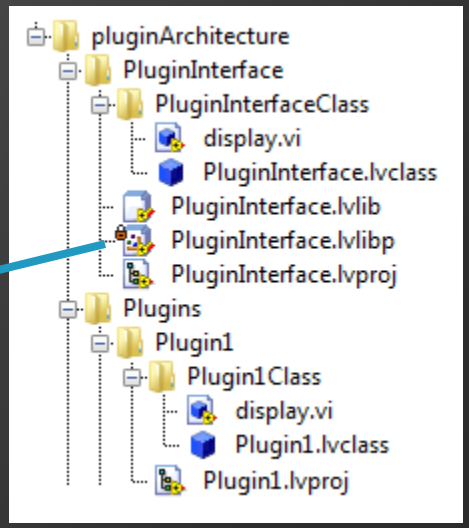
Plug-in Architecture with Packed Project Libraries

Step 5 : Create a project and a new plugin class. Include the built interface in the project.

In LabVIEW

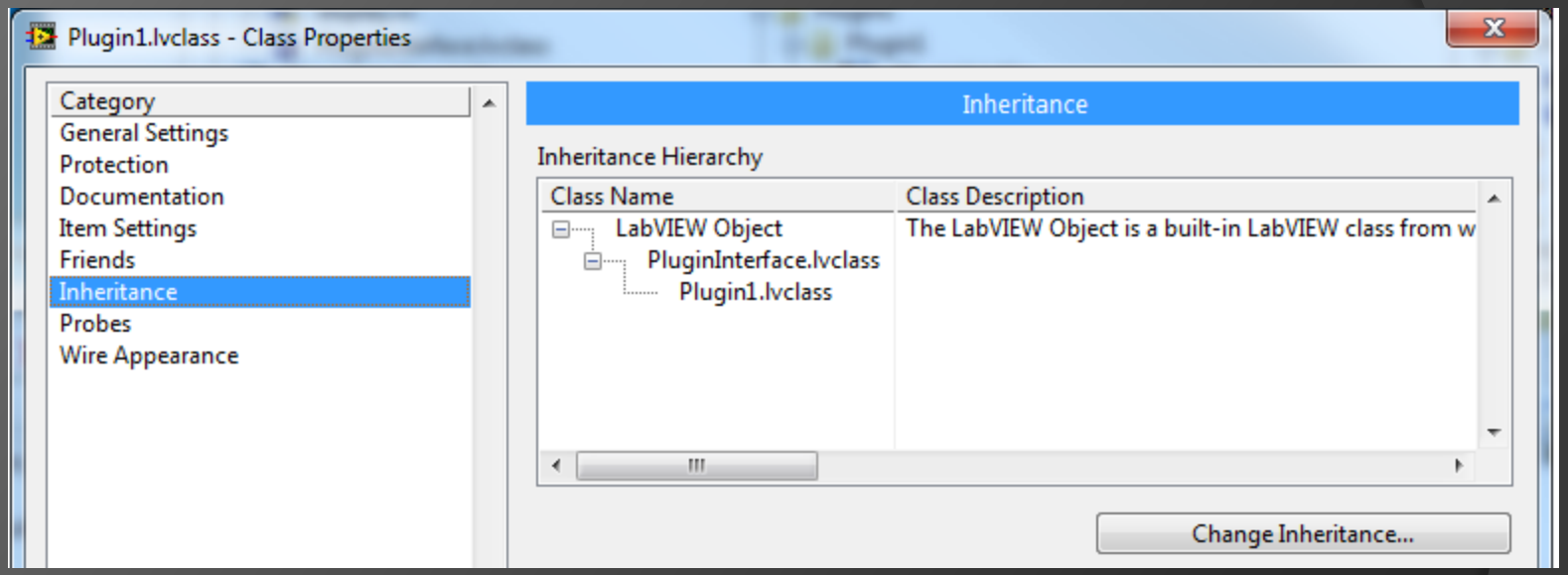


On Disk



Plug-in Architecture with Packed Project Libraries

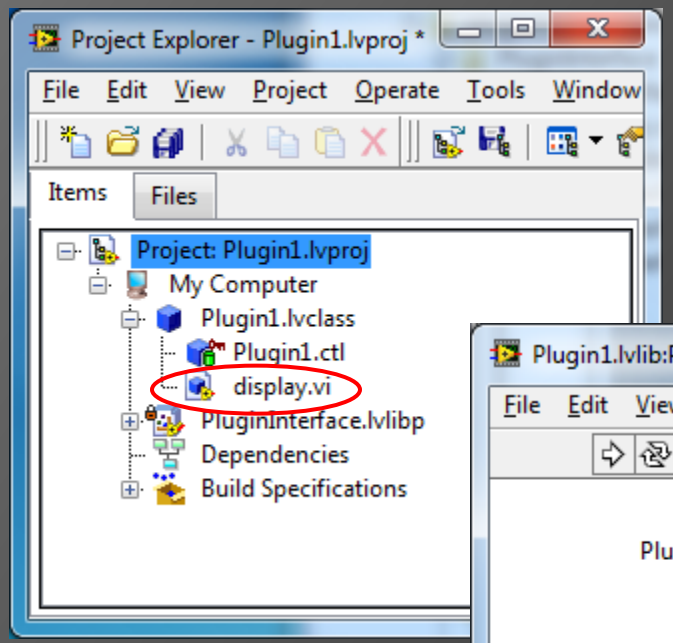
Step 6 : Change to inherit from PluginInterface.

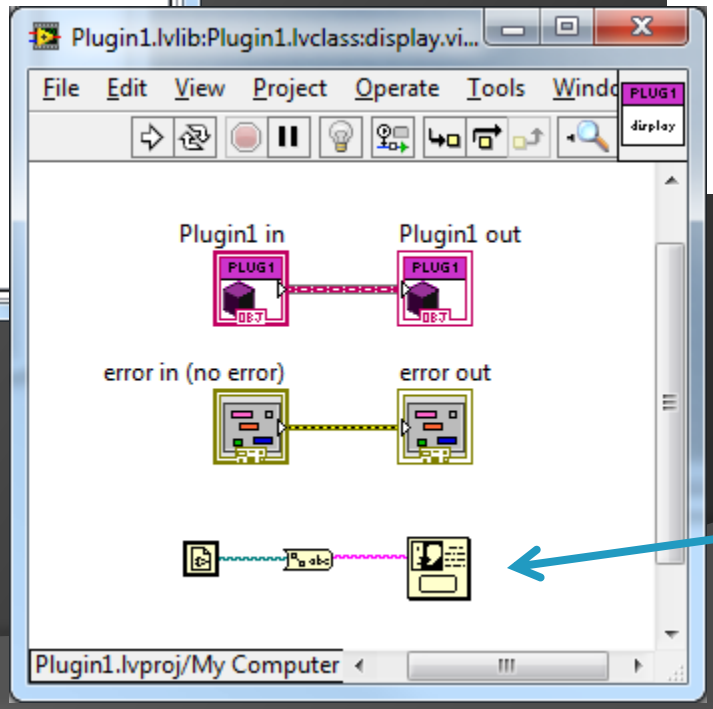


Plug-in Architecture with Packed Project Libraries

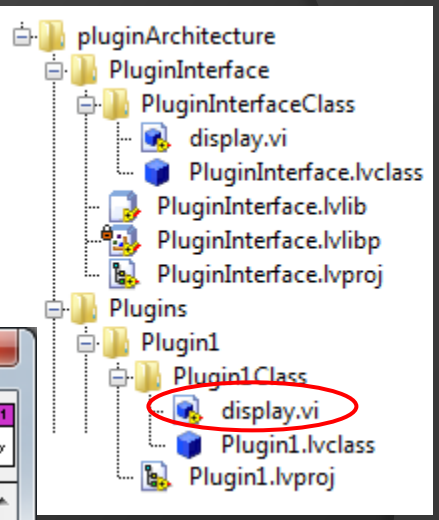
Step 7 : Create override VIs for all interface methods.

In LabVIEW





On Disk

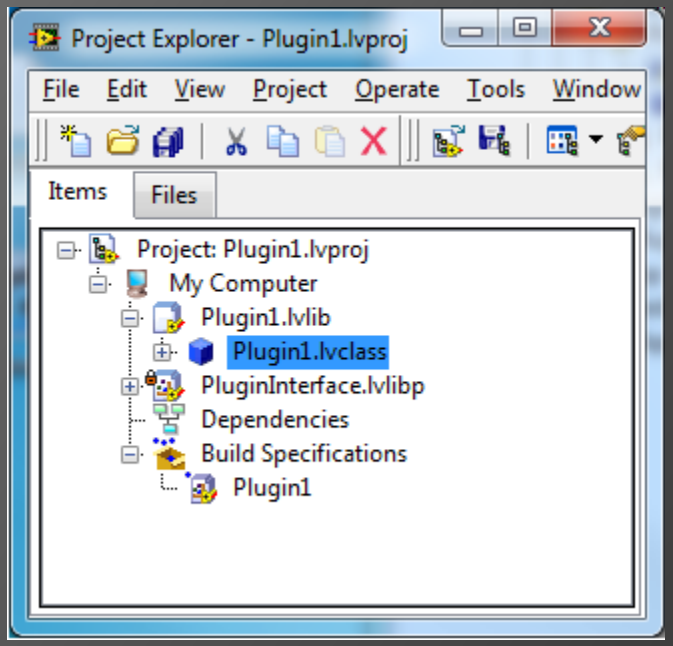


Pop-up dialog to display the path to this plugin method.

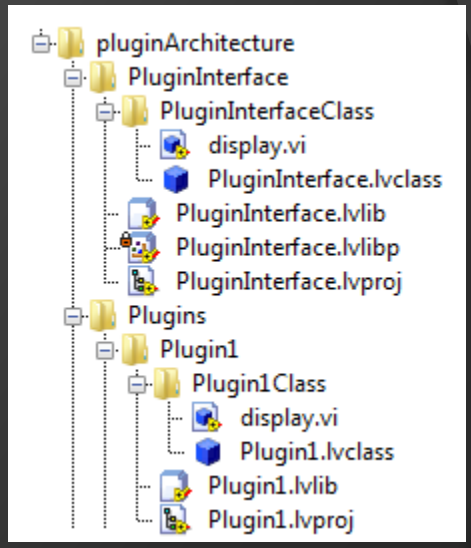
Plug-in Architecture with Packed Project Libraries

Step 8 : Create .lvlib to contain the Plugin class
Create a build spec for a packed project library

In LabVIEW



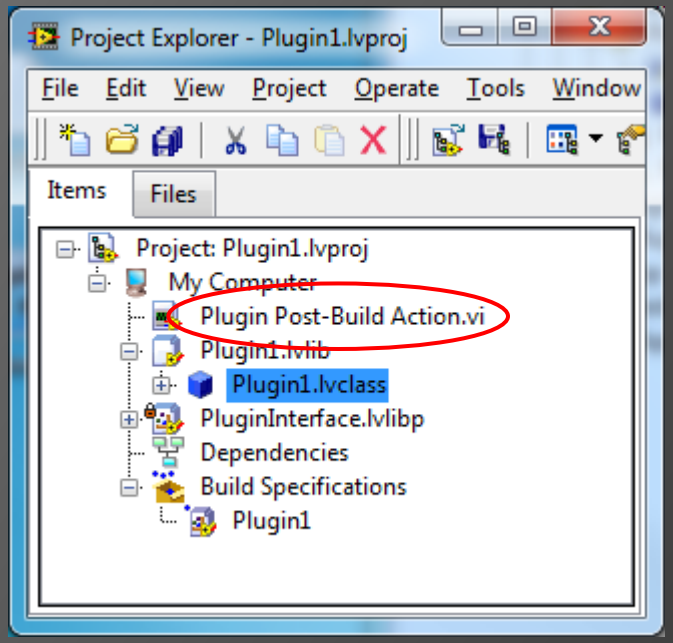
On Disk



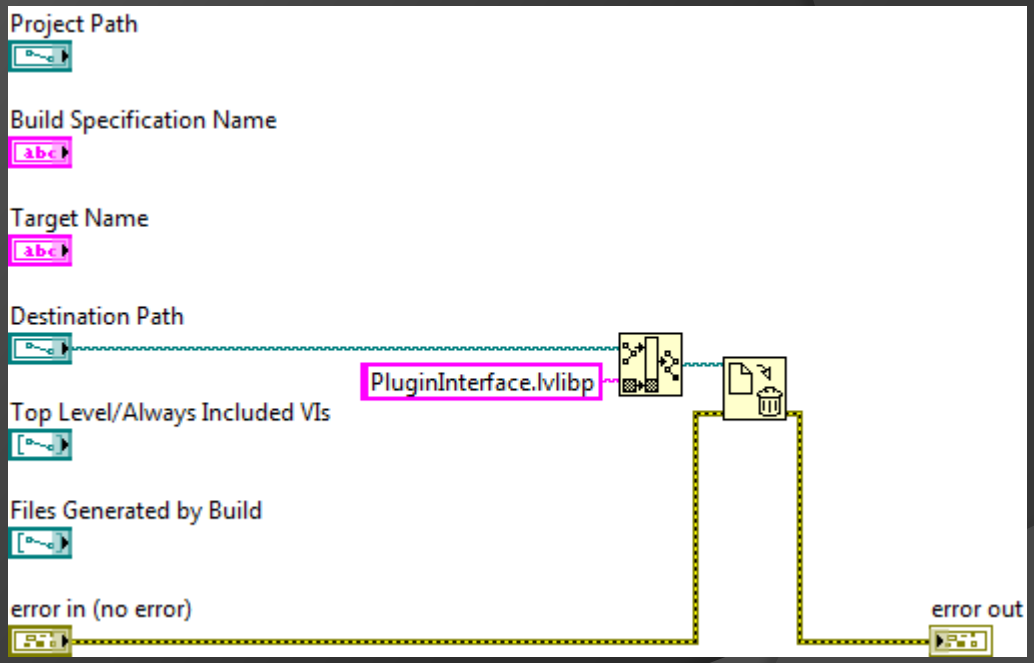
Plug-in Architecture with Packed Project Libraries

Step 9 : Add a Post-Build Action VI that will delete the Interface copy made during the build process.

In LabVIEW



Plugin Post-Build Action.vi



Why delete the copy? – Because we don't need or want an extra copy of the PluginInterface on disk.
Won't this break the Plugin class since it links to the copy it created during the build?
Actually yes, it will break it. You couldn't call Plugin1.lvlibp:Plugin1.lvclass:display.vi directly from another VI. But you can call it if the PluginInterface.lvlibp is already loaded into memory, as it will be in the main application that uses the interface.

Plug-in Architecture with Packed Project Libraries

Step 9 : Add a Post-Build Action VI that will delete the Interface copy made during the build process.

Why delete the copy, doesn't LabVIEW know what it's doing?
Delete it because we don't need or want an extra copy of the PluginInterface on disk.

No, LabVIEW doesn't know what it's doing. In fact, in LabVIEW 2010, the newly built Plugin1.lvlibp actually still depends on the ORIGINAL PluginInterface.lvlibp, not the new copy it just made! That's a bug, fixed in 2011.

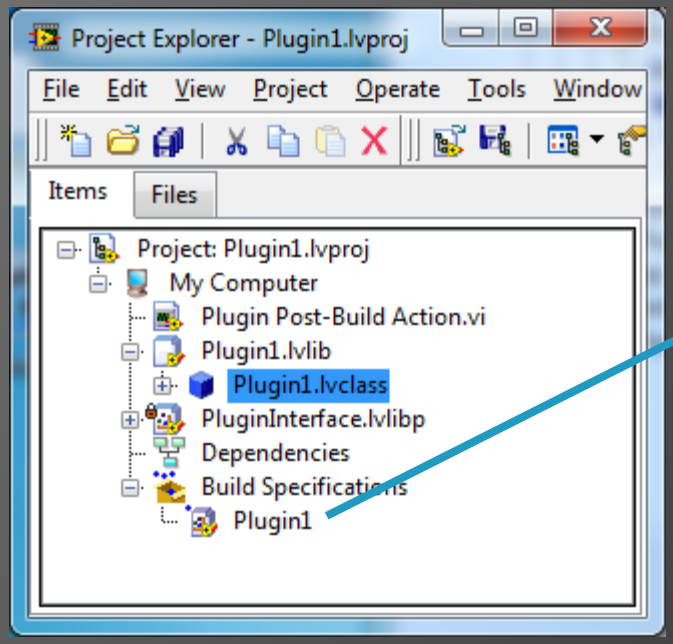
The copy of the lvlibp sometimes makes sense as you'll see when we build an executable of the main application.

Won't this break the Plugin class since it links to the copy it created during the build?
Actually yes, it will break it. You couldn't call Plugin1.lvlibp:Plugin1.lvclass:display.vi directly from another VI if the PluginInterface.lvlibp isn't already loaded in memory (as it will be in the main application that uses the interface).

Plug-in Architecture with Packed Project Libraries

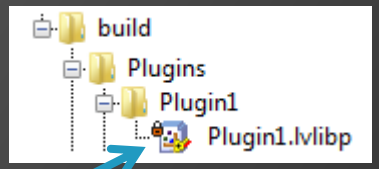
Step 10: Set the destination to a builds directory.

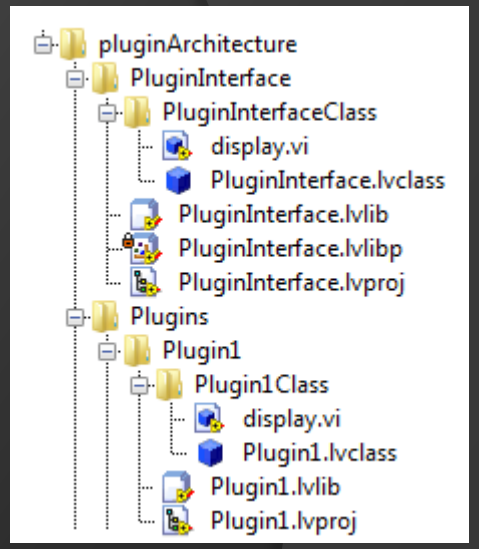
In LabVIEW



Destination

On Disk

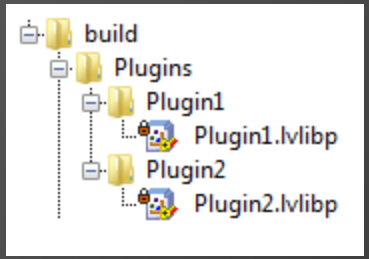




Why did the Interface lvlibp go in the source directory but the plug-in goes to builds?
This is because we're treating the Interface as source code while the plug-in lvlibp will only be loaded in a built executable at Run-time. We'll never need to write code that calls a plugin directly.

Plug-in Architecture with Packed Project Libraries

Step 11 : Create as many plug-ins as you need for now, giving each a new project and built lvlibp.



Why did the Interface lvlibp go in the source directory but the plug-ins go to builds? This is because we're treating the Interface as source code while the plug-in lvlibp will only be loaded in a built executable at Run-time. Using this architecture we'll never need to write code that calls a plugin directly.

Plug-in Architecture with Packed Project Libraries

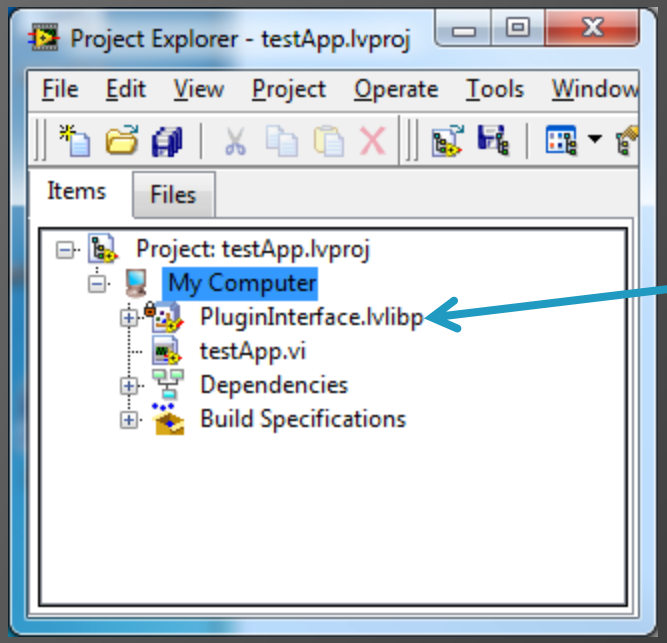
Now it's time to create the main application!

(this could have been done prior to creating the plugins since the main app will only depend on the Plugin Interface)

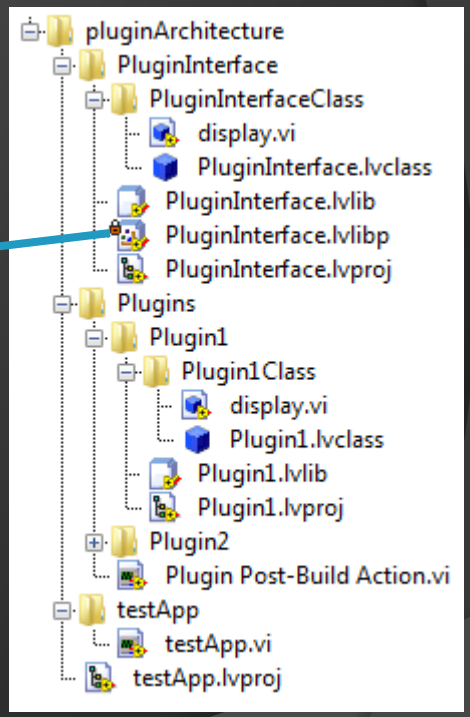
Plug-in Architecture with Packed Project Libraries

Step 12 : Create a new project and a test application. Include the PluginInterface packed proj library.

In LabVIEW



On Disk



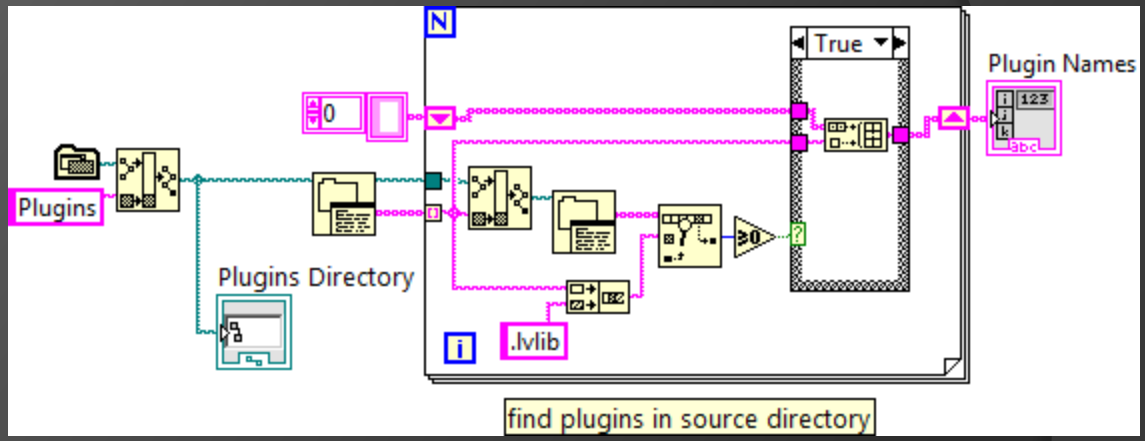
Plug-in Architecture with Packed Project Libraries

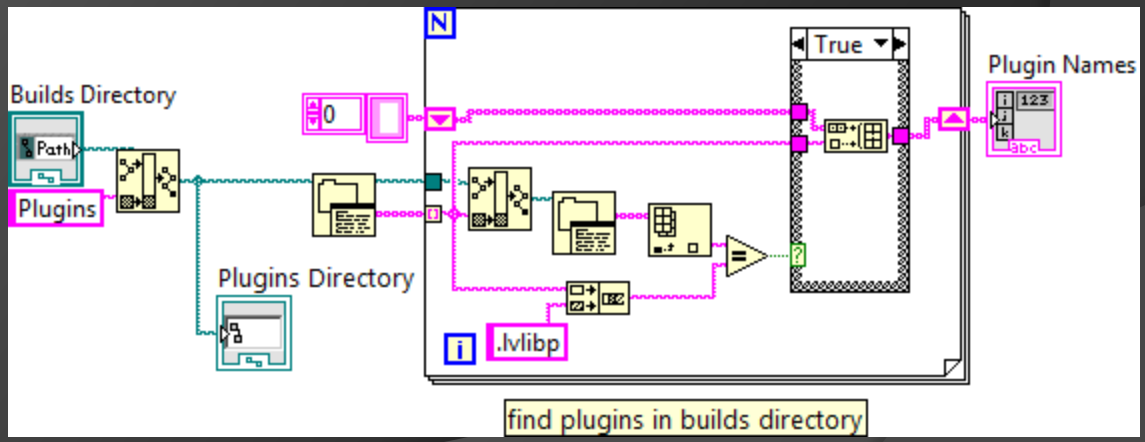
Step 13 : Main application can locate available plug-ins.

Locate all plugin classes in source code.

OR

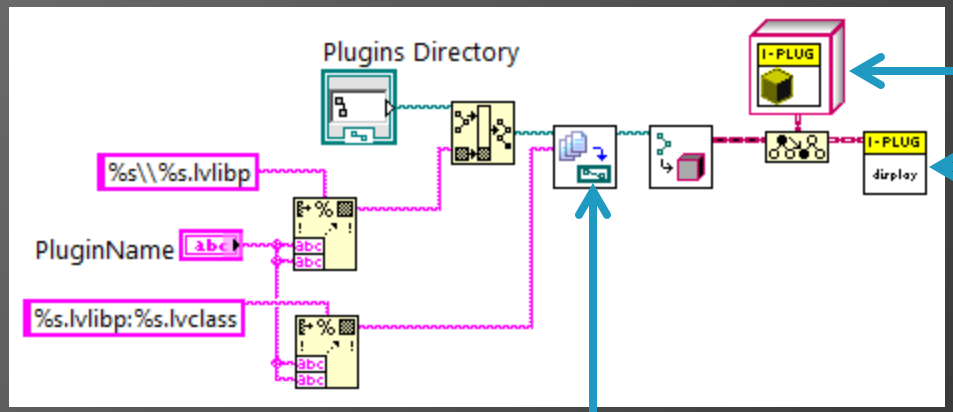
Locate all plugins in builds directory.





Plug-in Architecture with Packed Project Libraries

Step 14 : Main application can then load Plugins dynamically from disk using ‘Get LV Class Default Value’



Typecast to the PluginInterface class and call the interface method.

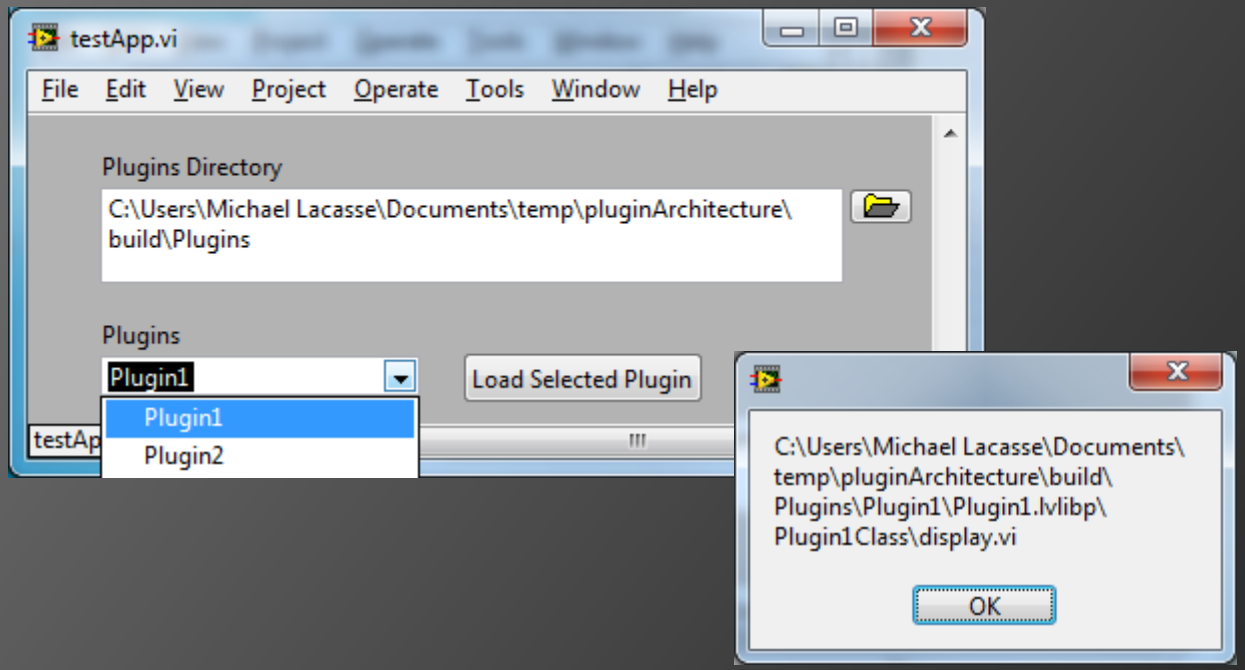
Where did that function come from and what does it do?

It came from the tools pallette 'File I/O>>Advanced File Functions>>Packed Library‘. There are only 2 VIs in this pallette in LabVIEW 2010, but three in LabVIEW 2011. The new function 'Get Exported File Path' in LabVIEW 2011 is precisely what is needed in this situation.

This function locates the requested file within packed library and outputs the path to it, which might otherwise be tricky to form because disk hierarchy is preserved within the lvlibp.

Plug-in Architecture with Packed Project Libraries

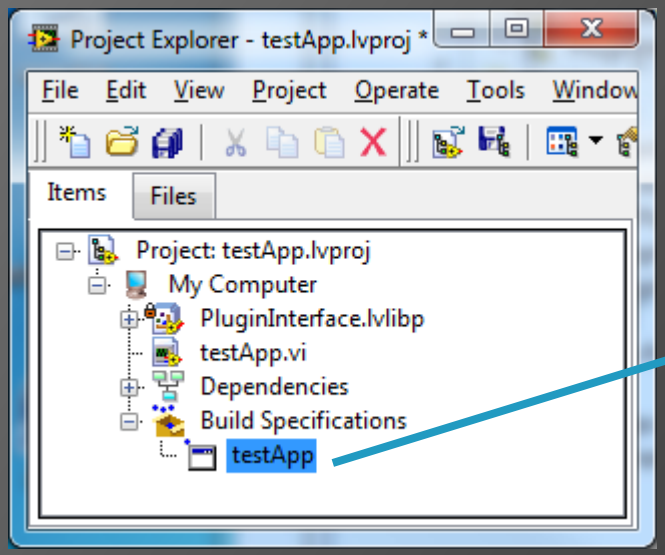
Step 15 : Test Run!



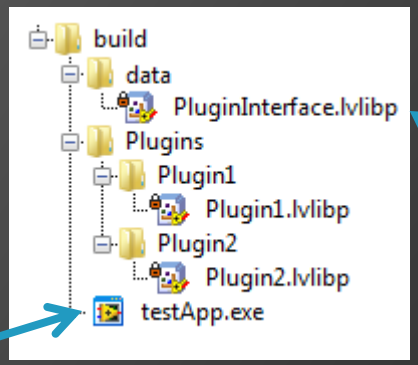
Plug-in Architecture with Packed Project Libraries

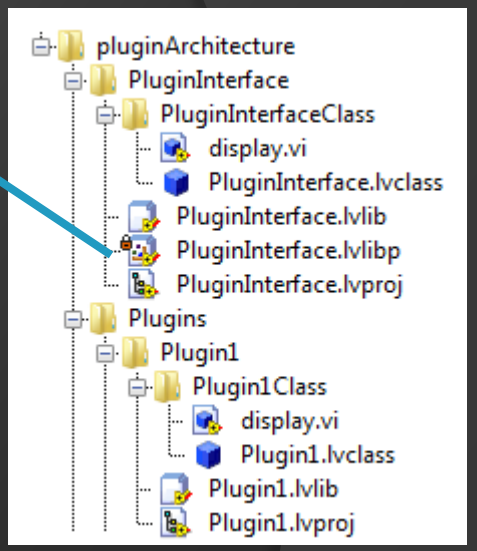
Step 16: Create an executable for the application.

In LabVIEW



On Disk



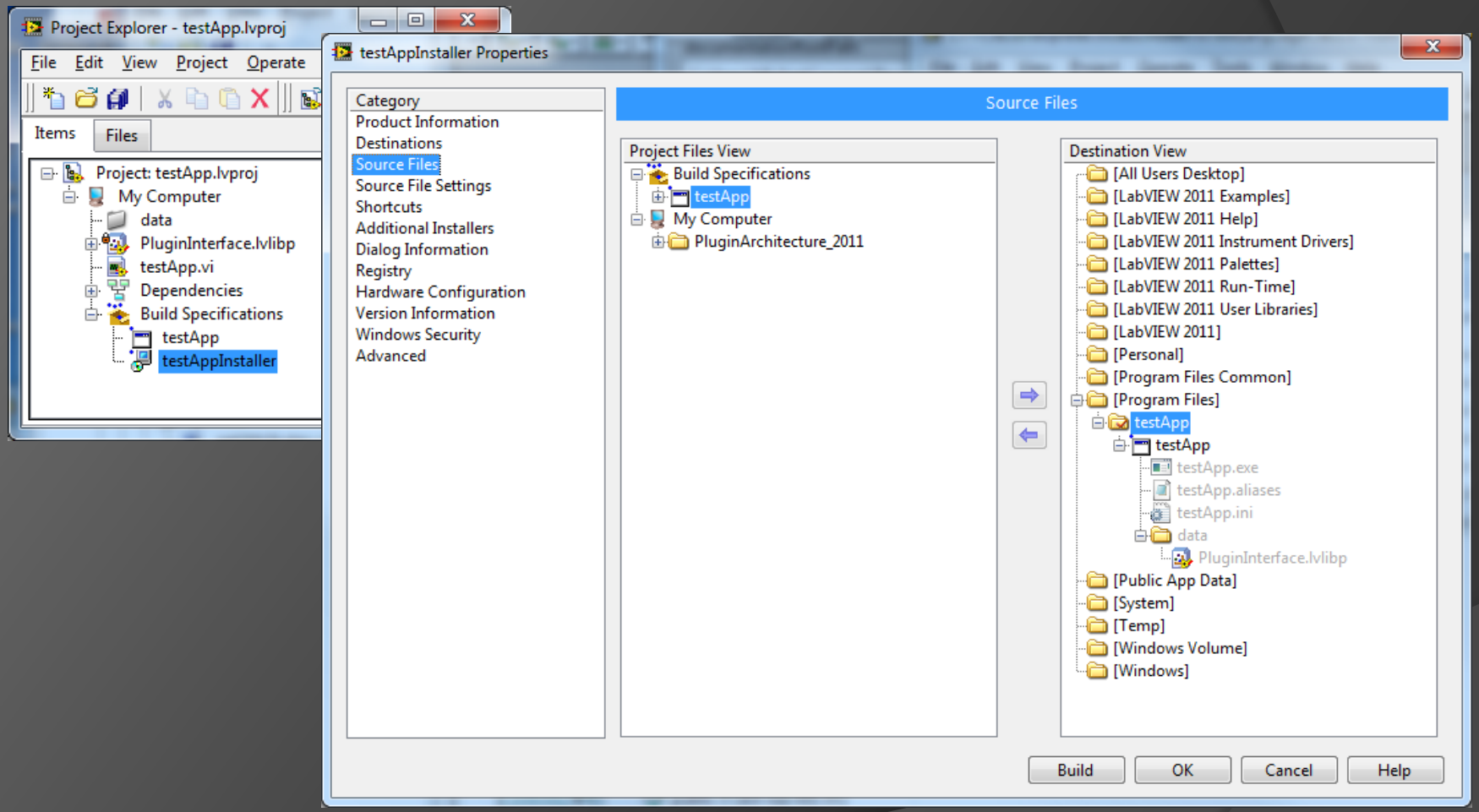


LabVIEW automatically places the PluginInterface in the Support Directory since testApp depends on it.

As long as the path to the Plugins is correctly formed for the run-time environment, the testApp will execute perfectly!

Plug-in Architecture with Packed Project Libraries

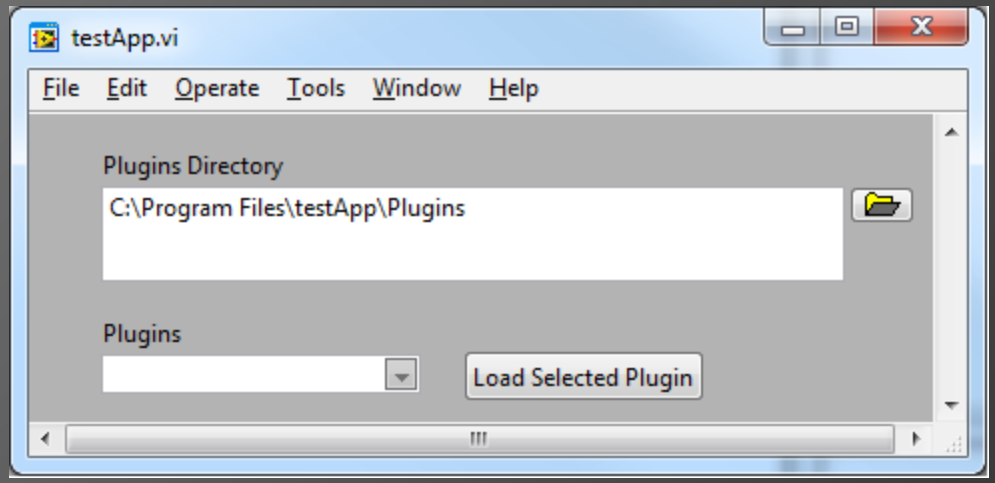
Step 17: Create an installer for the application.



From within the testApp project create an installer that will install the built application.

Plug-in Architecture with Packed Project Libraries

Step 18: Build the installer and run Volume/setup.exe
Run the installed testApp.exe



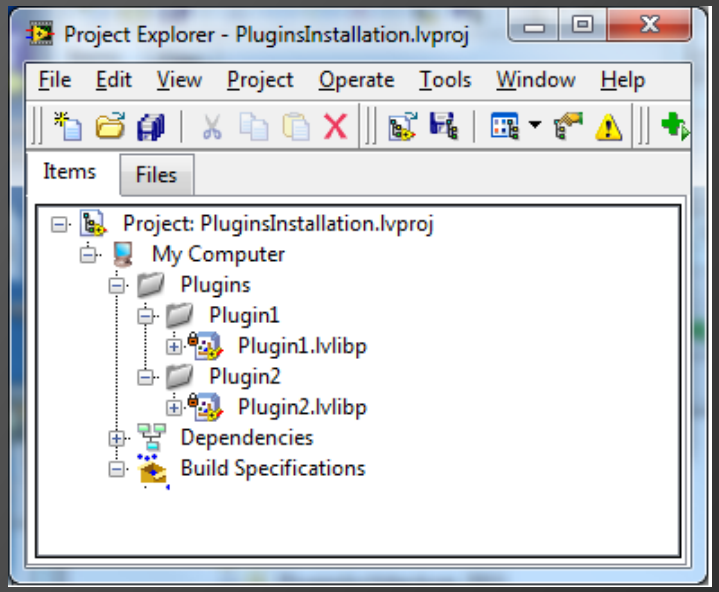
Why are no plugins listed?

No plugins are available to load because none were installed. The beauty of this architecture is that plugins can be added later without modification to the installed application.

Note the 'Plugins Directory' path on the UI. This is where the plugins will need to be installed.

Plug-in Architecture with Packed Project Libraries

Step 19: Create a new project for the plugins installation. Add the ‘builds/Plugins’ directory to the project.

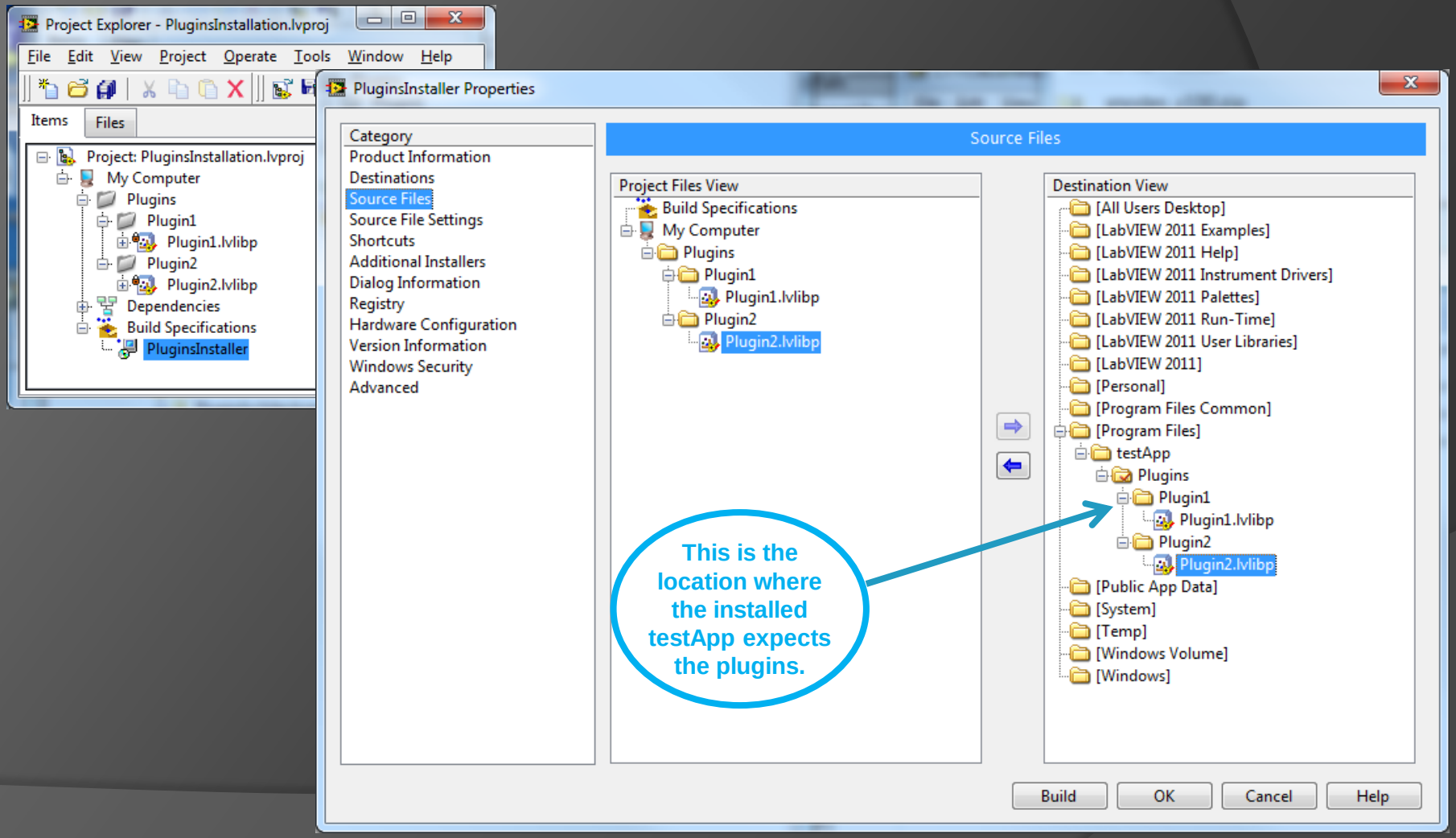


Why a new project? Why not create the plugin installers from the plugin projects?
If the installer is created using the plugin lvlibp build in the plugin project, LabVIEW will attempt to include the PluginInterface.lvlibp in the install. It won’t be found since the post-built VI deleted it. Besides, the application install already included the PluginInterface so the plugin installers don’t need to.

This also allows groups of plugins to be installed simultaneously.

Plug-in Architecture with Packed Project Libraries

Step 20: Create an installer for the Plugins.



Plug-in Architecture with Packed Project Libraries

Step 21: Install the plugins then run the installed testApp.exe