LabVIEW

Advanced Object-Oriented Design Patterns Hands-On

Elijah Kerry, LabVIEW Product Manager

Revision: Q1 2011

NATIONAL INSTRUMENTS™

Application Walk-through........2
Producer consumer Queued State Machine Without Classes........5
Command Pattern (Object Oriented State Machine)........7
Factory Method Design Pattern (Creating a Plugin)........17
Sending Messages to Plugins Using User Events........25
Object-Oriented Hardware Abstraction Layer........29
More Information........33

Solution Walk-through

Goal

Understand and explore a working LabVIEW application that combines multiple design patterns that we will later explore individually in later exercises.

Scenario

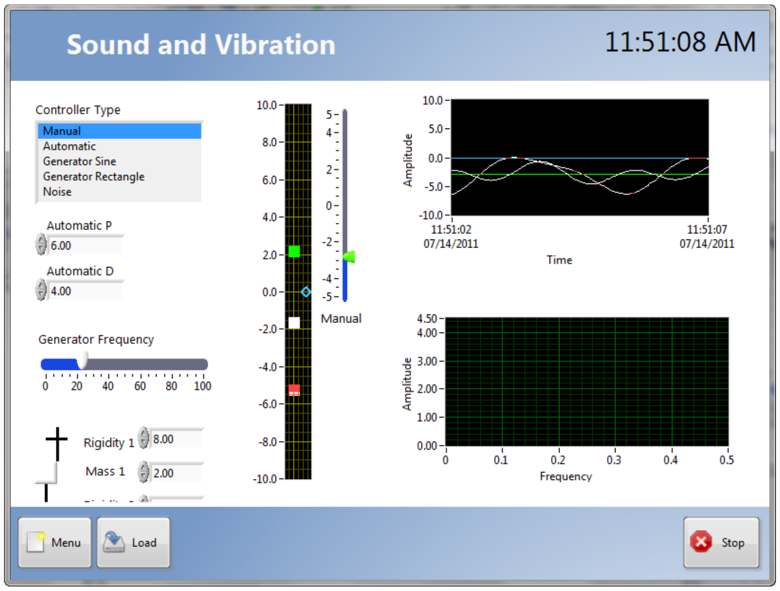
This application is a high-level UI framework that calls into an arbitrary number of plugins. The plugins could be performing any number of tasks, including data acquisition tasks or general visualizations. The architecture is such that separate developers could work on different sections of the application without interfering with one another thanks to established APIs. In this first exercise we will change the set of plugins that are loaded by the framework dynamically.

Concepts Covered

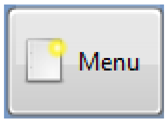
- LabVIEW applications often utilize complex architectures
- Architectures are comprised of multiple design patterns
- Overview of application for demonstration

Setup

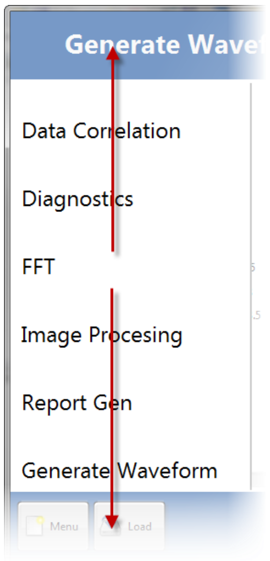
- Open the 'UI Plugin Solution.lvproj' from the UI Plugin Framework Solution Folder
- Launch 'main.vi' and click **Run**
- Ensure that the menu options show and launch correctly
- Press 'Stop' to terminate the application



1. Explore the application's functionality
    a. Click the **Run VI** button
    b. Click the **Menu Button** in the bottom left to show the menu



    c. The list in the menu is populated as a result of plugins that are detected and installed. Later in this exercise you will change the plugins that are running without restarting the application. In the exercise titled 'Factory Method Design Pattern (Creating a Plugin)' you'll see how to create and install a plugin for this application using a standard factory design pattern.
    d. Use the mouse to click and drag up and down on the menu. Point out that they're looking at a reusable UI component that has been built using an XControl and was designed for use on a touch-screen.



    e. Click on **Sound and Vibration**, and explain that this is going to run one on the plugins.
2. Demonstrate that the framework can communicate with plugins
    a. Click the **Menu Button** to open the menu. Explain that the showing plugin receives a message when the menu appears, giving the plugin the option of performing a task. In this case, opening the menu pauses the acquisition to conserve resources. Clicking the button again closes the menu and send a message to the plugin, which then opts to resume the acquisition.
    b. As we will see later on, all of these plugins are complete atomic, self contained applications that can be developed and modified independent of the calling framework.
3. Load different plugins during run-time
    a. Click the **Load** button in the bottom left corner to open a file dialog that allows you to select the location on disk of new plugins. The default location should be the root where all of these demos are located.
    b. Navigate into the folder entitled '**Plugin Libraries.'** Choose from any of the other available libraries and select **Current Folder** in the bottom right corner.

c. Notice that the responsiveness of the UI is not affected while the new plugins are being loaded into memory.

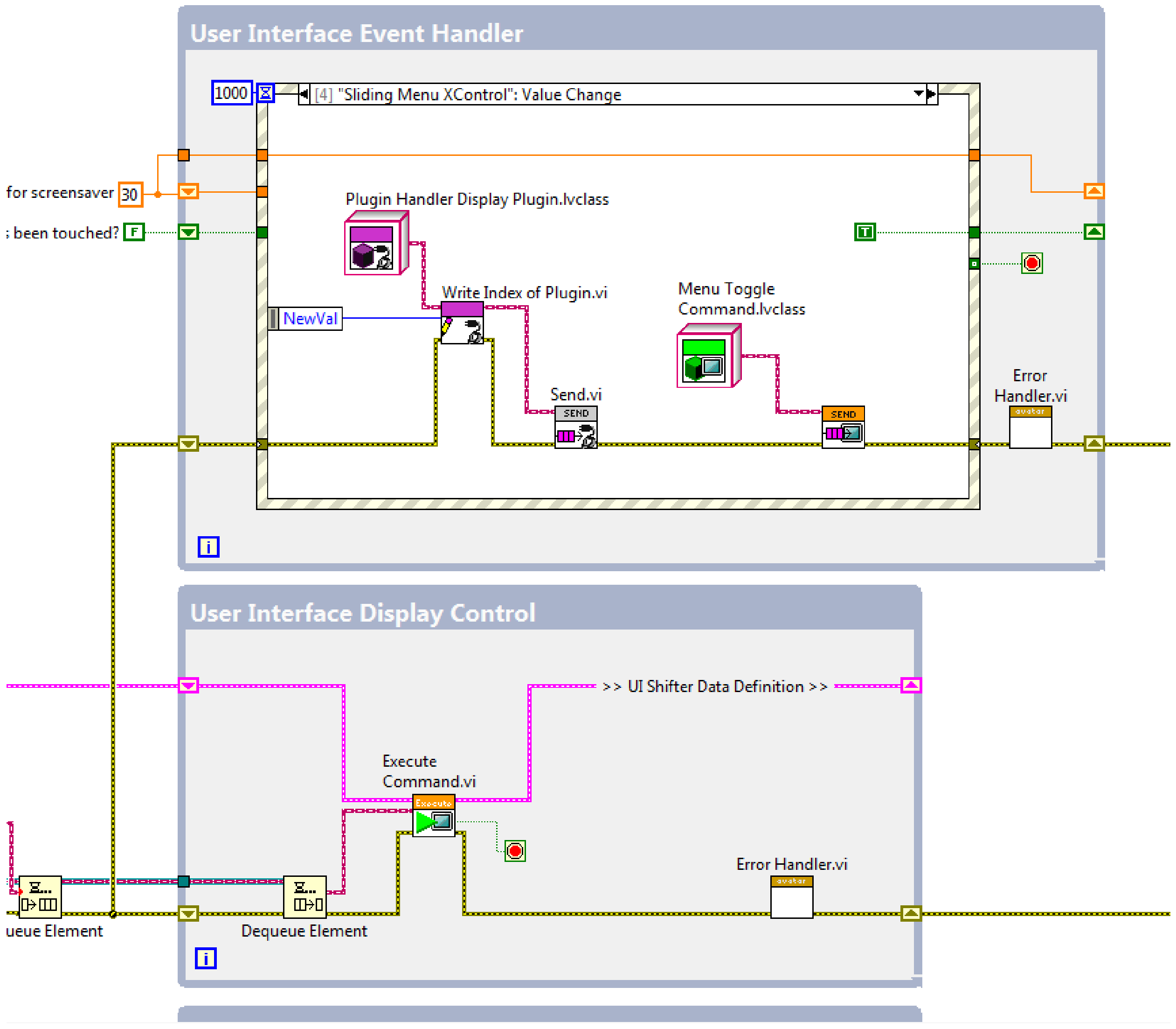
d. After a few moments, a different set of plugins should be shown in the menu.

4. Show the Block Diagram

   a. Stop the application by clicking 'Stop'

   b. Switch to the block-diagram by pressing CTRL+E

   c. This application communicates amongst multiple parallel processes using a combination of producer consumer loops, queued state machines and a factory pattern. This code is built using LabVIEW objects, so some elements may not be immediately recognized, but the same underlying principles apply and we'll look at both an OO and non-OO implementation.


User Interface Event Handler
1000
[4] "Sliding Menu XControl": Value Change
for screensaver 30
; been touched? F
Plugin Handler Display Plugin.lvclass
NewVal
Write Index of Plugin.vi
Menu Toggle Command.lvclass
Send.vi
SEND
Error Handler.vi
User Interface Display Control
>> UI Shifter Data Definition >>
Execute Command.vi
Execute
Error Handler.vi
ueue Element
Dequeue Element

Producer consumer Queued State Machine Without Classes

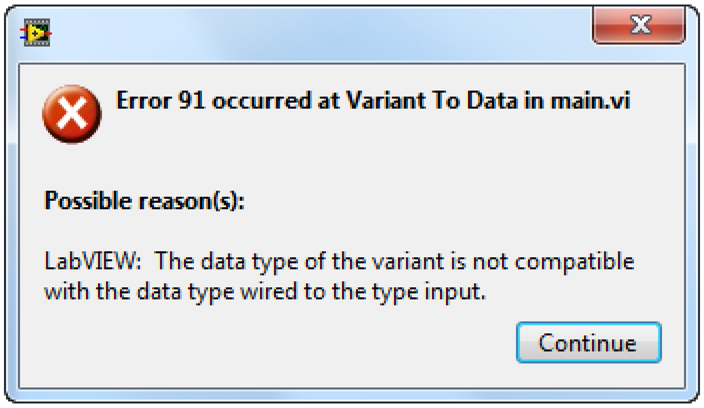
Goal

This exercise will illustrate an alternate implementation of the UI Framework that does not use classes. It's a simple and quick demo to bridge to the oo-based example and demonstrate some of the potential drawbacks of the enum/variant method to passing data.

Scenario

Clicking on a menu item invokes the 'toggle menu' item command, but enqueues the wrong data type, creating a run-time error.

1. Introduce alternative implementation without classes
    a. Open the Project entitled ‘UI Framework (PC-QSM).lvproj’ in the ‘UI Framework (PC-QSM)’ Folder.
    b. Open **main.vi** and switch to the block diagram, which uses messages built with enums and variants. The application is functionally equivalent, but it poses some additional risk due to the lack of data encapsulation.
    c. This framework will call into and run the same plugins as the one explored in the previous exercise.
    d. Switch to the front panel and run the application to illustrate that it works.
2. Generate a run-time error
    a. Open the menu by clicking on the button in the bottom left, this is enqueing a message to the UI Display control state machine to run the toggle menu command.
    b. This command also gets run when a menu item is clicked on; however, the message to toggle the menu has been composed incorrectly in the “Sliding Menu XControl”: Value Change case. Instead of a Boolean data type, it uses an integer. As a result, clicking on a menu item will send an incorrect datatype at run-time, which cases an error.


Error 91 occurred at Variant To Data in main.vi

Possible reason(s):

LabVIEW: The data type of the variant is not compatible with the data type wired to the type input.

    c. This can be avoided by encapsulating data with classes.

Command Pattern (Object Oriented State Machine)

Goal

LabVIEW Classes provide an alternative implementation of what is typically referred to as a queue-driven state machine for dispatching and executing commands. By representing the commands with a class hierarchy, we can utilize encapsulation to ensure that data types are enforced at compilation and make it possible to add new commands without modifying the calling code.

Scenario

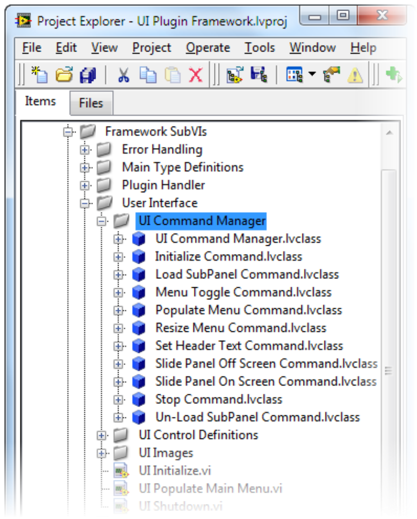
In the first part of this exercise, we will execute an existing command by invoking the send method.

Concepts Covered

- Class hierarchy window
- Inheritance
- Dynamic Dispatch
- Accessor Methods

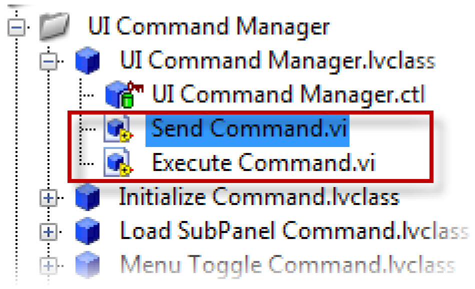
PART ONE: SENDING A MESSAGE

3. Explore the command hierarchies for both the UI and Plugin State Machines
   a. Navigate to the folder 'UI Framework Exercises, and open 'UI Framework Exercises.lvproj.'
   b. This application has two separate class hierarchies that are used in separate command patterns. Expand 'Framework SubVIs > User Interface > UI Command Manager' to show the class hierarchy for the UI Command state machine.


Project Explorer - UI Plugin Framework.lvproj
File Edit View Project Operate Tools Window Help
Items Files

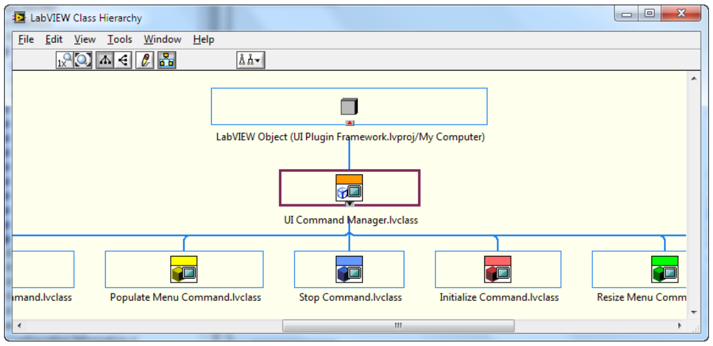
Framework SubVIs
Error Handling
Main Type Definitions
Plugin Handler
User Interface
UI Command Manager
UI Command Manager.lvclass
Initialize Command.lvclass
Load SubPanel Command.lvclass
Menu Toggle Command.lvclass
Populate Menu Command.lvclass
Resize Menu Command.lvclass
Set Header Text Command.lvclass
Slide Panel Off Screen Command.lvclass
Slide Panel On Screen Command.lvclass
Stop Command.lvclass
Un-Load SubPanel Command.lvclass
UI Control Definitions
UI Images
UI Initialize.vi
UI Populate Main Menu.vi
UI Shutdown.vi

   a. Expand the contents of 'UI Command Manager.lvclass.' This is the parent class for all commands for the UI State Machine. It contains two methods and no data. The methods are **Send Command.vi** and **Execute Command.vi**. 'Send Command.vi' is a static method we will use to enqueue child classes and we will override 'Execute Command.vi' to customize the behavior of all the child classes.

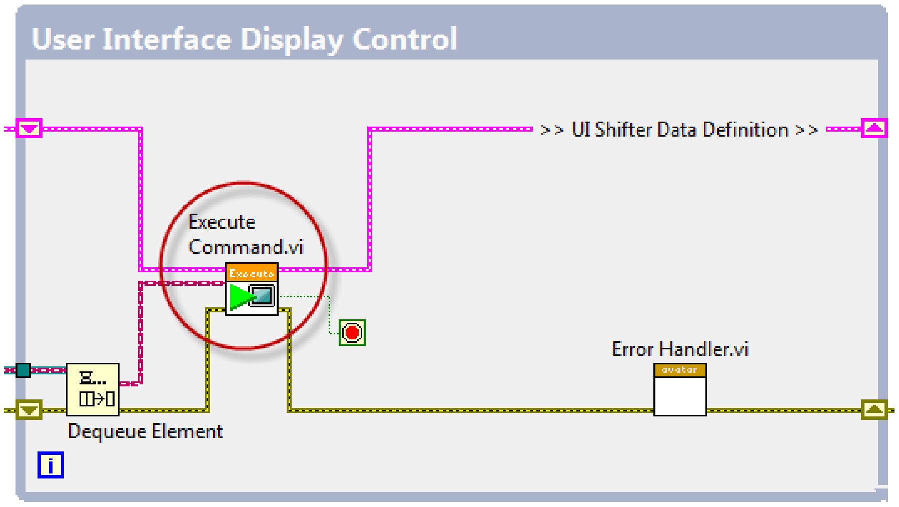


UI Command Manager
UI Command Manager.lvclass
UI Command Manager.ctl
Send Command.vi
Execute Command.vi
Initialize Command.lvclass
Load SubPanel Command.lvclass
Menu Toggle Command.lvclass

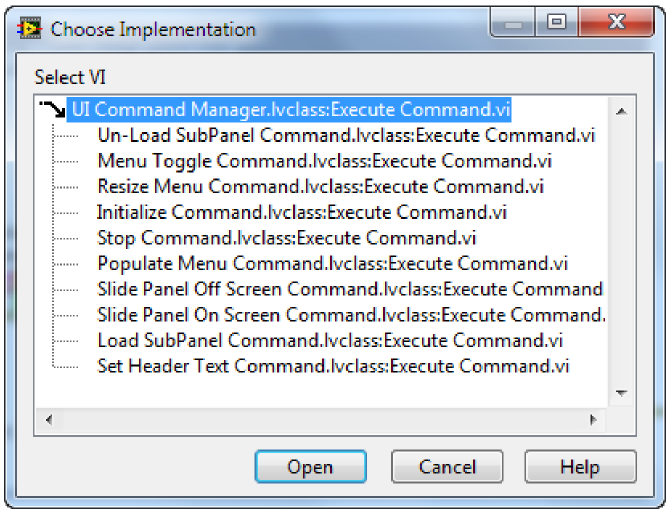
   b. Right-click on **UI Command Manager.lvclass** and select 'Show Class Hierarchy' to illustrate that this is the parent class and that all the other commands inherit and from it.


LabVIEW Class Hierarchy
File Edit View Tools Window Help
LabVIEW Object (UI Plugin Framework.lvproj/My Computer)
UI Command Manager.lvclass
mand.lvclass
Populate Menu Command.lvclass
Stop Command.lvclass
Initialize Command.lvclass
Resize Menu Comm

c. Navigate to the block diagram of **main.vi**. The loop labeled ‘User Interface Display Control’ de-queues objects from this class hierarchy and runs the appropriate version of **Execute Command.vi.**


User Interface Display Control
>> UI Shifter Data Definition >>
Execute Command.vi
Execute
Error Handler.vi
Dequeue Element

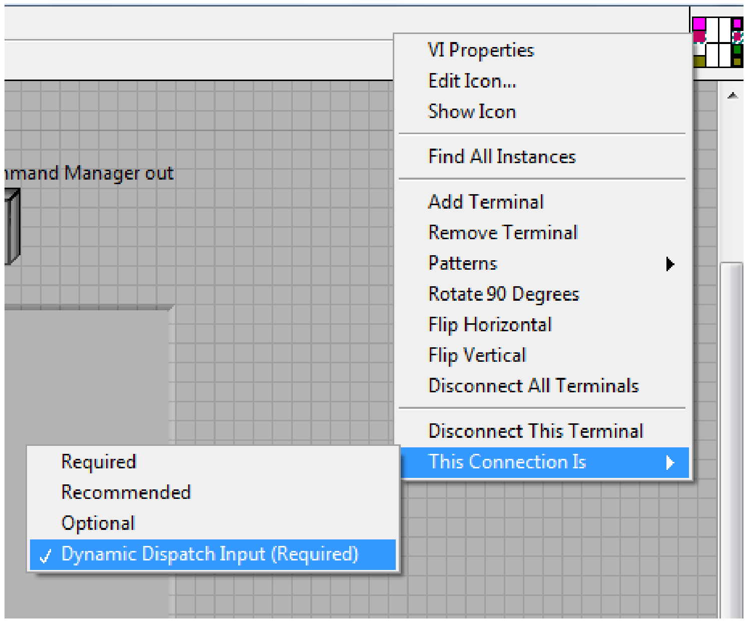
d. Double-click on **Execute Command.vi**. Unlike a normal subVI, this subVI is dynamically dispatched, which means the run-time engine decides which version of ‘execute command.vi’ to run based on the object that is sent to the dynamic terminal. As a result, double-clicking on the VI brings up a menu that lists all the instances of ‘Execute Command.vi’ that could potentially be run during execution.


Choose Implementation

Select VI

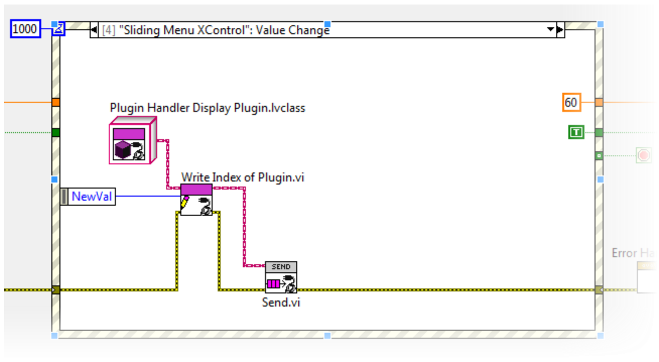
UI Command Manager.lvclass:Execute Command.vi
Un-Load SubPanel Command.lvclass:Execute Command.vi
Menu Toggle Command.lvclass:Execute Command.vi
Resize Menu Command.lvclass:Execute Command.vi
Initialize Command.lvclass:Execute Command.vi
Stop Command.lvclass:Execute Command.vi
Populate Menu Command.lvclass:Execute Command.vi
Slide Panel Off Screen Command.lvclass:Execute Command
Slide Panel On Screen Command.lvclass:Execute Command.
Load SubPanel Command.lvclass:Execute Command.vi
Set Header Text Command.lvclass:Execute Command.vi

a. Double-click on the first item in the list, which is the copy of 'Execute Command.vi' belonging to the parent, UI Command Manager.lvclass. View the connector pane of this icon and right-click on the input terminal for the object. When a VI belongs to a class hierarchy, you can specific that the input terminal for the class is a Dynamic Dispatch Input, which enables children to override this VI. Navigate to the options for 'This Connection Is' and ensure that 'Dynamic Dispatch Input' has been selected. Verify that the input terminal is set to **Dynamic Dispatch Input (Required).**



4. Send a command to 'Toggle Menu' when a user selects a menu item
   a. Return to the front panel of **main.vi** and run the application.
   b. Click on the **menu button** to expand the list of plugins. Clicking on one of these items displays that plugin's front panel in the subPanel, but we currently have to go and manually click the menu button to collapse this list. To automatically close the menu after a user has selected a plugin, we're going to dispatch an instance of 'Menu Toggle Command.lvclass' to the UI Interface Display Control.
   c. Stop the application and switch to the block diagram.

d. Navigate to the case in the Event Structure labeled “**Sliding Menu XControl”: Value Change.”** This case currently handles a user selecting an item from the menu and sends the command to display the appropriate plugin based on the updated value of the control.


1000
[4] "Sliding Menu XControl": Value Change
Plugin Handler Display Plugin.lvclass
Write Index of Plugin.vi
NewVal
60
SEND
Send.vi
Error H

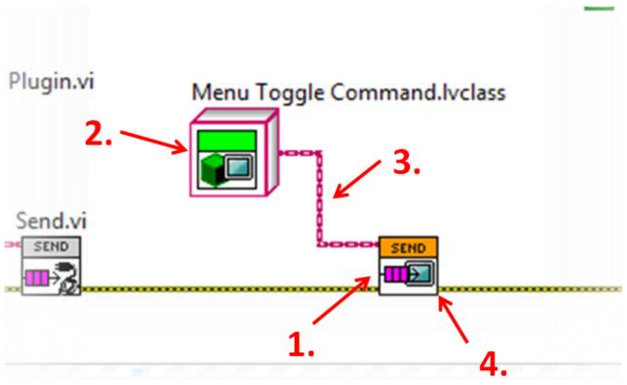
e. Open the Project Explorer and navigate to ‘Framework SubVIs > User Interface > UI Command Manager > UI Command Manager.lvclass.’ We will be using ‘send command.vi’ to enqueue an instance of ‘Menu Toggle Command.lvclass.’ Perform the following steps to implement the new command as shown:

*1. Drag and drop a copy of **Send Command.vi** to the block diagram*

*2. Drag and drop a copy of **Menu Toggle Command.lvclass** to the block diagram*

*3. Connect the wire from the object to the input terminal of **Send Command.vi***

4. Connect the error wires



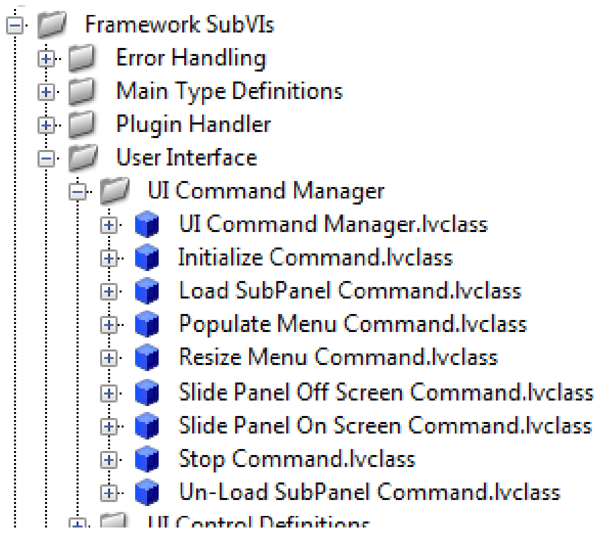
f. Return to the front panel of **main.vi** and click the run button. Expand the menu and show that selecting an item now closes the menu.

PART TWO: CREATING A NEW COMMAND

1. Create a new command to set the header of the display to show the name of the plugin that is running.

Note: It's important to realize that this task will be accomplished without modifying the top level VI

a. In the LabVIEW Project Explorer, navigate to and expand 'Framework SubVIs > User interface.' The list of classes shown represents the commands that have currently been implemented.

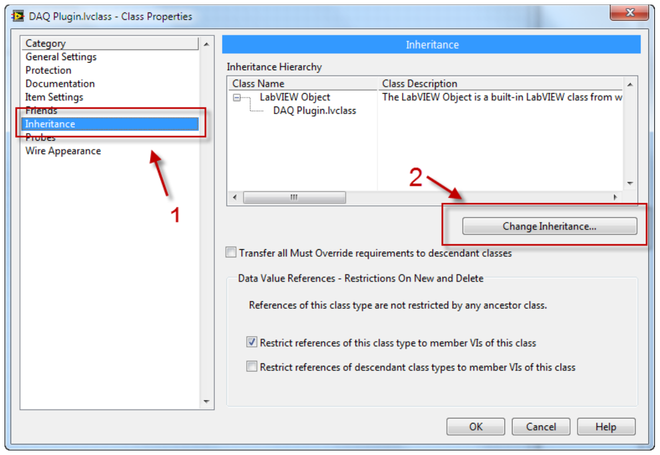


```
Framework SubVIs
    Error Handling
    Main Type Definitions
    Plugin Handler
    User Interface
        UI Command Manager
            UI Command Manager.lvclass
            Initialize Command.lvclass
            Load SubPanel Command.lvclass
            Populate Menu Command.lvclass
            Resize Menu Command.lvclass
            Slide Panel Off Screen Command.lvclass
            Slide Panel On Screen Command.lvclass
            Stop Command.lvclass
            Un-Load SubPanel Command.lvclass
```

b. Right click on the 'User Interface' Folder and select 'New > Class.' When prompted, select the title 'Set Header Text Command' and click **OK**. The new class will appear in the Project Explorer.

c. To function properly as a command, this class will need to inherit the capabilities of the 'UI Command Manager' class. To specify this relationship, right click 'Set Header TextCommand.lvclass' and select **Properties.**

d. In the Properties Dialog that appears for this class, select **Inheritance** from the list of Categories. On the right side of the screen, select **Change Inheritance**.



DAQ Plugin.lvclass - Class Properties

Inheritance Hierarchy

Class Name	Class Description
LabVIEW Object	The LabVIEW Object is a built-in LabVIEW class from w
DAQ Plugin.lvclass	

☐ Transfer all Must Override requirements to descendant classes

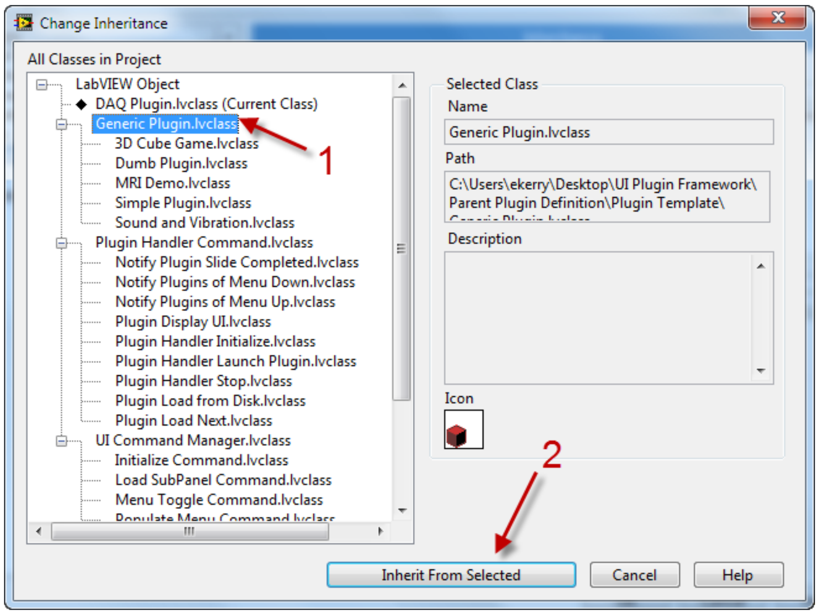
Data Value References - Restrictions On New and Delete

References of this class type are not restricted by any ancestor class.

☑ Restrict references of this class type to member VIs of this class

☐ Restrict references of descendant class types to member VIs of this class

e. The 'Change Inheritance' dialog will appear, which lists all of the Classes currently loaded in the Project Explorer. Find and select 'UI Command Manager.lvclass,' and click **Inherit from Selected.**



All Classes in Project:

- LabVIEW Object
  - DAQ Plugin.lvclass (Current Class)
  - Generic Plugin.lvclass
    - 3D Cube Game.lvclass
    - Dumb Plugin.lvclass
    - MRI Demo.lvclass
    - Simple Plugin.lvclass
    - Sound and Vibration.lvclass
  - Plugin Handler Command.lvclass
    - Notify Plugin Slide Completed.lvclass
    - Notify Plugins of Menu Down.lvclass
    - Notify Plugins of Menu Up.lvclass
    - Plugin Display UI.lvclass
    - Plugin Handler Initialize.lvclass
    - Plugin Handler Launch Plugin.lvclass
    - Plugin Handler Stop.lvclass
    - Plugin Load from Disk.lvclass
    - Plugin Load Next.lvclass
  - UI Command Manager.lvclass
    - Initialize Command.lvclass
    - Load SubPanel Command.lvclass
    - Menu Toggle Command.lvclass

Name: Generic Plugin.lvclass

Path: C:\Users\ekerry\Desktop\UI Plugin Framework\Parent Plugin Definition\Plugin Template\

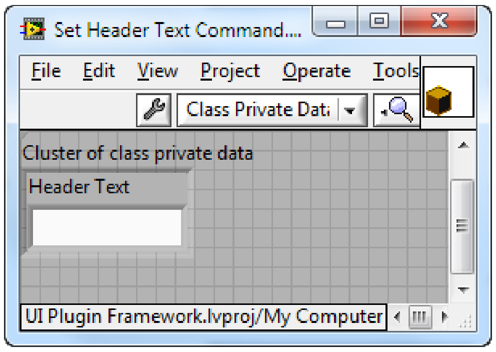
Description

Icon

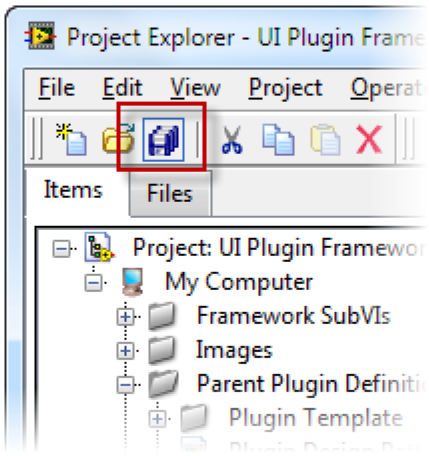
f. The Change Inheritance dialog will close, returning to the Class Properties dialog. Click **OK** to save the changes.

g. The new class will contain a control where the data members can be defined. Double click on the control, 'Set Header Text Command.ctl'

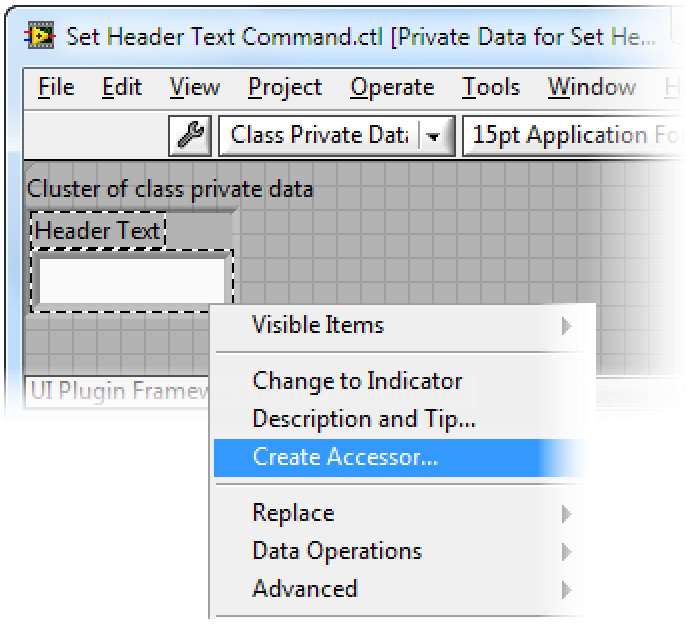
h. In the data cluster for the class, put a new string control and name it 'Header Text.' This will be the private data that only this command has access to.


Set Header Text Command....
File
Edit
View
Project
Operate
Tools
Class Private Data
Cluster of class private data
Header Text
UI Plugin Framework.lvproj/My Computer

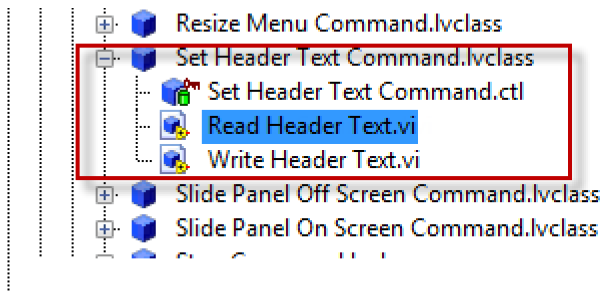
i. Before proceeding, ensure that the new 'Set Header Text Command.lvclass' class has been saved to disk. To do this, return to the Project Explorer and click the toolbar icon for 'Save All,' as shown below:



j. Return to the front panel for 'Set Header Text Command.ctl.' Right click on the border of the string control and select the option to **Create Accessor**, as shown. This will generate wrapper VIs that allow us to set and read this control value.



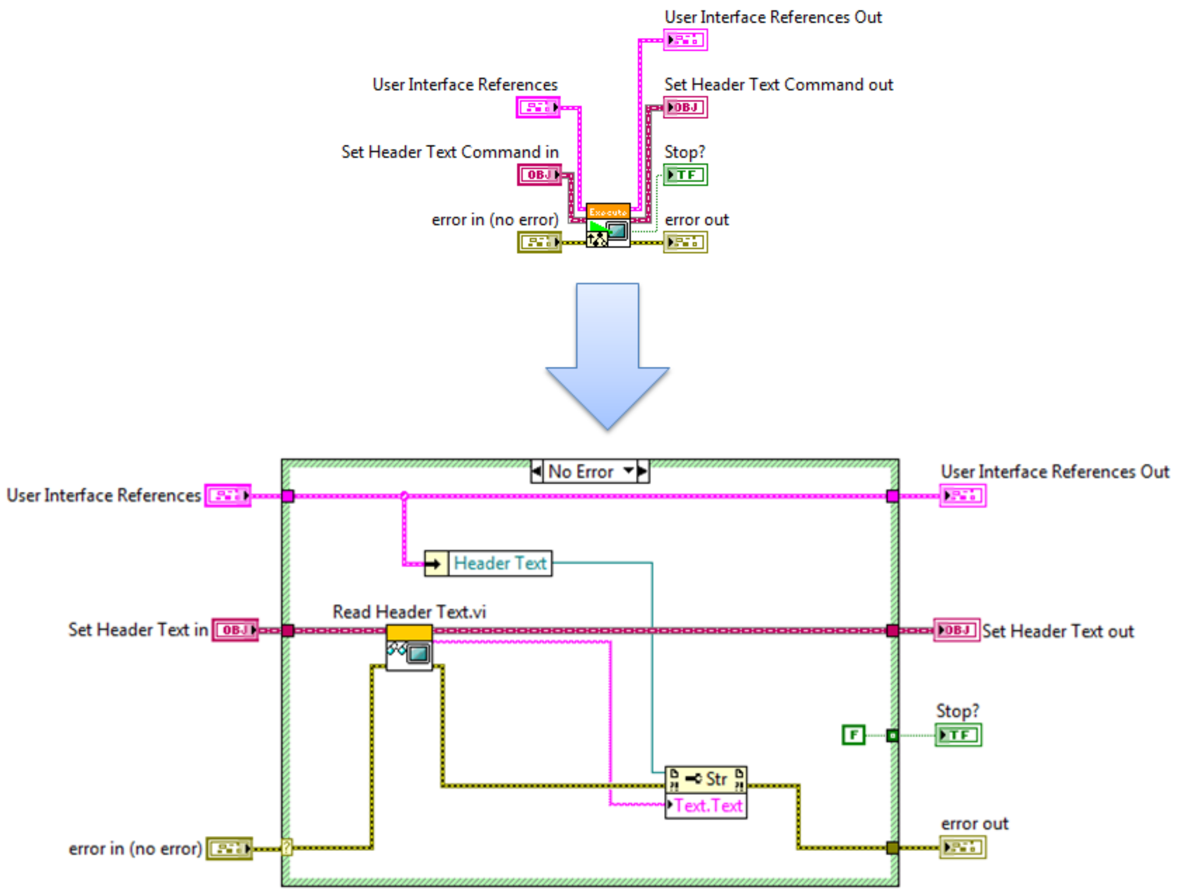
k. A configuration dialog will appear, prompting us to specify which methods we want to create. Select 'Read and Write' from the drop-down, ensure 'Create static accessor' is selected, and press **OK**. Two new VIs should now appear in the Project Explorer as shown:


Resize Menu Command.lvclass
Set Header Text Command.lvclass
Set Header Text Command.ctl
Read Header Text.vi
Write Header Text.vi
Slide Panel Off Screen Command.lvclass
Slide Panel On Screen Command.lvclass

l. The final step is to create the new copy of Execute.vi, which will be run by the state machine when this command is called. Right click on the 'Set Header Text Command.lvclass' and select 'New > VI for Override.'

m. In the dialog that appears, select 'Execute.vi.' LabVIEW will generate a VI as shown at the top of the image below. Make the necessary changes to implement the state's functionality. An example is shown at the bottom of the image. The following are the recommended steps to build it:

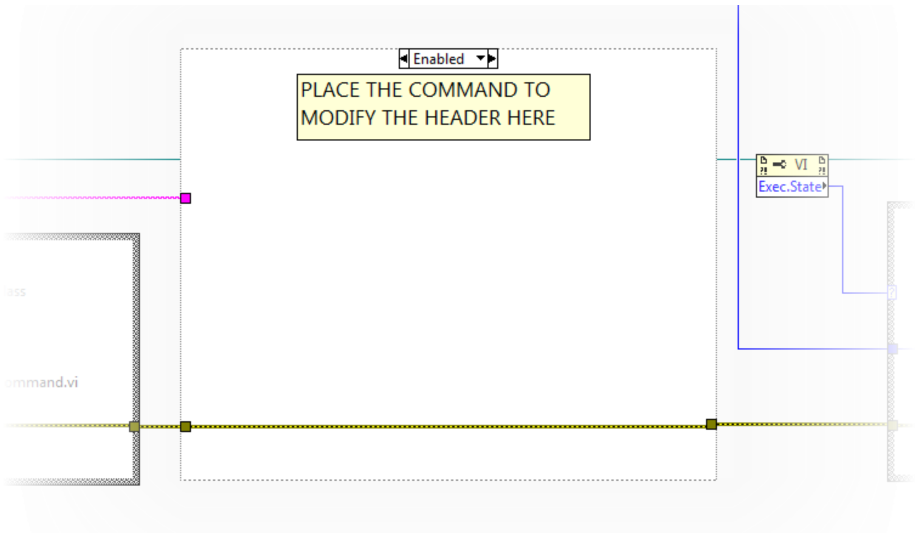
   i. Delete the subVI in the generated code (parent copy of Execute.vi)
   ii. Reconnect the User Interfaces References Wire
   iii. Place an 'Unbundle by Name' and select 'Header text' reference
   iv. Place the accessor created in previous steps to read the header text, 'Read Header Text.vi'
   v. Wire a False constant into the Stop Indicator
   vi. Wire the reference to the header text to the property node
   vii. Wire the output of 'Read Header Text.vi' to the value of the property node


User Interface References Out
User Interface References
Set Header Text Command out
Set Header Text Command in
Stop?
error in (no error)
error out
Execute
No Error
User Interface References
User Interface References Out
Header Text
Read Header Text.vi
Set Header Text in
Set Header Text out
Stop?
F
Str
Text.Text
error in (no error)
error out

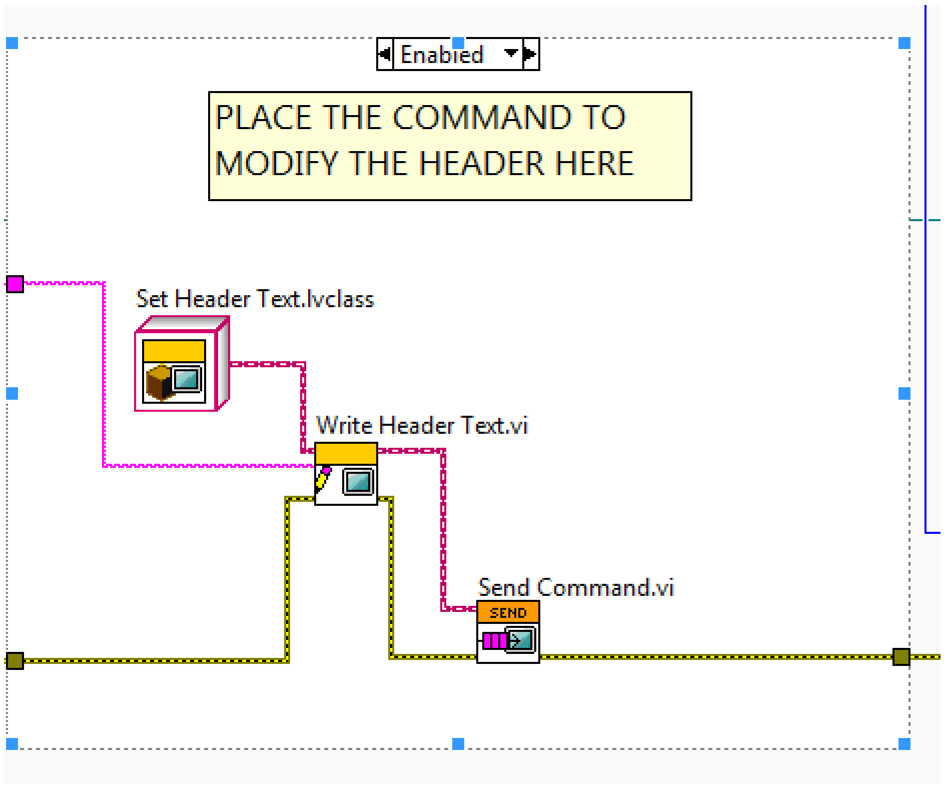
n. Once again, save everything and close the windows that are currently open.

o. Finally, we must call this command from the appropriate location as we did in the earlier example for the toggle menu command. In this case, we will call it when the plugin handler receives a request to load a plugin. Navigate to 'Framework SubVIs > Plugin Handler > Plugin Display UI > Plugin Display UI.lvclass > Execute.vi.'

p. On the block diagram, scroll to the location where this command will be sent from, as shown below:



q. Build the command by implementing the following code



r. Save all the changes and open and run **main.vi**. When a plugin is clicked, the header will now be populated with the same string that is used to identify the plugin in the menu.

Factory Method Design Pattern (Creating a Plugin)

Goal

'Factory Method Pattern' refers to a well-know object-oriented pattern for creating objects at run-time. These objects are typically derived from a parent, and they extend the functionality for the specific object through the use of inheritance.

Scenario

We have an application that runs and displays applications that are loaded at run-time as plugins. In this case, the plugins typically consist of a user interface that allows the user to perform an arbitrary and independent task. After running the plugin framework, the menu displays the names of all the plugins that were found on disk. Clicking on any item displays the front panel and sends a message to the plugin that its user interface is showing. In this scenario, we want to take a simple data acquisition application and turn it into a plugin that this application can run and load.

Description

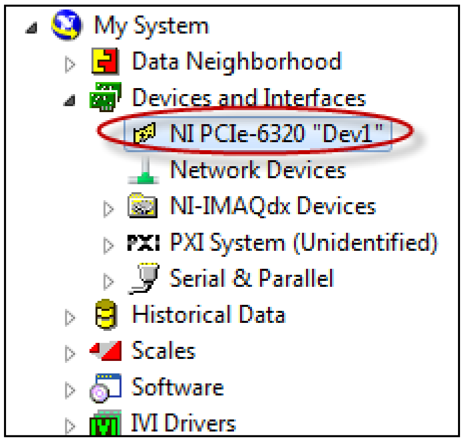
The VI that we want to use in the plugin has already been created. In this demonstration, the VI will be added to a new object we create such that it will be launched by the framework. The framework will send user events to the plugin for communication purposes. After registering these events with the plugin, we will be able to understand and act upon these events.

Concepts Covered

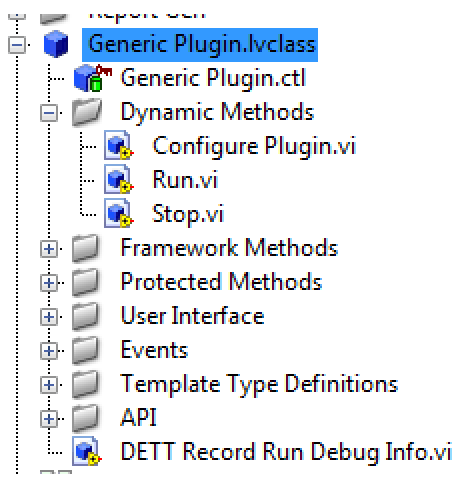
- Dynamically loading classes
- Defining and creating a new child of a class hierarchy

Setup

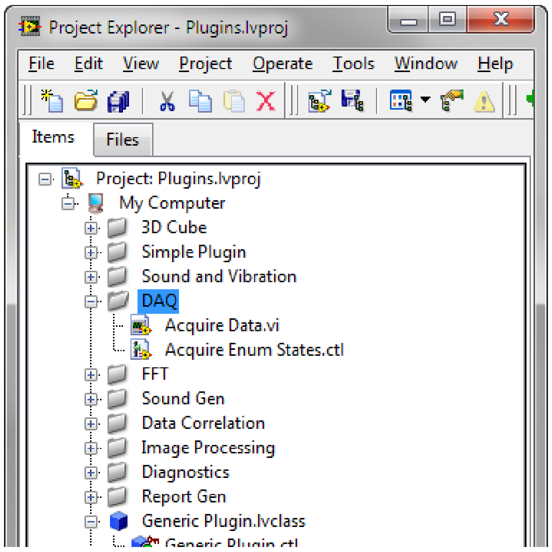
- The plugin that we will require uses the DAQmx API. Make sure it is installed
- Ensure a device (Dev0) has been simulated and appears as shown below in Measurement Explorer.



1. Introduce the Plugin Template Class
    a. From the root of this hands-on, open 'Project Libraries> Measurement Tasks' and launch the Plugins.lvproj to see the Project Explorer containing all the plugins.
    b. Navigate to 'Parent Plugin Definition > Plugin Template > Generic Plugin.'
    c. Right-click on 'Generic Plugin.lvclass' and select **Show Class Hierarchy**. This illustrates that all the plugins currently running in the framework are children of this generic plugin definition. Close this window.
    d. The class, 'Generic Plugin.lvclass' contains all the private and public methods that are called by the framework to interact with a plugin. Some of them can be overridden to customize the behavior of a plugin, such as the run and stop functionality.



2. Introduce the application that we will be creating a plugin from
    a. A simple data acquisition application is included in the Plugins project for the sake of creating a new plugin. In the Project Explorer, navigate to the folder labeled 'DAQ' and you should see a vi entitled 'Acquire Data.vi'

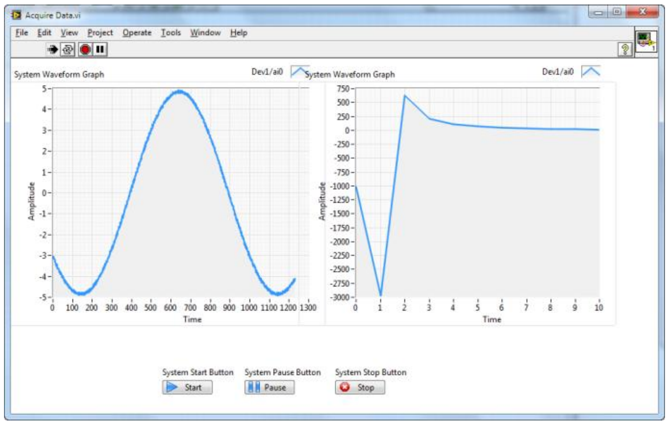


    b. Close the UI Plugin Framework Project and any other open projects that call these classes (this will un-lock the plugin classes so that you can modify inheritance)

c. In the Plugins project, open the folder ‘DAQ’
d. Open the application and click the run button.
e. Click **Start** to begin the acquisition.

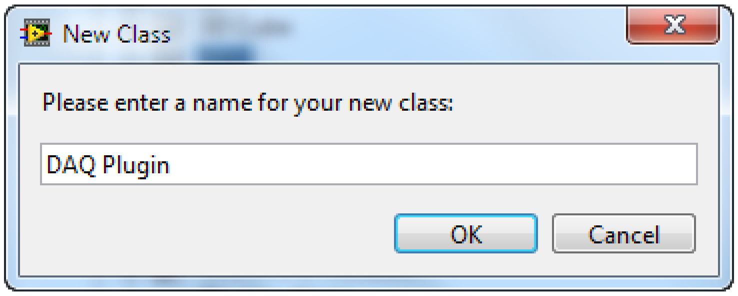
Note: this front panel has been designed to scale and resize all the controls appropriately. This will be important when loading into a framework as a plugin with variable width and height parameters. To see how this has been configured, use the mouse to grab the bottom right corner and resize the panel.

f. Click **Pause** to temporarily stop the acquisition.



g. Click **Stop** to terminate execution of the VI.
h. Switch to the block diagram to see how the application has been implemented. A simple producer consumer pattern is used to dispatch commands from UI interaction to the consumer, which is a simple queued state machine that controls the DAQ task.

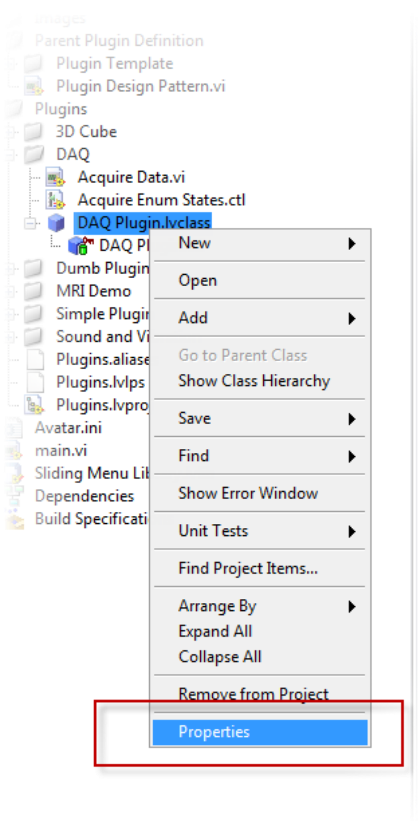
3. Create an Object that will store and run this VI
   a. In the Project Explorer, navigate to the folder that contains the data acquisition application, which should be ‘Plugins > DAQ.’
   b. Right click on the folder and select ‘New > Class.’ When prompted, enter ‘DAQ Plugin’ as the name for the class. Click **OK** and ‘DAQ Plugin.lvclass’ should now appear under the ‘DAQ’ folder in the Project Explorer.


New Class

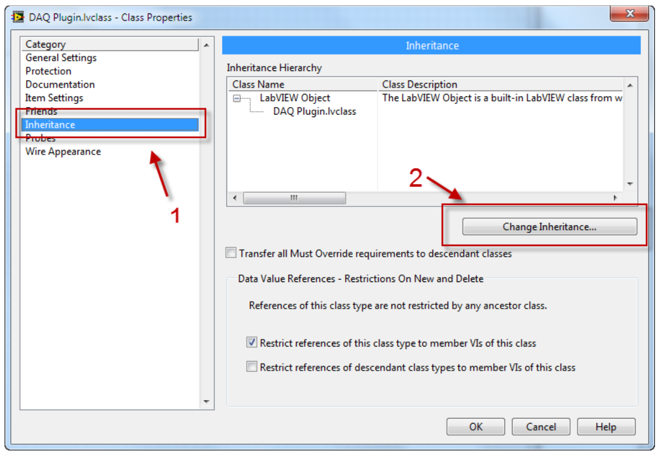
Please enter a name for your new class:

DAQ Plugin

   c. To function properly as a plugin, this class will need to inherit the capabilities of the Generic Plugin class that we looked at in section 1. To specify this relationship, right click ‘DAQ Plugin.lvclass’ and select **Properties** as shown below.



d. In the Properties Dialog that appears for this class, select **Inheritance** from the list of Categories. On the right side of the screen, select **Change Inheritance**.



Class Name	Class Description
LabVIEW Object	The LabVIEW Object is a built-in LabVIEW class from w
DAQ Plugin.lvclass	

☐ Transfer all Must Override requirements to descendant classes

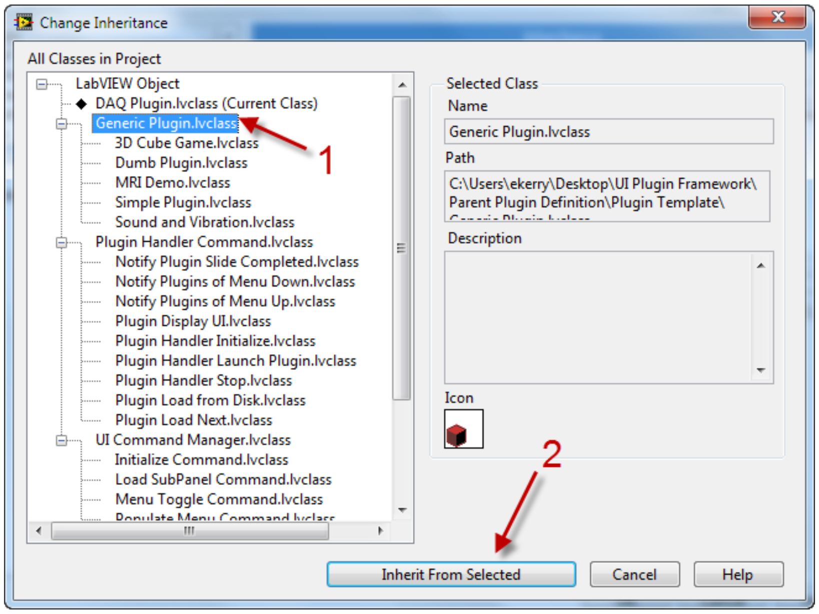
Data Value References - Restrictions On New and Delete

References of this class type are not restricted by any ancestor class.

☑ Restrict references of this class type to member VIs of this class

☐ Restrict references of descendant class types to member VIs of this class

e. The 'Change Inheritance' dialog will appear, which lists all of the Classes currently loaded in the Project Explorer. Find and select 'Generic Plugin' and click **Inherit from Selected.**


Change Inheritance

All Classes in Project

LabVIEW Object
◆ DAQ Plugin.lvclass (Current Class)
Generic Plugin.lvclass
3D Cube Game.lvclass
Dumb Plugin.lvclass
MRI Demo.lvclass
Simple Plugin.lvclass
Sound and Vibration.lvclass
Plugin Handler Command.lvclass
Notify Plugin Slide Completed.lvclass
Notify Plugins of Menu Down.lvclass
Notify Plugins of Menu Up.lvclass
Plugin Display UI.lvclass
Plugin Handler Initialize.lvclass
Plugin Handler Launch Plugin.lvclass
Plugin Handler Stop.lvclass
Plugin Load from Disk.lvclass
Plugin Load Next.lvclass
UI Command Manager.lvclass
Initialize Command.lvclass
Load SubPanel Command.lvclass
Menu Toggle Command.lvclass
Populate Menu Command.lvclass

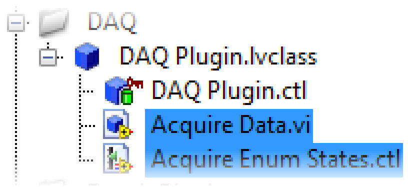
Selected Class
Name: Generic Plugin.lvclass
Path: C:\Users\ekerry\Desktop\UI Plugin Framework\Parent Plugin Definition\Plugin Template\Generic Plugin.lvclass
Description
Icon

f. The Change Inheritance dialog will close, returning to the Class Properties dialog. Click **OK** to save the changes.

g. Before proceeding, ensure that the new 'DAQ Plugin.lvclass' class has been saved to disk. To do this, return to the Project Explorer and select File > Save All.

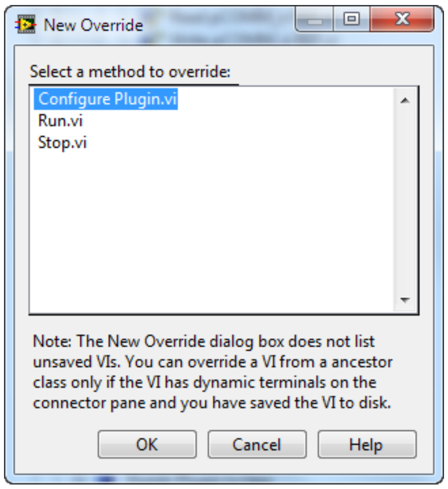
h. When prompted, save the lvclass file in the same directory as the 'Acquire Data.vi,' which should be [root]\Plugins\DAQ, where [root] refers to the directory of the Project. **DO NOT SAVE IT ANYWHERE ELSE AS THIS IS THE DIRECTORY THAT THE FRAMEWORK WIL LOOK IN**

i. Hold shift to select 'Acquire Data.vi' and 'Acquire Enum States.ctl.' Click and drag them under the 'DAQ Plugin Class.lvclass' file to make it a member of this class. It should appear as shown below:



j. Right-click on the Class and select 'New > VI for Override.' This will allow customization of behaviors that were defined by the parent 'Generic Plugin.lvclass' for the specific purpose of 'DAQ Plugin.lvclass' class.

k. We can override the behaviors of multiple methods, including 'Run,' 'Stop,' and 'Configure,' which allows us to completely customize what this plugin does for each. In this example, we are going to customize the implementation of 'Confiure Plugin.vi.' Select 'Configure Plugin.vi' and select **OK.**



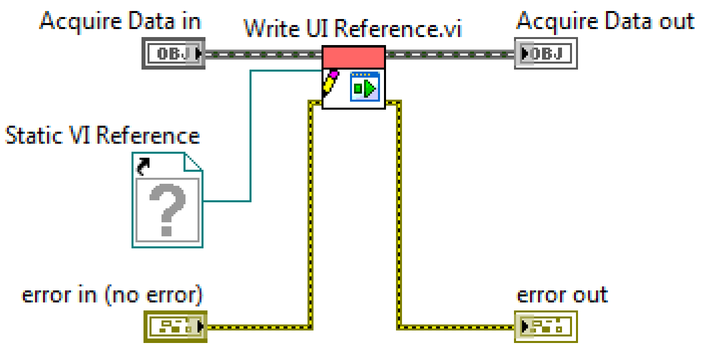
Select a method to override:

Configure Plugin.vi
Run.vi
Stop.vi

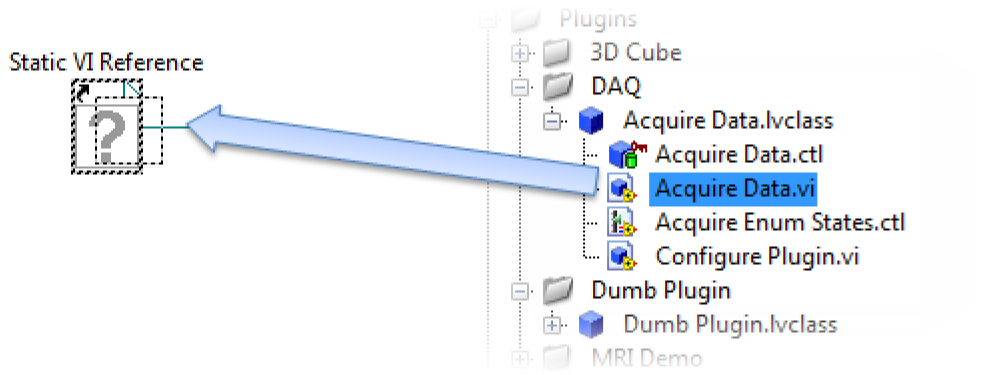
Note: The New Override dialog box does not list unsaved VIs. You can override a VI from a ancestor class only if the VI has dynamic terminals on the connector pane and you have saved the VI to disk.

l. LabVIEW will generate a new VI named 'Configure Plugin.vi.' Switch to the block diagram and delete the subVI that appears (it's the parent's copy of Configure Plugin.vi).

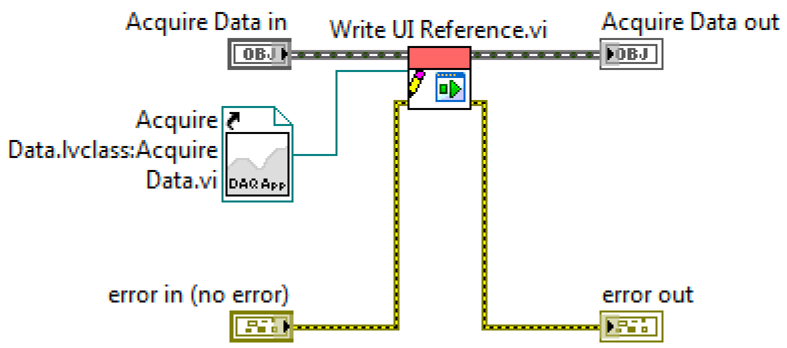
m. Open Quick-Drop by pressing CTRL + Space. Type 'Write UI Reference.vi. This VI is the private data modifier that allows us to store a reference to the VI we want this plugin to run. Drop it on the block diagram and connect it. Finally, place 'Static VI Reference' from the application palette to build the block diagram shown below:



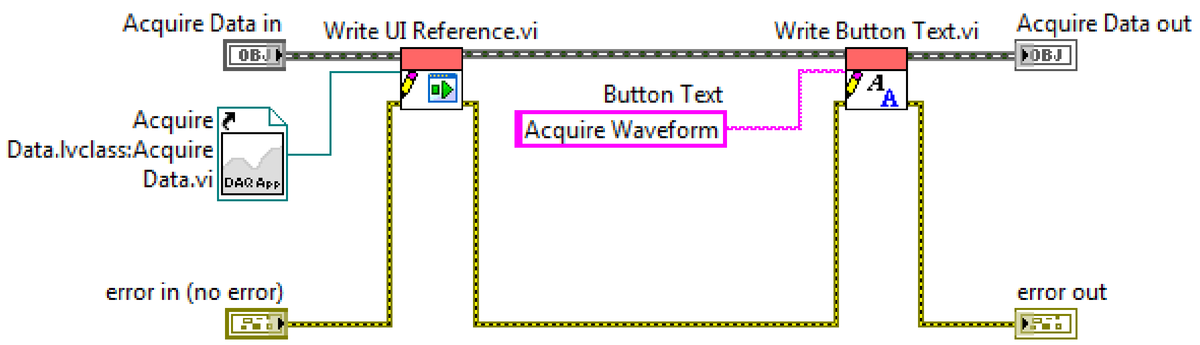
n. In the Project Explorer, select 'Acquire Data.vi,' which is the VI we want this plugin to run. Drag and drop it into the Static VI Reference.



o. The finished code should look like the diagram below:



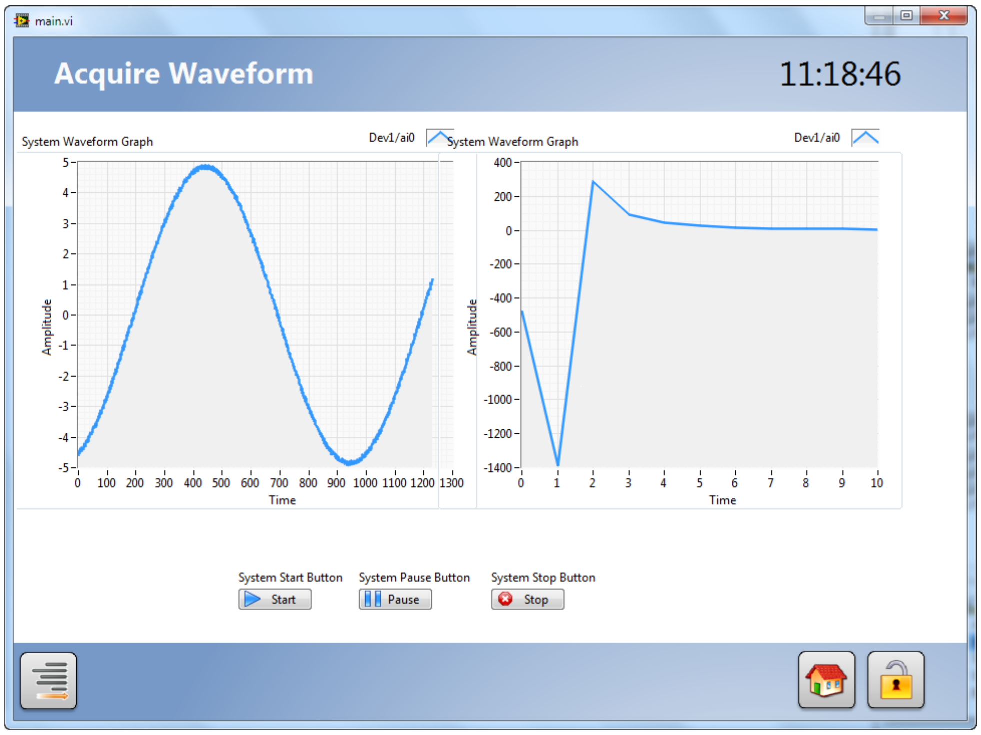
p. You can also modify the value of the string that will appear in the menu and in the header by modifying the code as shown:



q. Return to the Project Explorer and click the 'Save All' button again.

r. Open the Project, UI Plugin Framework.lvproj

s. Open **main.vi** and run the application. The menu should now be populated with one additional item to represent the newly created plugin. Click on it and verify that it appears in the subpanel as shown below:


main.vi
Acquire Waveform
11:18:46
System Waveform Graph
Dev1/ai0
Amplitude
Time
System Waveform Graph
Dev1/ai0
Amplitude
Time
System Start Button
Start
System Pause Button
Pause
System Stop Button
Stop

Sending Messages to Plugins Using User Events

Goal

We want to send basic commands to the plugin that are then handled in such a way to minimize the work being done in the background. By doing so, we can ensure that the application continues to run smoothly as the user navigates between multiple objects.

Scenario

We want to configure the newly created Acquire Measurement.vi application to pause the acquisition and display when it is not being shown, or when a user opens the menu.

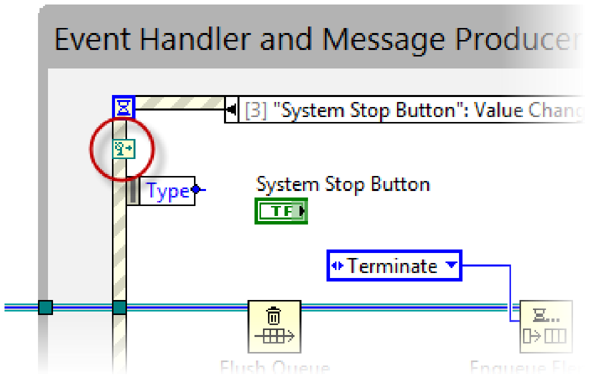
Description

Sending messages to plugins is a common challenge for plugins, as it represents a scenario in which you may need to broadcast the same information to N objects. In this case, sending individual messages to every recipient is generally inefficient and difficult. This exercise will use user events to broadcast information to plugins, who can then choose to act upon it as they see fit.

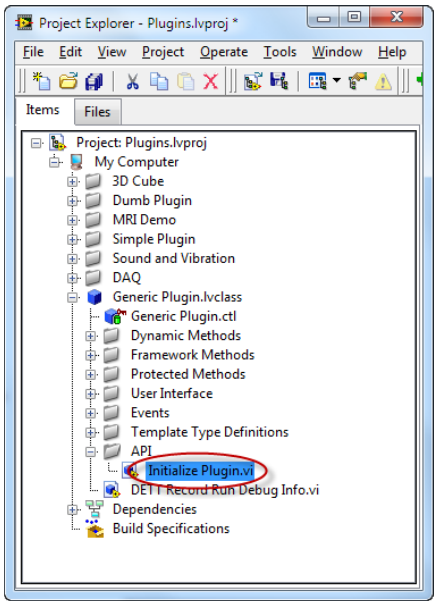
Concepts Covered

- Dynamic event registration
- User APIs

1. Handle the user event ‘Menu Open,’ which should terminate execution of the plugin.
   a. Open the Plugin project and close the UI Plugin Framework Project
   b. In the Project Explorer for Plugins.lvproj, navigate to ‘Plugins > DAQ > Acquire Data.vi’ and open the block diagram.
   c. Right-click on the border of the event structure and select **Show Dynamic Events Terminal**, which will add the following input terminal to the structure:

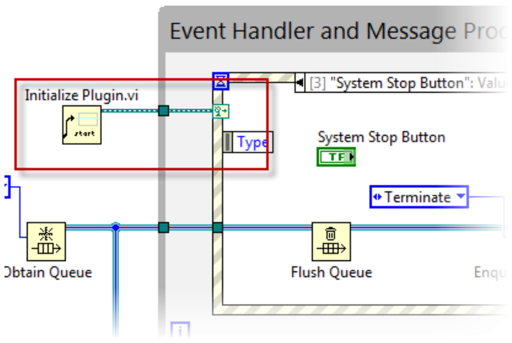

Event Handler and Message Producer
[3] "System Stop Button": Value Chang
Type
System Stop Button
TF
Terminate
Flush Queue
Enqueue Ele

   d. In the Project Explorer, navigate to ‘Parent Plugin Definition > Plugin Template > API’ and explore the folder.


Project Explorer - Plugins.lvproj *
File Edit View Project Operate Tools Window Help
Items Files

```
Project: Plugins.lvproj
  My Computer
    3D Cube
    Dumb Plugin
    MRI Demo
    Simple Plugin
    Sound and Vibration
    DAQ
    Generic Plugin.lvclass
      Generic Plugin.ctl
      Dynamic Methods
      Framework Methods
      Protected Methods
      User Interface
      Events
      Template Type Definitions
      API
        Initialize Plugin.vi
      DETT Record Run Debug Info.vi
    Dependencies
    Build Specifications
```

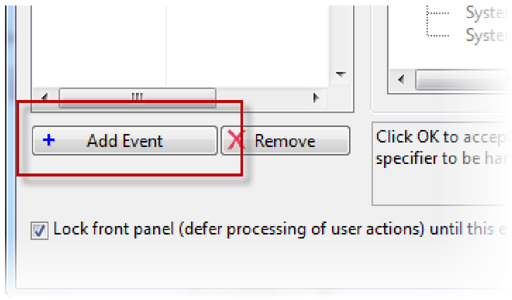
   e. Initialize Plugin.vi will register all the necessary events that this plugin needs to respond to. Place it on the block diagram of the plugin and connect it as shown:



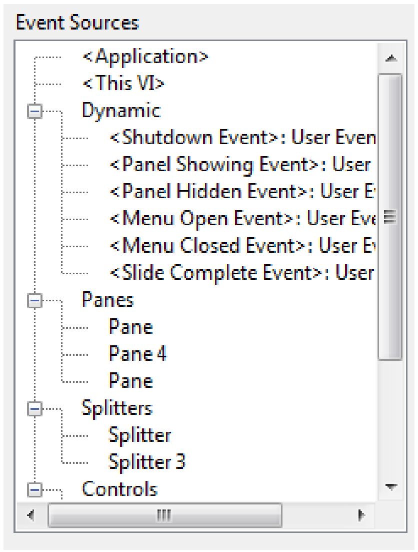
f. Navigate to the case in the Event Structure that handles "System Pause Button": Value Change. Note that this case currently handles the user pressing 'Pause' on the front panel.

g. To add the user event to this case, right-click and select 'Edit Events Handled by this Case"

h. Click **Add Event**


Add Event
Remove
Click OK to accep
specifier to be ha
Lock front panel (defer processing of user actions) until this e

i. The middle list-box contains all the items that can serve as an event source. The user events have been added to this list because of the inputs to the dynamic events terminal. Note that there are multiple user events that we can chose from:


Event Sources
<Application>
<This VI>
Dynamic
<Shutdown Event>: User Even
<Panel Showing Event>: User
<Panel Hidden Event>: User E
<Menu Open Event>: User Eve
<Menu Closed Event>: User E
<Slide Complete Event>: User
Panes
Pane
Pane 4
Pane
Splitters
Splitter
Splitter 3
Controls

j. Select 'Menu Open Event' and click **OK**.

k. When the message that the menu is open is fired, this plugin now handles it by stopping the acquisition.

l. Make sure that the front-panel has been closed and return to **main.vi**. Run the application and launch the plugin. Show that opening the menu now pauses the acquisition.

Object-Oriented Hardware Abstraction Layer

Goal

Demonstrate the use of an object-oriented design pattern to abstract hardware in an instrument control application. The goal of this abstraction is to make it simple to incorporate new hardware into an existing application without making modifications to the software framework.

Scenario

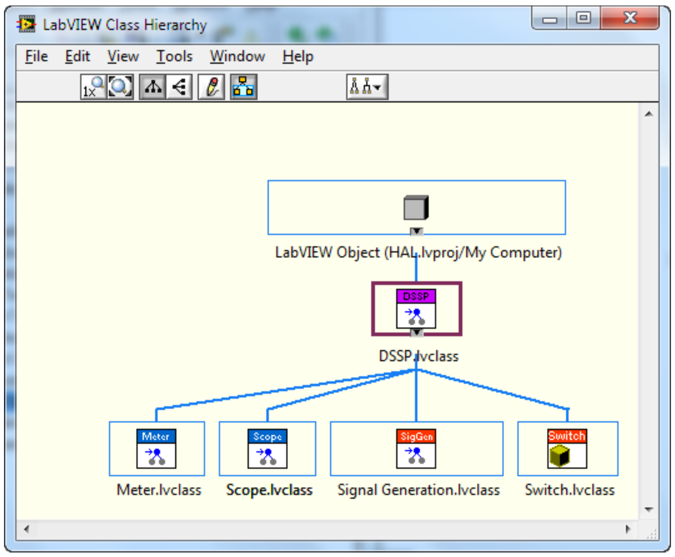
The problem with many test systems is that the overall system must be in operation longer than the individual system components are supported. Sometimes the device being tested has an active service life measured in decades, while many test instruments are obsolete and no longer supported after five years or less. Other times, the device being tested has an active service life measured in months. Both of these are examples of life-cycle mismatch.

Setup

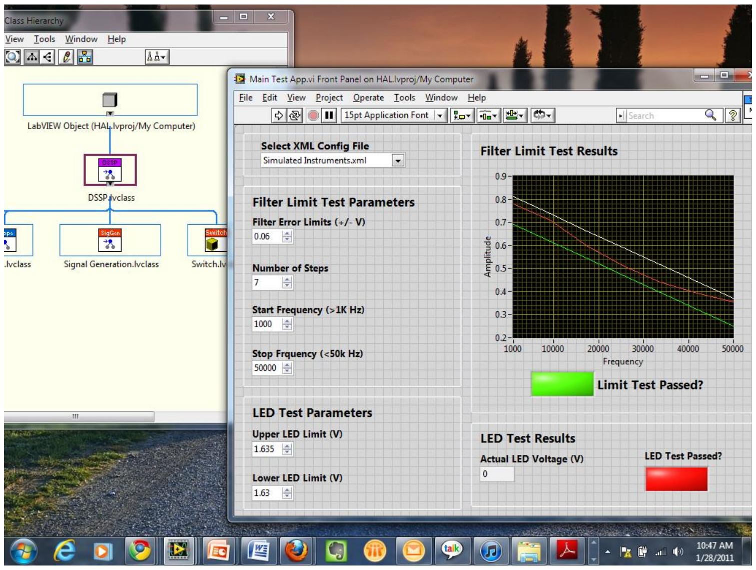
In order for all of the included instrument drivers to work, it's important to ensure that you've downloaded and installed the correct instrument drivers. In LabVIEW, go to Help > Find Instrument Drivers and make sure the following appear in the list:

- Fluke 884X Series
- Tktds3xx
- ag33xxx

For the sake of this demo, it's also important that it not have been run since the project was opened, as we will show the instrument drivers coming into memory for the first time using the VI Hierarchy Window. Ensure that the class hierarchy appears as shown below before proceeding. If it does not, simply close front panel of the top-level VI and re-open it.



1. Provide a brief explanation of the application and how it works.
    a. Open the Project, 'HAL.lvproj'
    b. Expand the 'Test Code' folder and open the **Main Test App.vi**
    c. Expand the Device-Specific Software Plugin Class, DSSP.lvclass. This is the parent for all device plugins and defines high-level generic functionality. In this case, the only methods of this class are 'Close Instrument' and 'Initialize Instrument.'
    d. Right-click on 'DSSP.lvclass' and select 'Show Class Hierarhcy.' Prior to running the application, only one layer of children will be visible, as shown in the image. Keep this window open and carefully positioned so that it can be seen next to the front panel of **Main Test App.vi**. If viewing on a projector with 1024 resolution, try to replicate the configuration shown below (taken on a monitor with 1024 x 768):


Class Hierarchy
View Tools Window Help
LabVIEW Object (HAL.lvproj/My Computer)
DSSP
DSSP.lvclass
.lvclass
SigGen
Signal Generation.lvclass
Switch
Switch.lv
Main Test App.vi Front Panel on HAL.lvproj/My Computer
File Edit View Project Operate Tools Window Help
15pt Application Font
Search
Amplitude
Frequency
10:47 AM
1/28/2011

Select XML Config File: Simulated Instruments.xml

Filter Limit Test Parameters
Filter Error Limits (+/- V): 0.06
Number of Steps: 7
Start Frequency (>1K Hz): 1000
Stop Frquency (<50k Hz): 50000

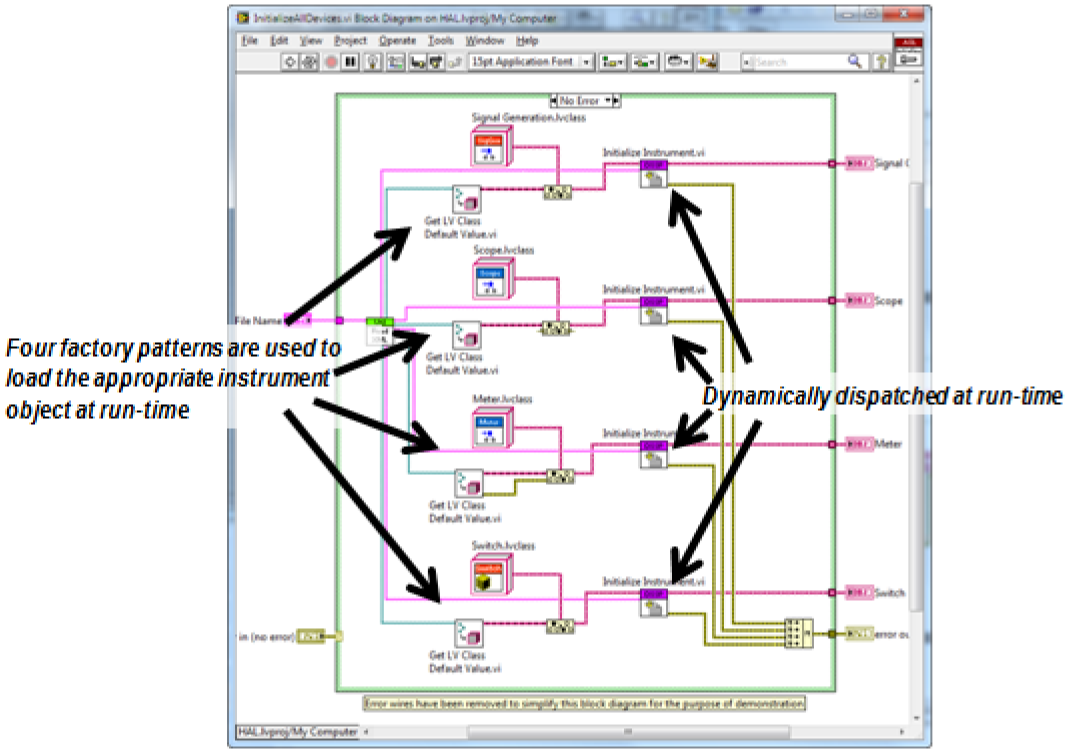
LED Test Parameters
Upper LED Limit (V): 1.635
Lower LED Limit (V): 1.63

Filter Limit Test Results
Limit Test Passed?

LED Test Results
Actual LED Voltage (V): 0
LED Test Passed?

    e. Show the block diagram of the application to review the operations that this application will be conducting.
    f. The first VI is 'Initialize All Devices.vi' which is built on top of a factory design pattern that products objects to represent the three devices based on the XML file that it is pointed to. Open this VI and look at the block diagram.

g. This VI outputs four objects: one for a signal generator, one for a scope, one for a meter and finally one for a switch. The actual class that defines the various methods appropriate to each decide will be loaded at run-time in the four parallel factory design patterns. The initialize instrument.vi will be dynamically dispatched based on the object that is loaded.


InitializeAllDevices.vi Block Diagram on HAL.lvproj/My Computer
No Error
Signal Generation.lvclass
Initialize Instrument.vi
Get LV Class Default Value.vi
Scope.lvclass
Meter.lvclass
Switch.lvclass
File Name
error in (no error)
Signal
Scope
Meter
Switch
error out
Error wires have been removed to simplify this block diagram for the purpose of demonstration
Four factory patterns are used to load the appropriate instrument object at run-time
Dynamically dispatched at run-time

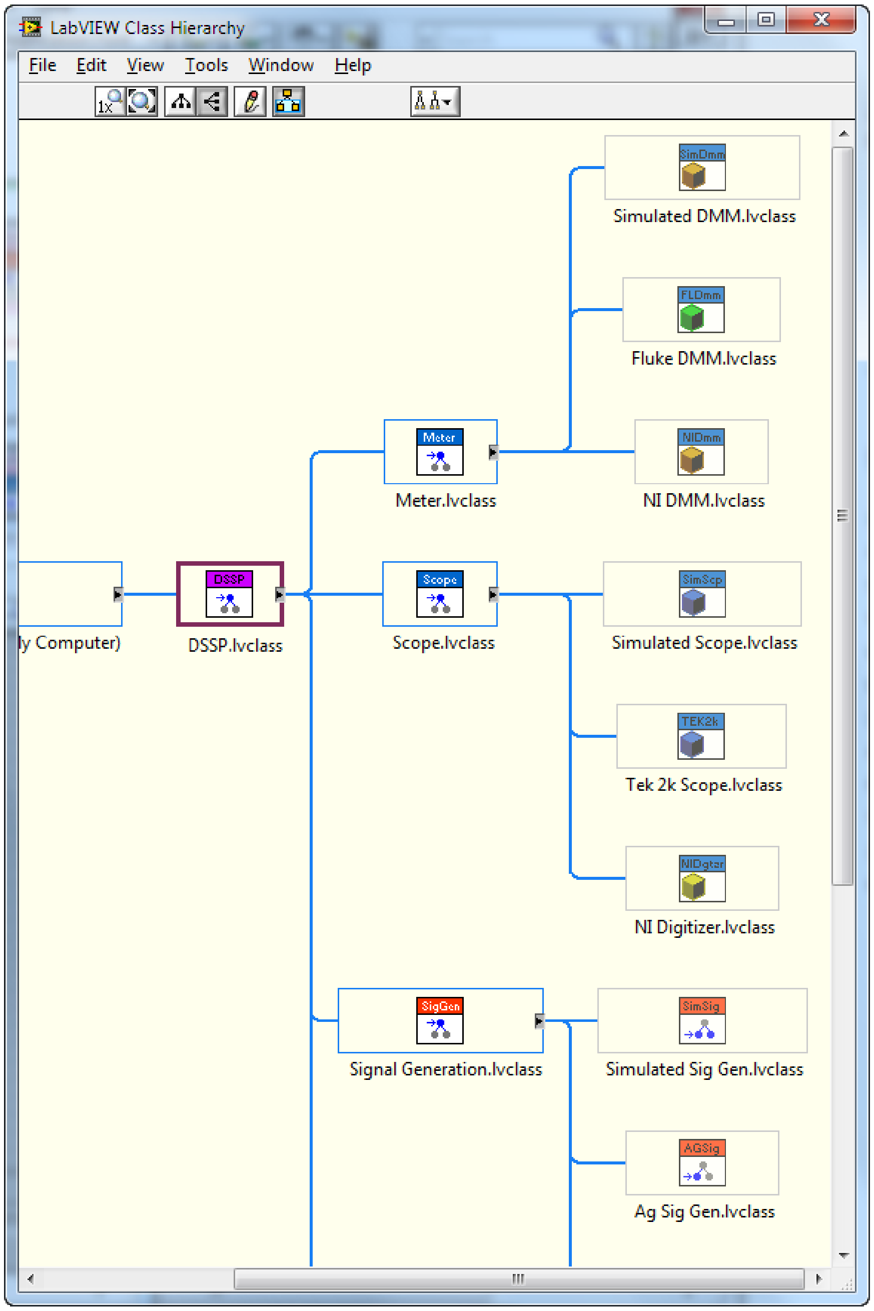
h. Set a breakpoint just before the case structure of this block diagram. This will make it easier to illustrate how and when the classes come into memory.

i. Close InitializeAllDevice.vi and return to the block diagram of the top-level VI. After the devices have all been initialized, they are then passed to other functions to complete the test sequence. In this case, a frequency sweep (which utilizes both the signal generator and the scope), an LED test (which needs only the voltmeter) and then finally, the instruments are closed.

j. Use the ring control to select 'Simulated Instruments.xml' and run the application (make sure the VI Hierarhcy window can be seen).

k. When the execution reaches the breakpoint, turn on highlight execution, ensure you can see the class hierarchy and run the code. The class hierarchy will populate the new objects as children of their generic parents as each factory pattern executes.

l. Repeat this process for the other enum operations. The resultant class hierarchy will appear as shown:



More Information

Online Resources

- **ni.com/largeapps** – find best practices, online examples and a community of LabVIEW users

Customer Education Classes

- Managing Software Engineering with LabVIEW
    - Learn to manage the development of a LabVIEW project from definition to deployment
    - Select and use appropriate tools and techniques to manage the development of a LabVIEW application
    - Recommended preparation for Certified LabVIEW Architect exam
- Object-Orientation in LabVIEW