Use Plug-ins

Contents

Introduction

Adding a plug-in system in your software makes it possible to add new functionality after a customer installed your software.

A plug-in system consists of following components.

- **Plug-in Host** – This can be an application accepting plug-ins and using plug-ins
- **Plug-in Objects** – Usually each plug-in object is in a DLL. The Plug-in Host loads the DLL and uses the plug-in object contained in the DLL.
- **Plug-in Definition** – This is usually a DLL used by both the plug-in host application and the plug-in object DLLs. The plug-in host uses the Plug-in Definition to know how to use plug-in objects before plug-in objects are loaded. The plug-in objects use the Plug-in Definition to define their application programming interface so that they can be used by the plug-in host.
- **Plug-in Manager** – This is for managing the plug-ins by the plug-in host application. For example, let the user select plug-in object DLL files, enable and disable plug-ins, sharing data between all plug-ins, etc.

This document presents a sample plug-in system, showing the creation of the above components. The sample projects described in this document can be found at
http://www.limnor.com/studio/PluginSample.zip

Plug-in Definition

Use Plug-in Base Class or Interface

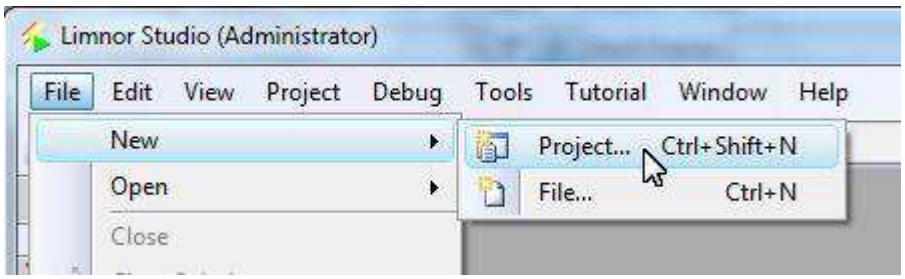
A plug-in definition can be a base class or an interface.

If all plug-ins must implement some same complex functionality then you may use a base class to implement such complex functionality. The problem is that a plug-in cannot inherit from any other classes. This could be a problem in many situations. If this is a problem then an interface should be used to define plug-ins.

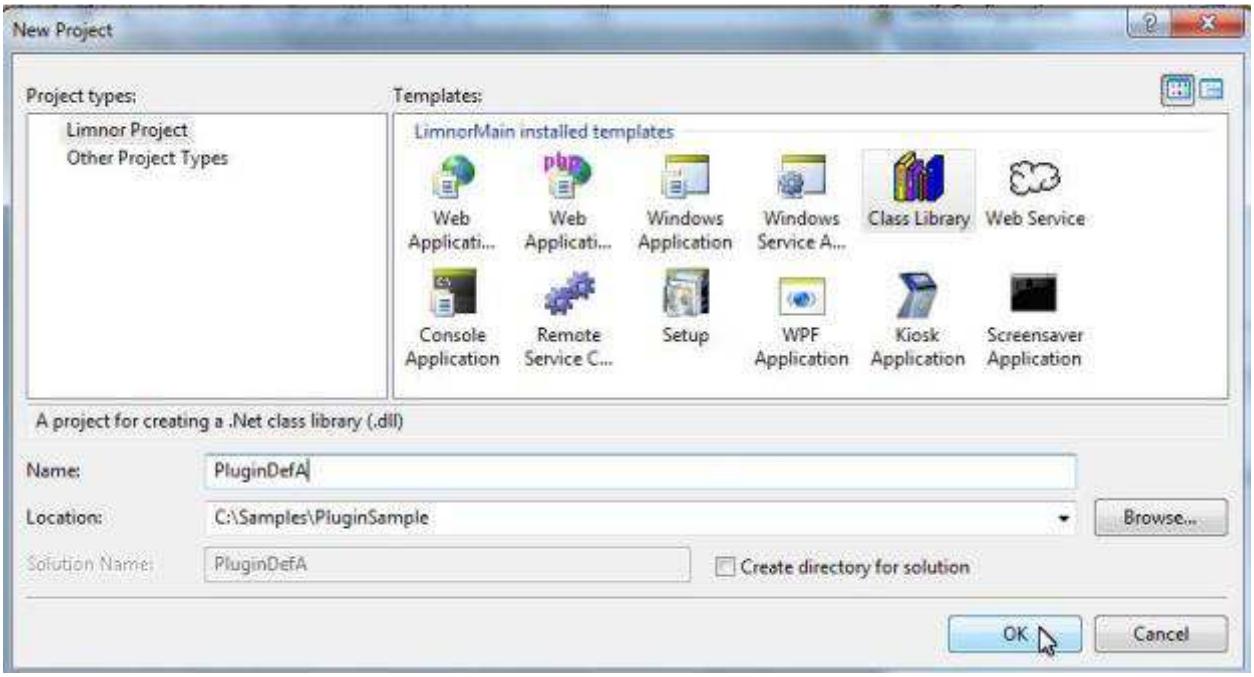
If an interface is used to define plug-ins then a plug-in can inherit from any unsealed class.

Create Plug-in Definition

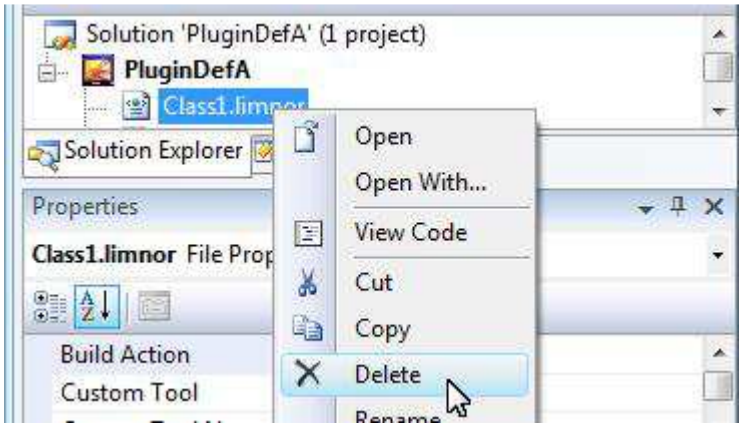
For this sample, we create an interface as our plug-in definition. We create a DLL project for it.



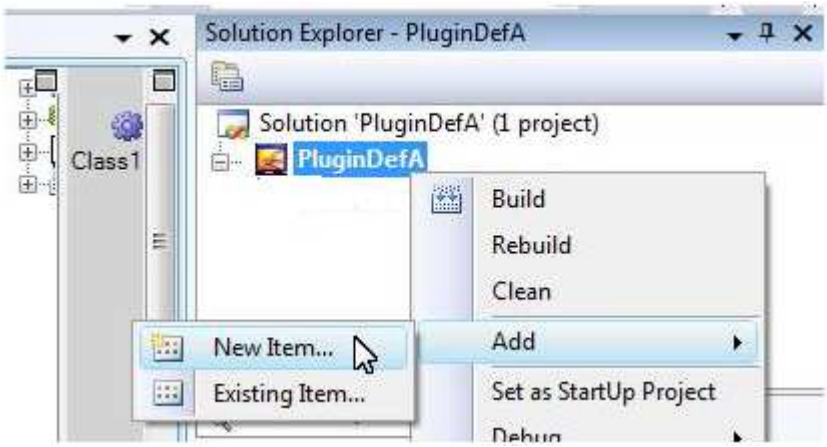
Name the project PluginDefA:



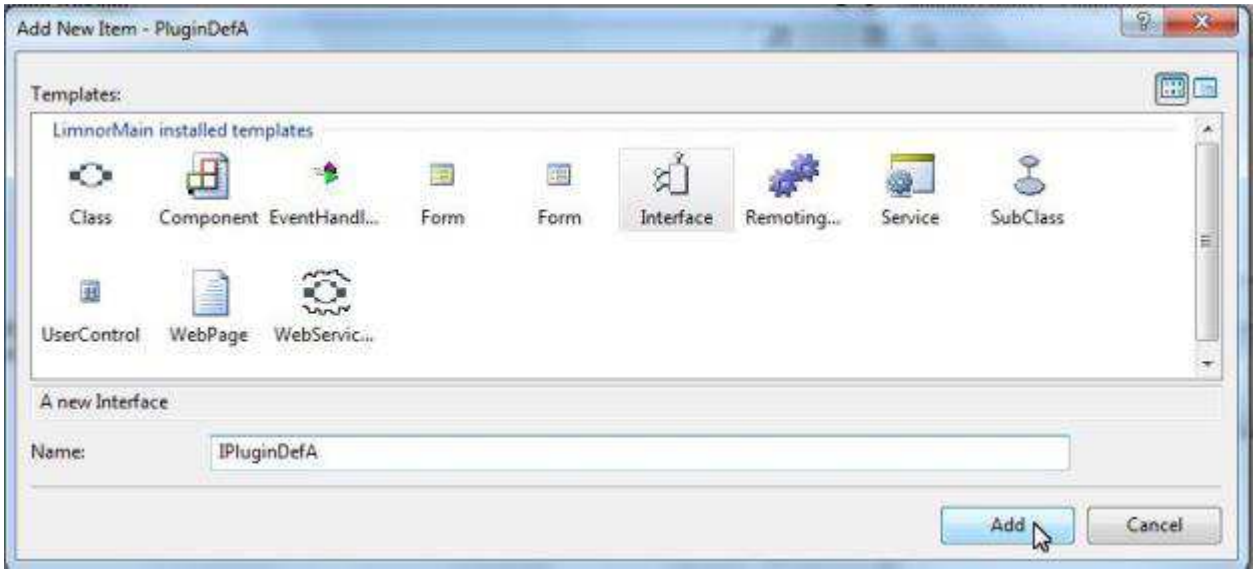
Delete automatically created class, Class1, because we are not going to use it:



Add an interface to the project as the plug-in definition to be created:



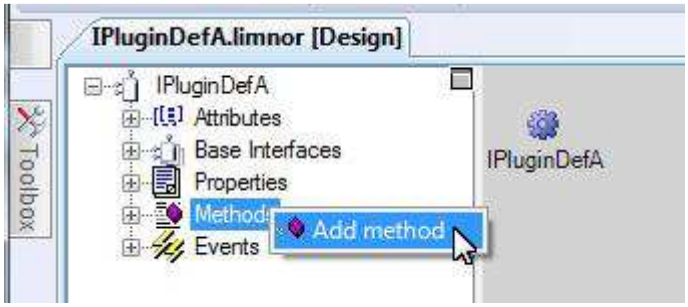
Name the interface IPluginDefA:



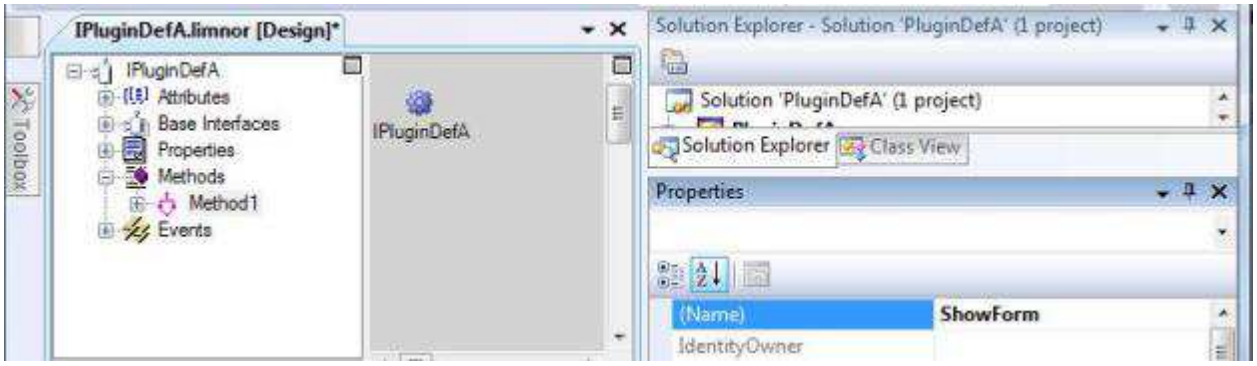
To define an interface is to add properties, methods, events, base interface and attributes to the interface:

To develop this interface, first we need to define what kinds of functionality should be plugged-in. For this sample, suppose we want to handle an image file list in various ways. Suppose we want to allow unlimited ways of handling the image file list. After installing the software, we want the software to be able to accept new plug-ins for providing new ways of handling the image file list.

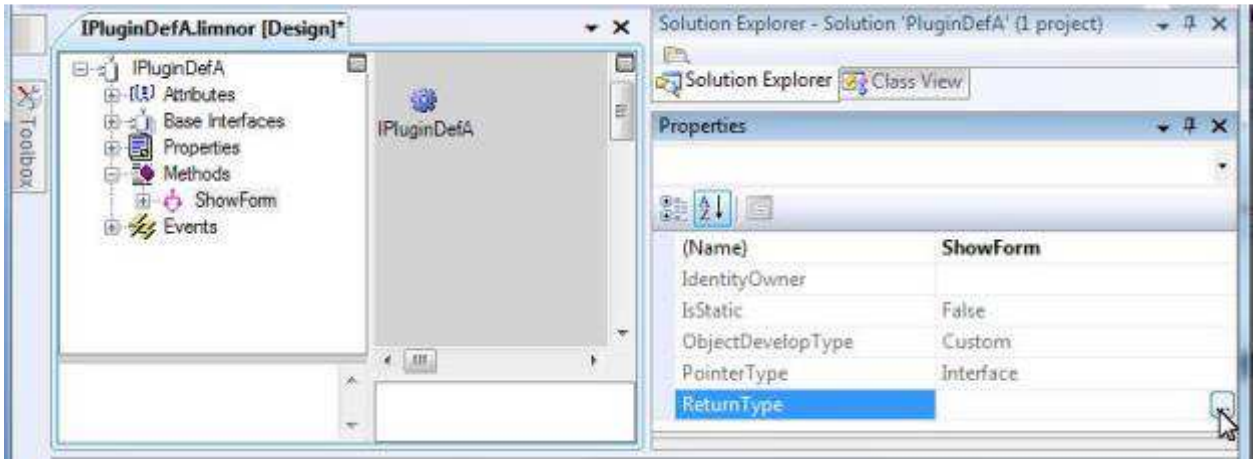
Suppose each way of handling the image list is presented in a Form. We may add a method for showing the form:

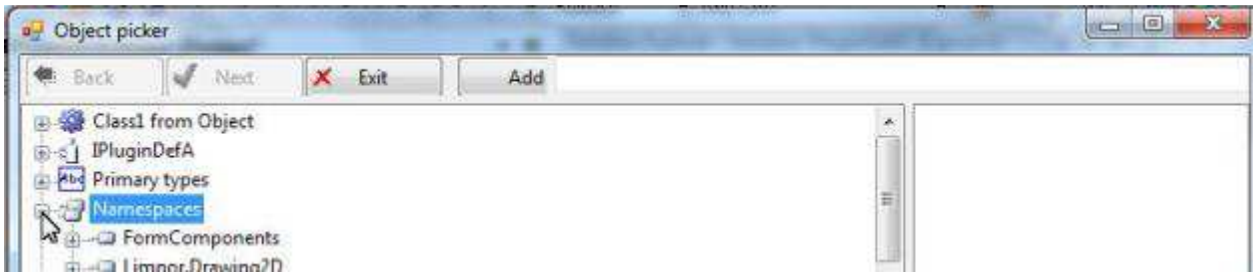


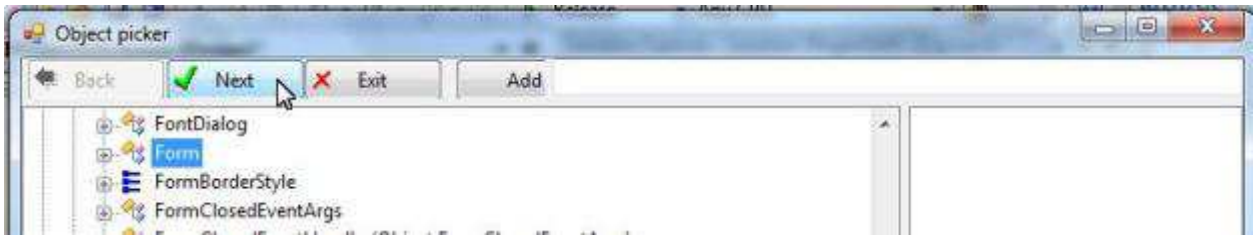
Name the method “ShowForm”:



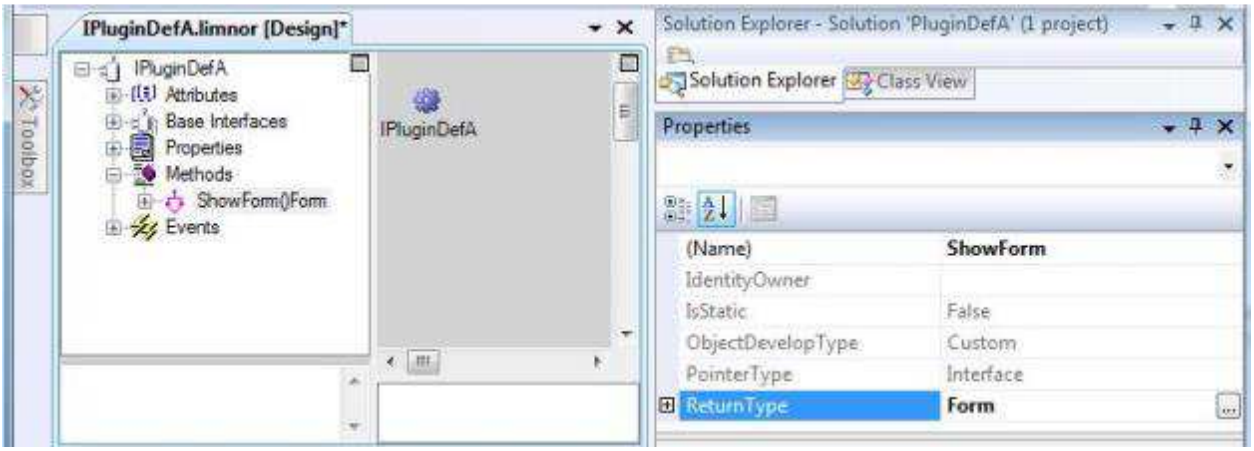
Change the method return type to Form for it to return the Form the method shows, so that the plug-in host may do additional handling on the form if needed:







This is a method we defined in this interface.

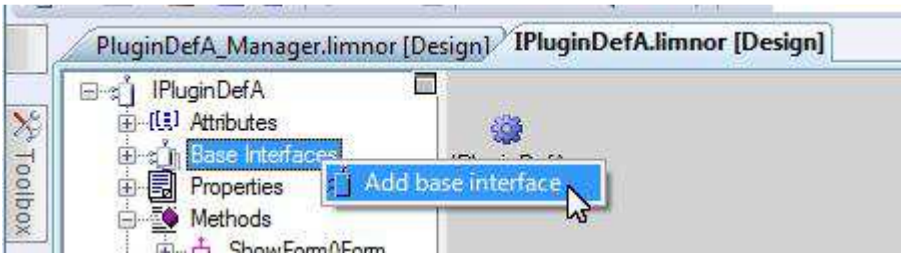


More members may be added to this interface for more functionality.

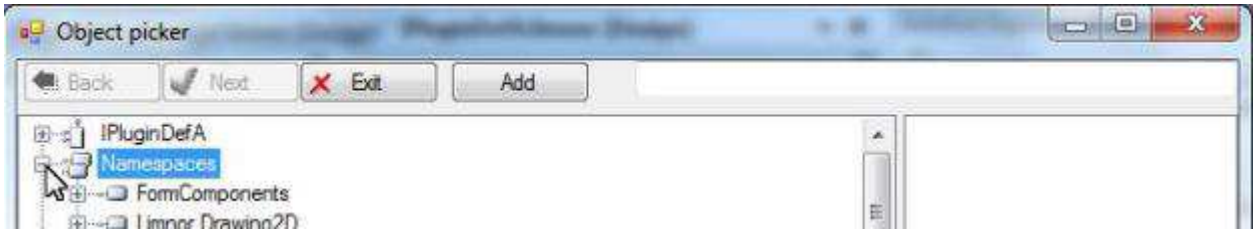
Data-sharing between Plug-ins

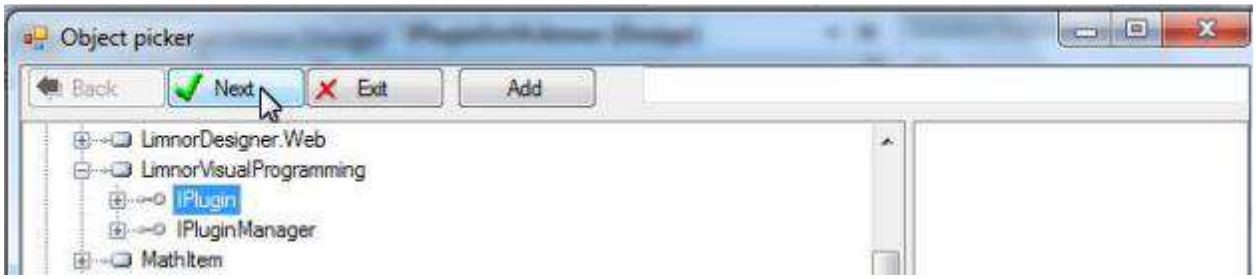
There is an interface, IPlugin, defined in namespace LimnorVisualProgramming. We may include this interface to our plug-in definition so that our plug-ins may send data-changed notifications to the plug-in host and receive data-changed notifications sent from other plug-ins. Our plug-ins may also access plug-in manager to get data stored in plug-in manager.

To include interface IPlugin in our plug-in definition interface, right-click “Base Interfaces”; choose “Add base interface”:

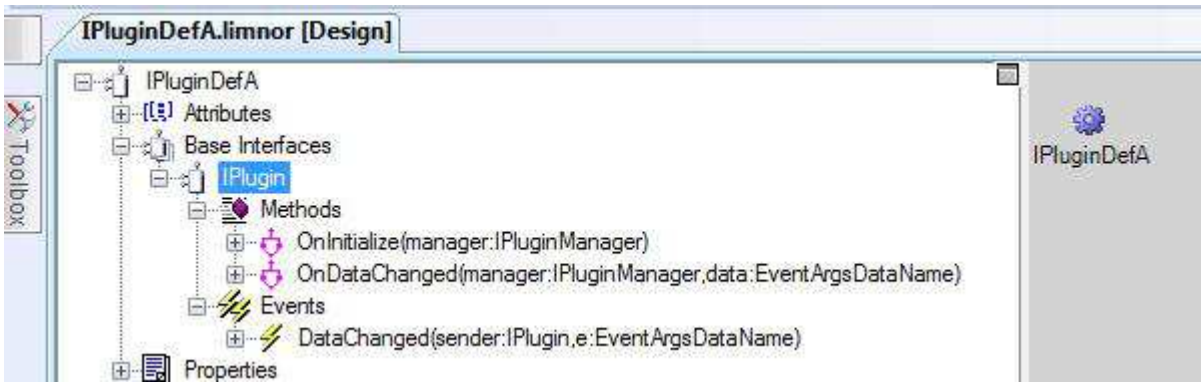


Select IPlugin under LimnorVisualProgramming:





IPlugin is included in IPluginDefA.



We can see that IPlugin provides following definitions:

- OnInitialize – This method will be called by the plug-in manager immediately after the plug-in object is created. The plug-in object may execute initialization actions inside this method. The plug-in manager is passed into this method. The plug-in object may access data stored in the plug-in manager. The plug-in object may remember the plug-in manager so that it may access it later.
- OnDataChanged – This method is executed when any one of plug-ins notifies the plug-in manager that data are changed. The plug-in object may refresh itself using the changed data.
- DataChanged – This is an event to be used to notify the plug-in manager that data changed. Plug-in fires this event to notify the plug-in manager that data changed. The plug-in manager will respond to this event and notify all other plug-ins by executing OnDataChanged on all other plug-ins.

We will demonstrate the implementation of these members by plug-ins.

Plug-in Manager

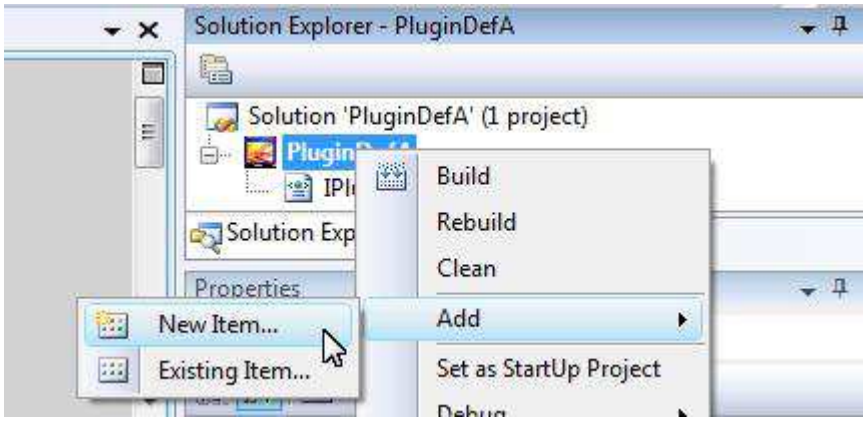
A plug-in manager will manage the plug-ins by allowing the user to select plug-in DLL files, allowing the user to enable and disable plug-ins.

Such functionality can be put inside the plug-in host application, or in a separate DLL file. In this demonstration, we put it in the plug-in definition DLL.

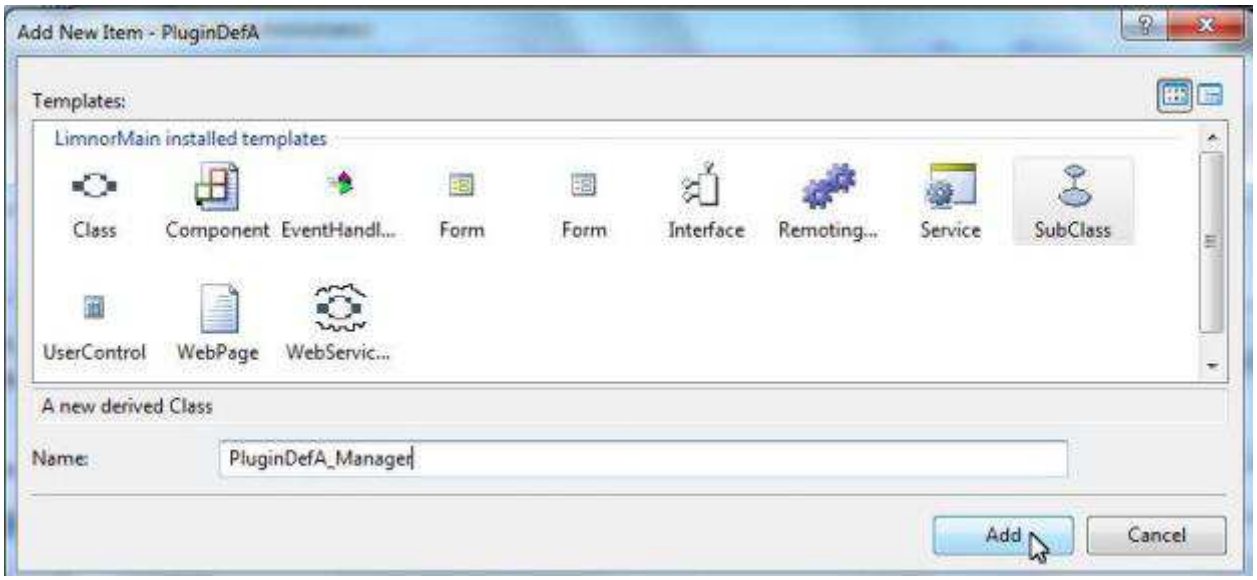
Limnor Studio ships with a plug-in manager class, which provides some basic plug-in manager functionality. We may derive our plug-in manager from it so that we do not need to development those basic plug-in managing functionality.

Derive a new plug-in manager

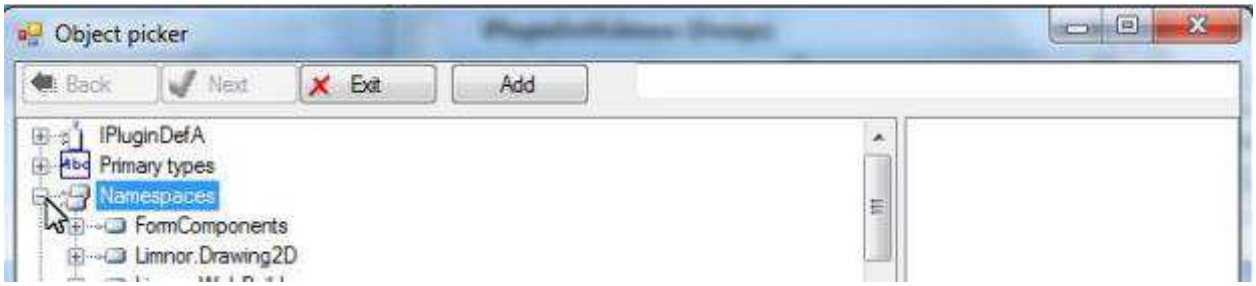
Create a new class as the plug-in manager:

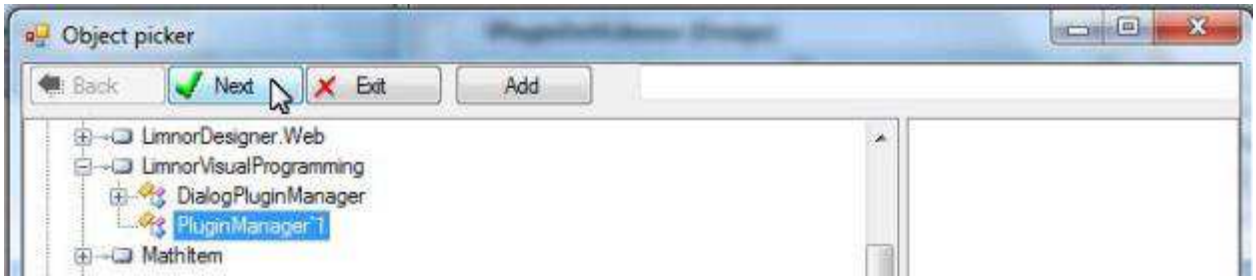


Select SubClass because we want our plug-in manager to be a sub class of the plug-in manager shipped with Limnor Studio. Name the new class PluginDefA_Manager:

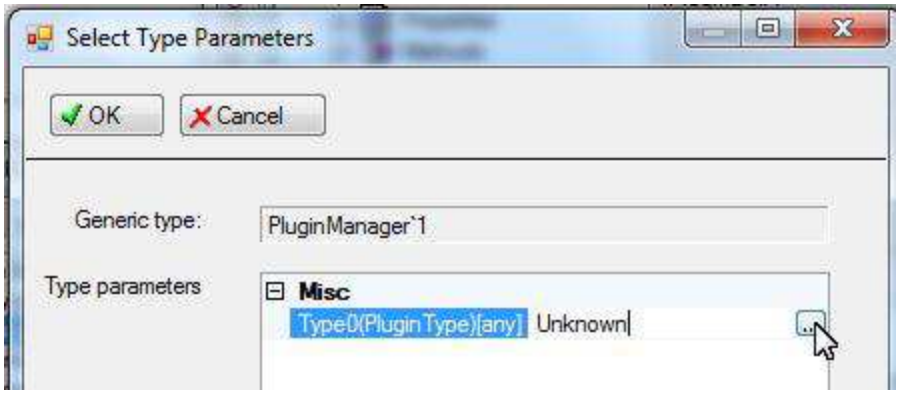


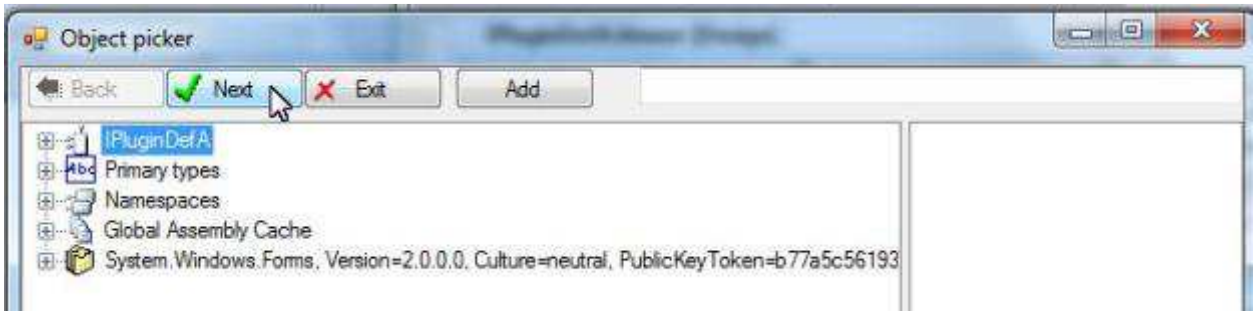
A dialogue box appears to let us select the base class. We want to select the plug-in manager shipped with Limnor Studio. It is under namespace LimnorVisualProgramming:

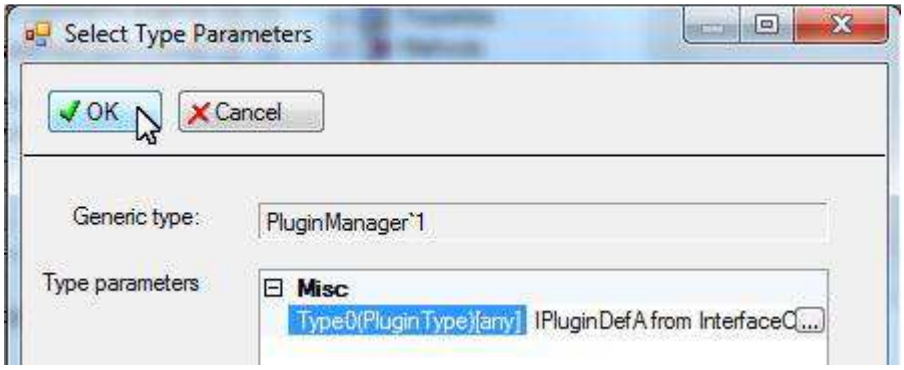




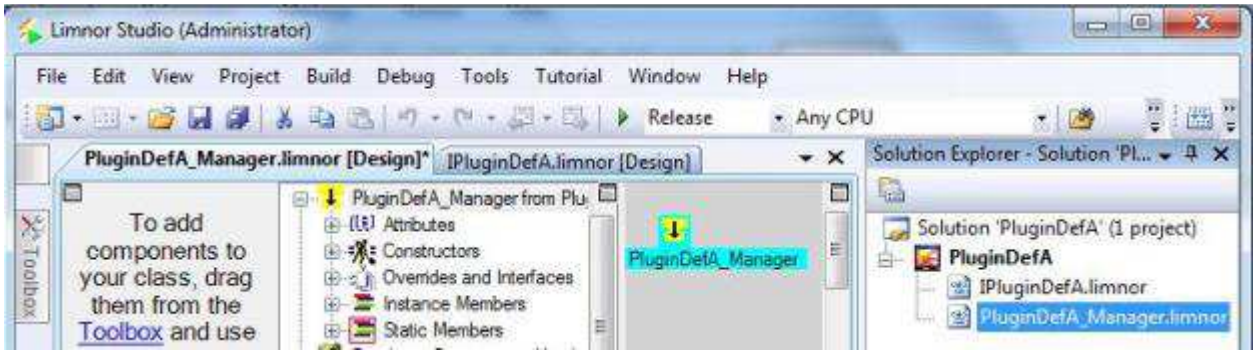
This base class contains a generic type parameter representing the plug-in definition. We may select the plug-in interface we just created:





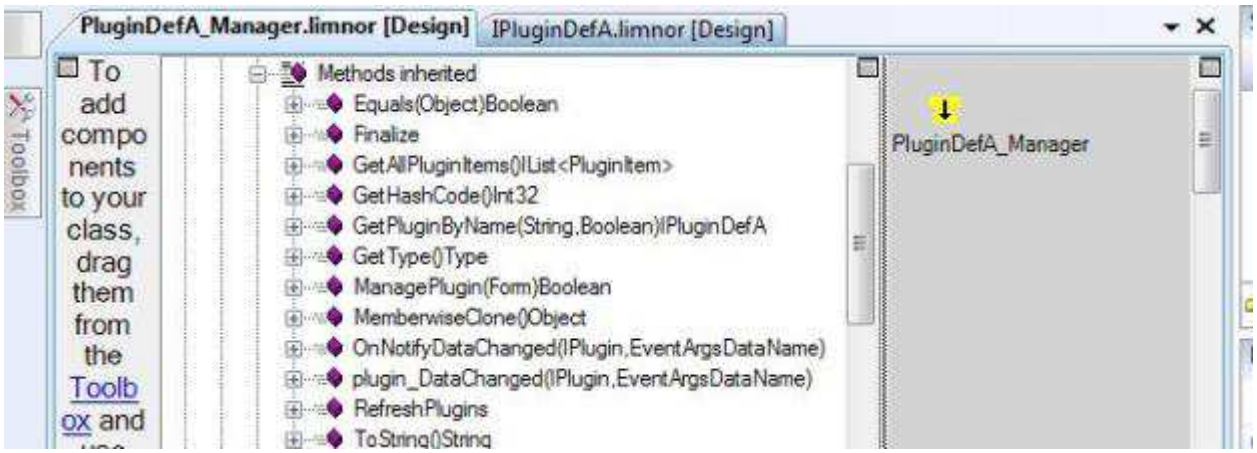


Our plug-in manager is created:



Inherited methods of the plug-in manager

The new plug-in manager inherits functionality from the base plug-in manager:



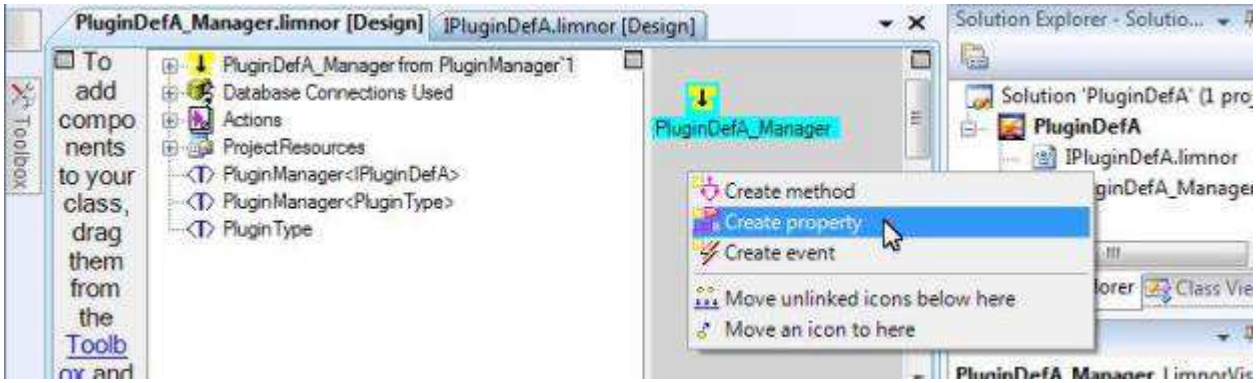
A plug-in host application may use following methods provided by the plug-in manager to manage plug-ins:

- GetAllPluginItems – It returns a list of all loaded plug-ins with plug-in names
- GetPluginByName – It returns the first plug-in by name-matching.

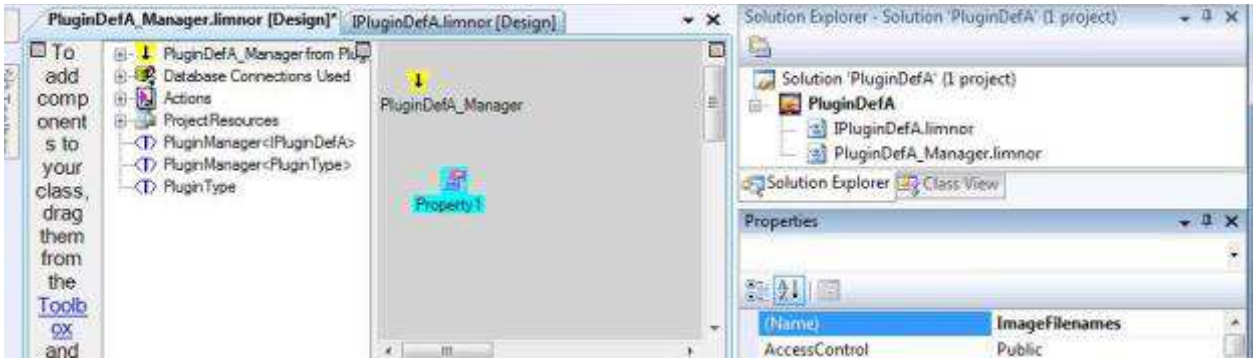
- ManagePlugin – It shows a dialogue for managing all plug-ins. The user may select plug-in DLL files to load plug-in objects; enable and disable loaded plug-in objects; give each plug-in object a name.
- OnNotifyDataChanged – It calls OnDataChanged of all plug-ins to notify all plug-ins that data changed. This method will be executed if a plug-in fires DataChanged event.
- RefreshPlugins – It creates plug-in objects of all enabled plug-ins.

Hold data in plug-in manager

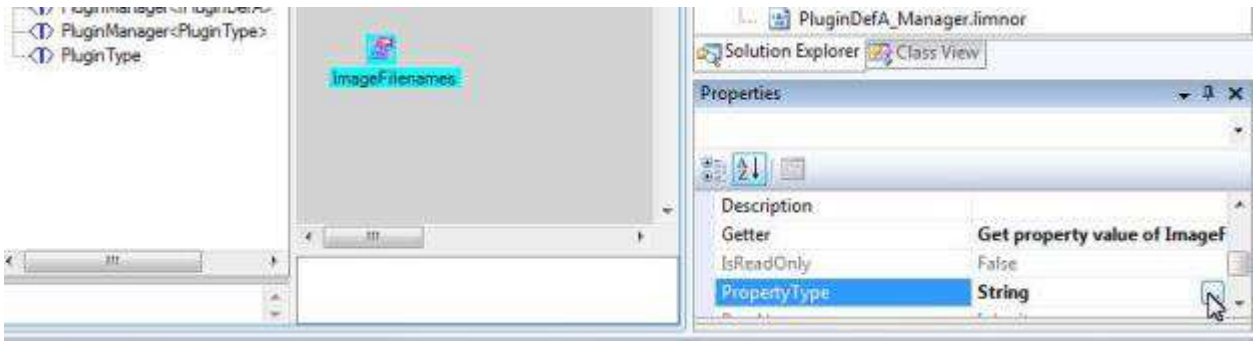
For this sample, we add a string collection property to the plug-in manager. Each string in the string collection is supposed to be an image file path.

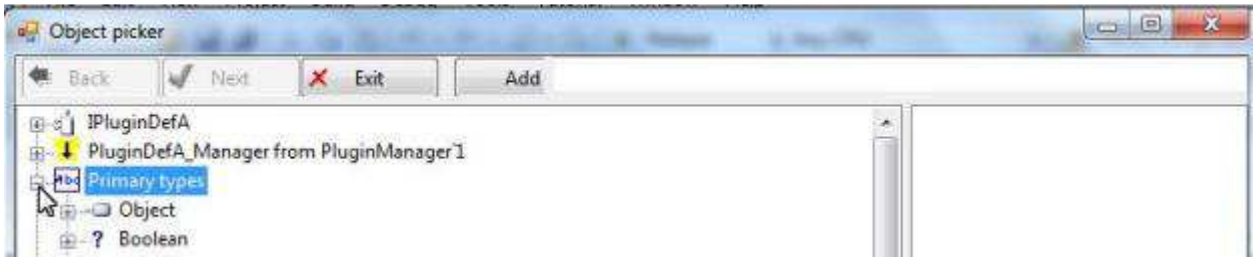


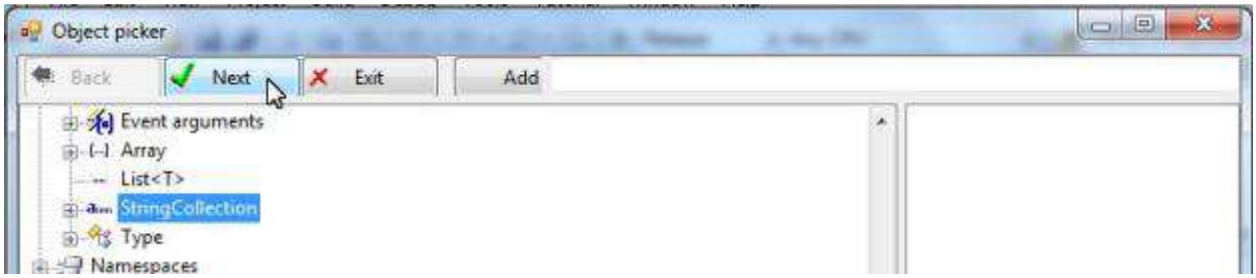
Change the property name to ImageFilenames:



Change the property type to StringCollection:



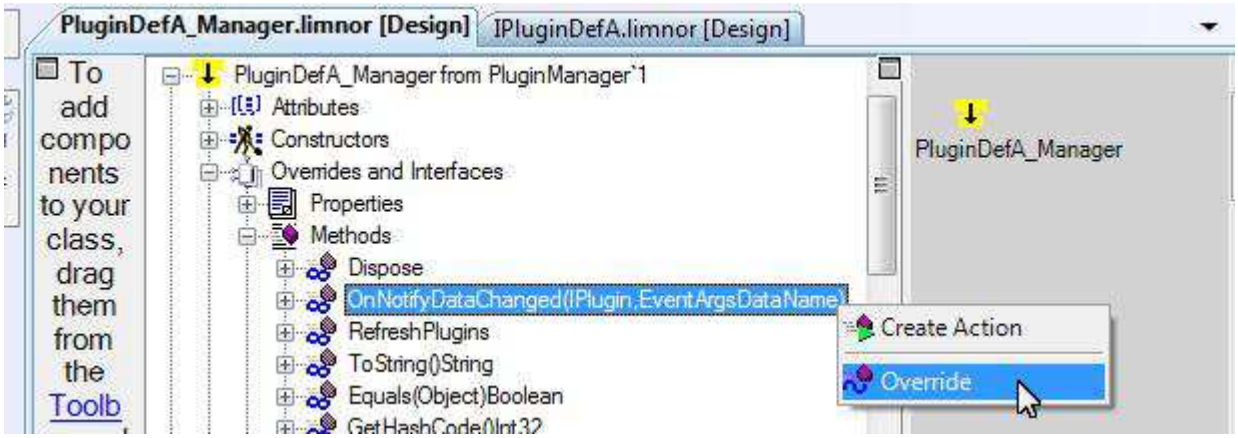




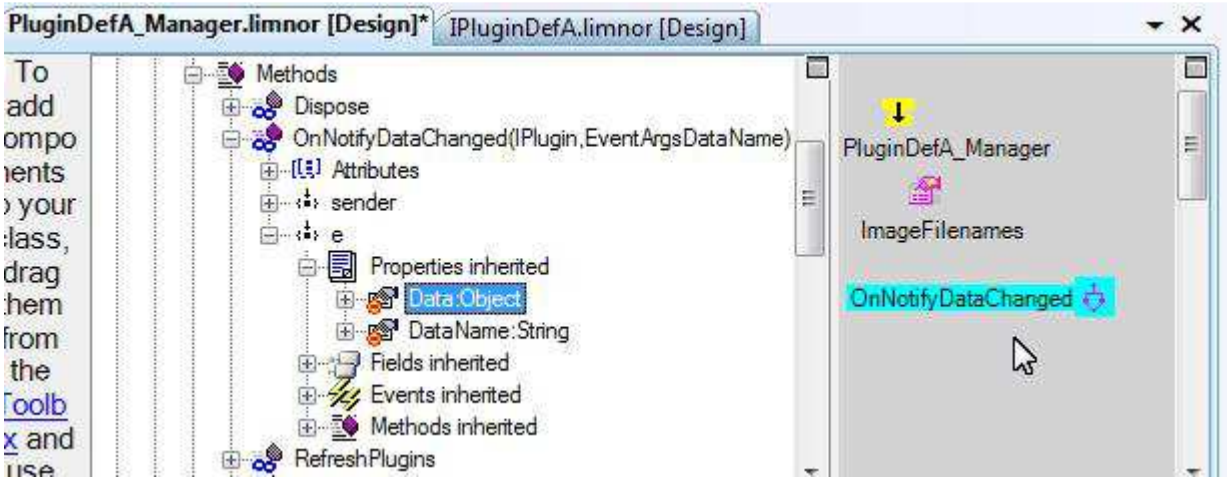
Respond to data change

The default behavior of the plug-in manager works in such a way: when a plug-in notifies data change, OnNotifyDataChanged on the plug-in manager is executed. OnNotifyDataChanged executes OnDataChanged on every plug-in except the plug-in sending the data change notice.

Because our new plug-in manager uses a property named ImageFilenames to hold data, we may use the opportunity of executing OnNotifyDataChanged to save the modified data to ImageFilenames. This is done by overriding OnNotifyDataChanged. Right-click OnNotifyDataChanged, choose “Override”:



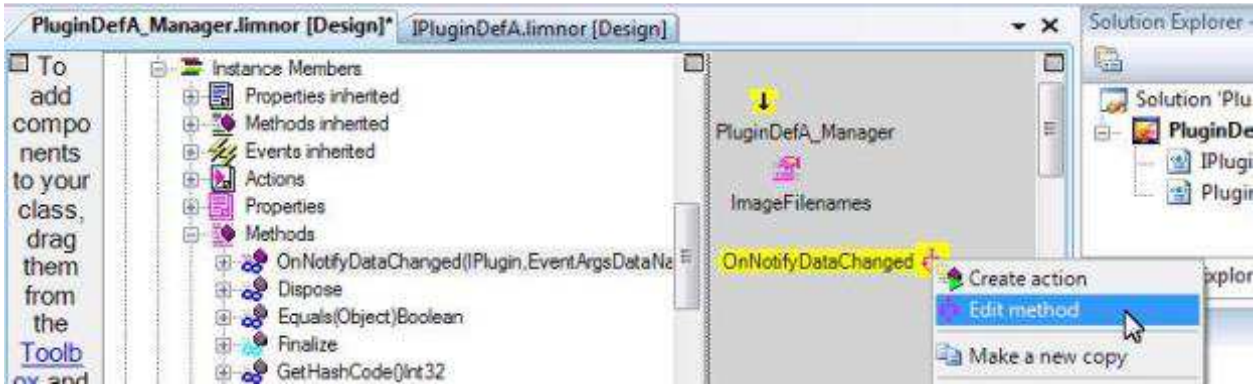
The override of OnNotifyDataChanged is created.



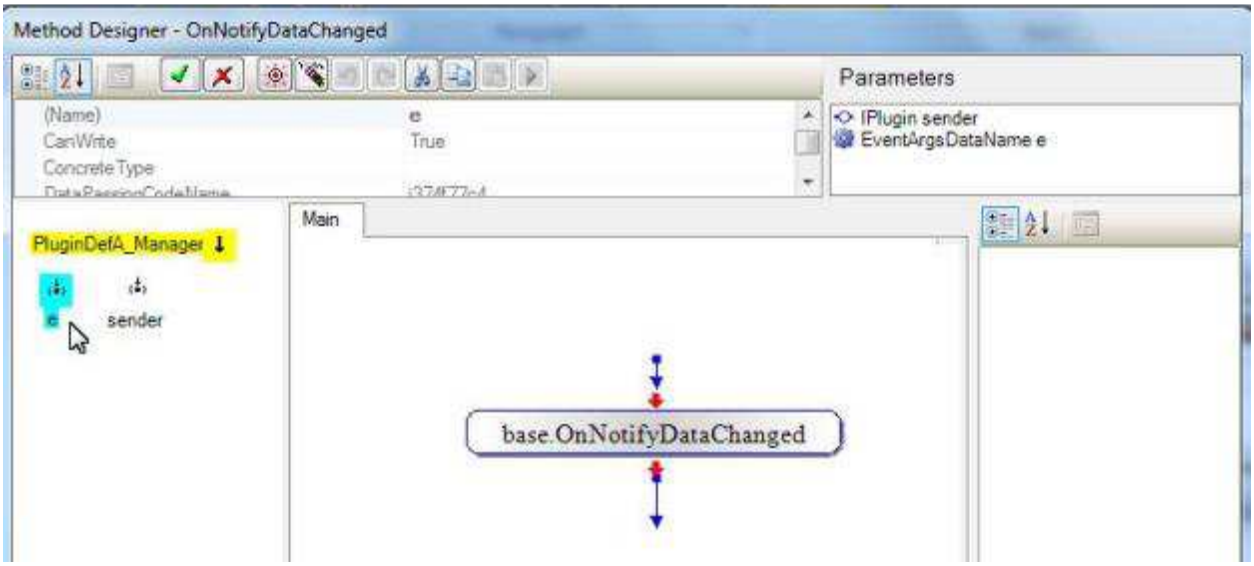
Note that parameter “sender” is the plug-in sending the data change notification. Parameter “e” describes the data changed. “Data” property of “e” is the new data. You may also use “DataName” of “e” to identify data changed. In our sample, we only have one “ImageFilenames”; we do not need to use data names to distinguish data.

We may edit the override of OnNotifyDataChanged to pass “Data” of “e” to “ImageFilenames”.

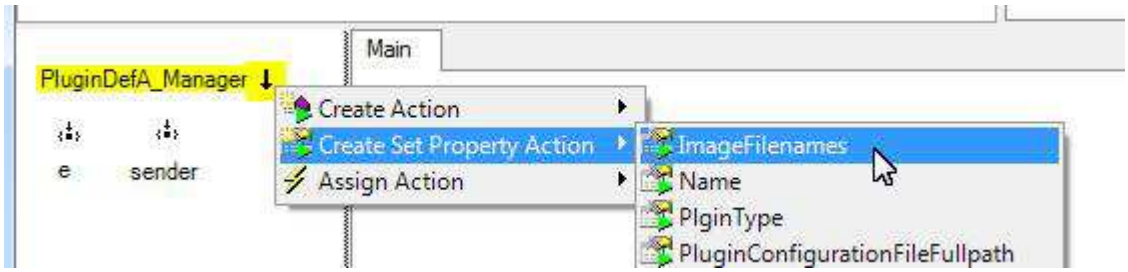
Right-click the override; choose “Edit” to modify the override:

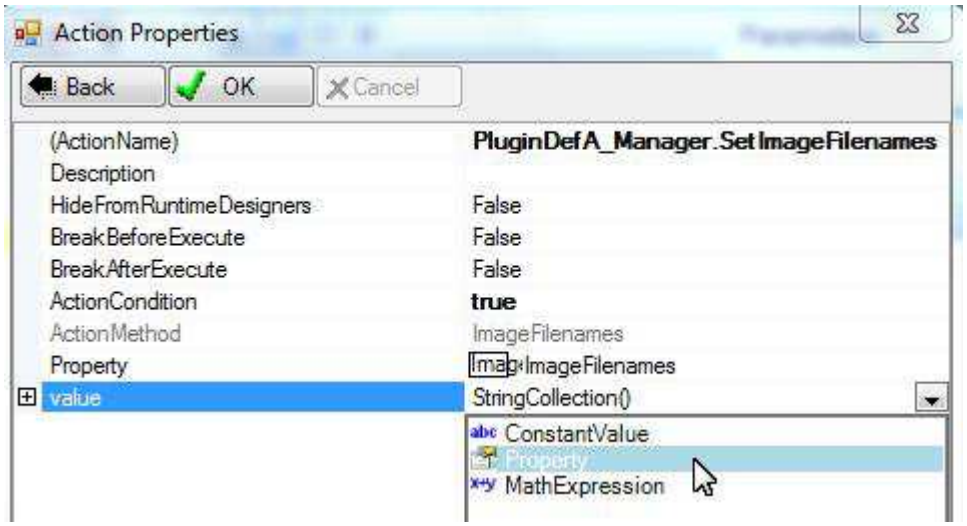


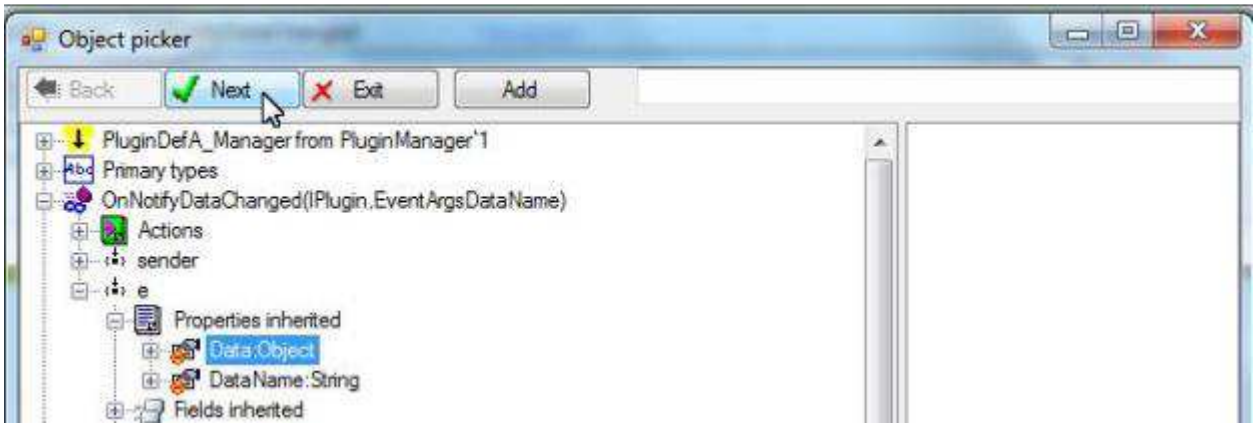
A method editor appears for editing the override.



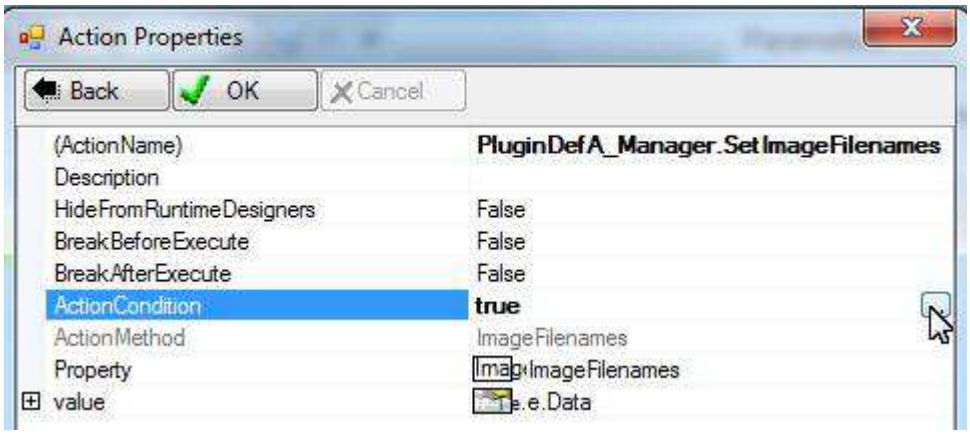
Create an action to set “ImageFilenames” to “Data” property of “e”:

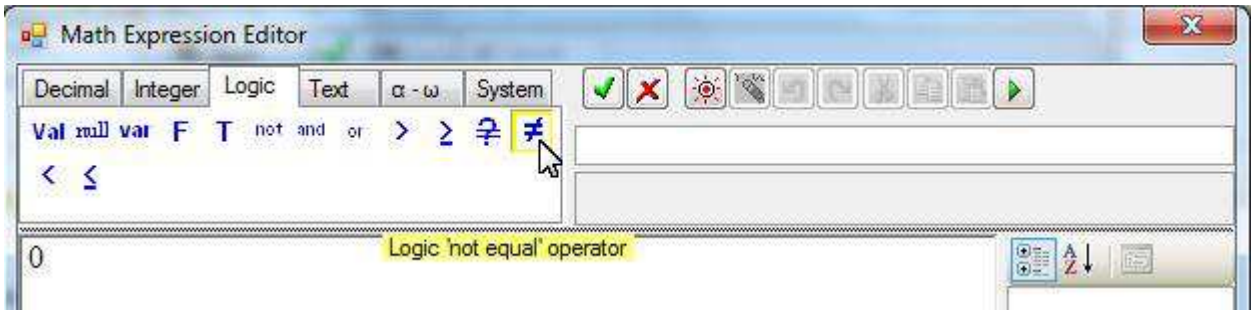


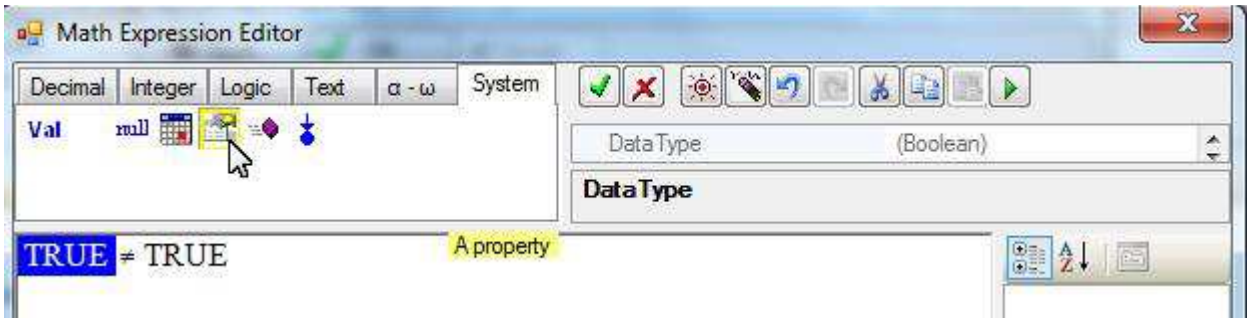


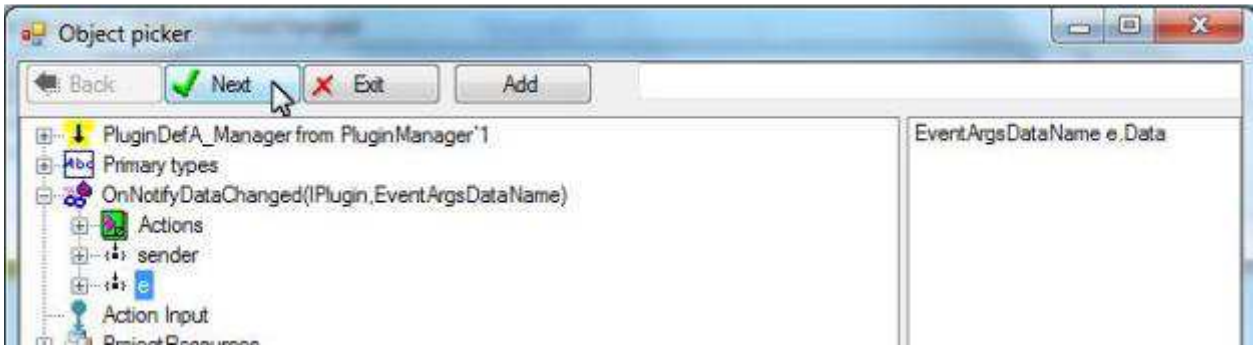


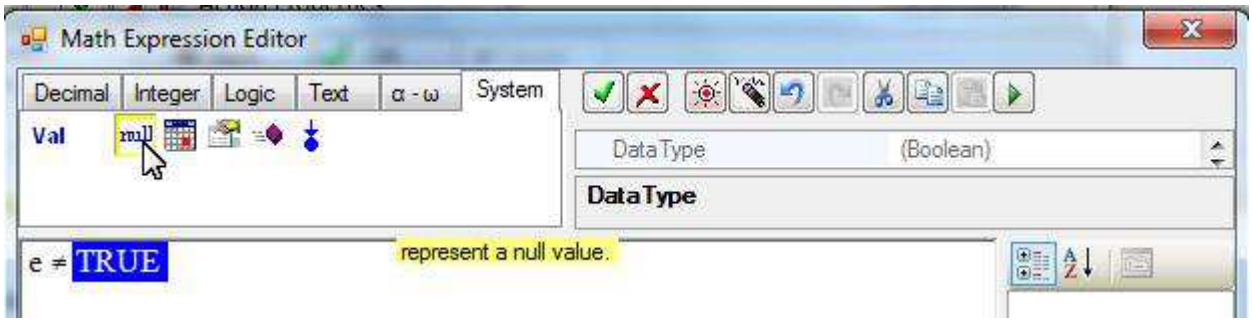
We need to set ActionCondition to check that “e” is not null and “Data” of “e” is not null:

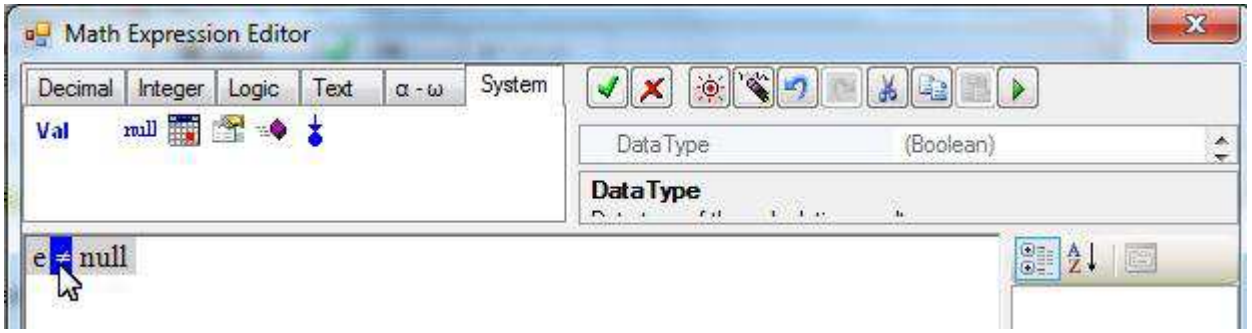


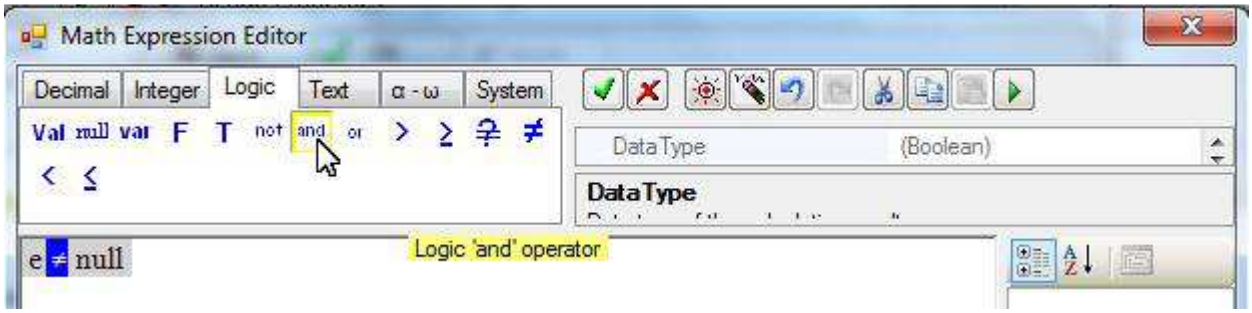


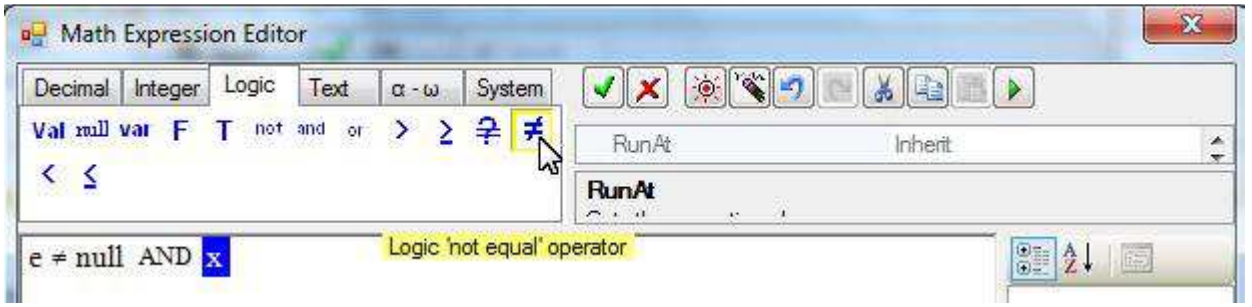


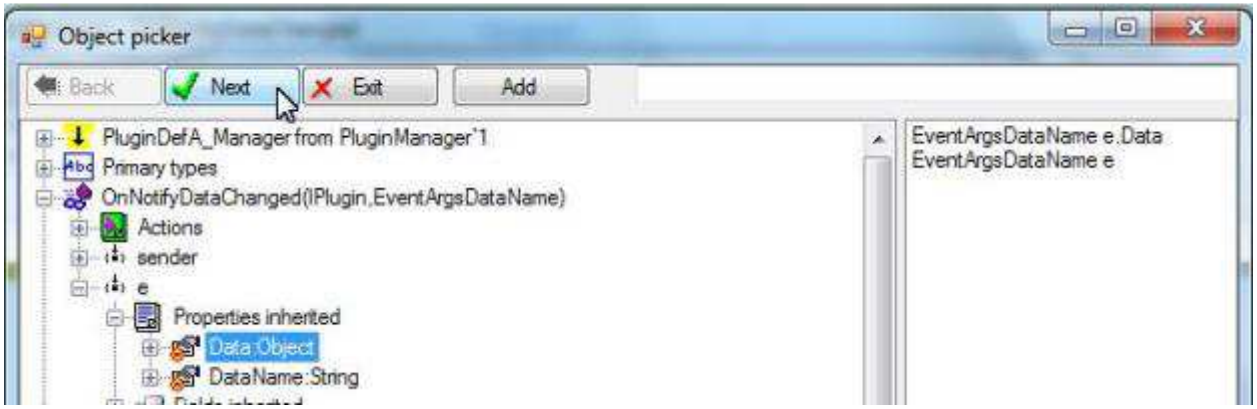

Object picker
Back
Next
Exit
Add
PluginDefA_Manager from PluginManager*1
Primary types
OnNotifyDataChanged(IPlugin,EventArgsDataName)
Actions
sender
e
Action Input
EventArgsDataName e.Data


Math Expression Editor
Decimal
Integer
Logic
Text
α - ω
System
Val
null
DataType
(Boolean)
DataType
e ≠ TRUE
represent a null value.

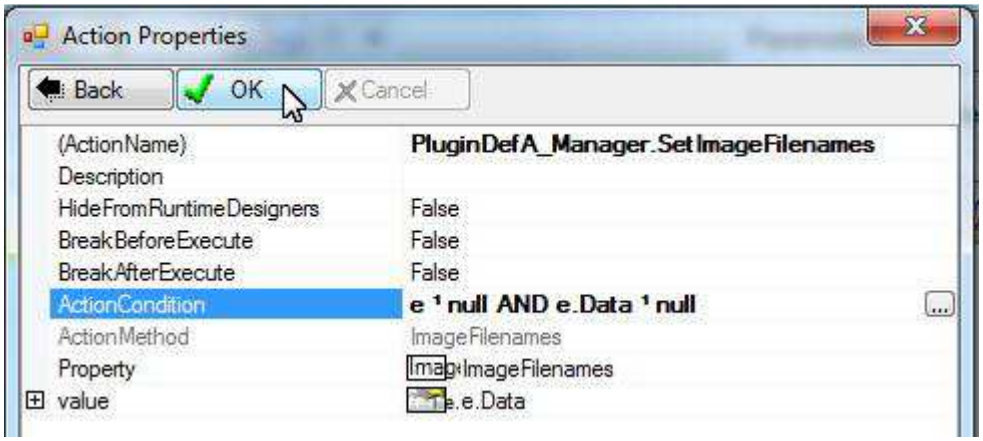

Math Expression Editor
Decimal
Integer
Logic
Text
α - ω
System
Val
null
DataType
(Boolean)
DataType
e ≠ null


Math Expression Editor
Decimal
Integer
Logic
Text
α - ω
System
Val null var F T not and or > ≥ ≠ < ≤
DataType
(Boolean)
DataType
e ≠ null
Logic 'and' operator

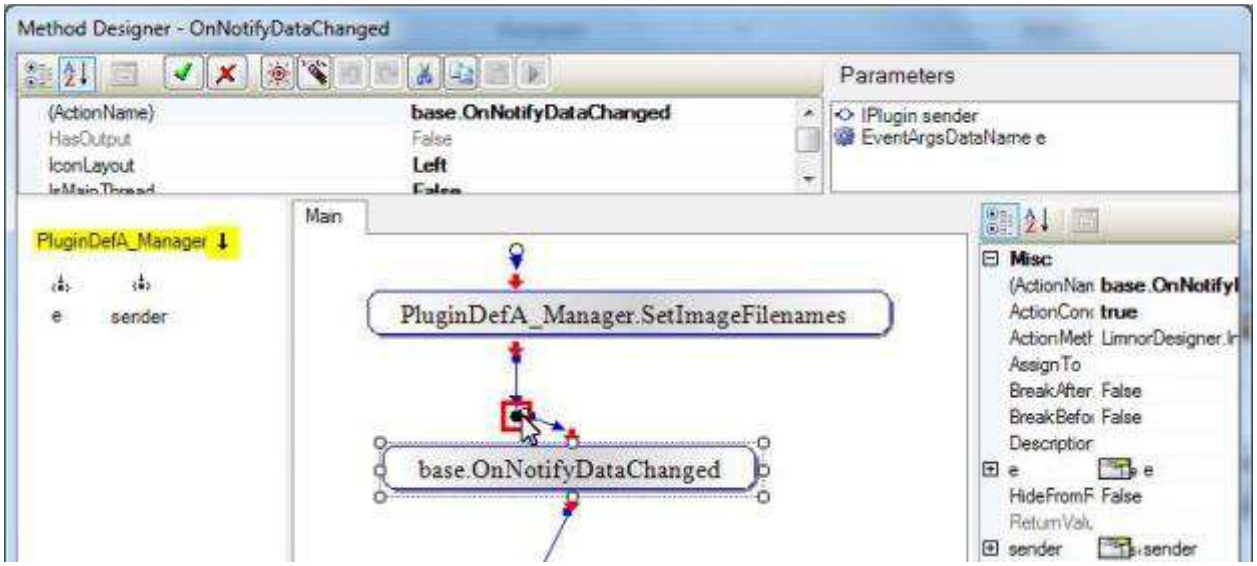

Math Expression Editor
Decimal
Integer
Logic
Text
α - ω
System
Val null var F T not and or > ≥ ≠ < ≤
RunAt
Inherit
RunAt
e ≠ null AND x
Logic 'not equal' operator



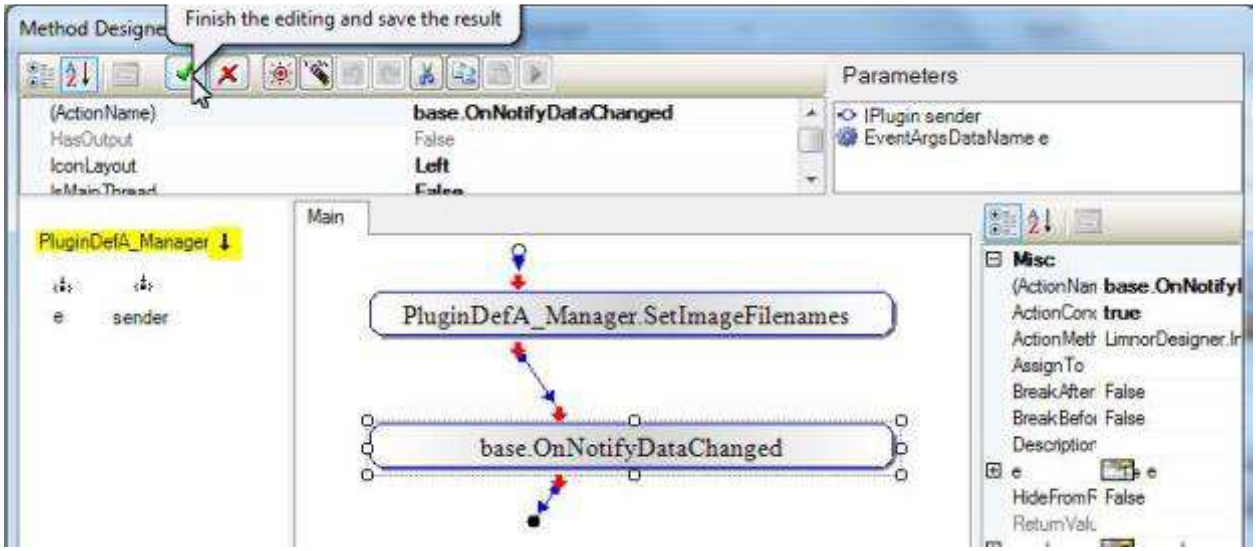
Click OK to finish creating the action:



Link the "base" action to the new action:

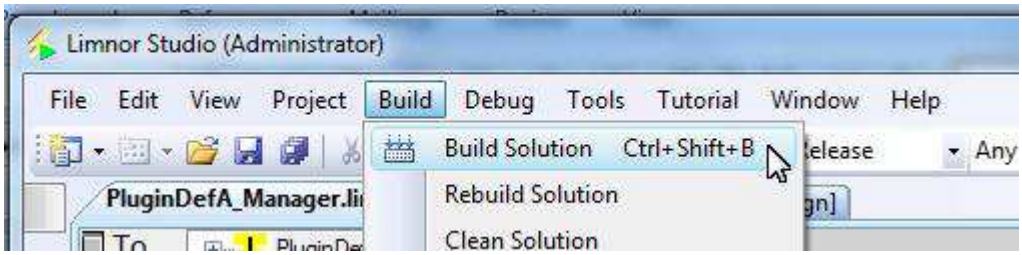


Action "base.OnNotifyDataChanged" needs to be the last action. Action "base.OnNotifyDataChanged" executes OnDataChanged of all plug-ins. Before its execution, we want to update data storage in the plug-in manager first.

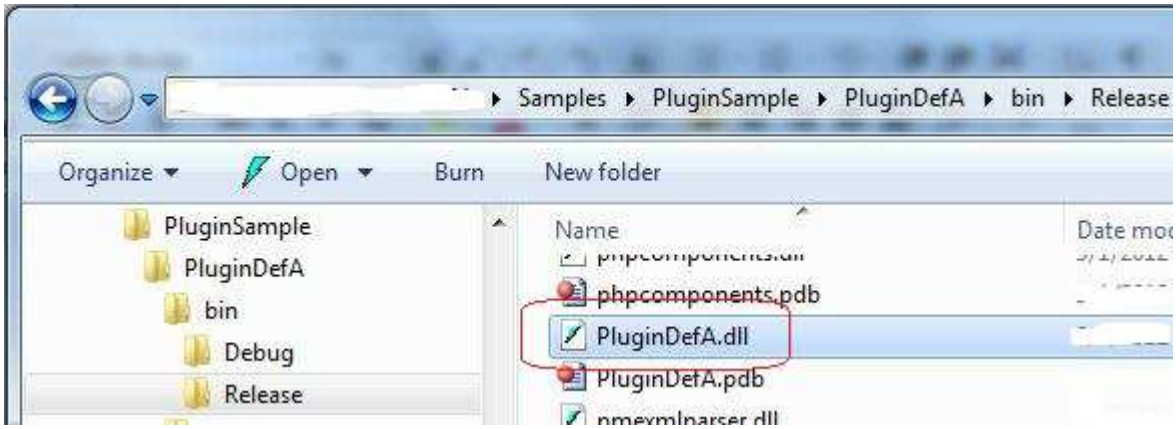


Compile DLL

For this demo, we do not need further development for our plug-in manager because the base class already provides all the functionality we want to use. We may compile the project:



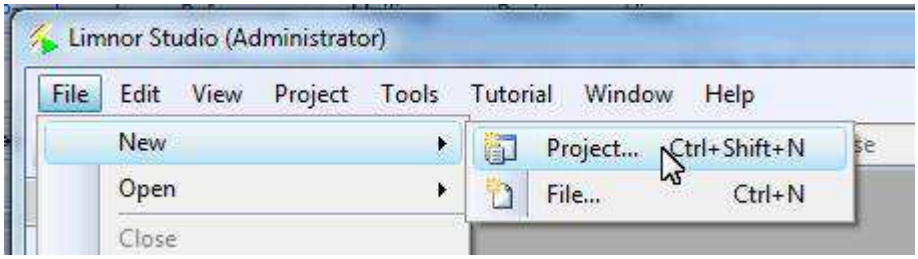
The DLL file is generated:



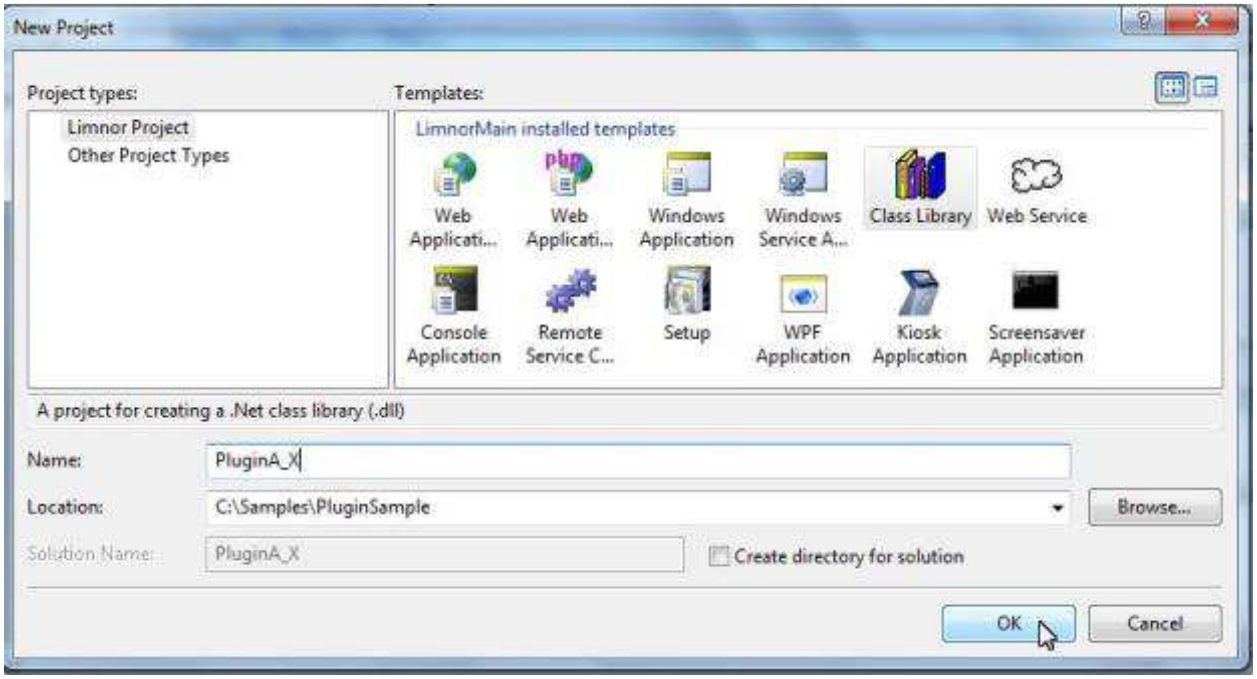
The first implementation of plug-in

Create a new DLL

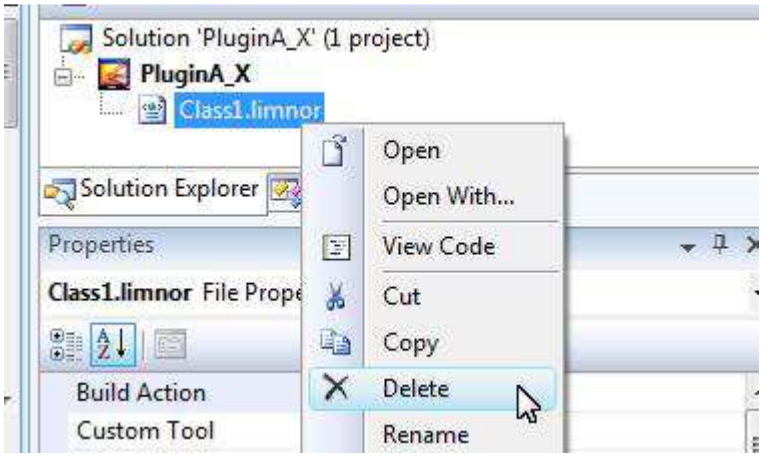
A plug-in implementation must be in a DLL so that a plug-in manager may load it.



Name the project PluginA_X:

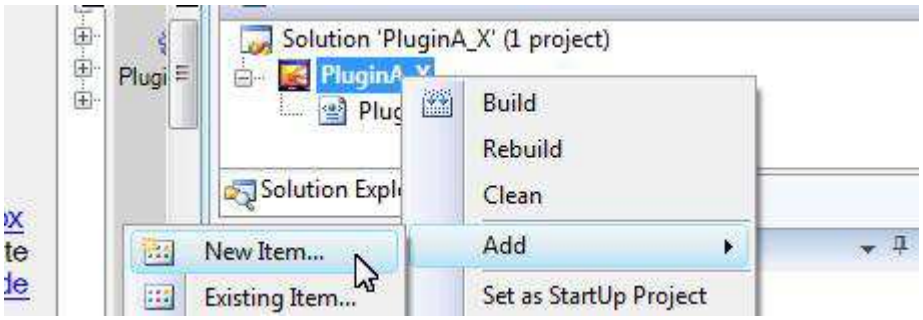


Delete Class1:

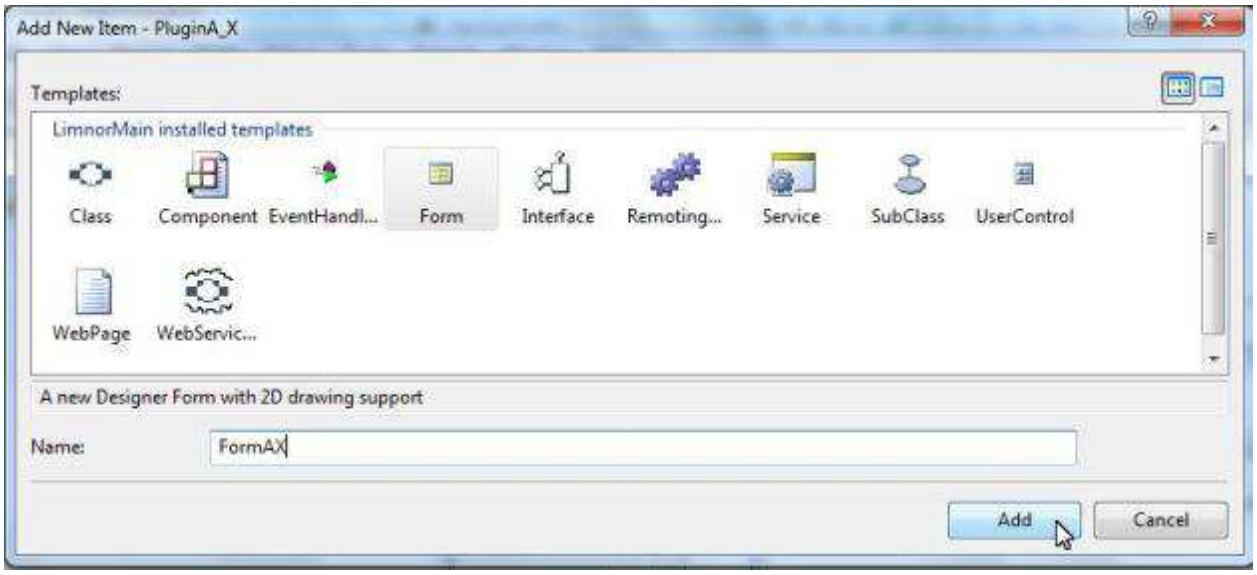


Add a new form

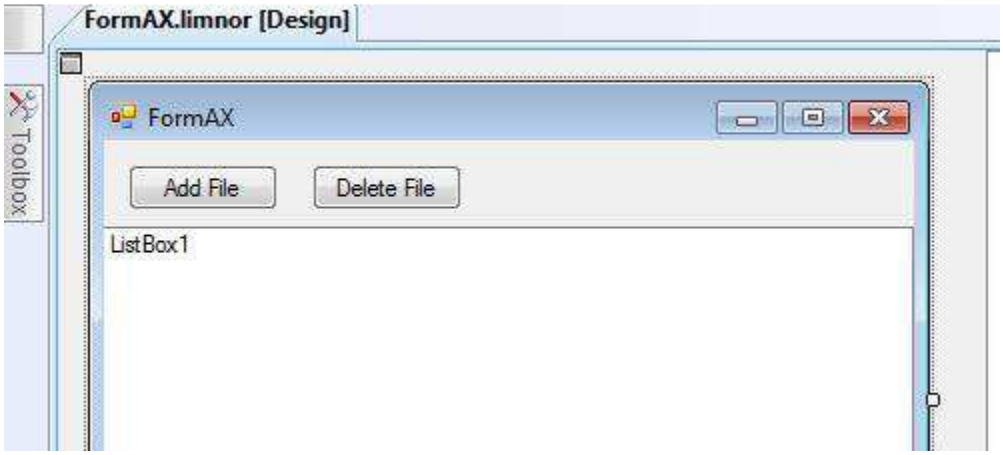
For this implementation, we want to use a list box to display all image file names. We add a Form to host a list box.



Name it FormAX:



Add UI elements to the form:

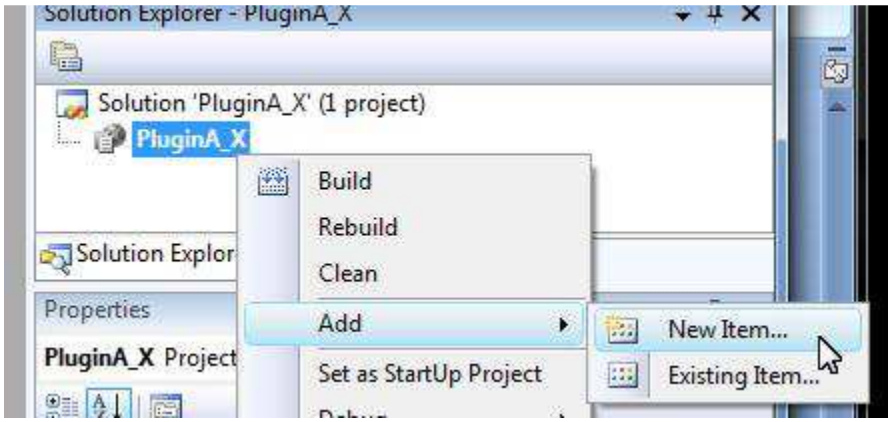


"Add File" button launches file browse dialogue to select a file and add file name to the list box. "Delete File" button removes selected list box item. We do not go into details of such programming.

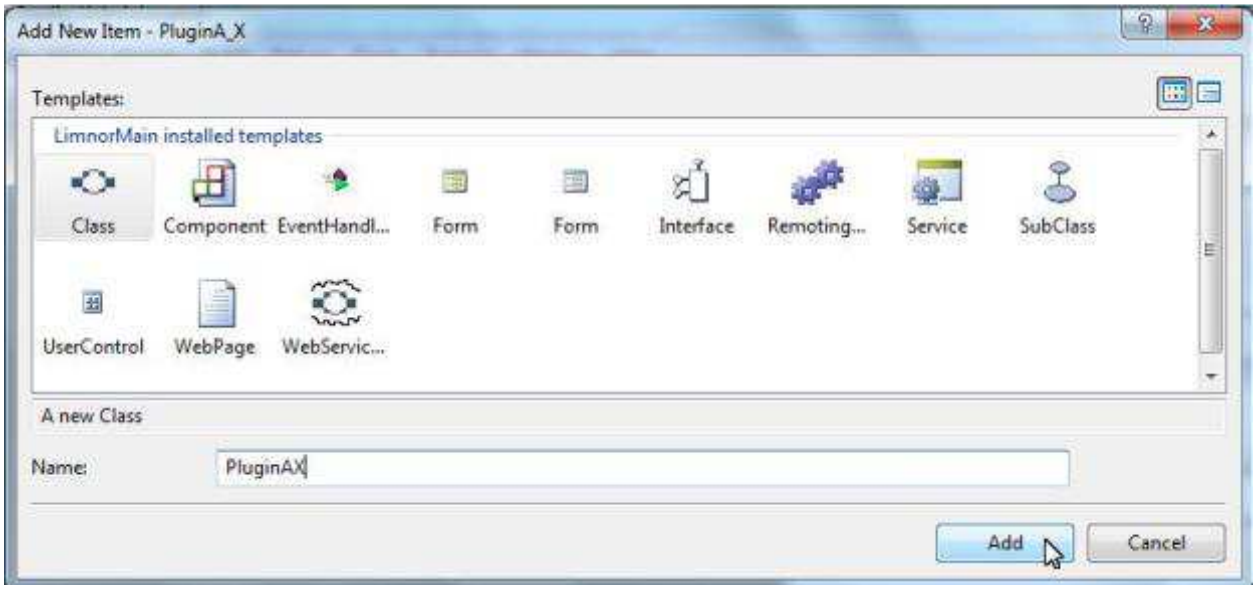
We also create a StringCollection property named "FileList". When setting "FileList", the strings are added to the list box. When getting "FileList", each item in the list box becomes a string in the StringCollection.

Create plug-in

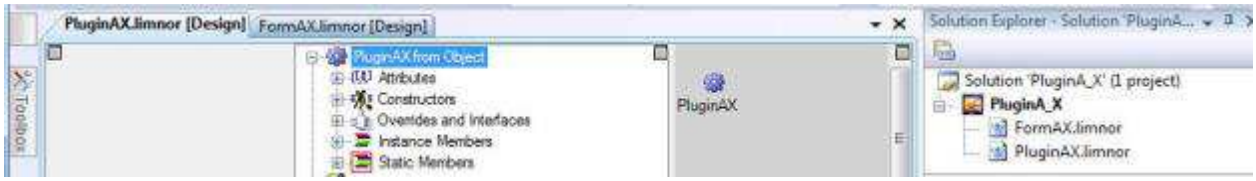
Add a new class:



Name the new class PluginAX:



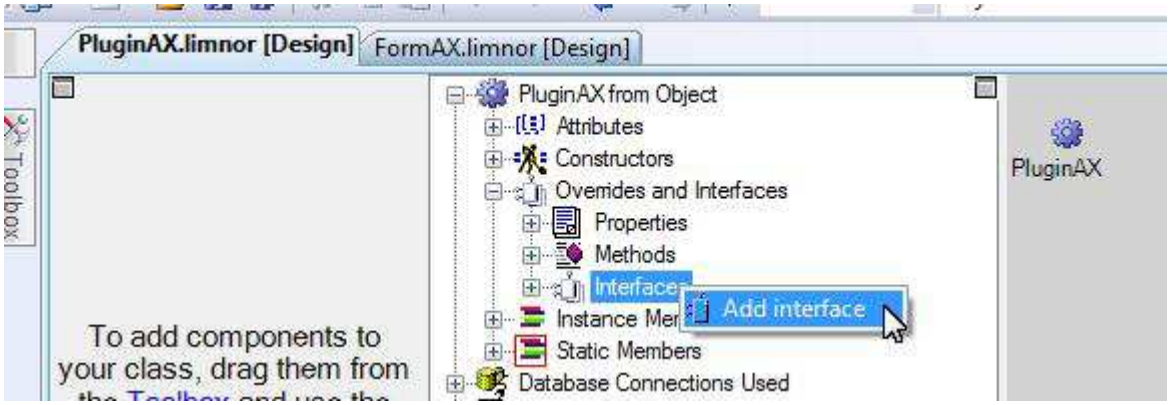
The new class is created:



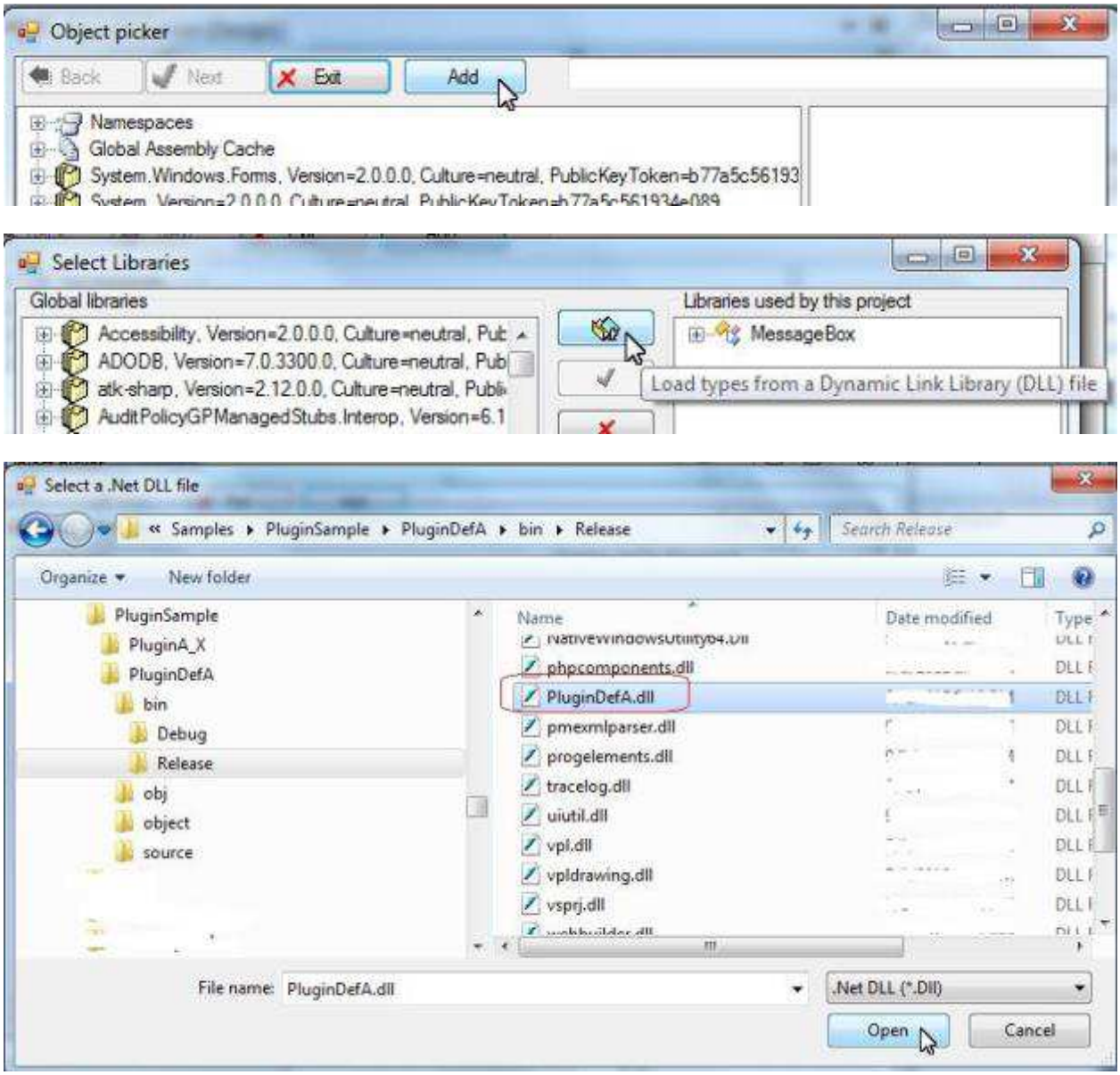
Implement Plug-in definition

Add plug-in definition interface

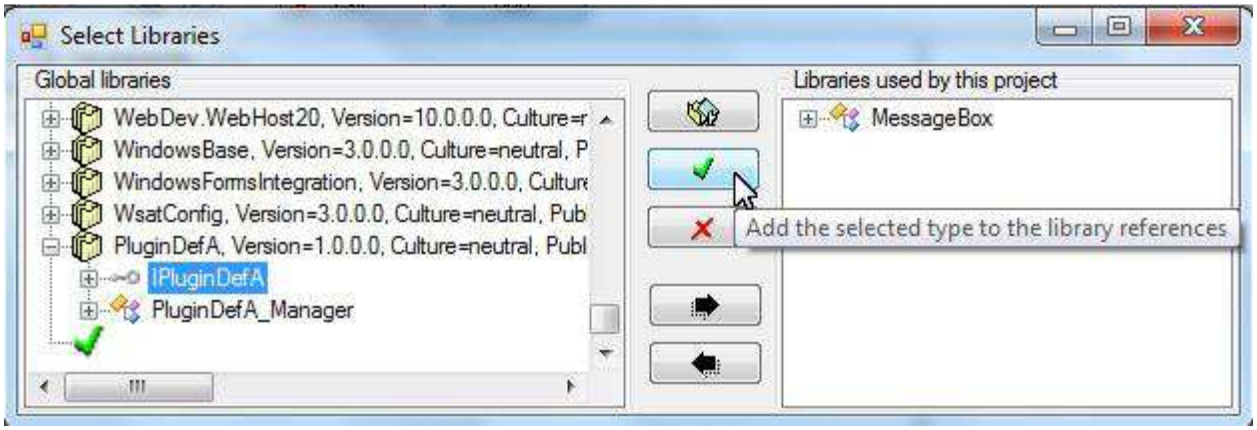
To make the class a plug-in, we need to add the plug-in definition interface to the class:



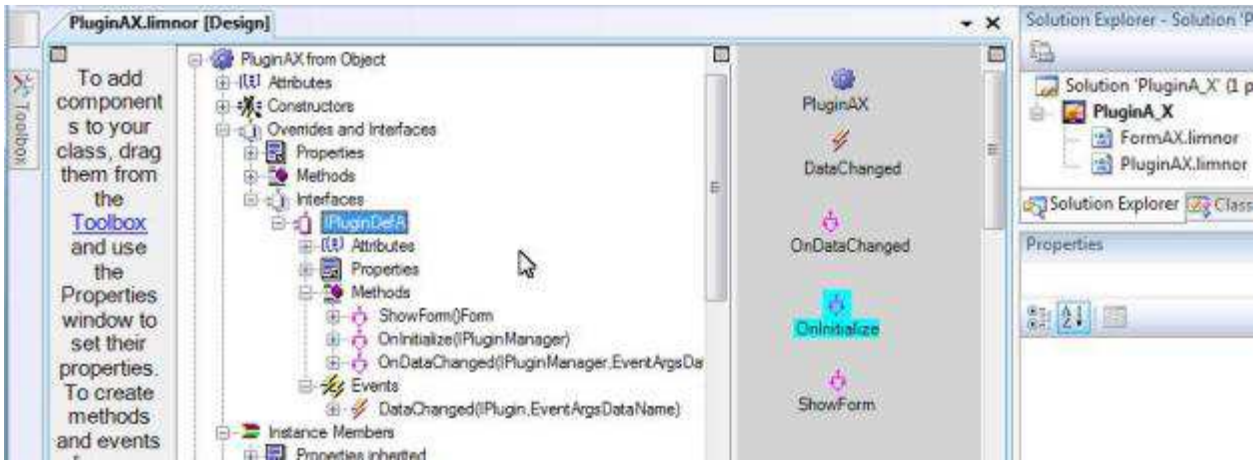
Click “Add” button to load plug-in definition DLL:



Select the plug-in definition interface:



On adding the plug-in definition interface, the members of the interface are created in the plug-in class:



Implement Plug-in Definition methods

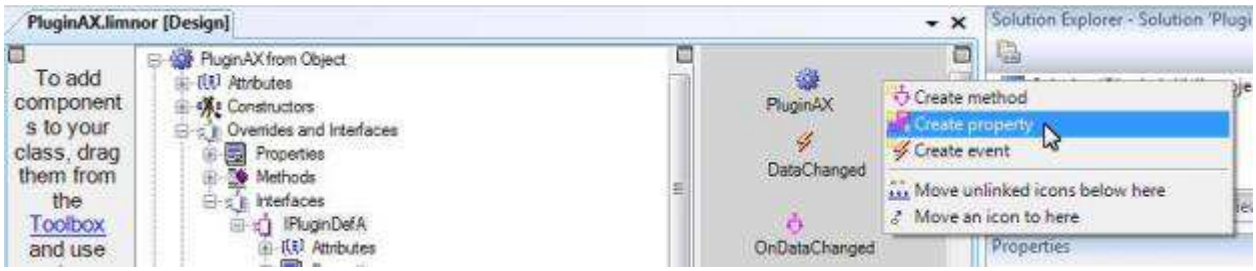
We need to edit methods, ShowForm, OnInitialize, and OnDataChanged, to complete the implementation of this plug-in.

Implement OnInitialize

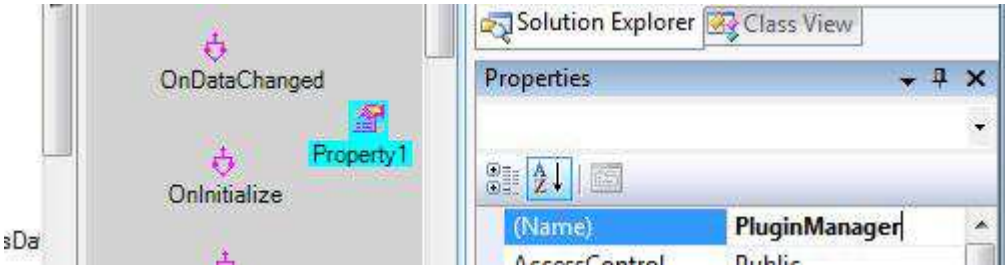
For this sample, we may do following things in this method:

1. Remember the plug-in manager for later access
2. Pass “ImageFilenames” property from the plug-in manager to the Form’s “FileList” property

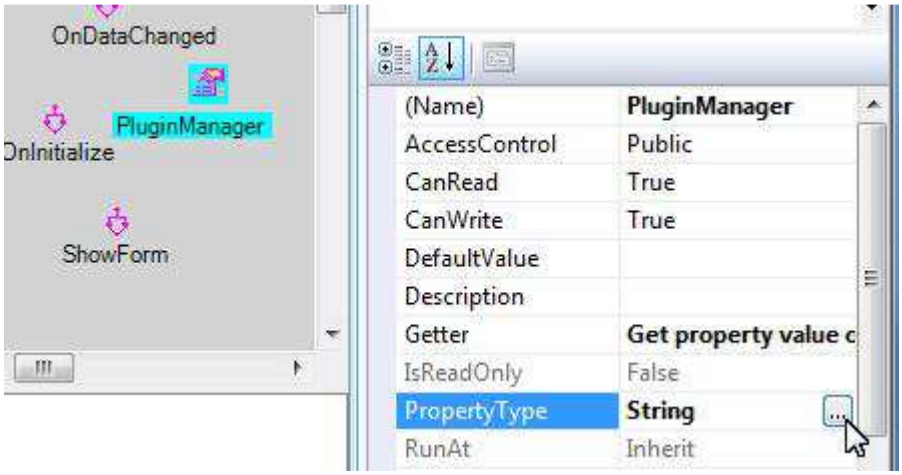
In order to do the first thing, we create a property to remember the plug-in manager:



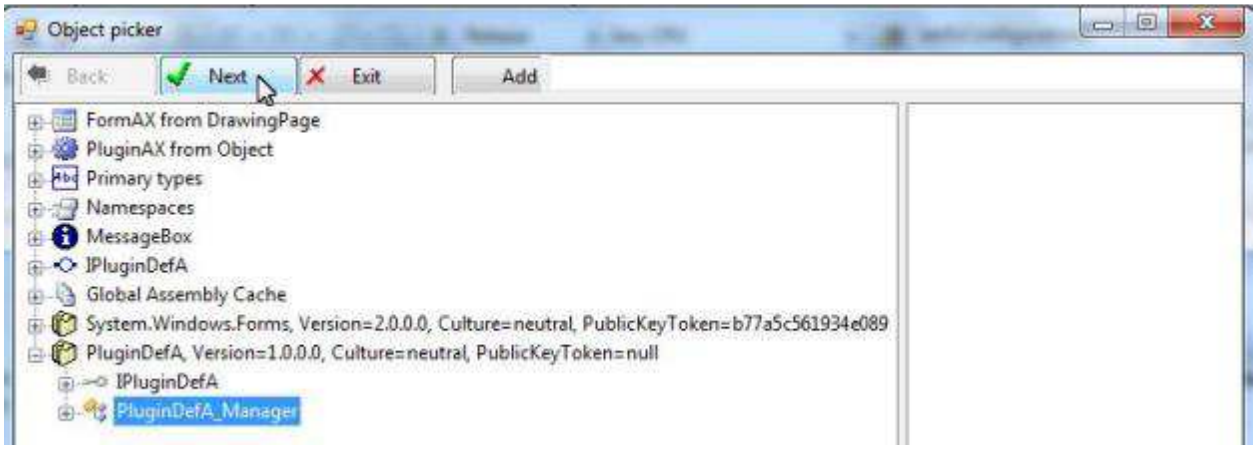
Rename the property to “PluginManager”:



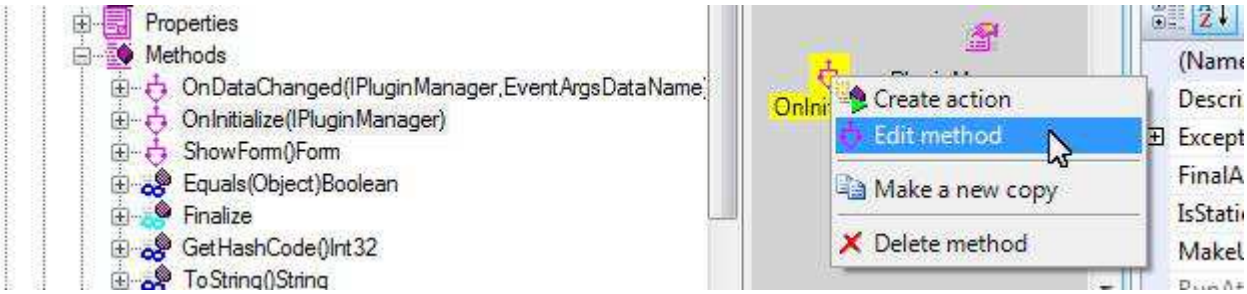
Change the property type to be the plug-in manager:



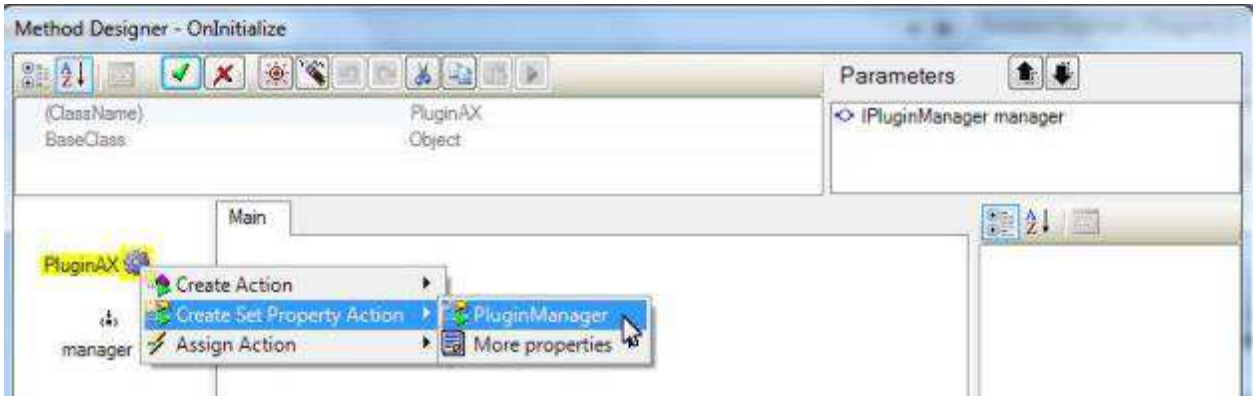
Select the plug-in manager:

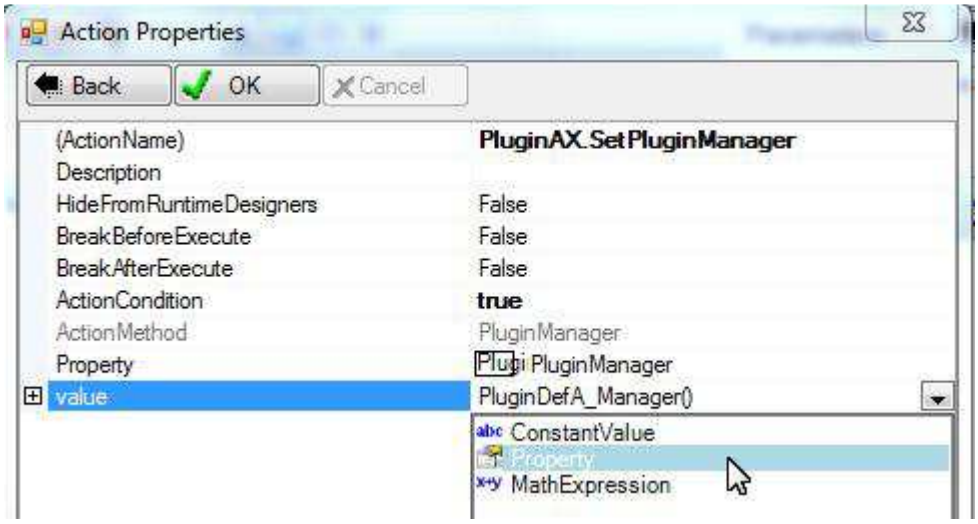


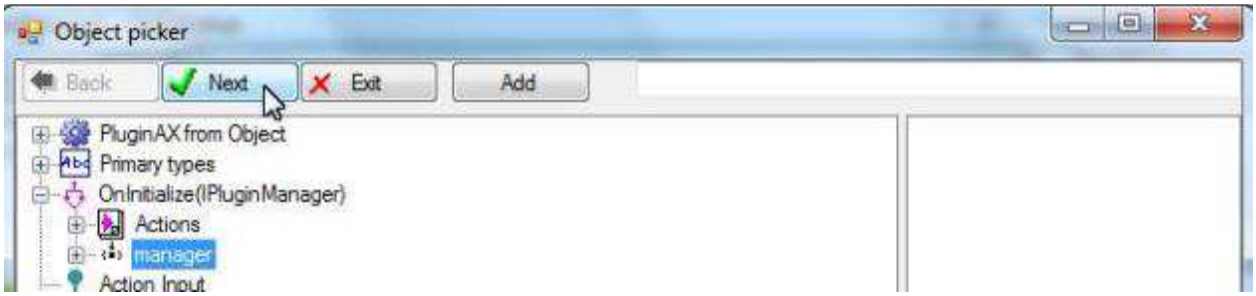
Now we may edit method OnInitialize to remember the plug-in manager.

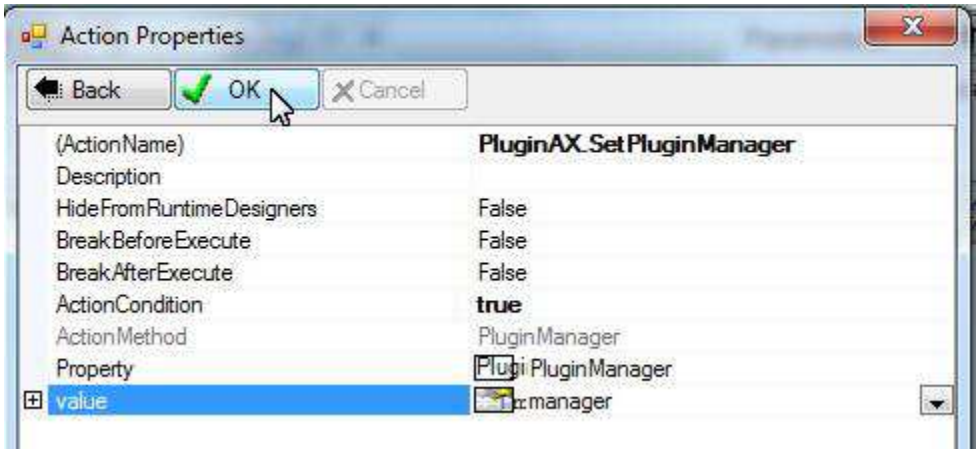


A method editor appears. Add an action to set the PluginManager:

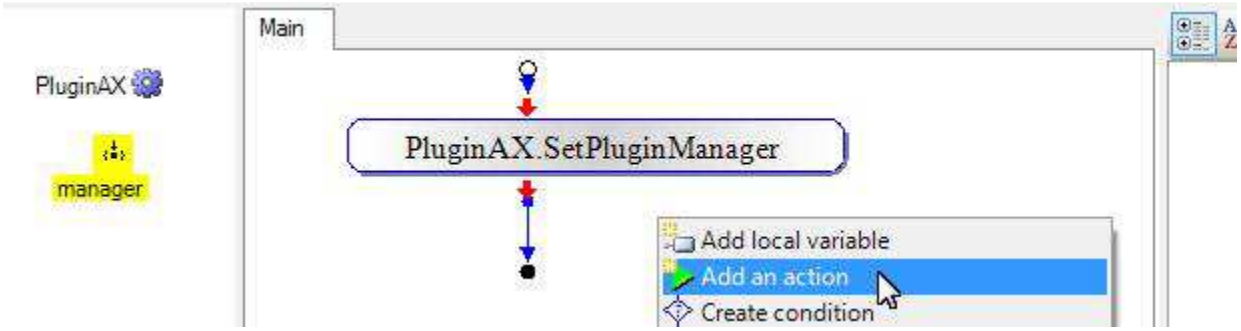


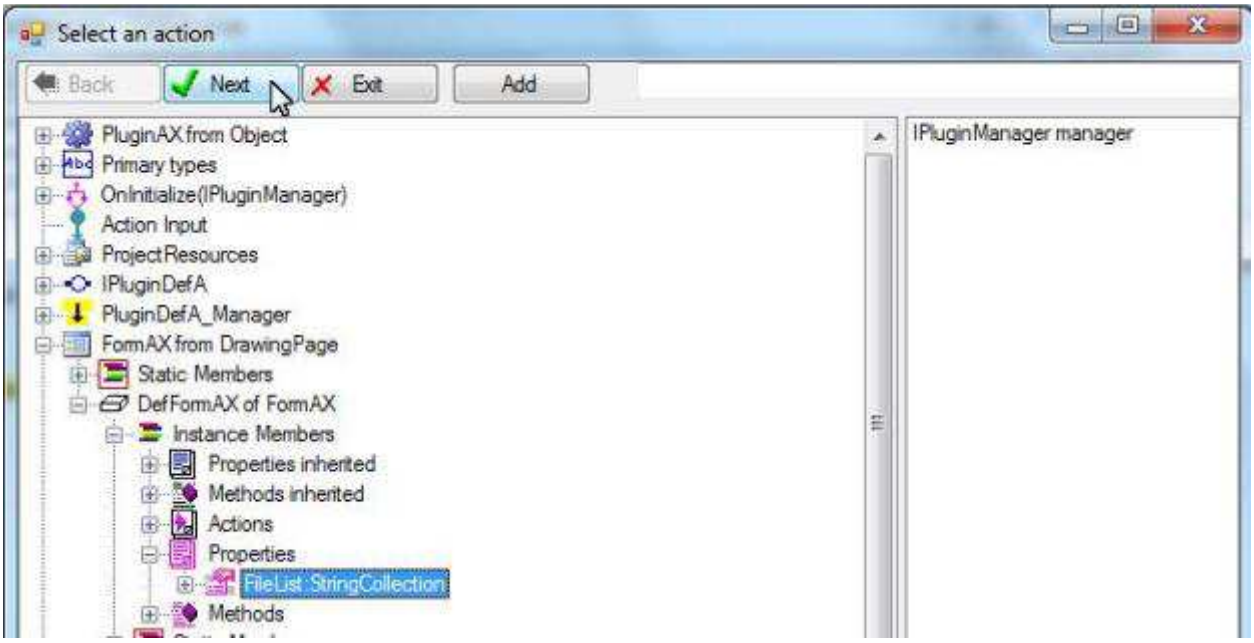


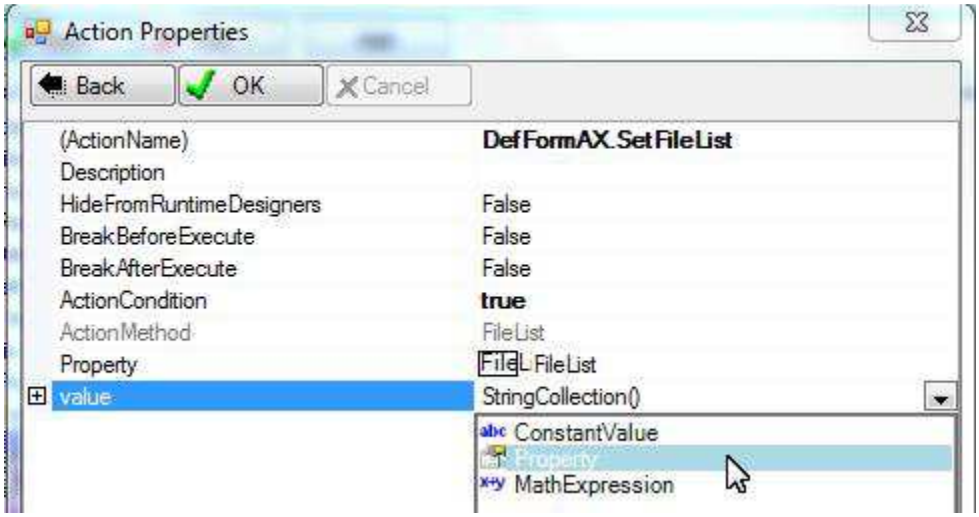


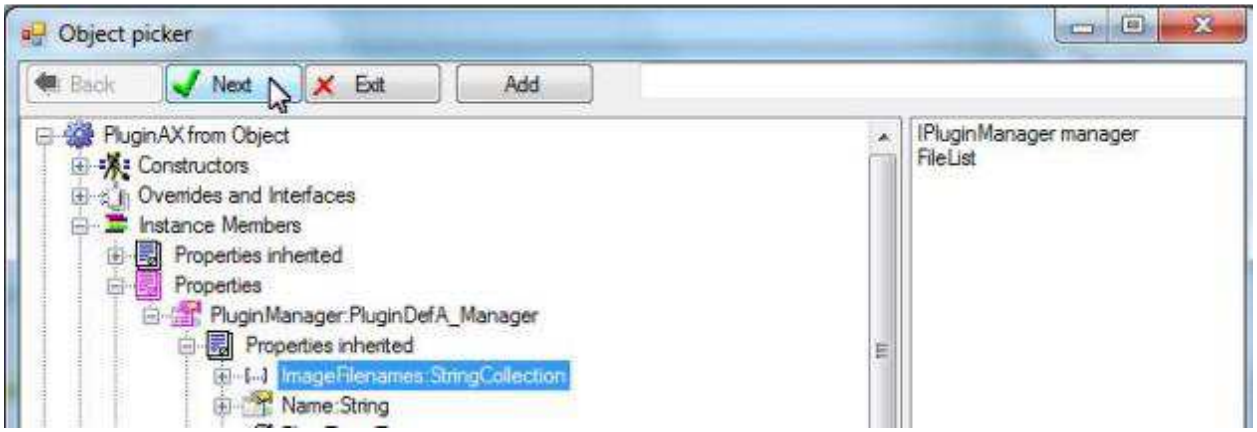


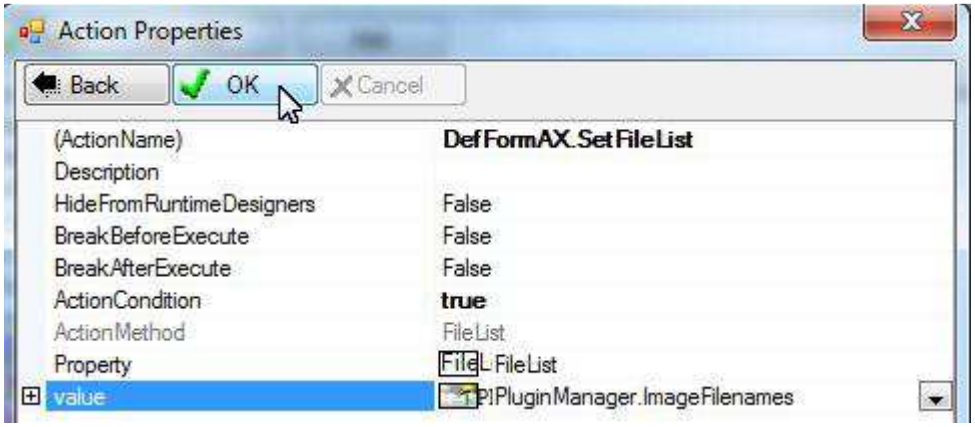
Add another action to pass “ImageFilenames” on the plug-in manager to “FileList” on the form:


Main
PluginAX
manager
PluginAX.SetPluginManager
Add local variable
Add an action
Create condition

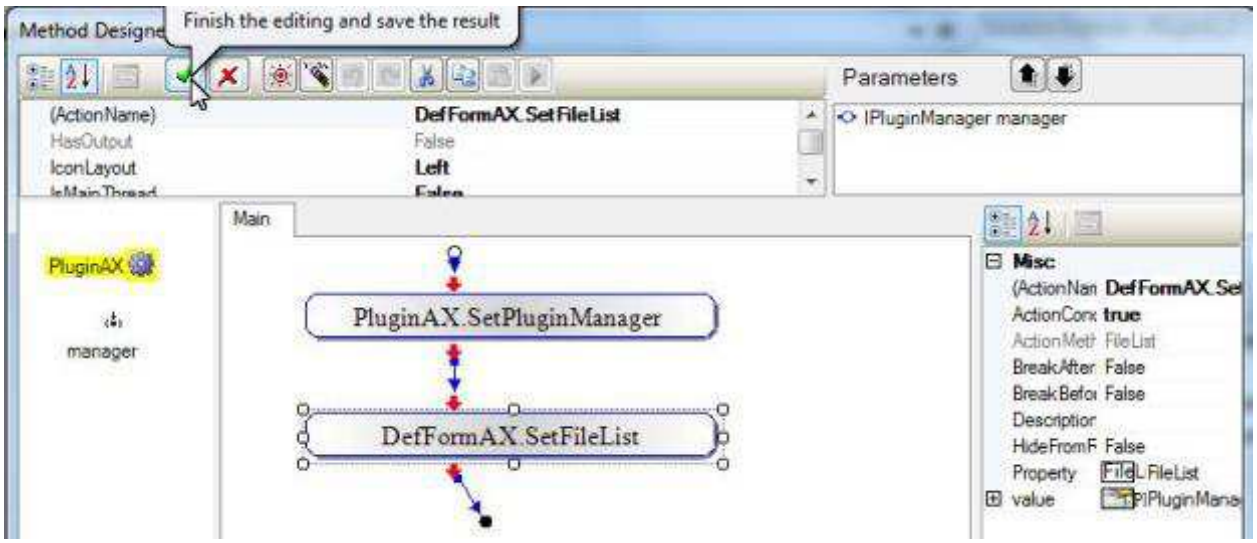

Select an action
Back
Next
Exit
Add
PluginAX from Object
Primary types
OnInitialize(IPluginManager)
Action Input
Project Resources
IPluginDefA
PluginDefA_Manager
FormAX from DrawingPage
Static Members
DefFormAX of FormAX
Instance Members
Properties inherited
Methods inherited
Actions
Properties
FileList StringCollection
Methods
IPluginManager manager


Action Properties
Back
OK
Cancel
(ActionName)
DefFormAX.SetFileList
Description
HideFromRuntimeDesigners
False
BreakBeforeExecute
False
BreakAfterExecute
False
ActionCondition
true
ActionMethod
FileList
Property
FileList
value
StringCollection()
ConstantValue
Property
MathExpression





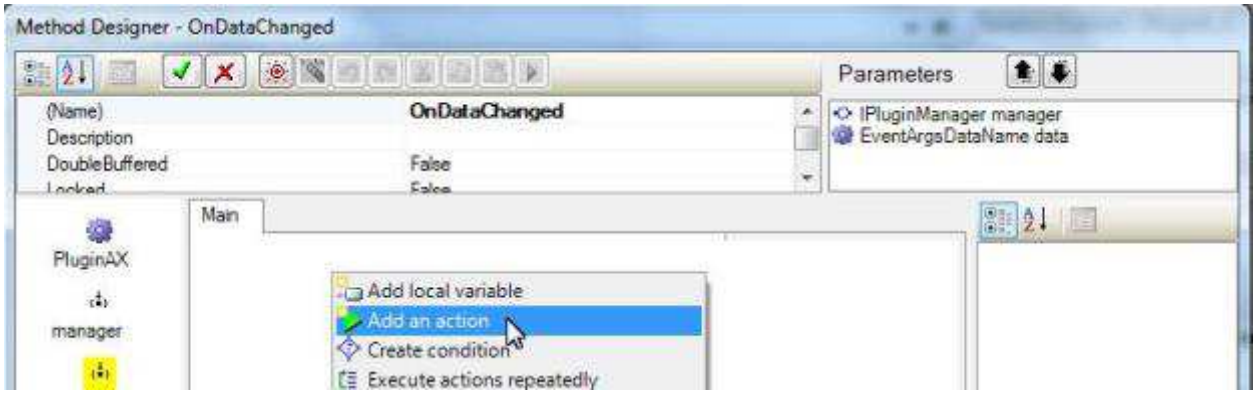
Link the two actions. We are done editing the method:

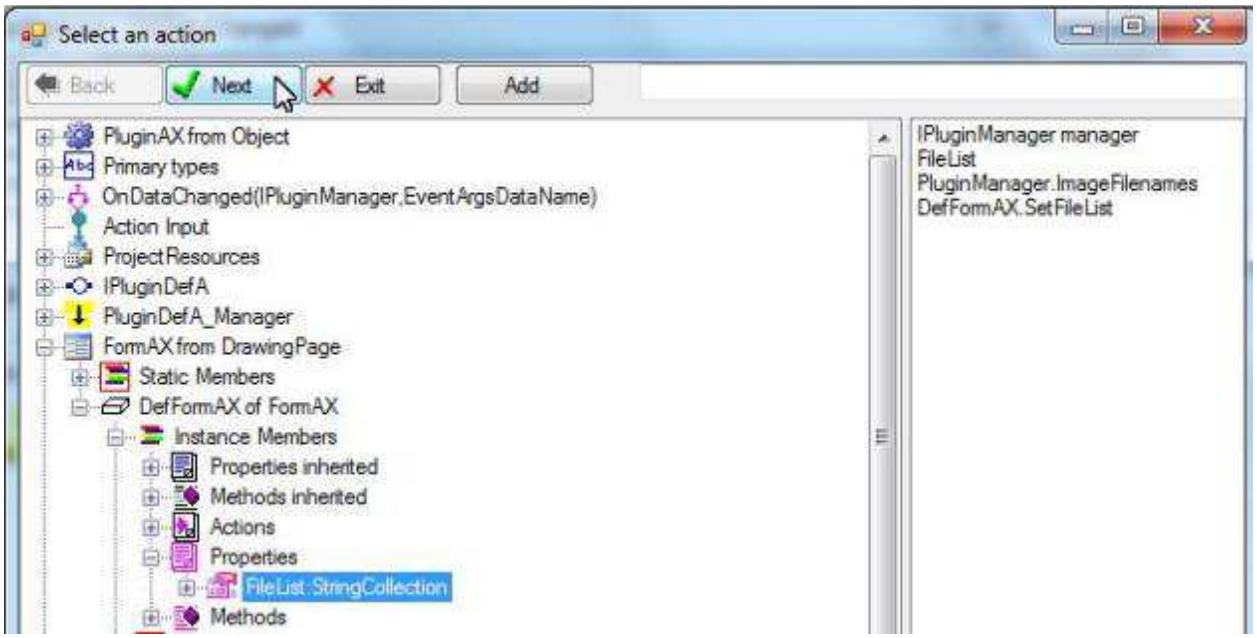


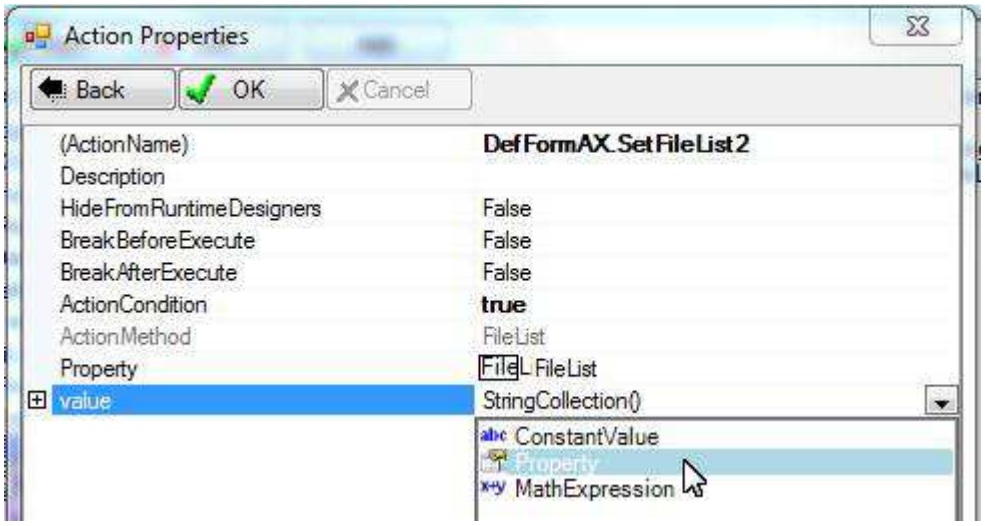
Implement OnDataChanged

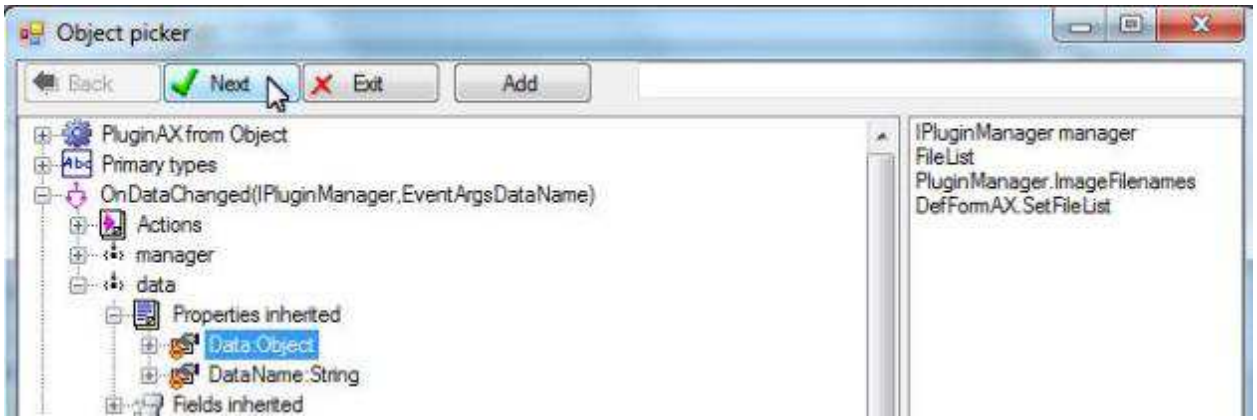
For this plug-in, we want to pass the data to “FileList” of the form.

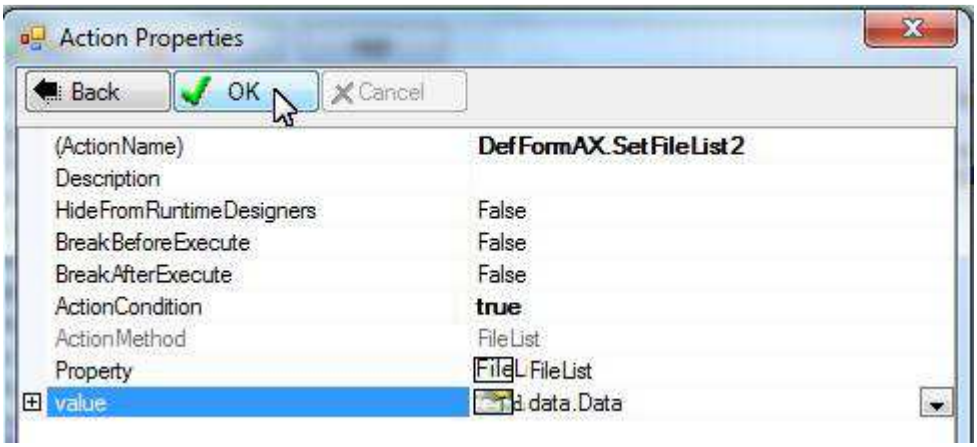

Events inherited
Actions
Properties
Methods
OnDataChanged(IPluginManager,EventArgsDataName)
OnInitialize(IPluginManager)
ShowForm()Form
OnDataChanged
Create action
Edit method
Make a new copy
Delete method
OnInitialize
FinalActions

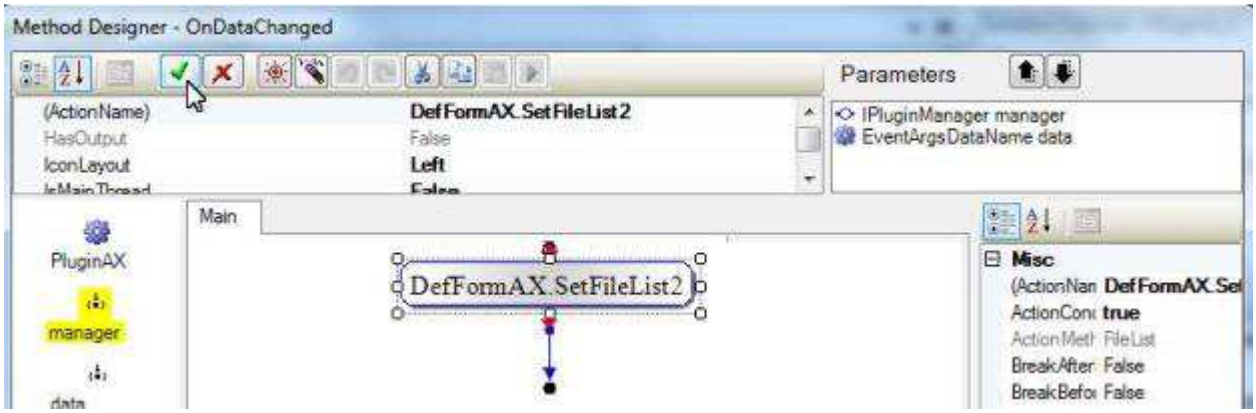

Method Designer - OnDataChanged
Parameters
(Name)
OnDataChanged
Description
DoubleBuffered
False
IPluginManager manager
EventArgsDataName data
Main
PluginAX
manager
Add local variable
Add an action
Create condition
Execute actions repeatedly


Select an action
Back
Next
Exit
Add
PluginAX from Object
Primary types
OnDataChanged(IPluginManager,EventArgsDataName)
Action Input
Project Resources
IPluginDefA
PluginDefA_Manager
FormAX from DrawingPage
Static Members
DefFormAX of FormAX
Instance Members
Properties inherited
Methods inherited
Actions
Properties
FileList StringCollection
Methods
IPluginManager manager
FileList
PluginManager.ImageFilenames
DefFormAX.SetFileList


Action Properties
Back
OK
Cancel
(ActionName)
DefFormAX.SetFileList2
Description
HideFromRuntimeDesigners
False
BreakBeforeExecute
False
BreakAfterExecute
False
ActionCondition
true
ActionMethod
FileList
Property
FileList
value
StringCollection()
ConstantValue
Property
MathExpression


Object picker
Back
Next
Exit
Add
PluginAX from Object
Primary types
OnDataChanged(IPluginManager,EventArgsDataName)
Actions
manager
data
Properties inherited
Data:Object
DataName:String
Fields inherited
IPluginManager manager
FileList
PluginManager.ImageFilenames
DefFormAX.SetFileList

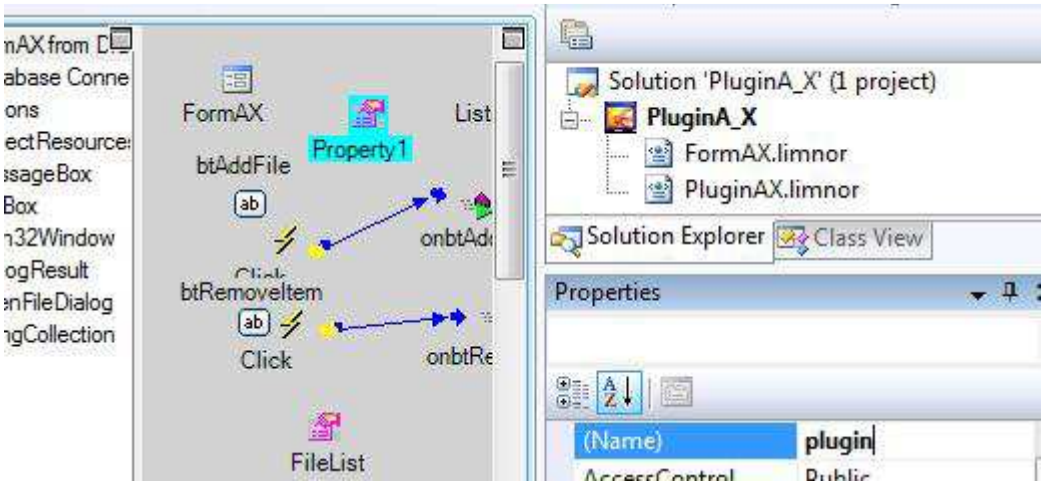

Action Properties
Back
OK
Cancel
(ActionName)
DefFormAX.SetFileList2
Description
HideFromRuntimeDesigners
False
BreakBeforeExecute
False
BreakAfterExecute
False
ActionCondition
true
ActionMethod
FileList
Property
FileList
value
data.Data



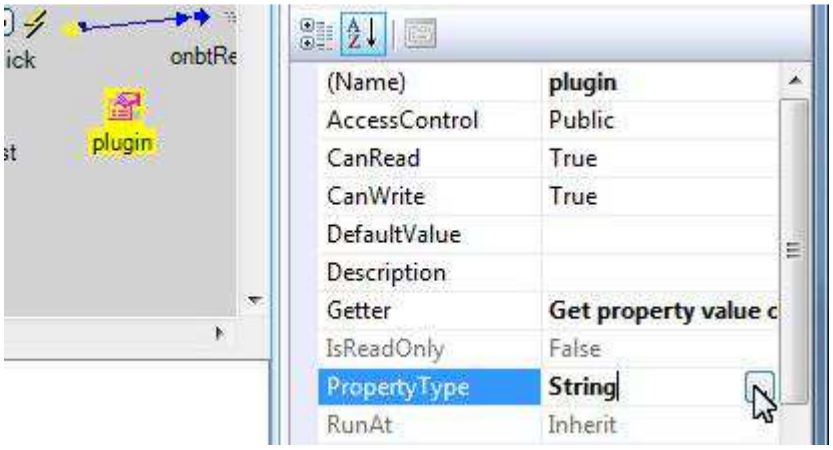
Implement ShowForm

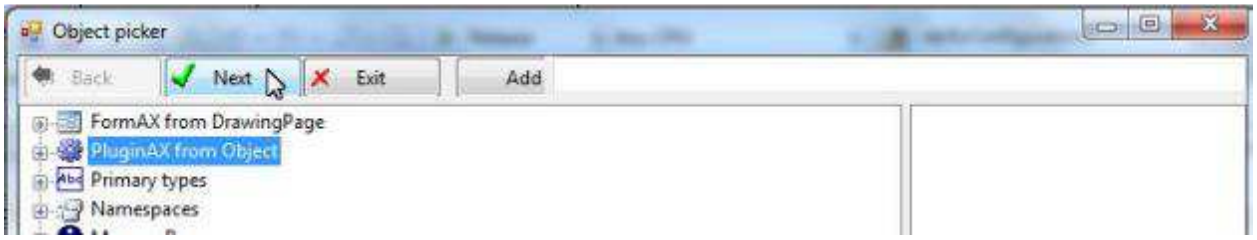
In this method, we need to execute a Show action of the form. We may also want the form to remember this plug-in object so that the form may use it later to send data to the plug-in manager. Let's first create a property in the form to remember the plug-in object:

Rename the property to "plugin":

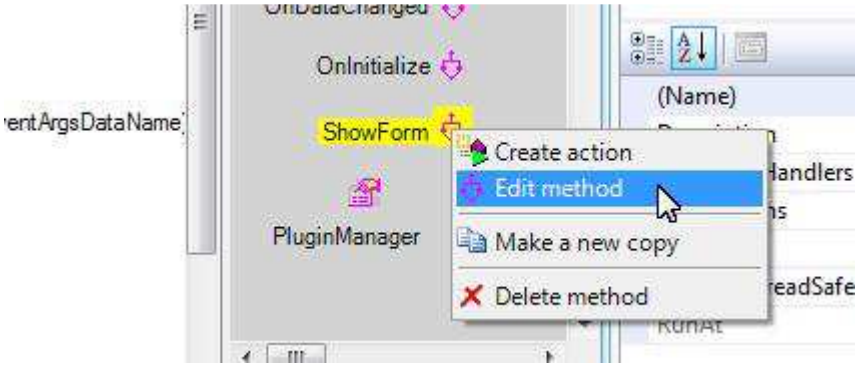


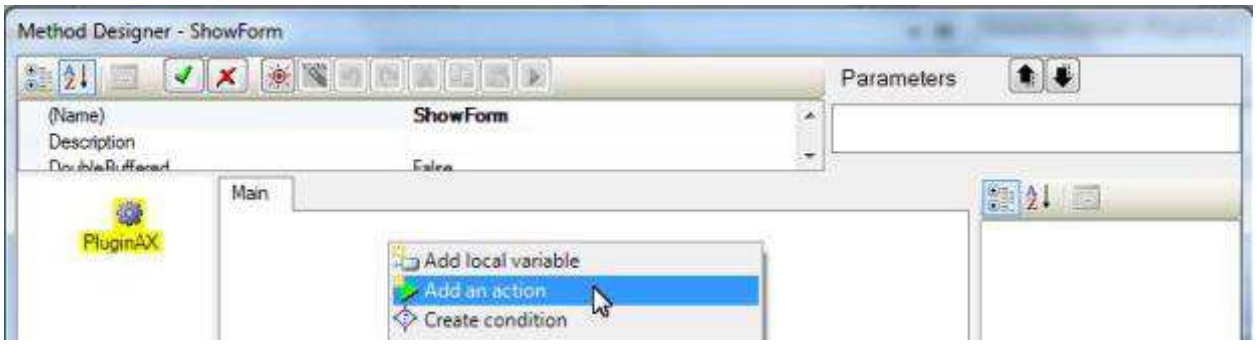
Change the property type to the plug-in class:

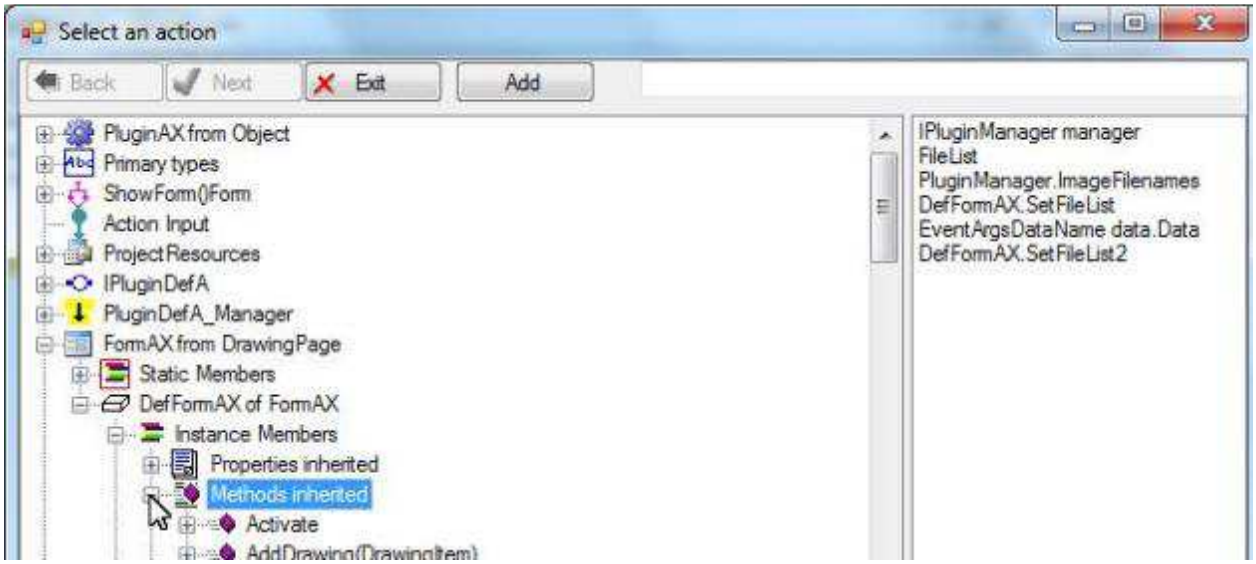


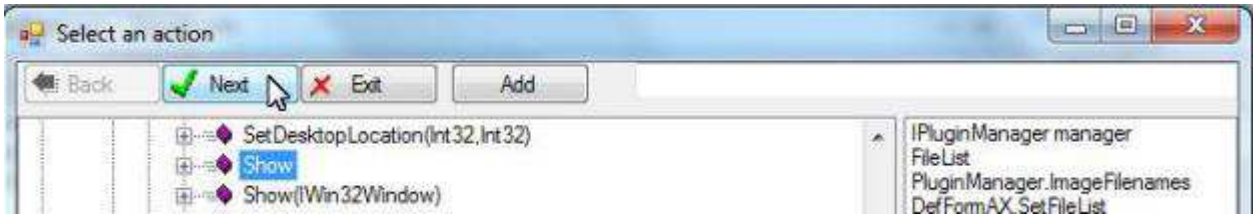


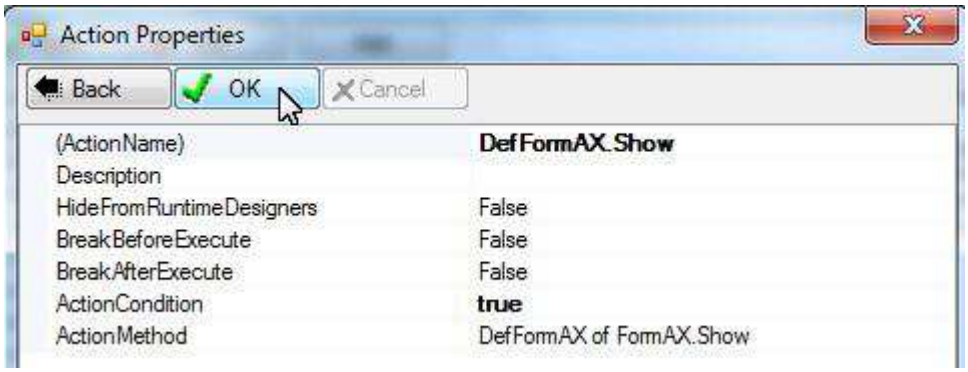
Now we may modify the method.

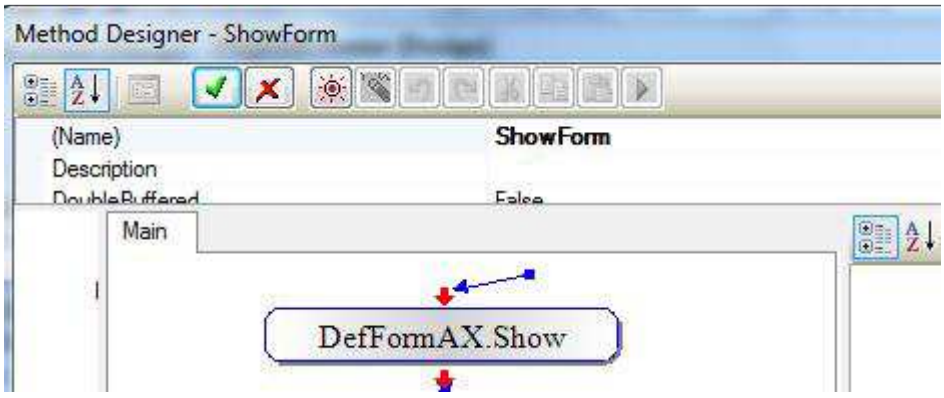




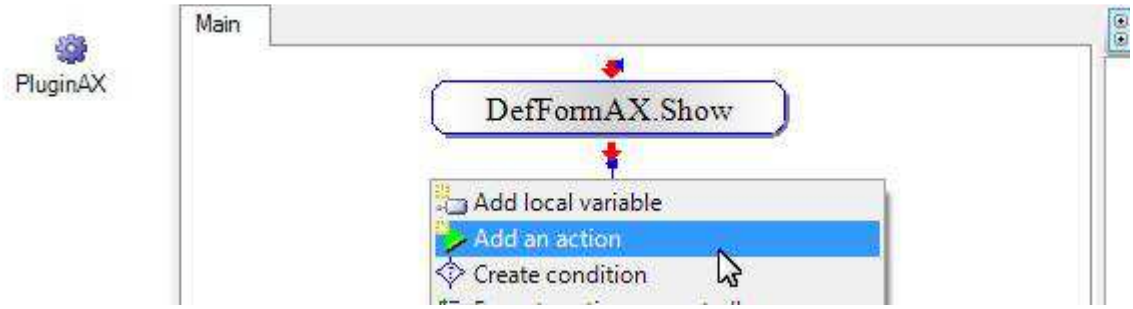


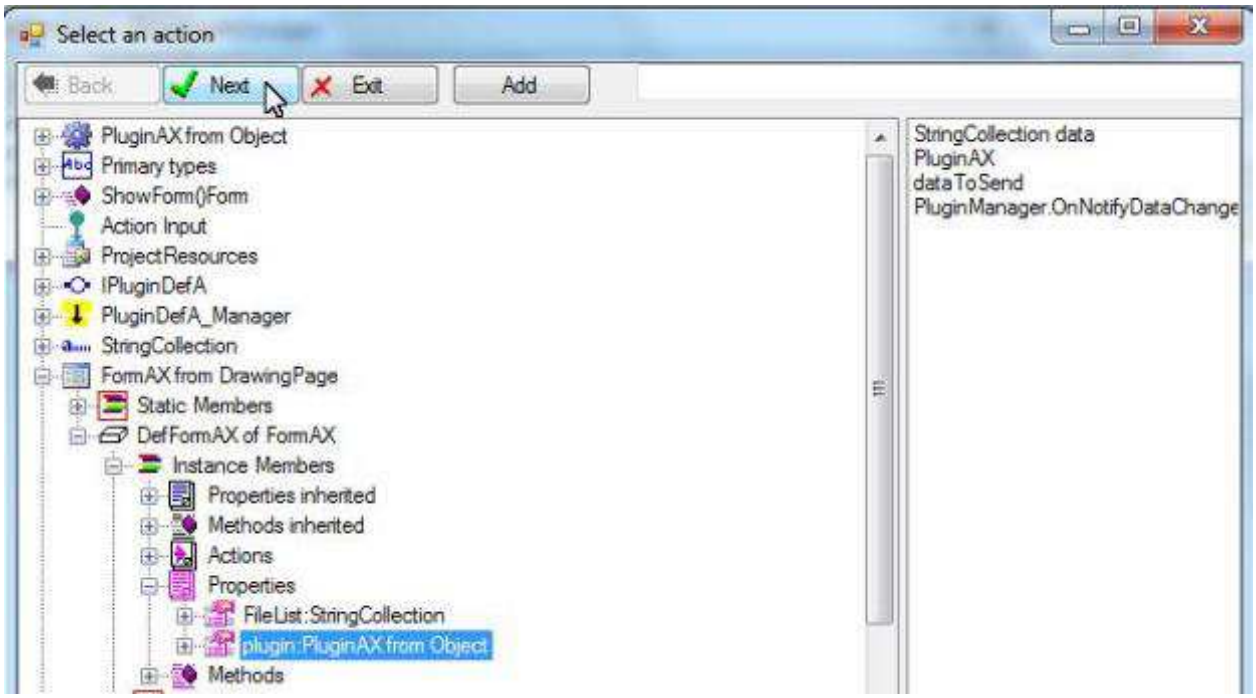


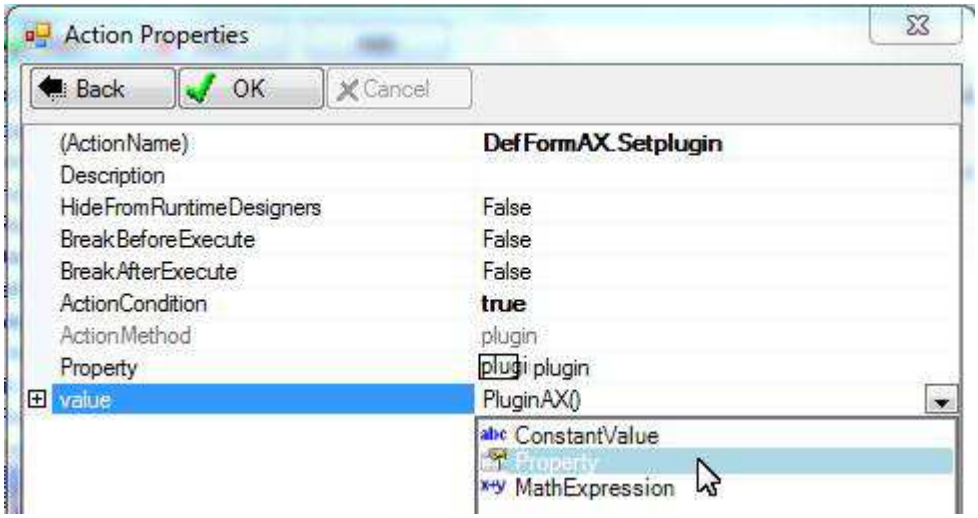


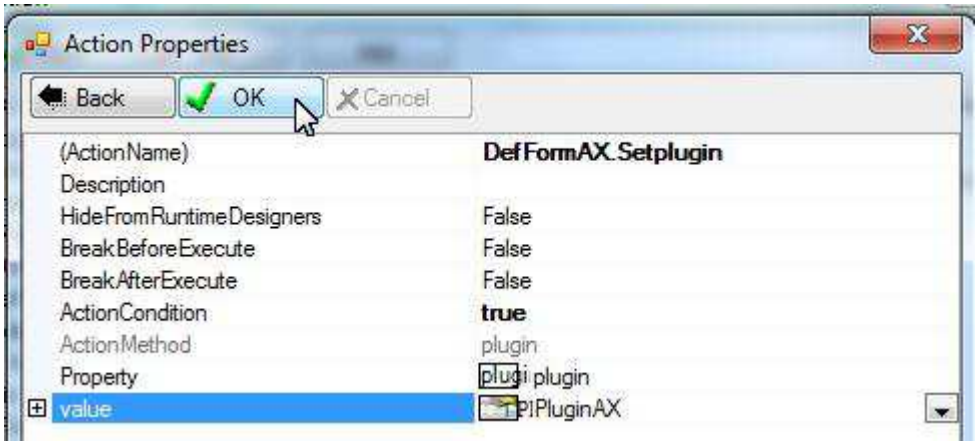


Create an action to let the form remember the plug-in object:

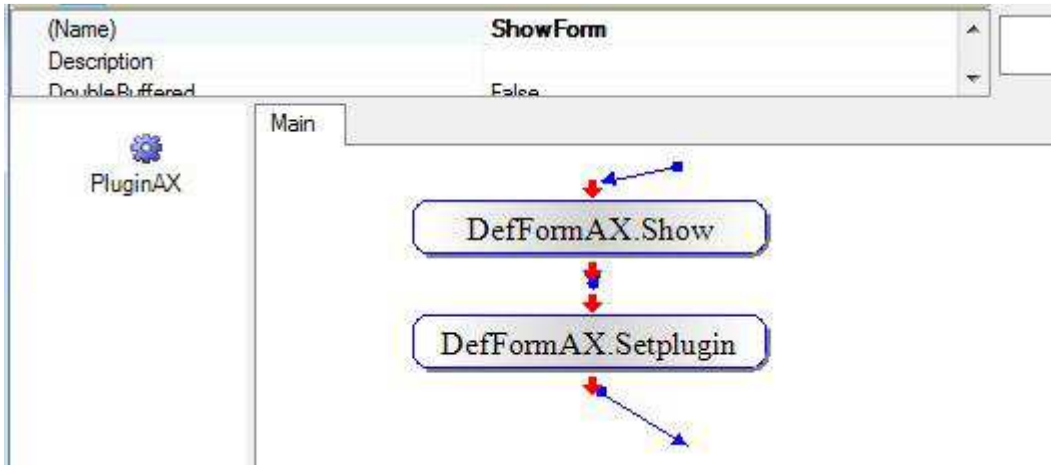

PluginAX
Main
DefFormAX.Show
Add local variable
Add an action
Create condition


Select an action
Back
Next
Exit
Add
PluginAX from Object
Primary types
ShowForm()Form
Action Input
Project Resources
IPluginDefA
PluginDefA_Manager
StringCollection
FormAX from DrawingPage
Static Members
DefFormAX of FormAX
Instance Members
Properties inherited
Methods inherited
Actions
Properties
FileList:StringCollection
plugin:PluginAX from Object
Methods
StringCollection data
PluginAX
dataToSend
PluginManager.OnNotifyDataChange

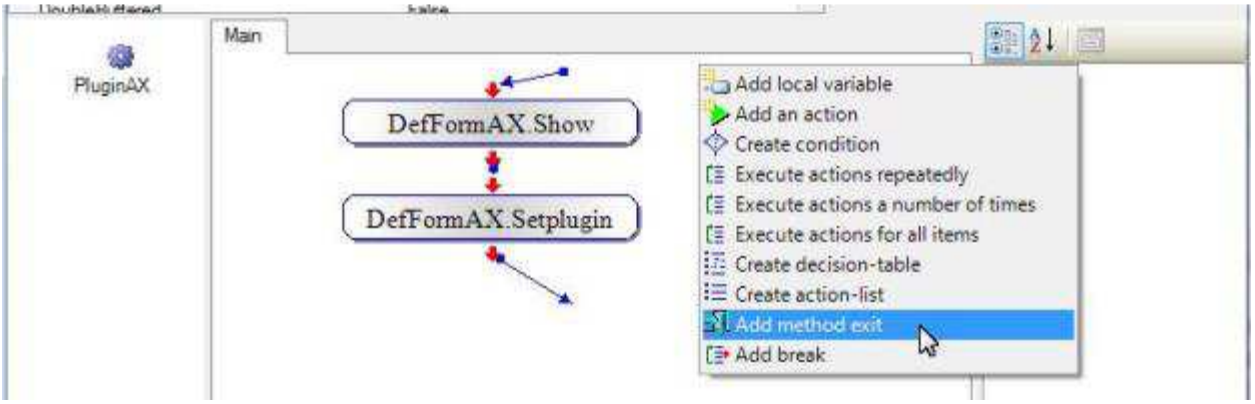

Action Properties
Back
OK
Cancel
(ActionName)
DefFormAX.Setplugin
Description
HideFromRuntimeDesigners
False
BreakBeforeExecute
False
BreakAfterExecute
False
ActionCondition
true
ActionMethod
plugin
Property
plugi plugin
value
PluginAX()
ConstantValue
Property
MathExpression



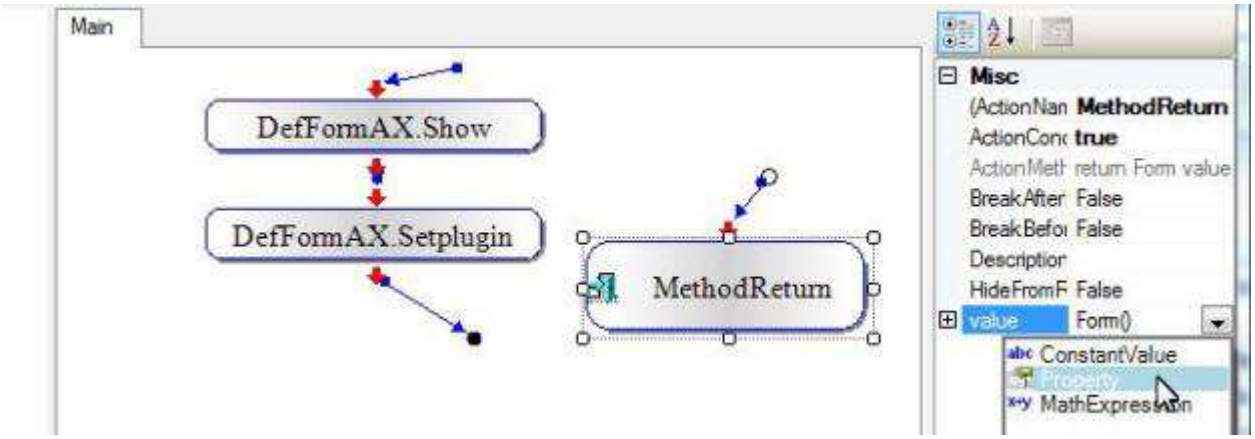
Link the two actions together:

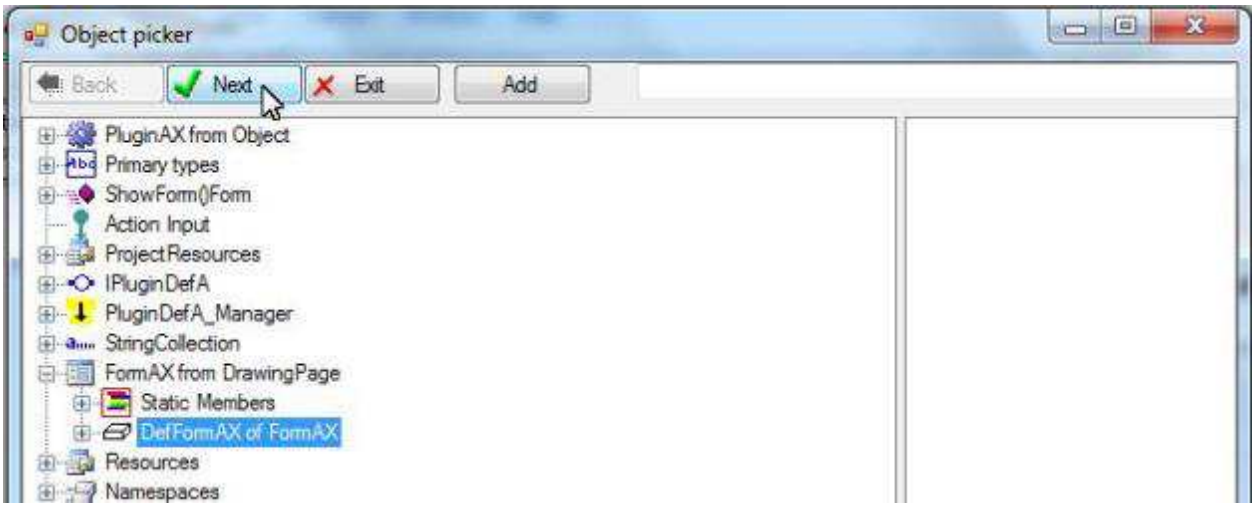


We need to return the form by this method:

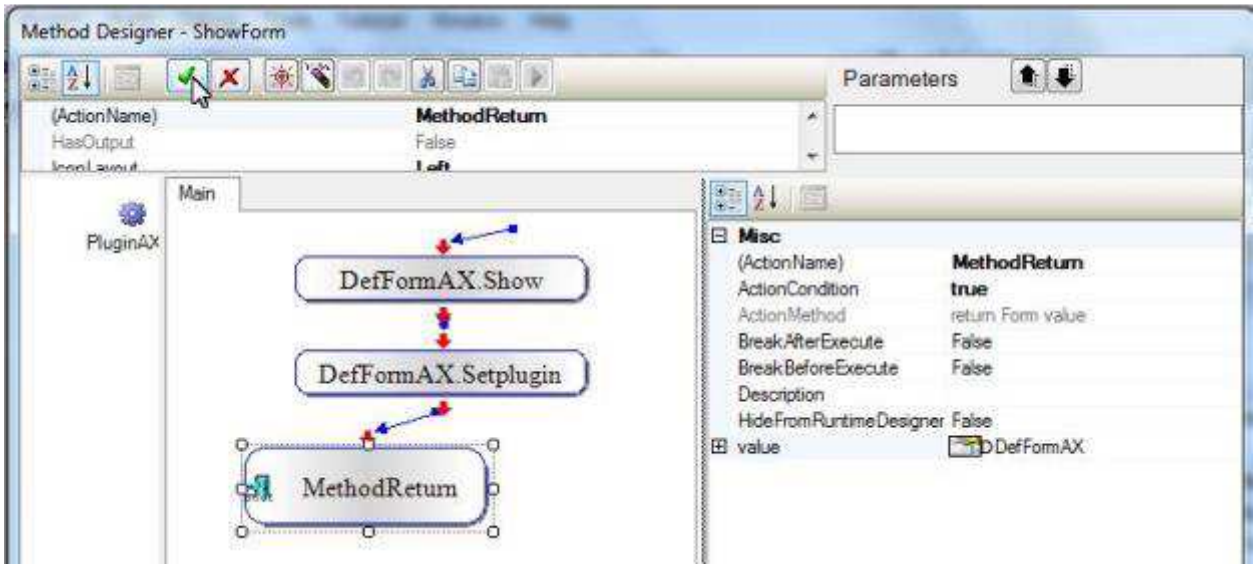


Select the form:





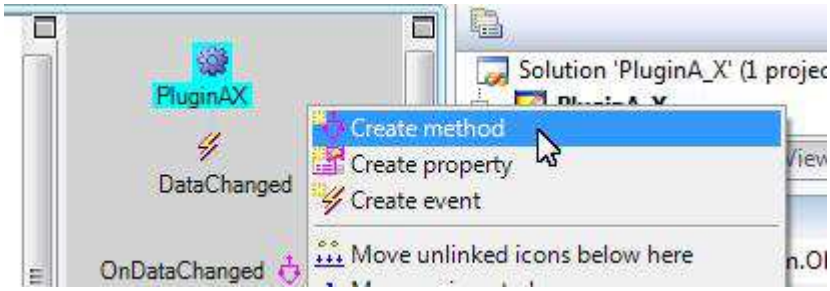
Link the action to the last action.



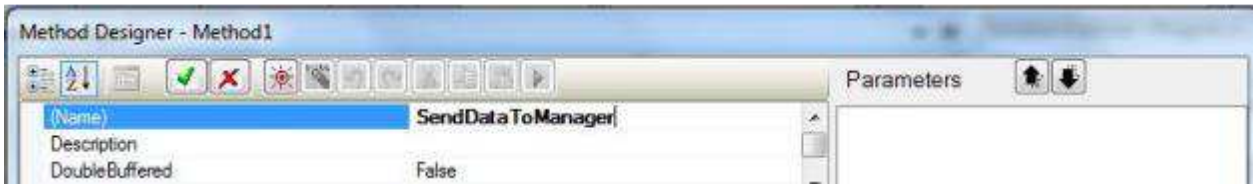
Notify Data Change

A plug-in may notify the plug-in manager when it changes the data.

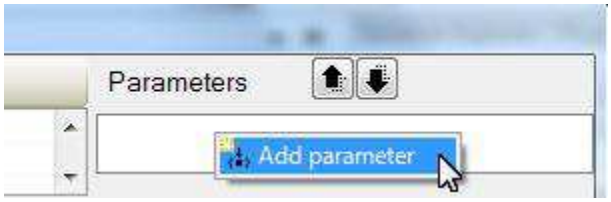
We need to create a method to execute an action to notify plug-in manage about data change.

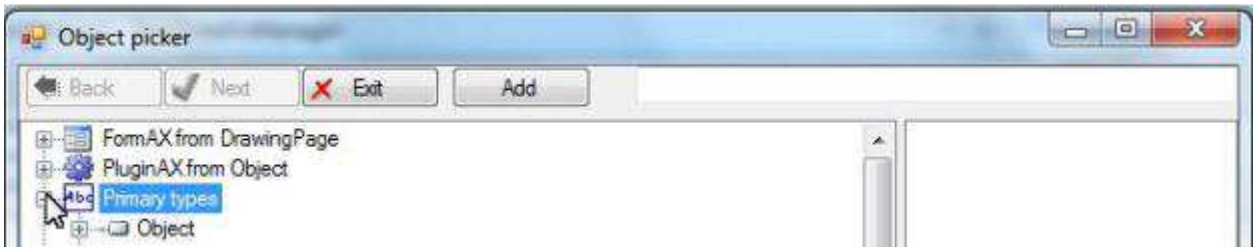


Rename the method to SendDataToManager:



Create a data parameter:

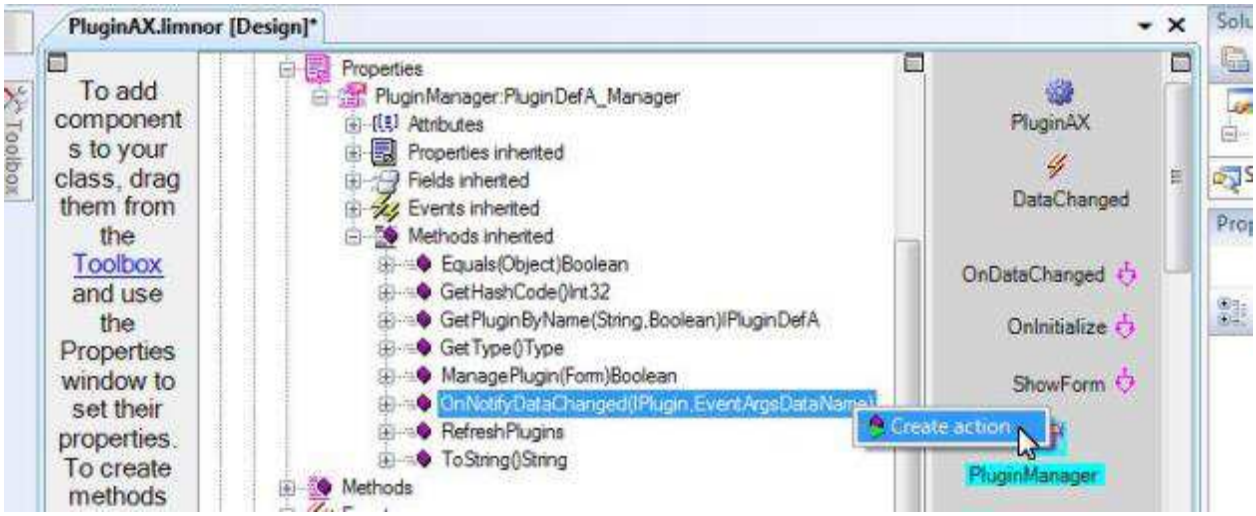




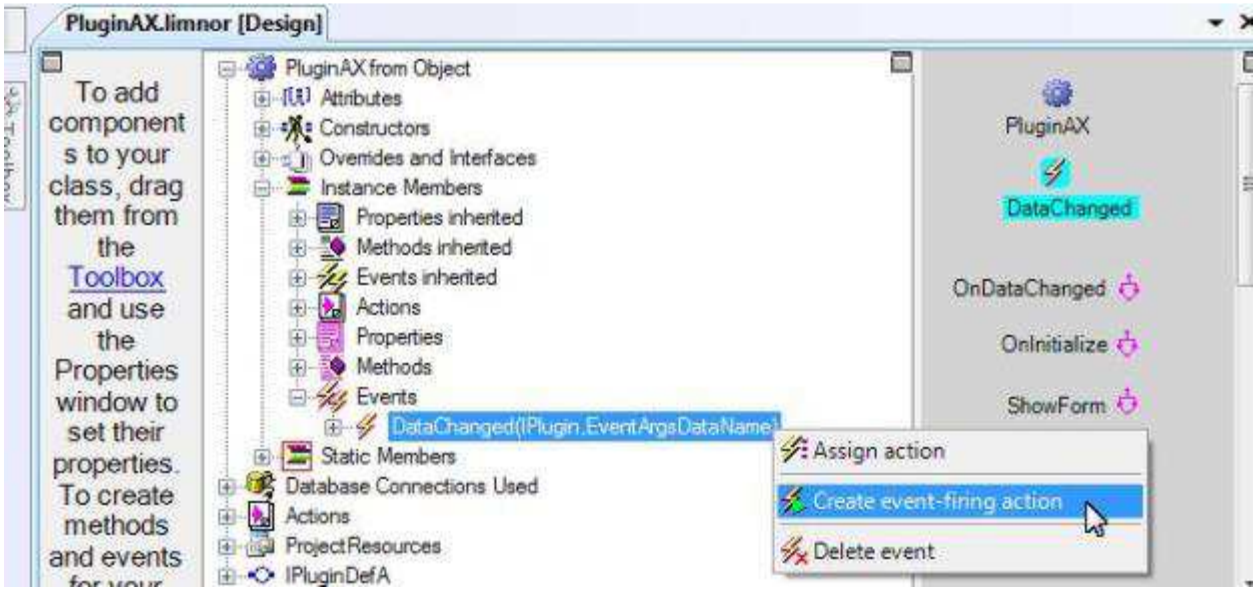
Rename the parameter to data:

There are two ways of notifying changes.

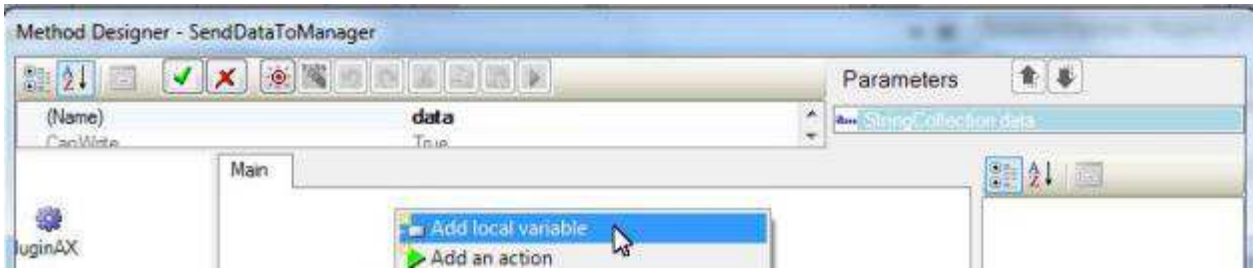
1. By calling OnNotifyDataChanged on the plug-in manager:

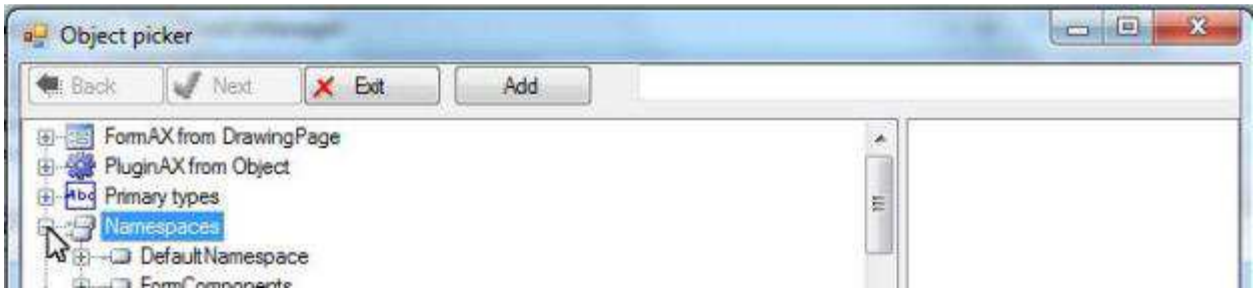


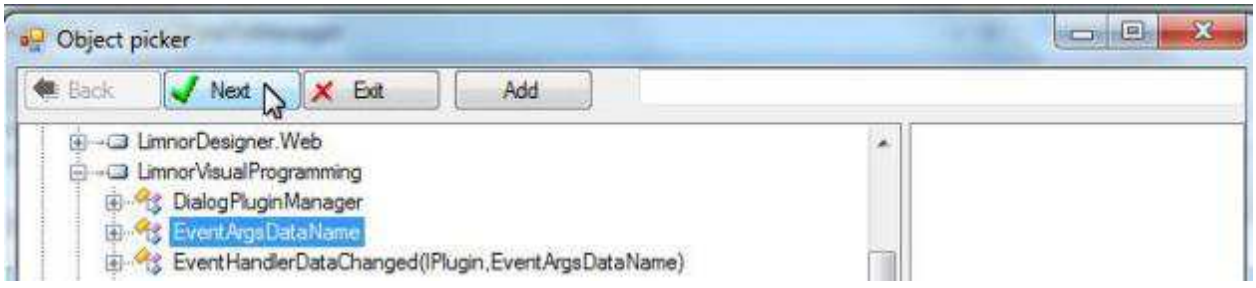
2. By firing DataChanged event:

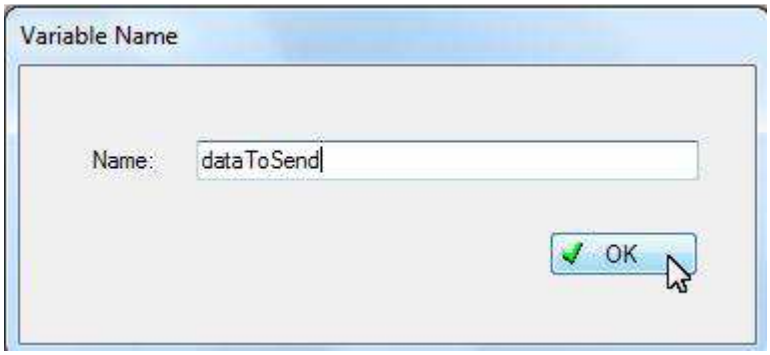


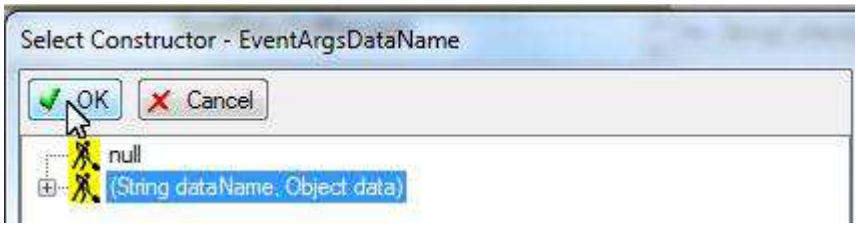
No matter which way we choose, we need to create a variable of type EventArgsDataName to hold the data for passing it to the plug-in manager:



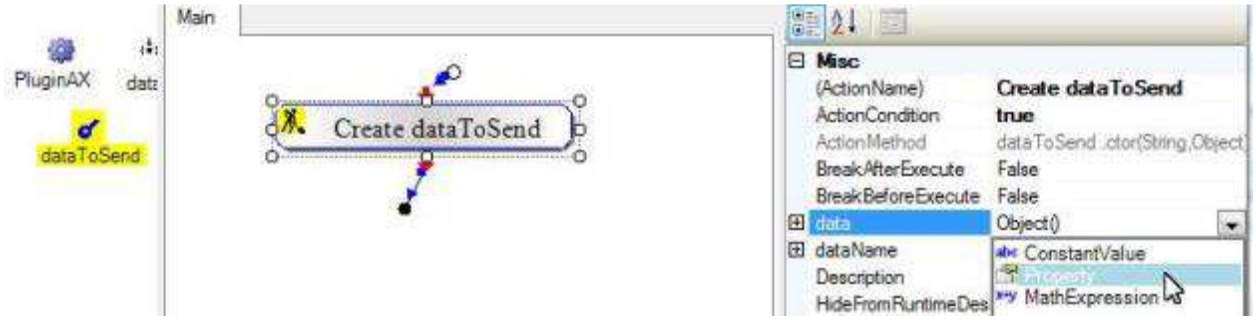




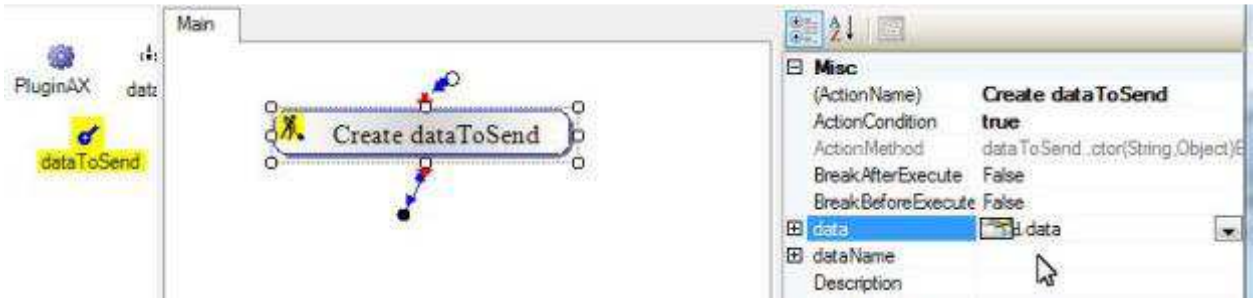




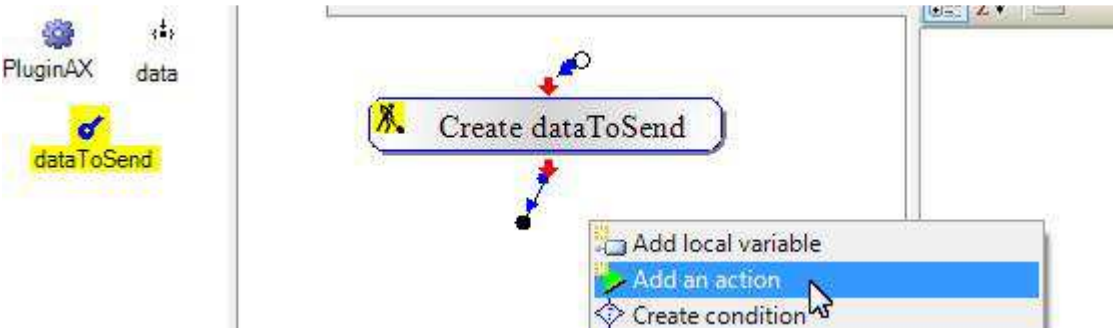
An action is created to create the variable. Set the data parameter for the action to the method parameter:

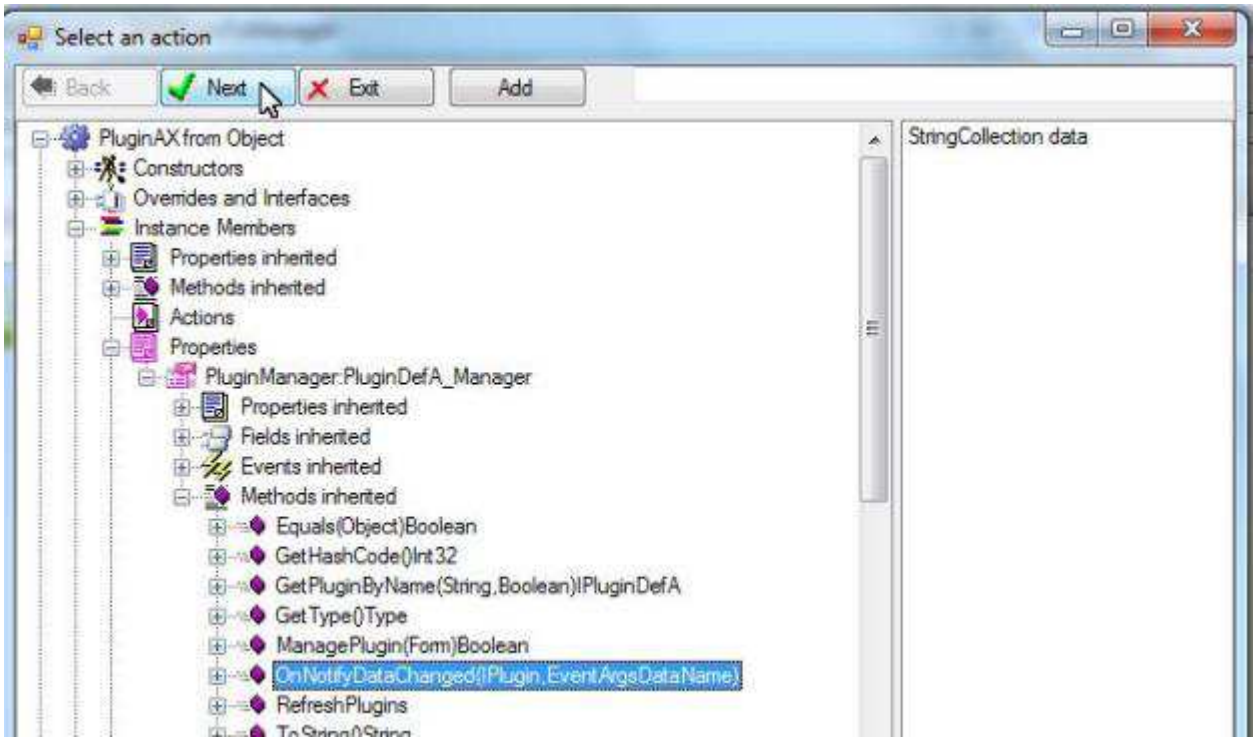


For this sample, we do not use data name. We may leave it blank.

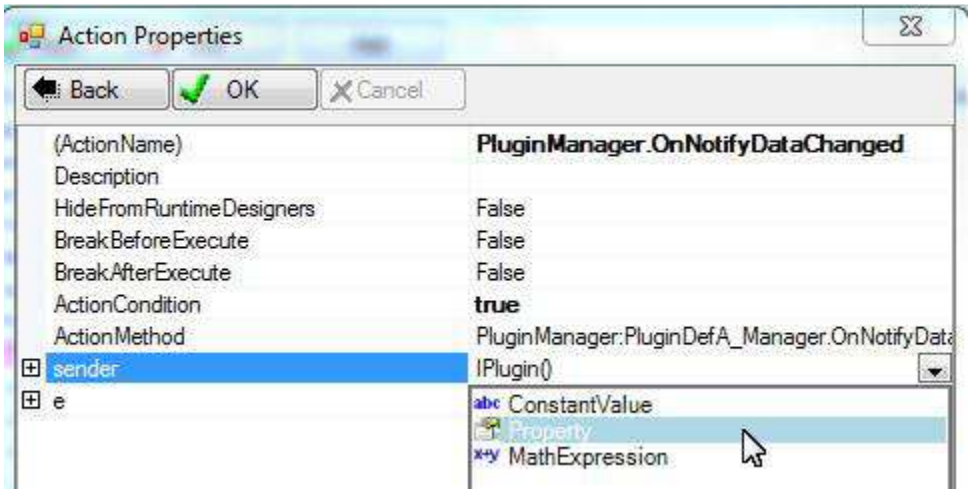


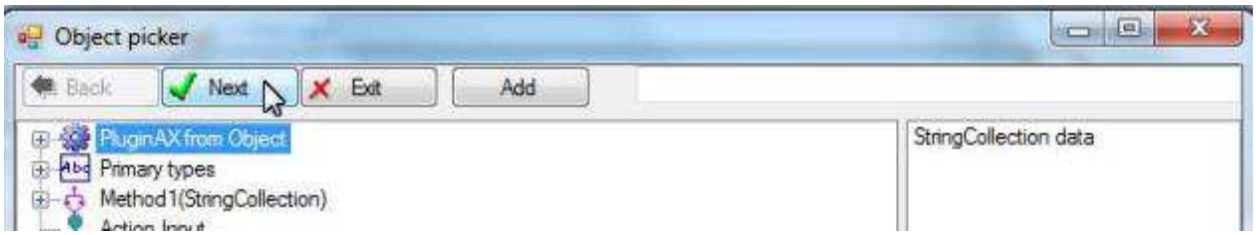
We use the first way of sending the data to the plug-in manager:



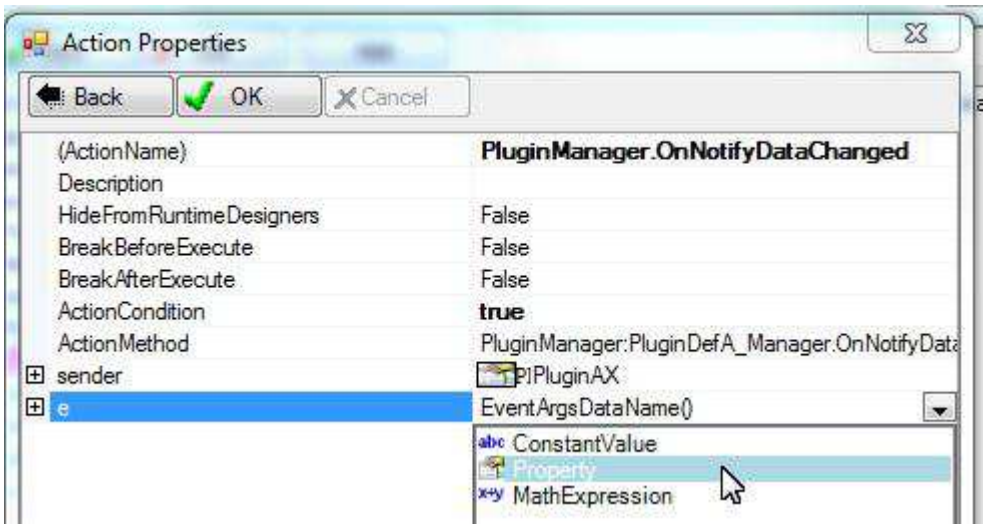


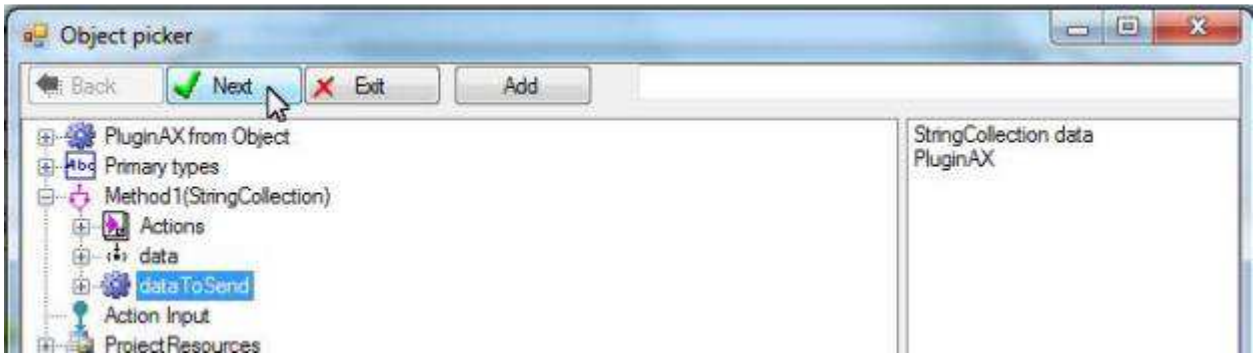
For “sender”, use the current plug-in class:

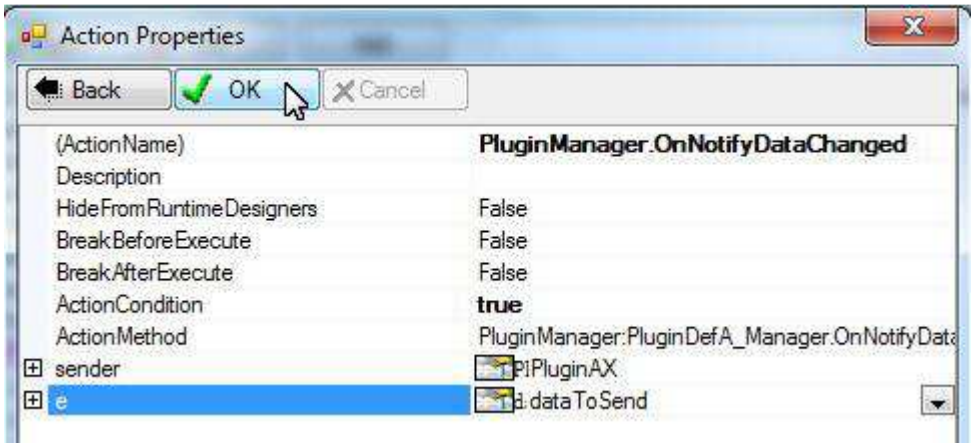




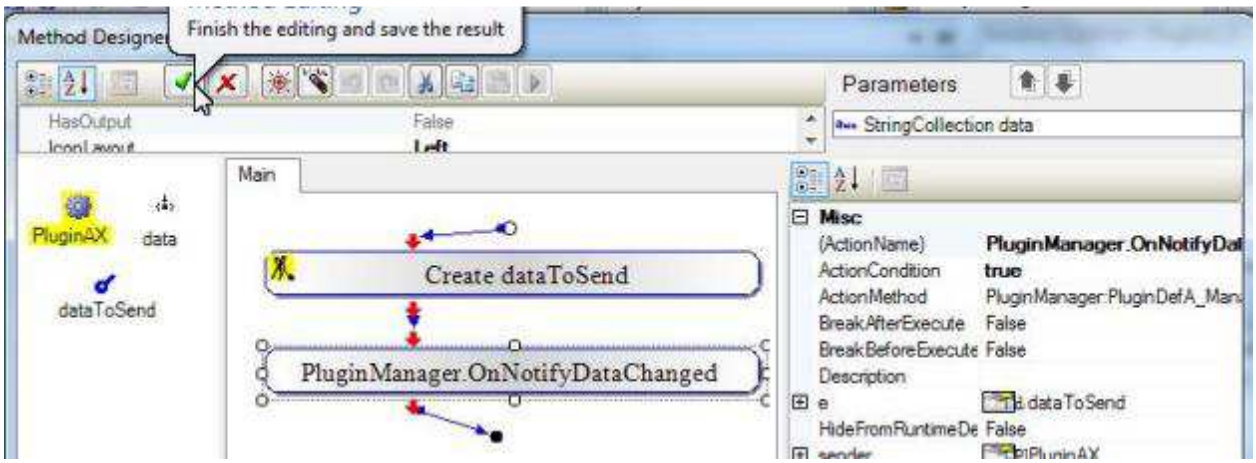
For “e”, use the variable “dataToSend”:







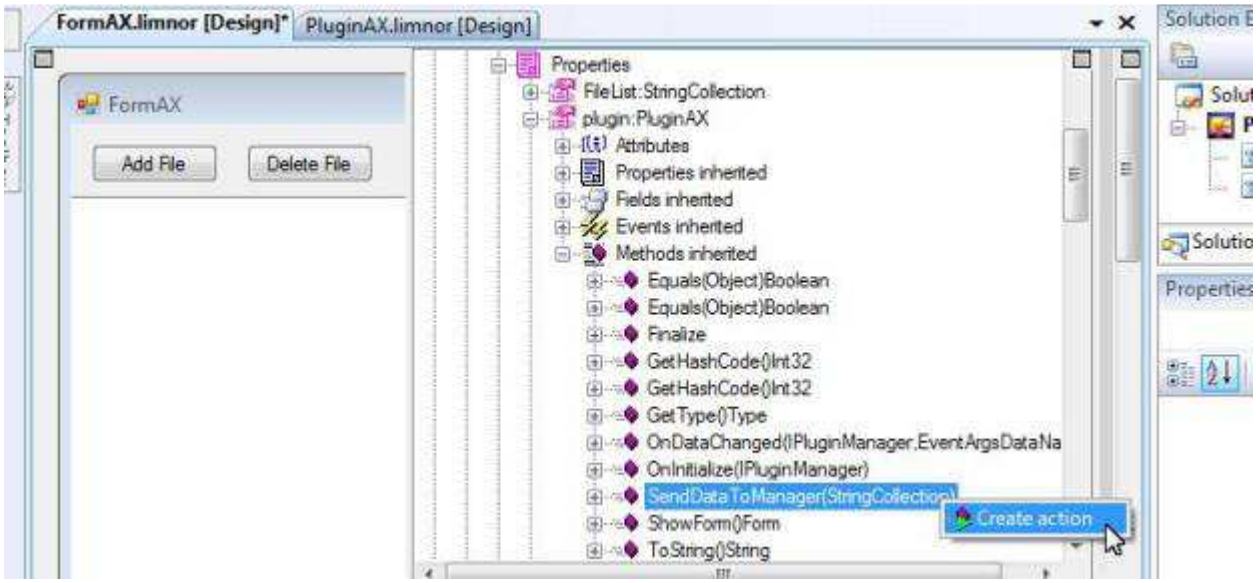
Link the two actions and finish the method editing:



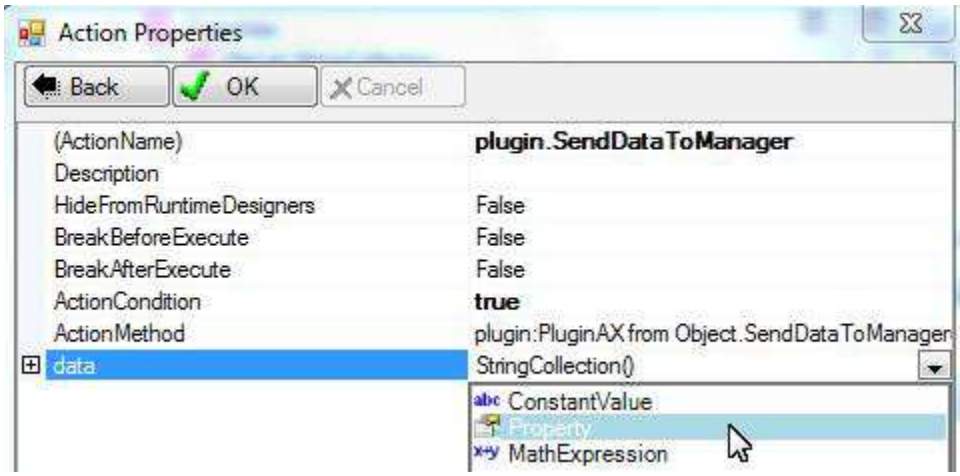
Send data to plug-in manager

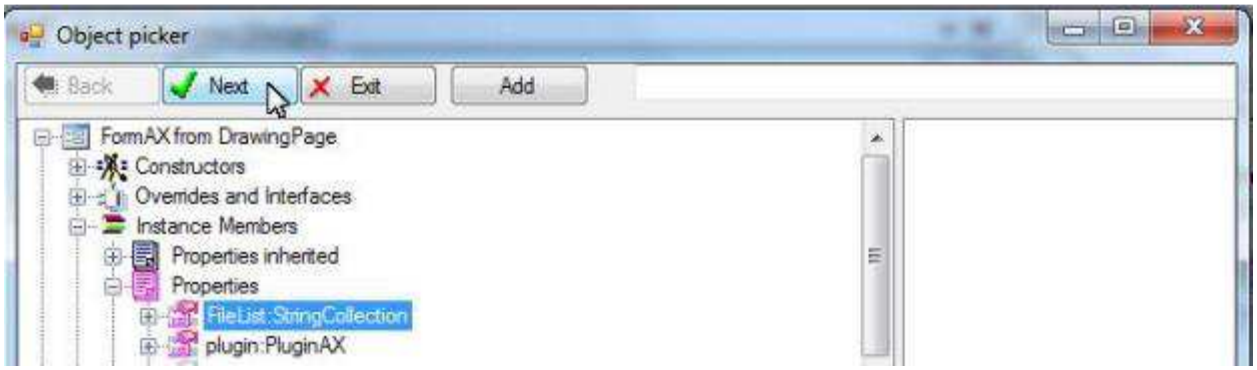
The user clicks “Add File” and “Delete File” buttons to modify the list. We may send data to the plug-in manager after changing the list.

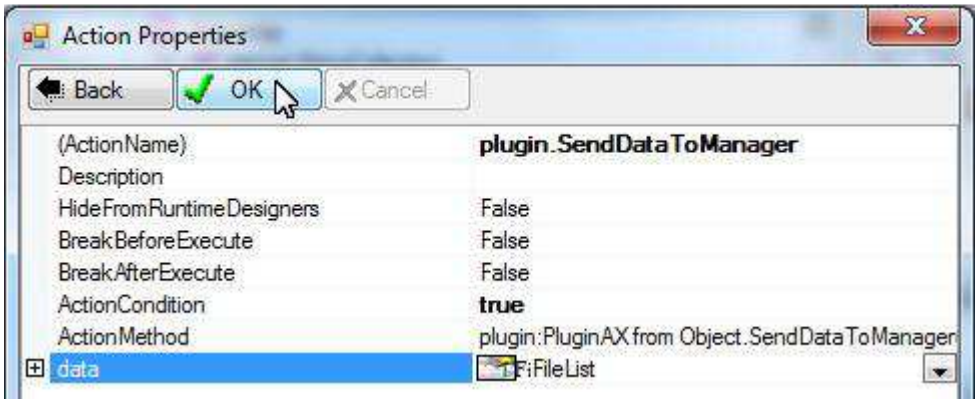
Create an action to send data to the plug-in manager:



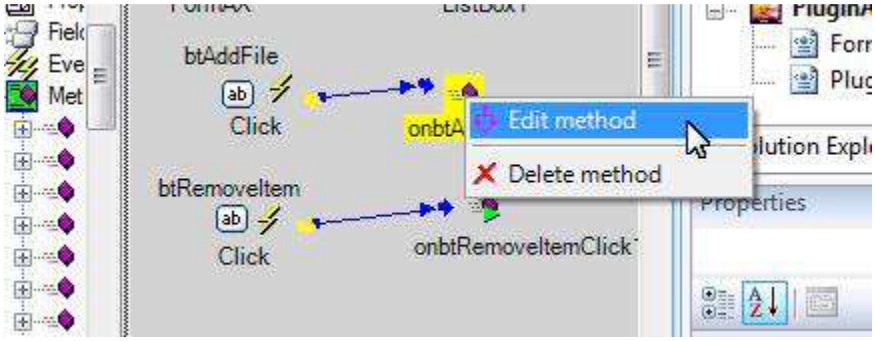
"FileList" is the data to be sent:

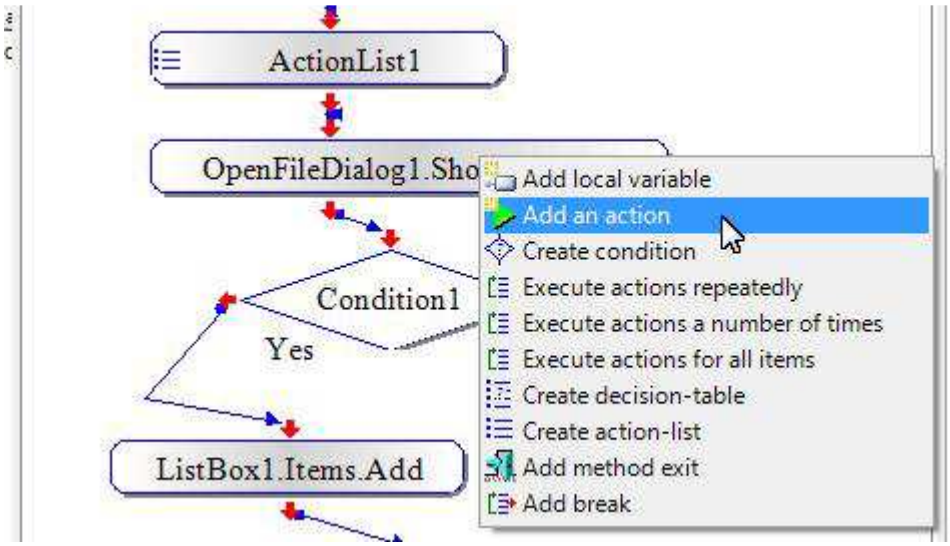


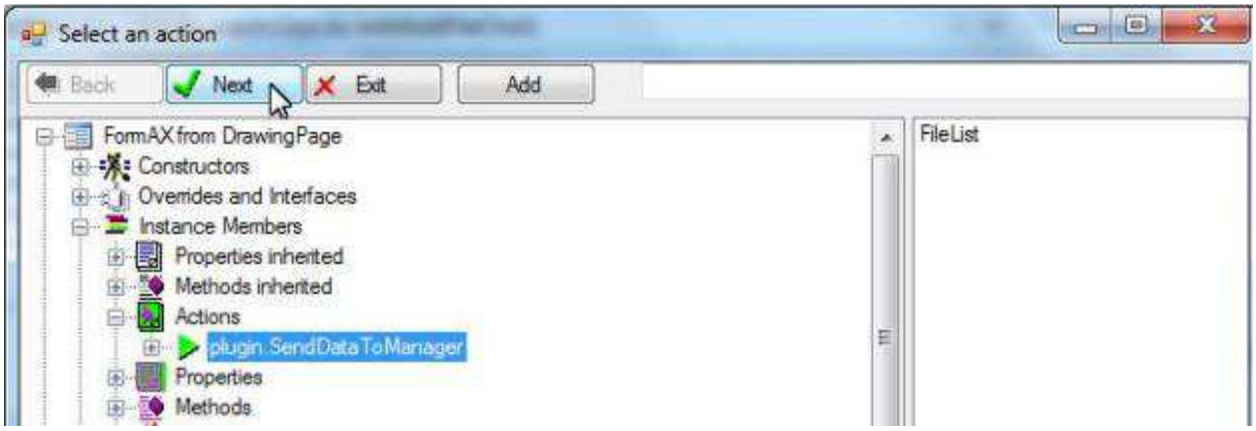


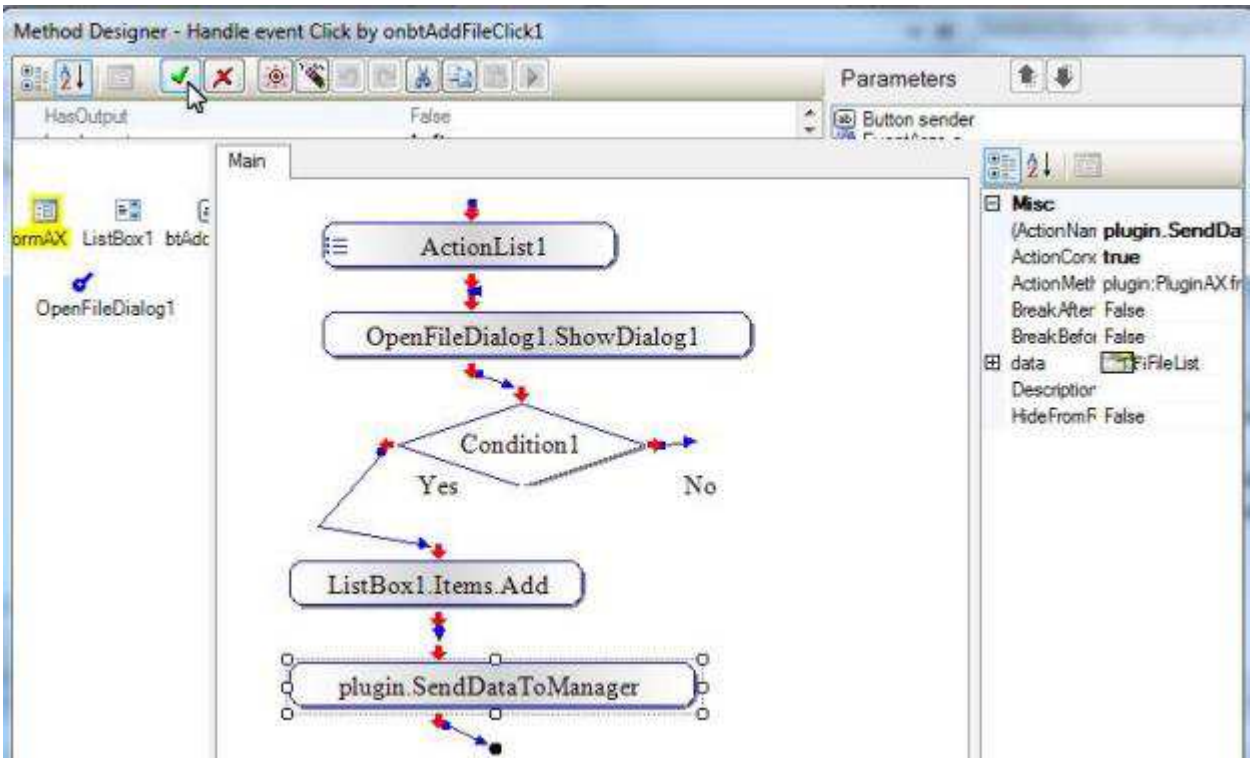


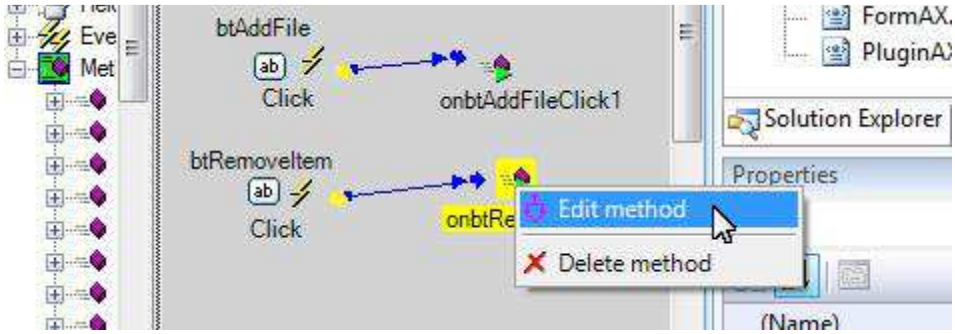
We may assign this action to both “Add File” button and “Delete File” button:

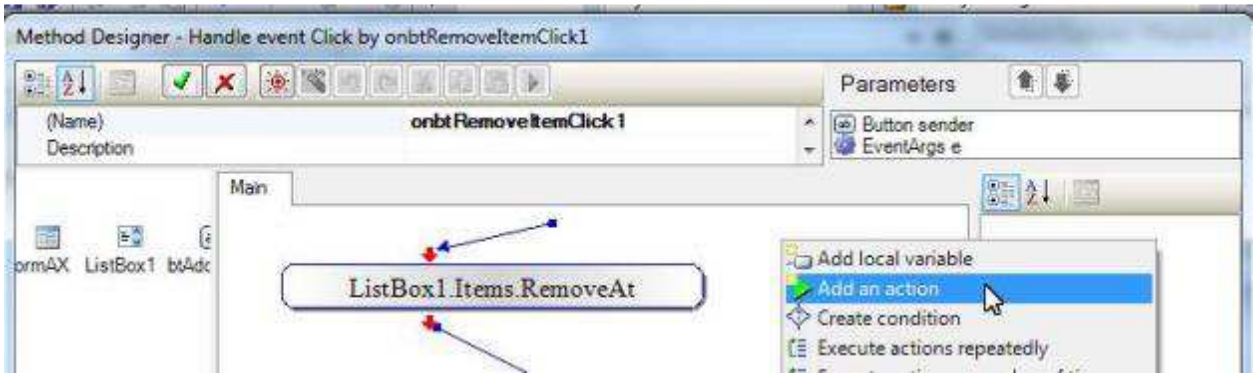


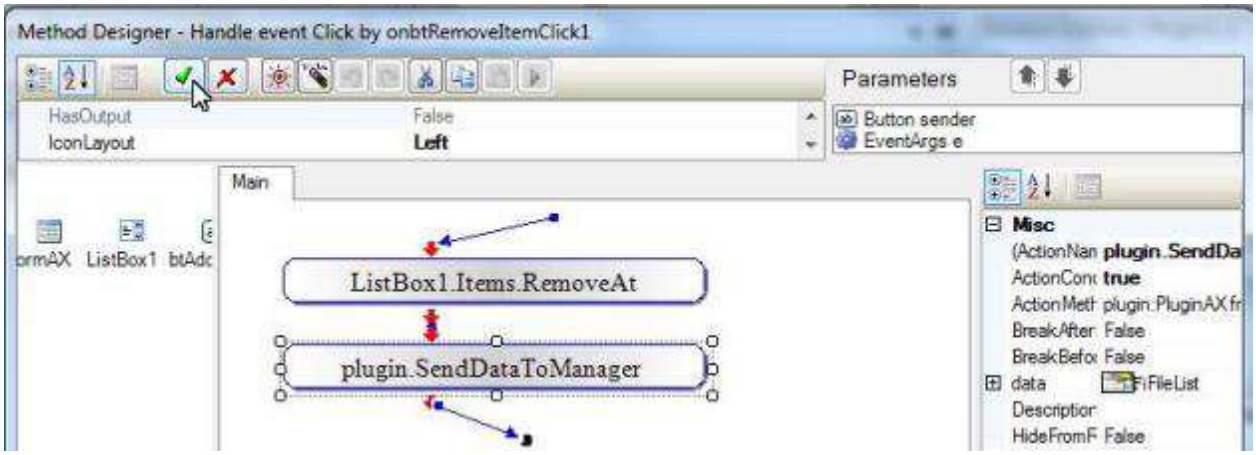



Select an action
Back
Next
Exit
Add
FormAX from DrawingPage
Constructors
Overrides and Interfaces
Instance Members
Properties inherited
Methods inherited
Actions
plugin.SendDataToManager
Properties
Methods
FileList


Method Designer - Handle event Click by onbtAddFileClick1
Parameters
HasOutput
False
Button sender
Main
ActionList1
OpenFileDialog1.ShowDialog1
Condition1
Yes
No
ListBox1.Items.Add
plugin.SendDataToManager
OpenFileDialog1
Misc
data
Description
HideFromF
False


btAddFile
Click
onbtAddFileClick1
btRemoveItem
Click
Edit method
Delete method
Solution Explorer
Properties
(Name)



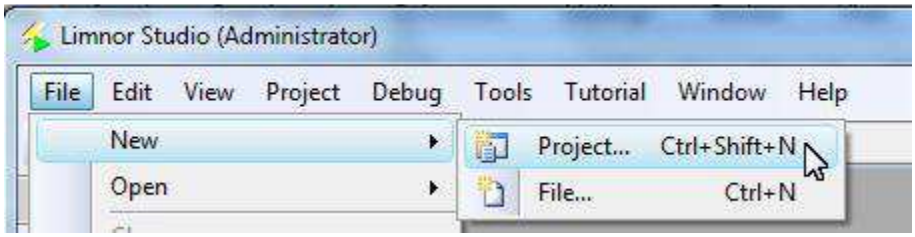


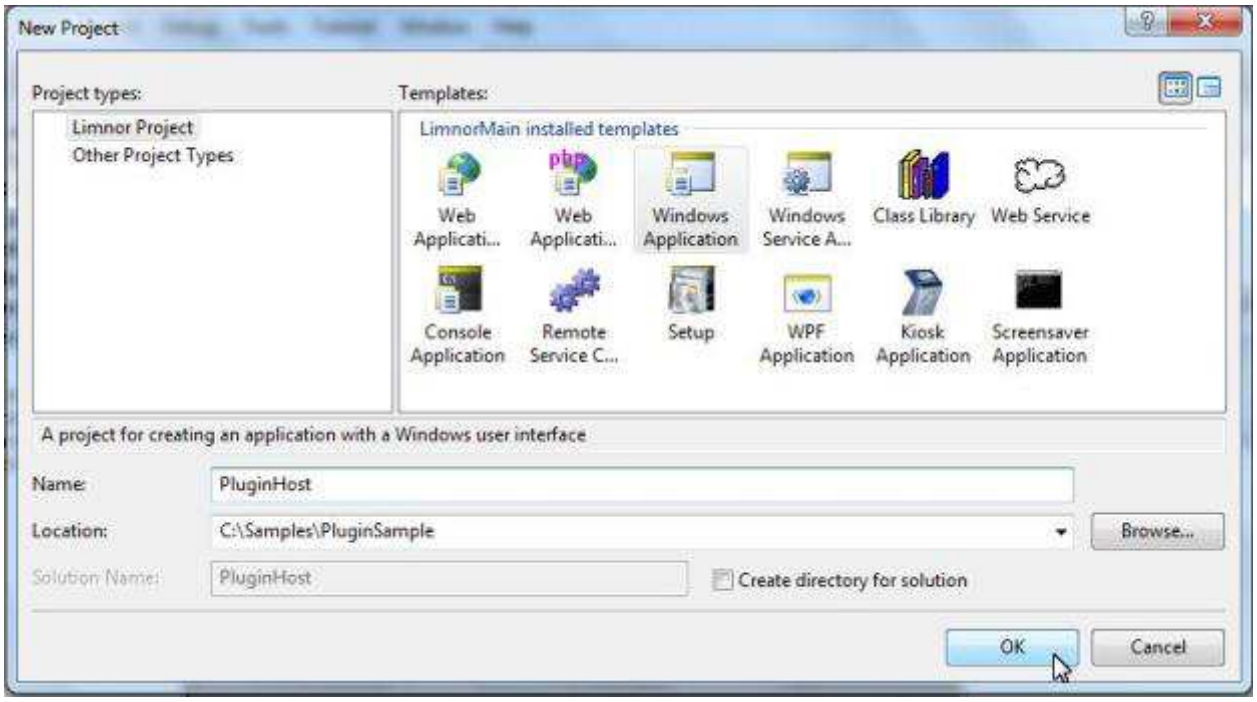
Plug-in Host Application

The last item in the puzzle is the plug-in host application which puts everything together.

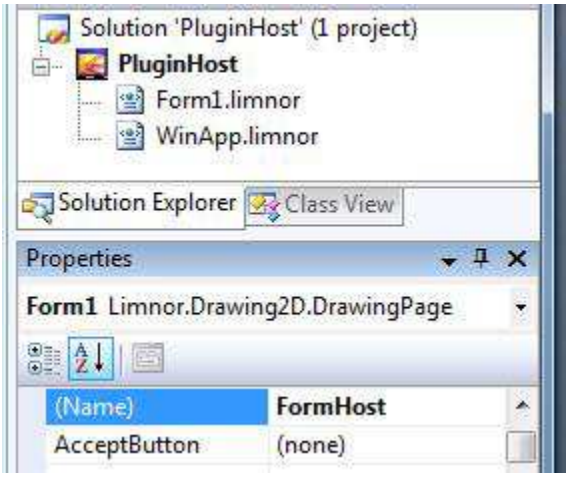
Create Windows Application

Let's create a Windows Form application as our plug-in host:

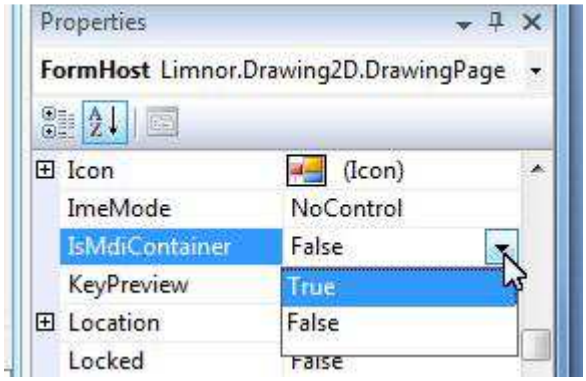




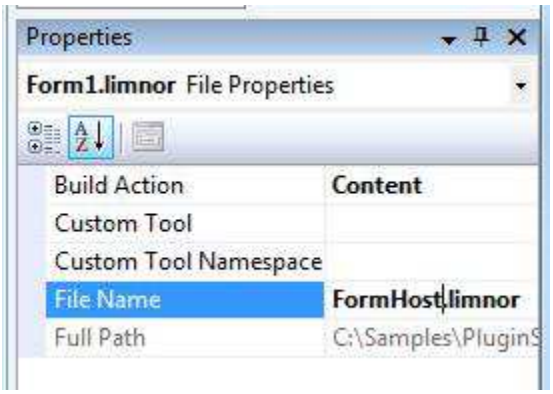
Rename Form1 to FormHost:



We want to use the form as a MDI container to contain forms loaded by plug-ins. Set IsMDIContainer to True:

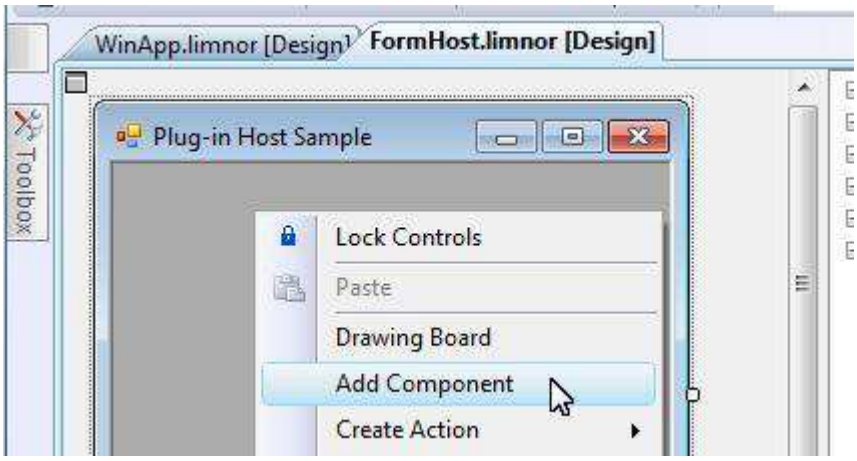


Rename the form file:

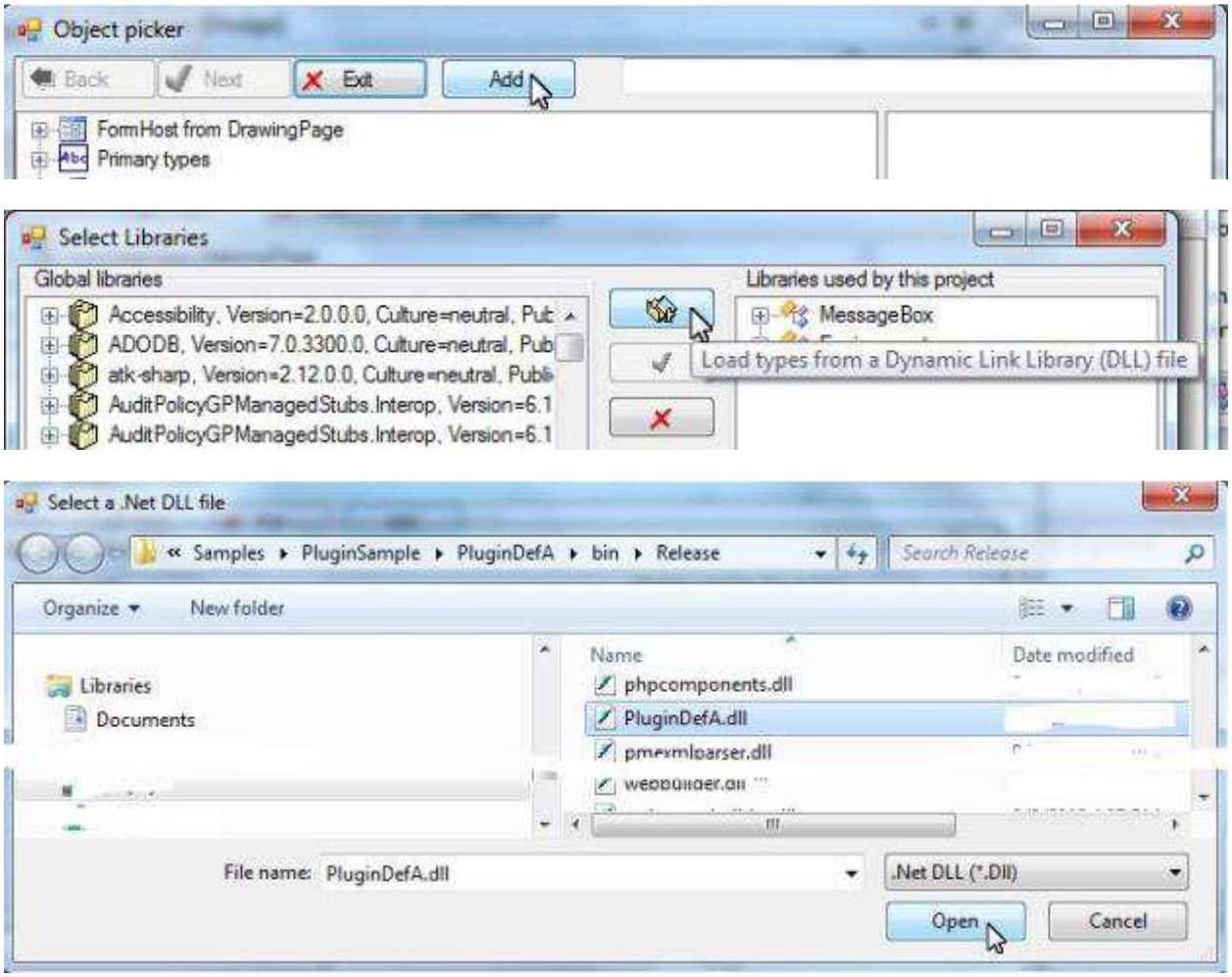


Add Plug-in Manager

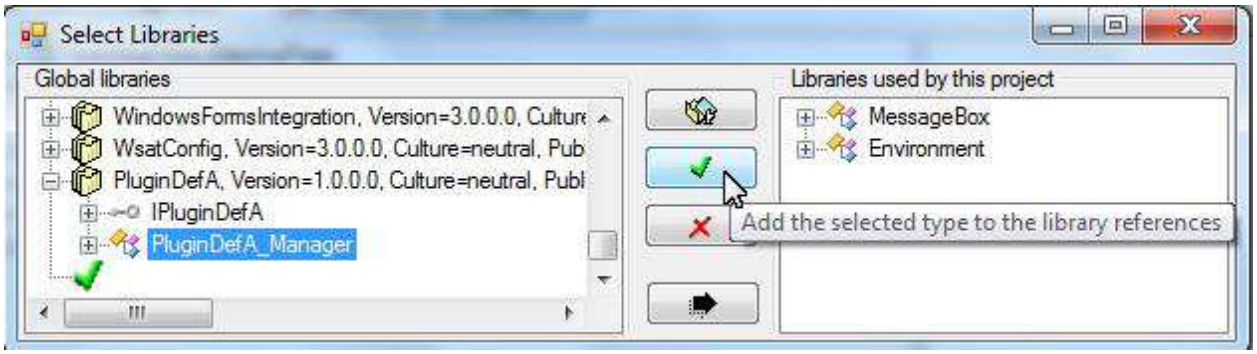
Add a plug-in manager to the form:

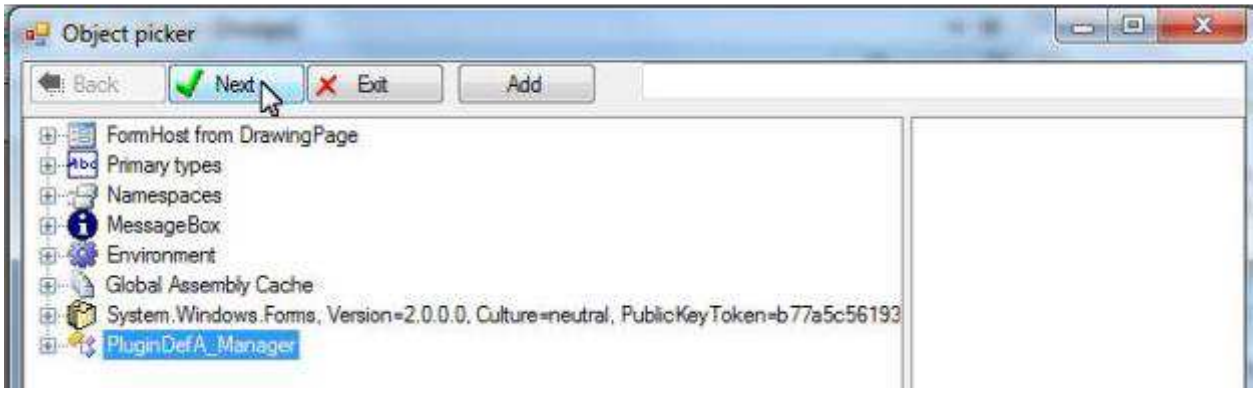


Click “Add” button to load the plug-in definition DLL:

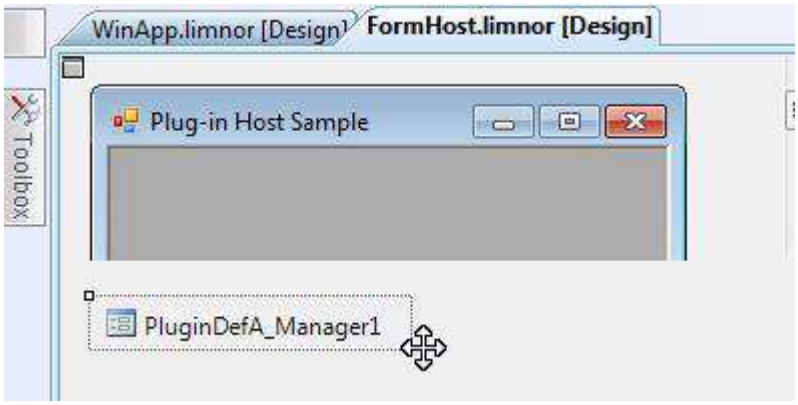


Select the plug-in manager contained in the DLL:





The plug-in manager appears in the form:

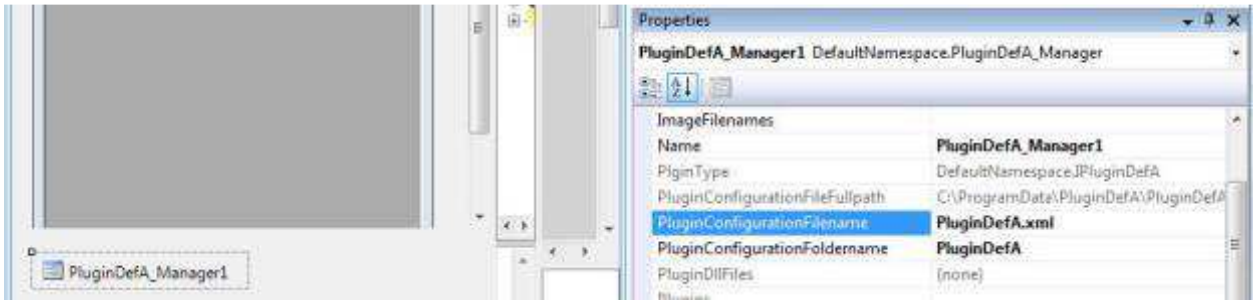


Set plug-in configuration file location

The plug-in manager uses an xml file under common application data folder to record loaded plug-ins. The xml file path is formed by following properties:

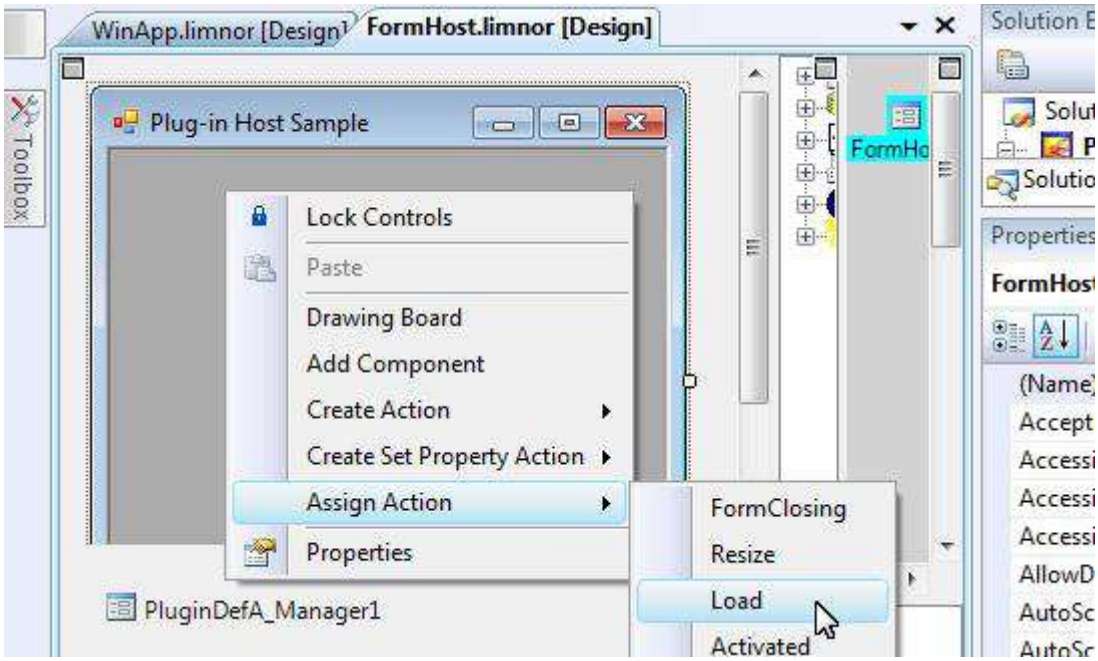
- Property “PluginConfigurationFoldername” indicates the folder name under the common application data folder.
- Property “PluginConfigurationFilename” indicates the file name.

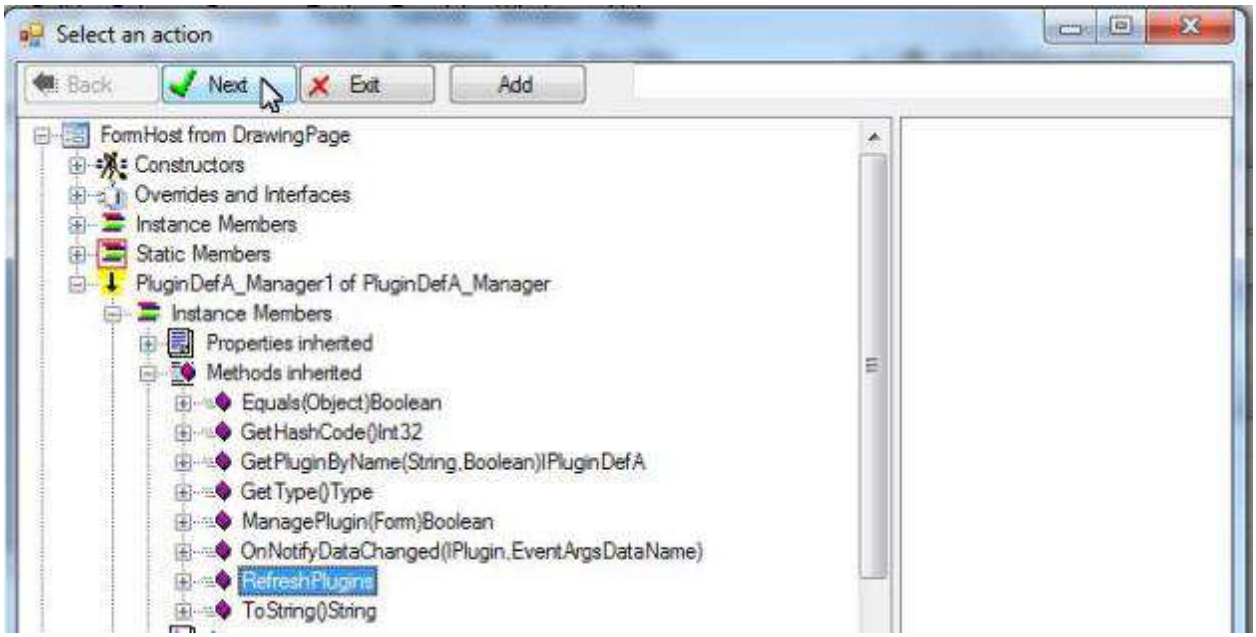
Suppose we set PluginConfigurationFoldername to PluginDefA, and PluginConfigurationFilename to PluginDefA.xml, then the full path for the configuration file could be “C:\ProgramData\PluginDefA\PluginDefA.xml”, “C:\Documents and Settings\AllUsers\Application Data\PluginDefA\PluginDefA.xml”, or another folder depending on your windows settings.

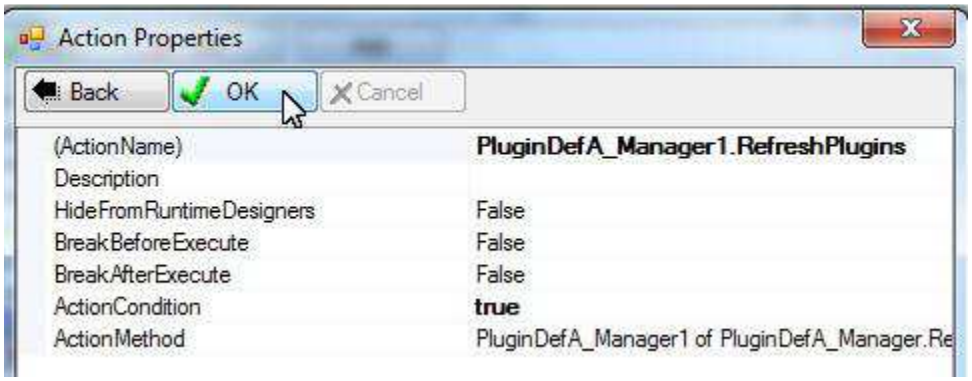


Load plug-ins on loading form

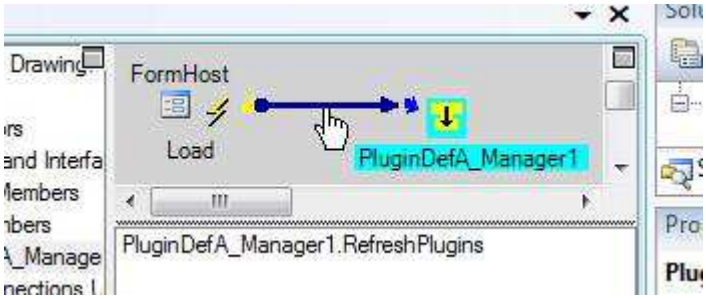
We may load all plug-ins at the time the form is loaded:





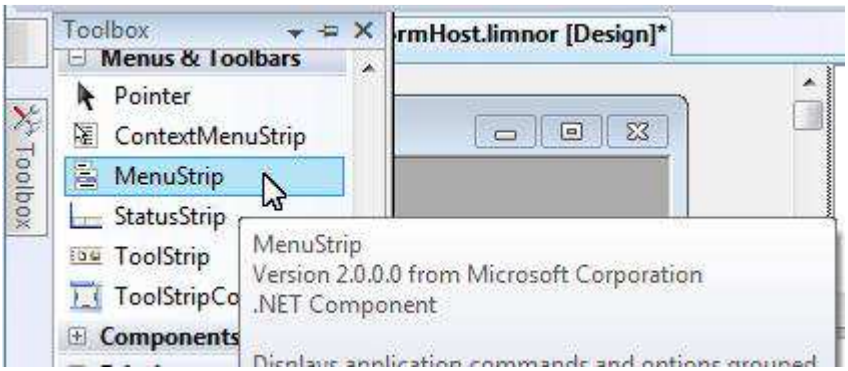


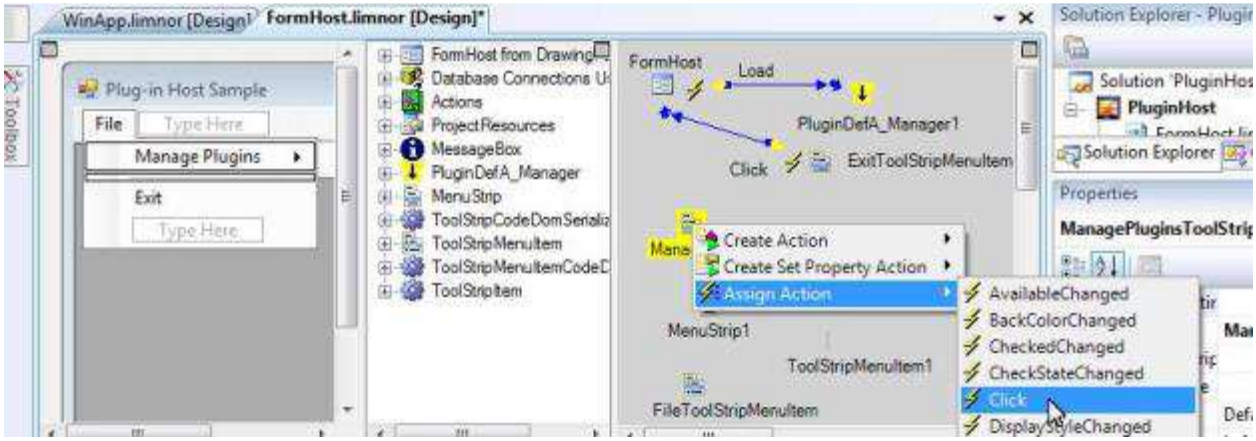
The action is created and assigned to event Load:

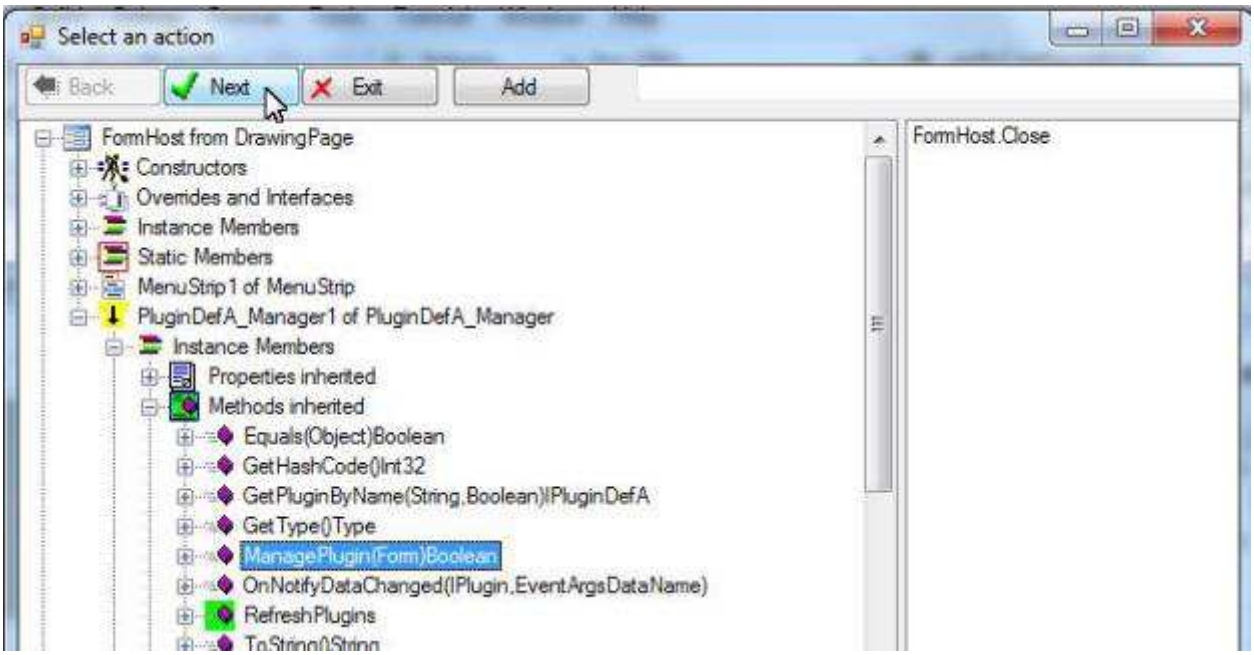


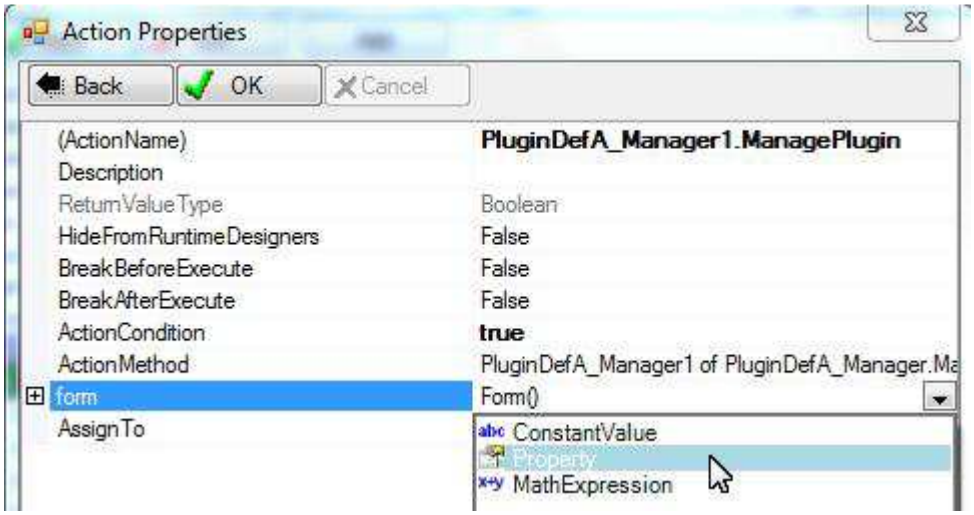
Manage Plug-ins

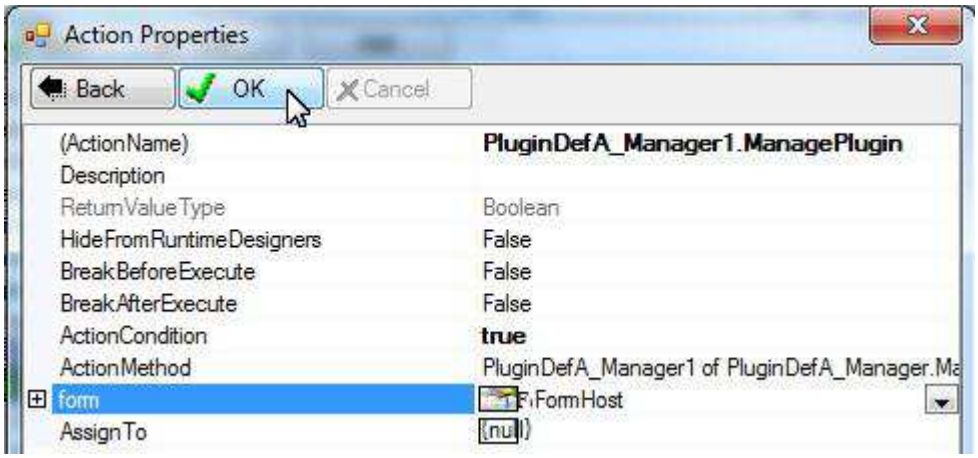
We use a menu to call ManagePlugin method of the plug-in manager:



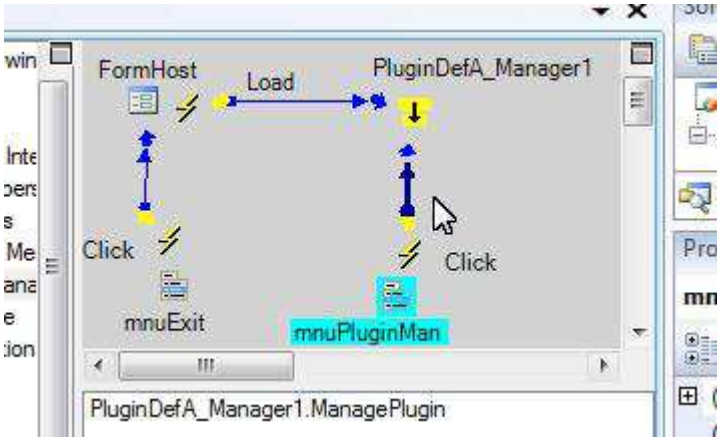

WinApp.limnor [Design]
FormHost.limnor [Design]*
Plug-in Host Sample
File
Type Here
Manage Plugins
Exit
Type Here
FormHost from Drawing
Database Connections
Actions
ProjectResources
MessageBox
PluginDefA_Manager
MenuStrip
ToolStripCodeDomSerializ
ToolStripMenuItem
ToolStripMenuItemCodeD
ToolStripItem
FormHost
Load
PluginDefA_Manager1
Click
ExitToolStripMenuItem
Create Action
Create Set Property Action
Assign Action
MenuStrip1
ToolStripMenuItem1
FileToolStripMenuItem
AvailableChanged
BackColorChanged
CheckedChanged
CheckStateChanged
Click
DisplayStyleChanged
Solution Explorer - Plugin
Solution 'PluginHos
PluginHost
Solution Explorer
Properties
ManagePluginsToolStrip


Select an action
Back
Next
Exit
Add
FormHost from DrawingPage
Constructors
Overrides and Interfaces
Instance Members
Static Members
MenuStrip1 of MenuStrip
PluginDefA_Manager1 of PluginDefA_Manager
Instance Members
Properties inherited
Methods inherited
Equals(Object)Boolean
GetHashCode()Int32
GetPluginByName(String,Boolean)IPluginDefA
GetType()Type
ManagePlugin(Form)Boolean
OnNotifyDataChanged(IPlugin,EventArgsDataName)
RefreshPlugins
ToString()String
FormHost.Close


Action Properties
Back
OK
Cancel
(ActionName)
PluginDefA_Manager1.ManagePlugin
Description
ReturnValueType
Boolean
HideFromRuntimeDesigners
False
BreakBeforeExecute
False
BreakAfterExecute
False
ActionCondition
true
ActionMethod
PluginDefA_Manager1 of PluginDefA_Manager.Ma
form
Form()
AssignTo
ConstantValue
Property
MathExpression



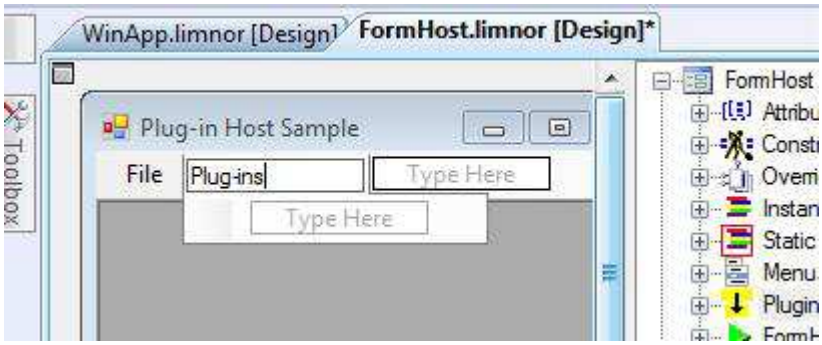
The action is created and assigned to the menu item:



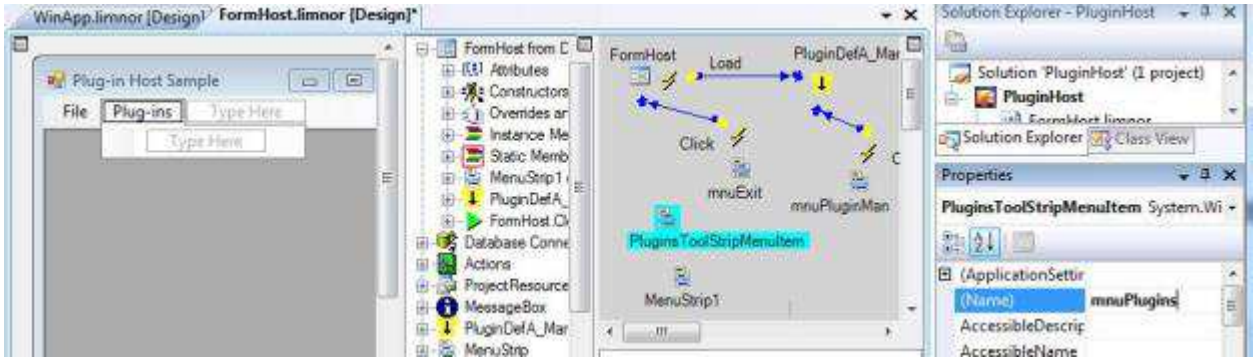
Use Plug-ins

All the efforts we have made previously are preparations for making it possible to do this last action. In this sample host application, we will create a menu named “Plug-ins”. Under this menu, we create one menu item for every available plug-in, and when the menu item is selected, we execute the ShowForm method of the corresponding plug-in.

Create "Plug-ins" main menu



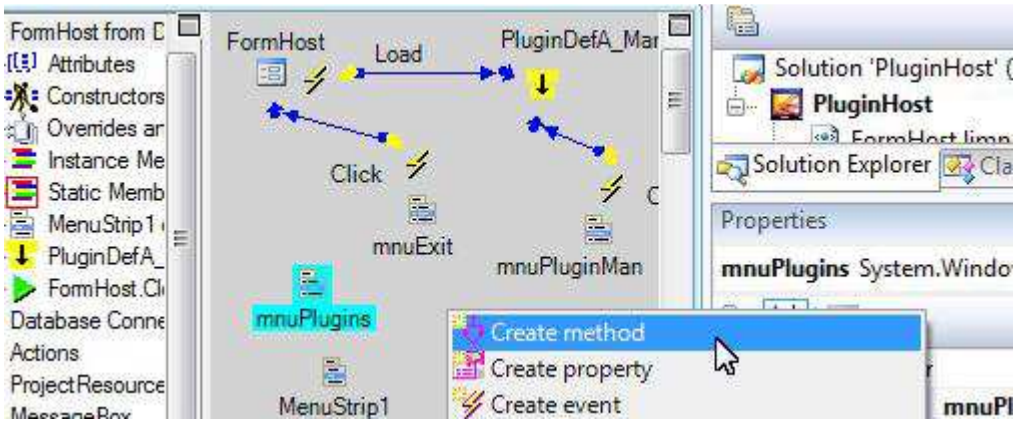
Rename the menu if we want to:



Create plug-in sub menus

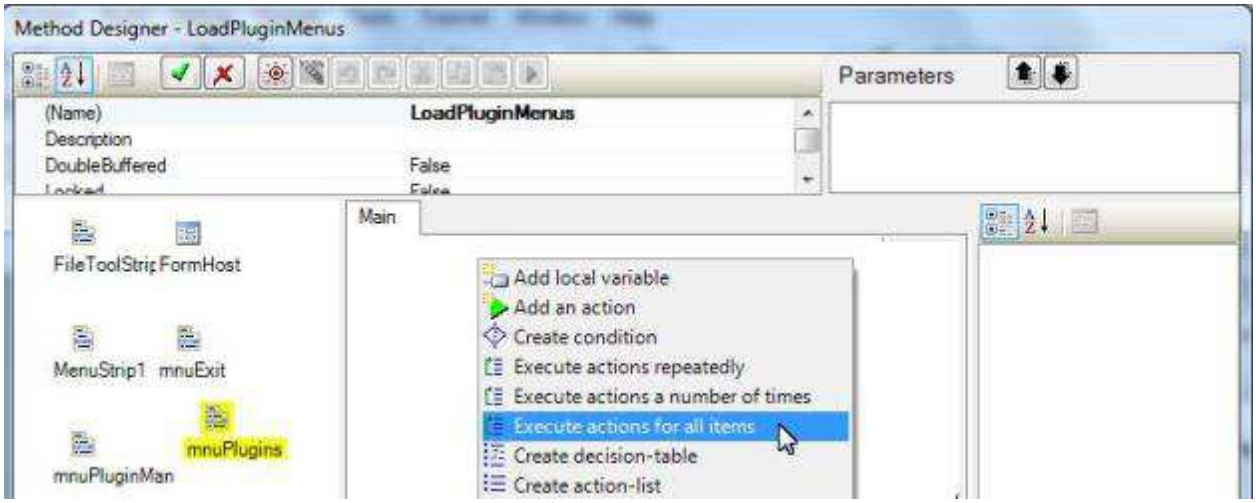
Process all plug-ins

We create one sub-menu under the "Plug-ins" main menu for each available plug-in. We create a method to do it. The method will be executed at the time of form loaded and when the user managed the plug-ins.

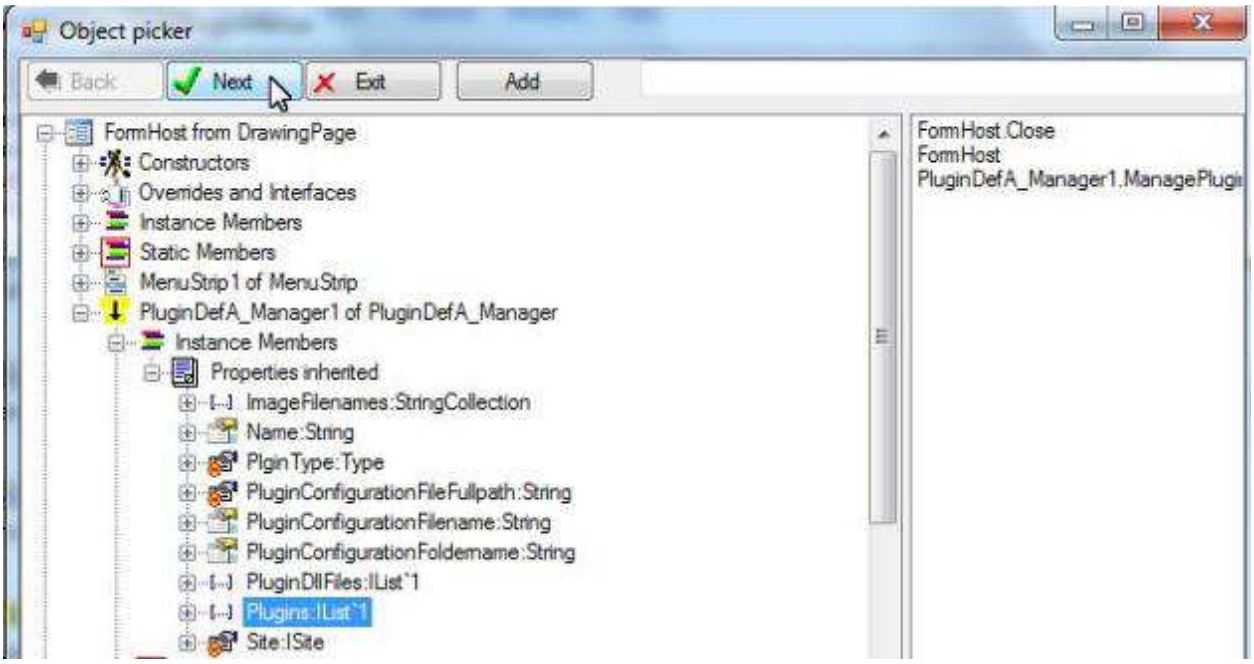


Rename the method:

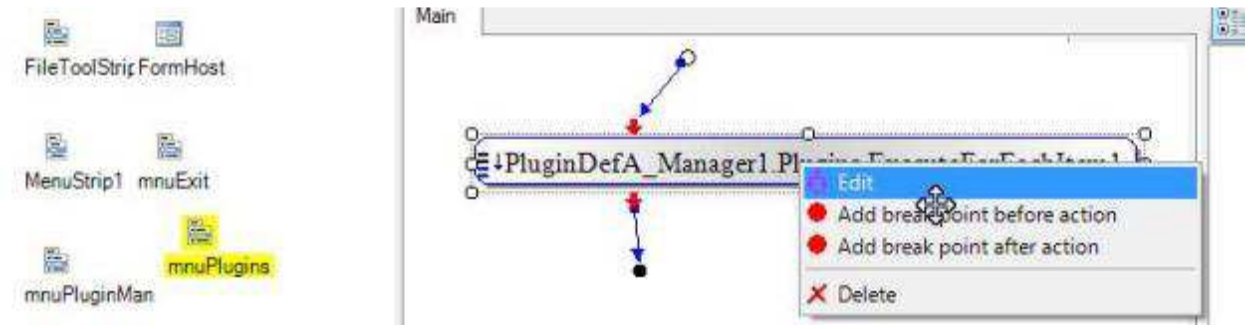
Create an “Execute actions for all items” action to go through all available plug-in object:



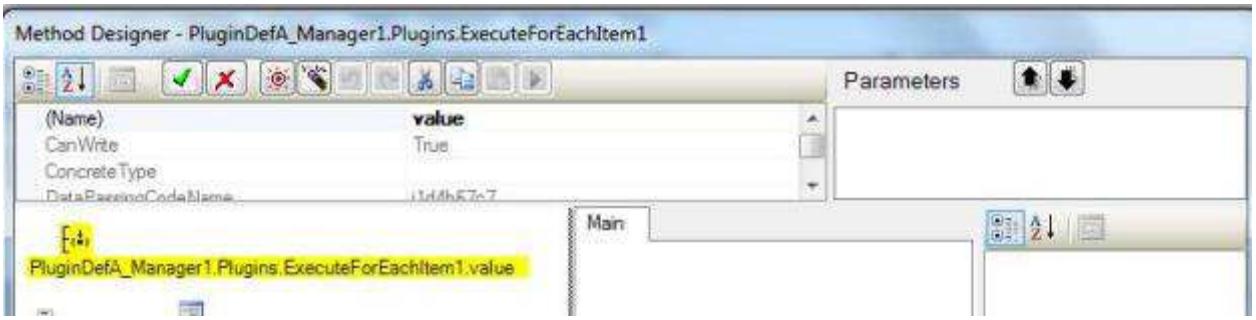
Select “Plugins” property of the plug-in manager:



Edit the action to work on each available plug-in:

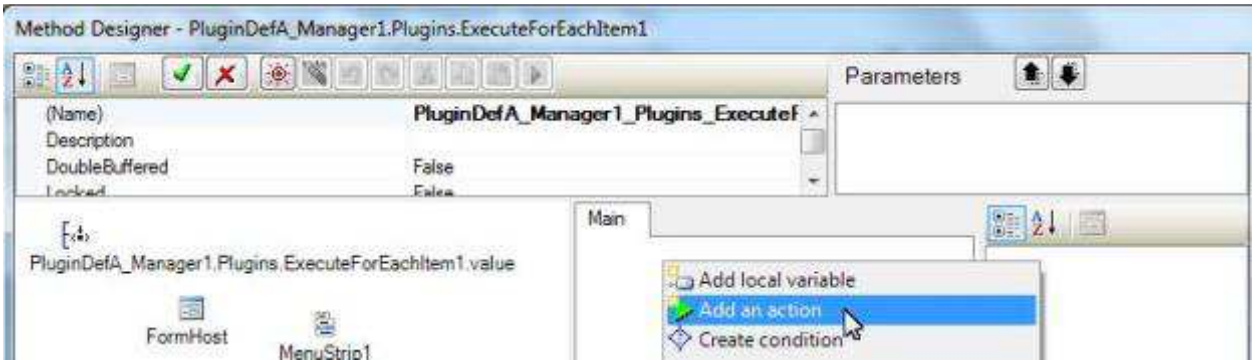


Note that the "value" variable represents the plug-in to be processed:

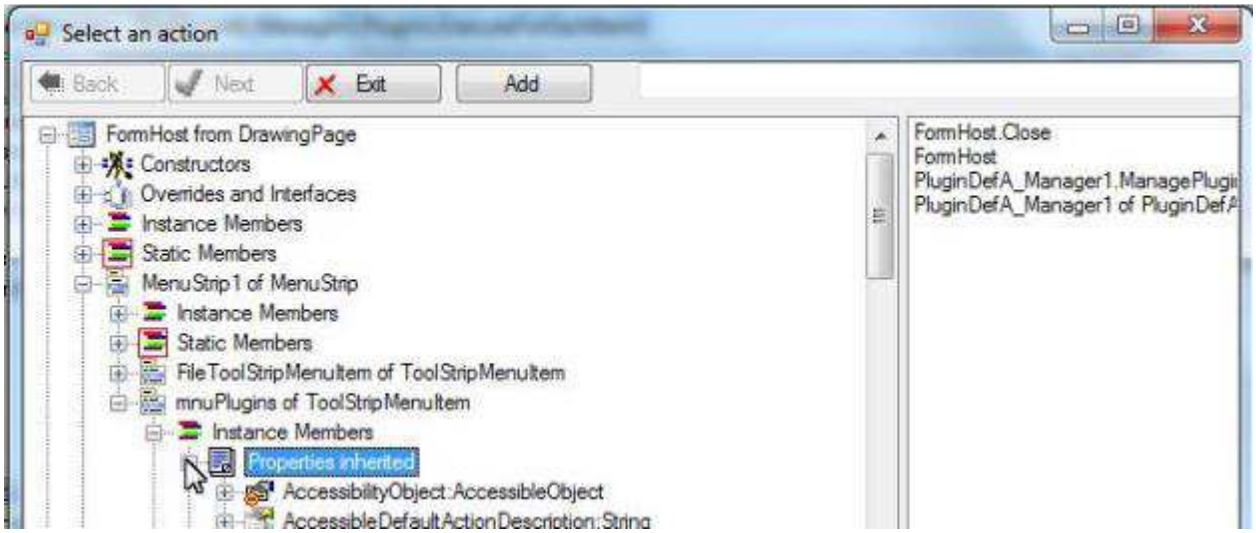


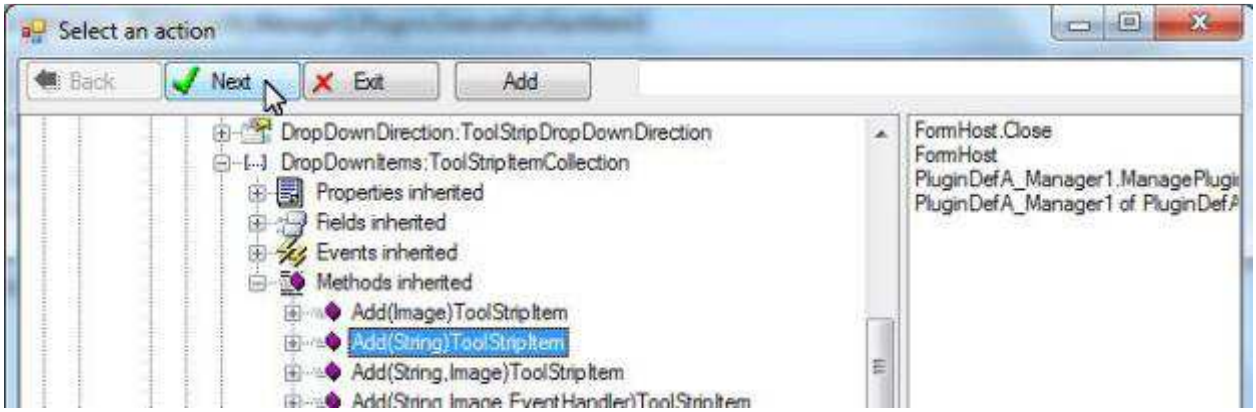
Add plug-in sub menu item

Create an action to add a menu item under "Plug-ins" menu:

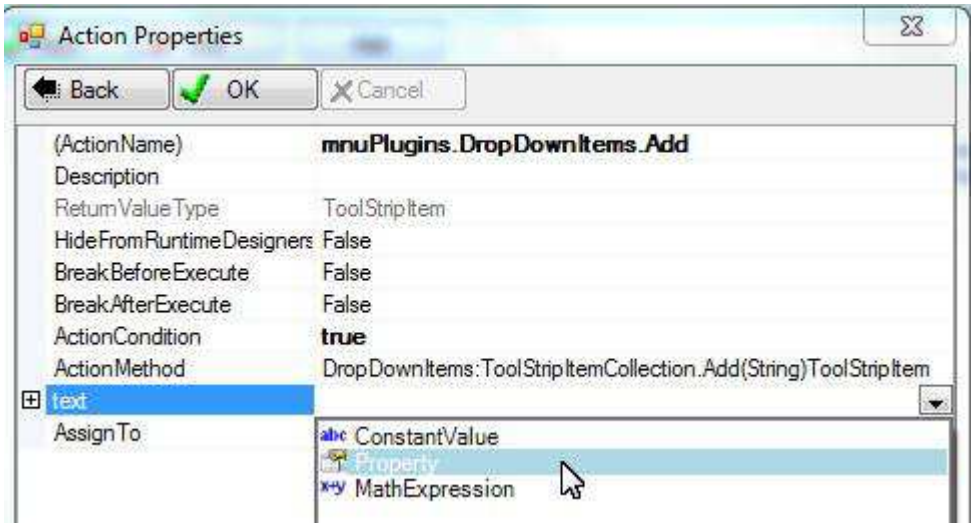


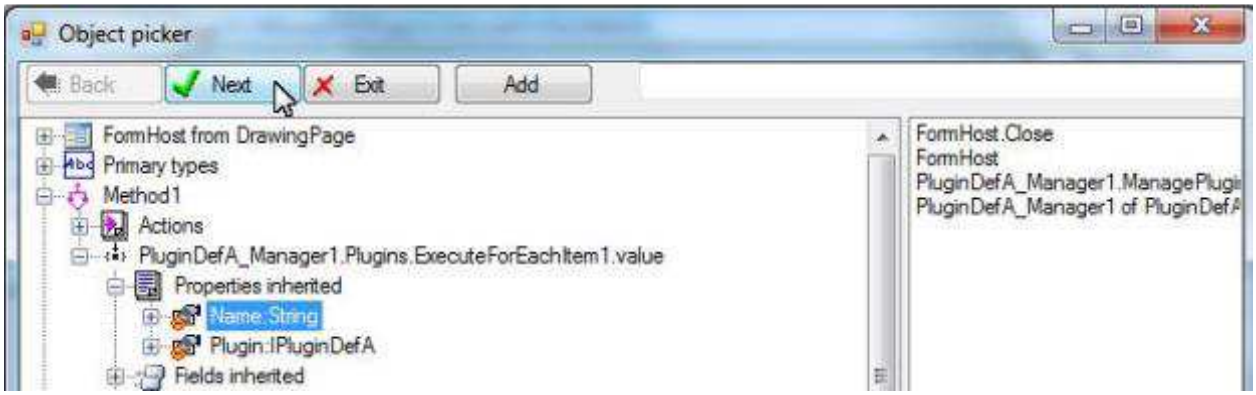
Select "Add" method of DropDownItems of menu "Plug-ins":



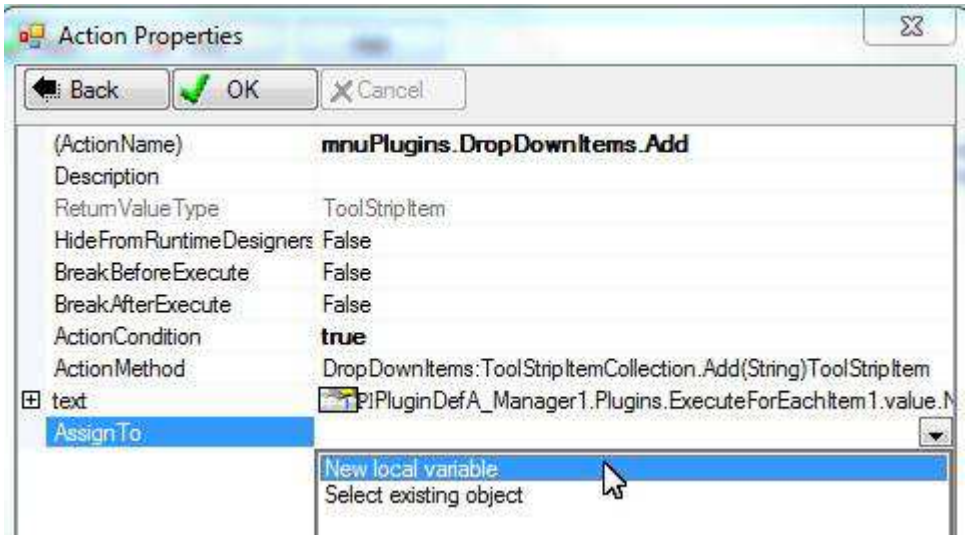


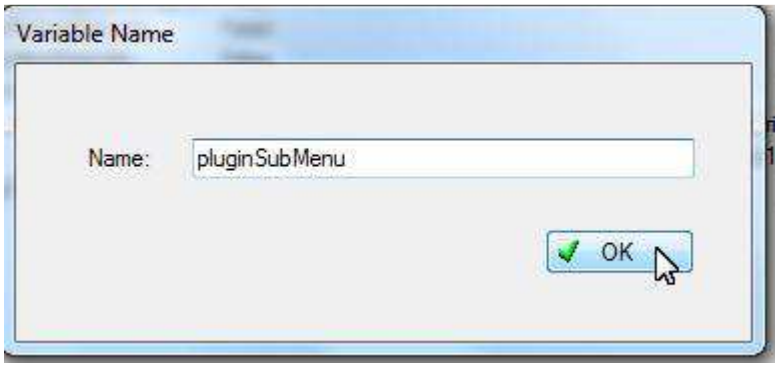
Use the name of the plug-in as the menu text:

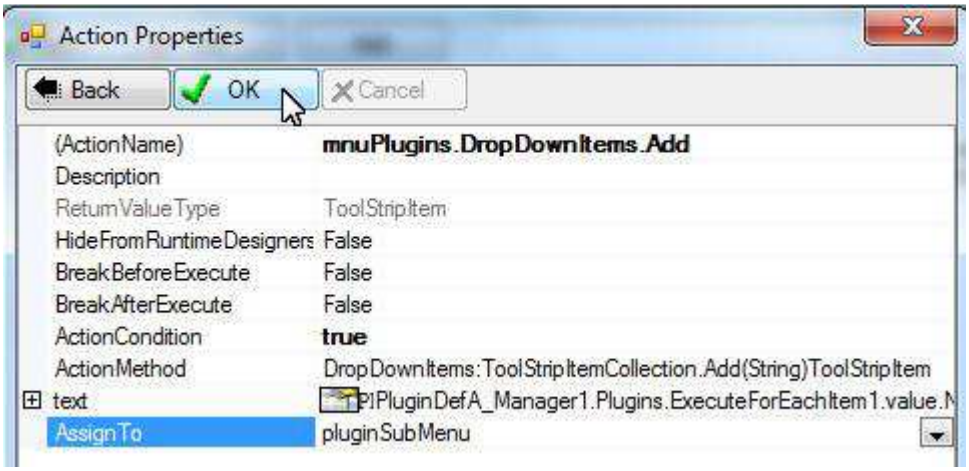




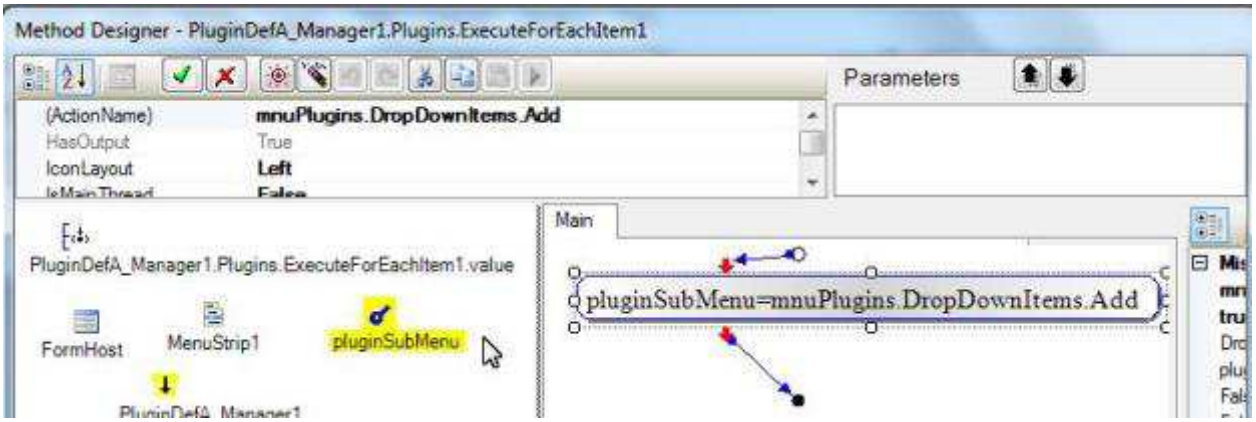
Assign the new menu item to a new variable:





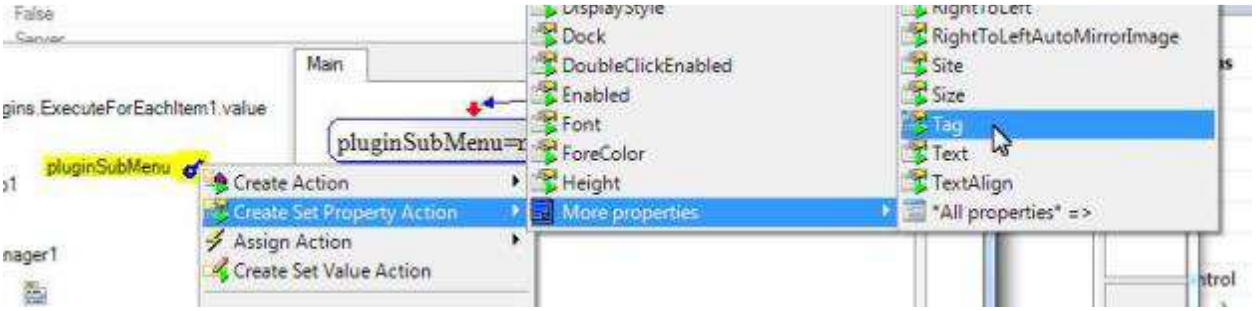


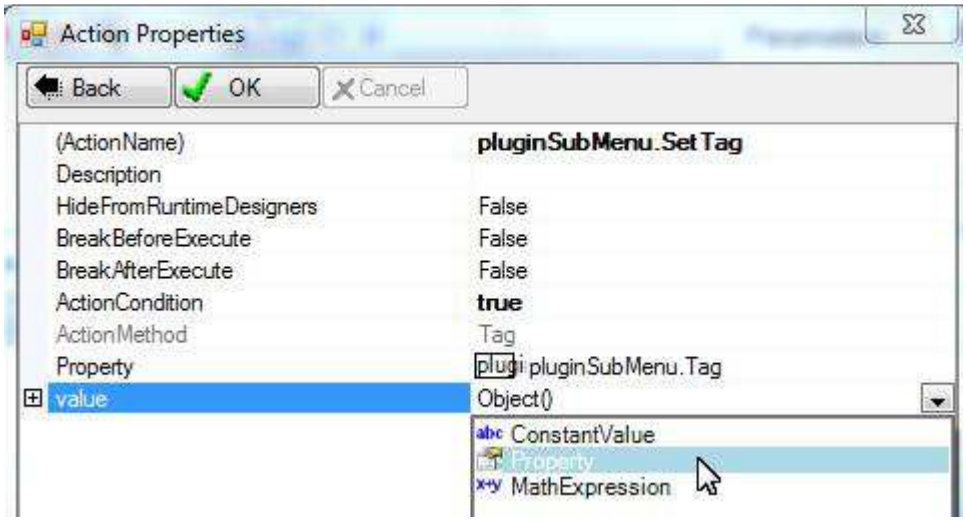
The action is created. Note that the variable, pluginSubMenu, is the new plug-in menu item added to the main menu:

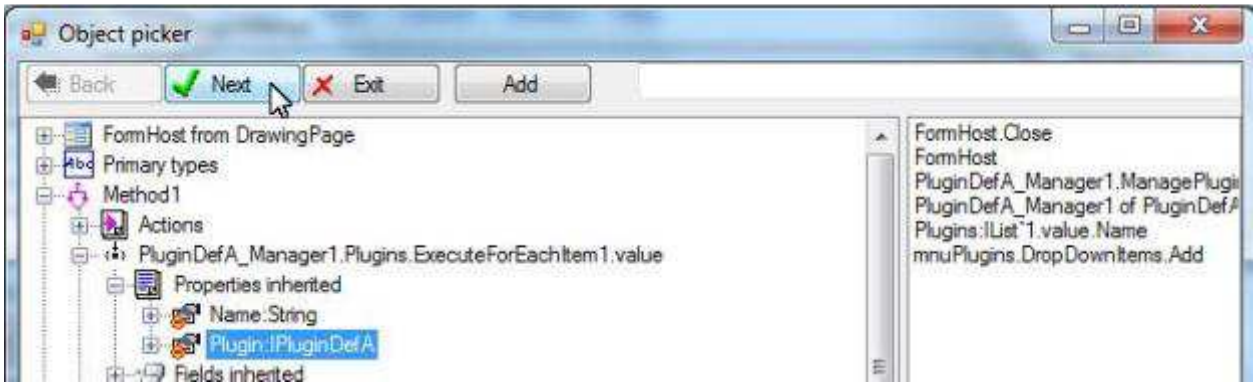


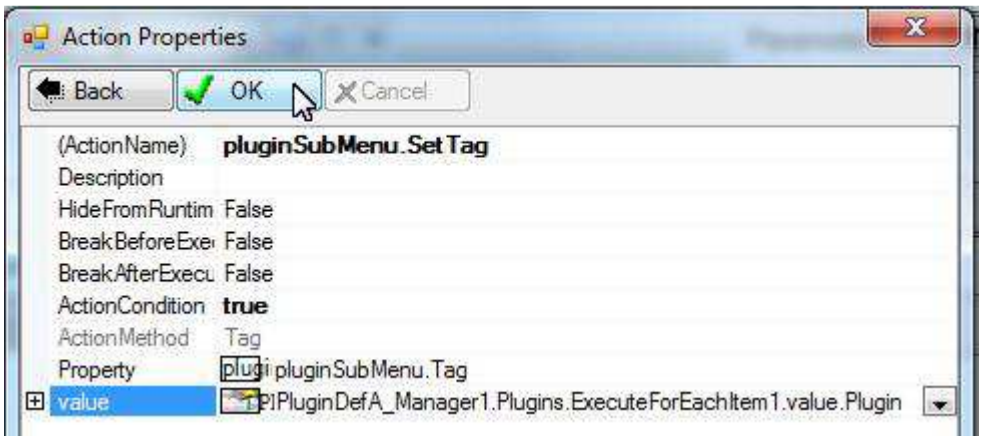
Associate plug-in object with sub menu item

When the sub menu item is selected, we need to know which plug-in the sub menu item is associated. We may use the menu text to search for the plug-in because the plug-in manager has a GetPluginByName method. We may also set the plug-in object to the Tag property of the menu item:

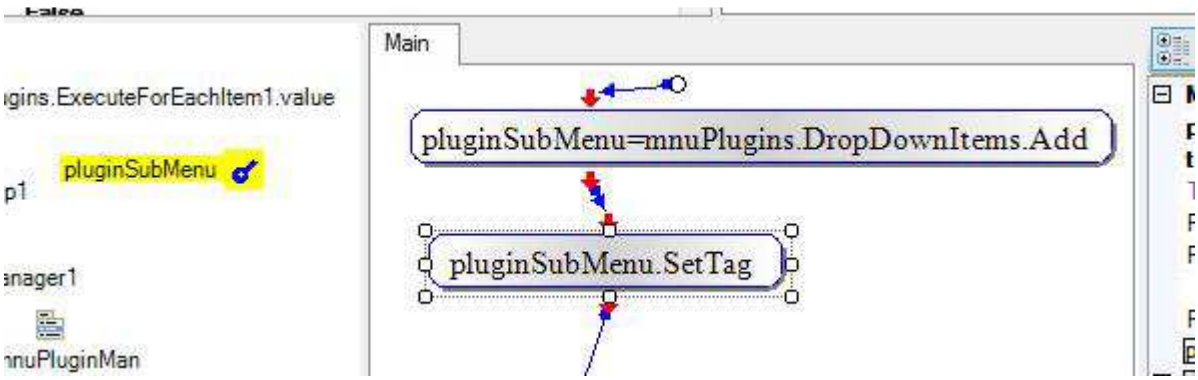






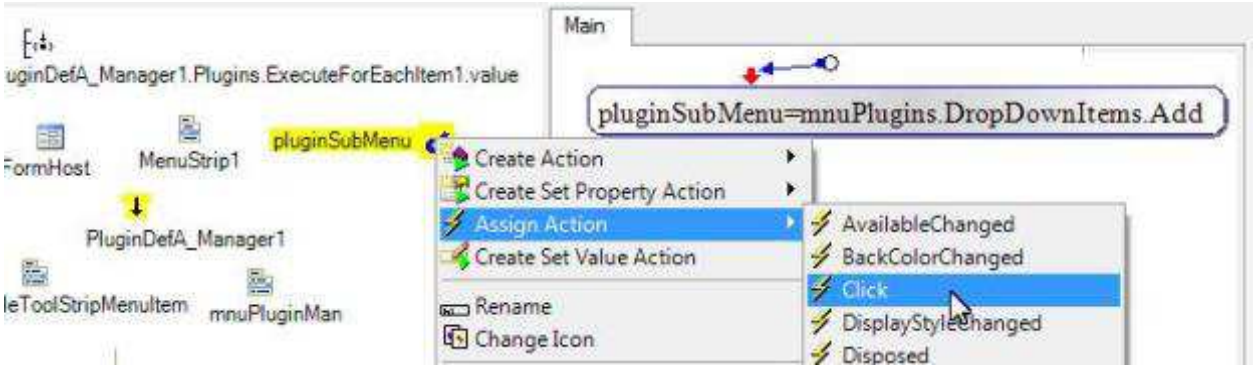


Link the actions:



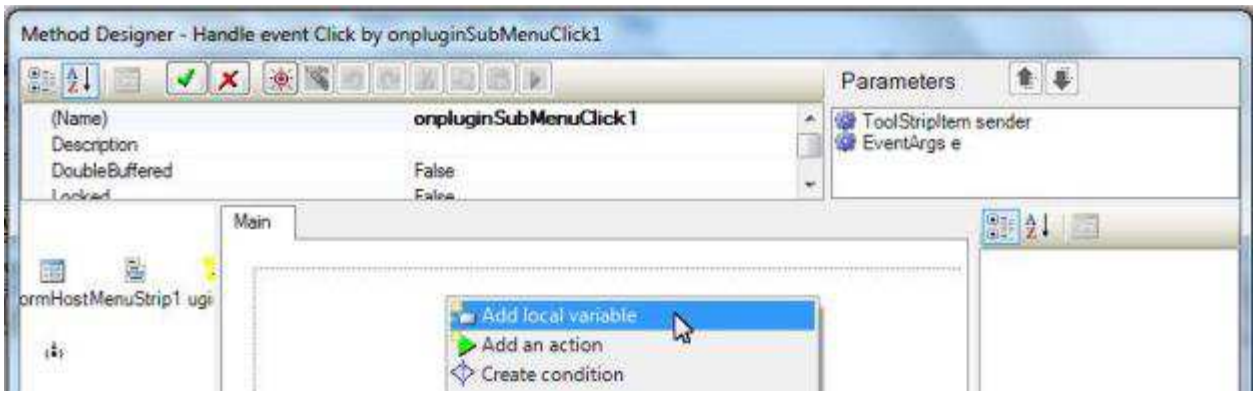
Assign actions to sub menu item

We need to assign actions to the Click event of the new menu item:

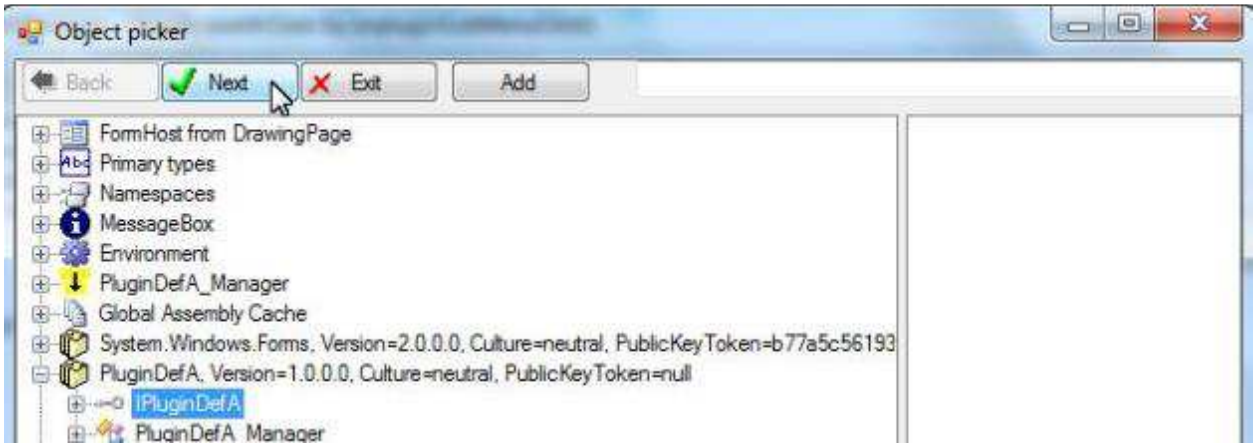


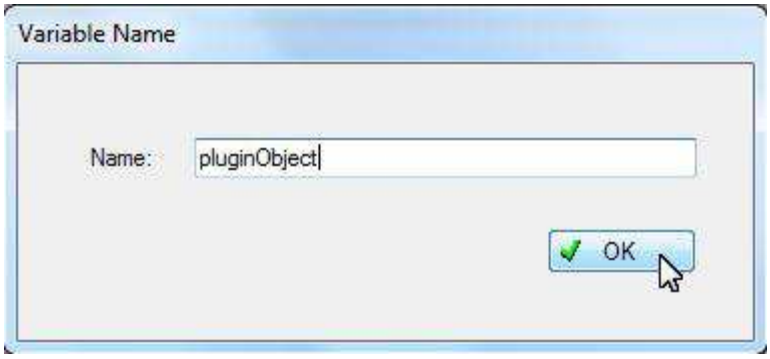
A method editor appears to let us adding actions to be executed when the new menu item is clicked.

We want to add an action to execute ShowForm method of the plug-in. But first we need to retrieve the plug-in object from the Tag property of the menu item. Let's use a local variable to do it:

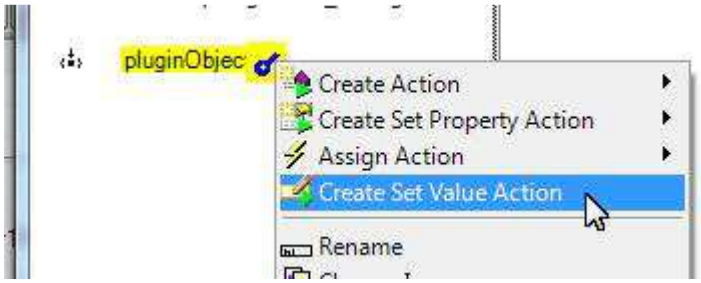


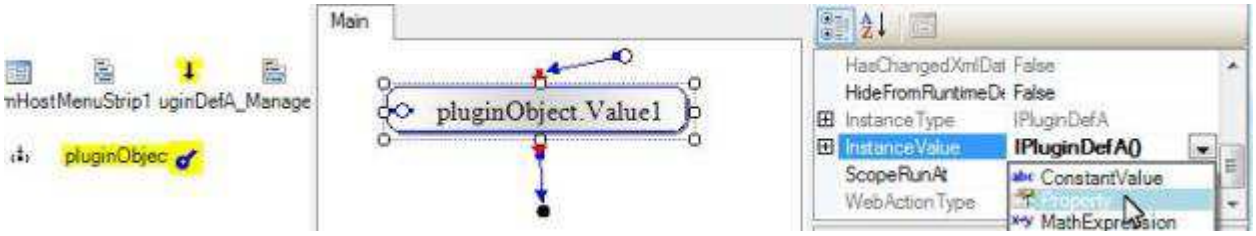
Select the plug-in definition interface as the variable type:

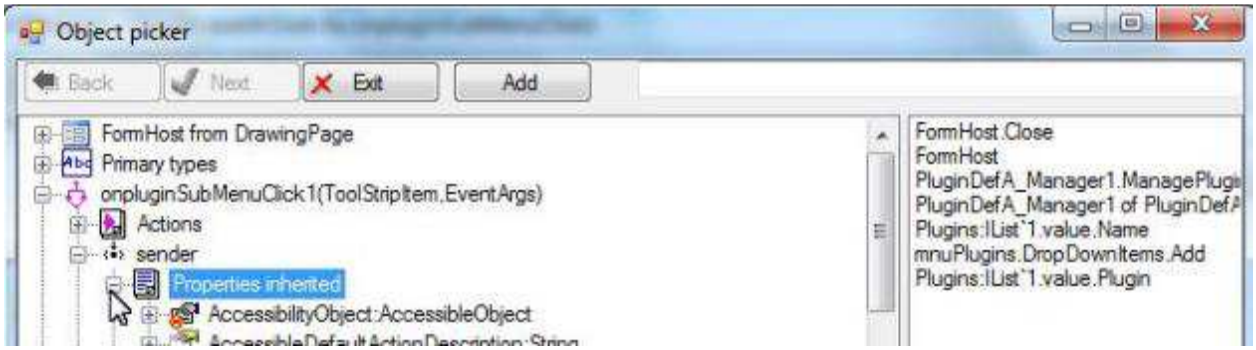


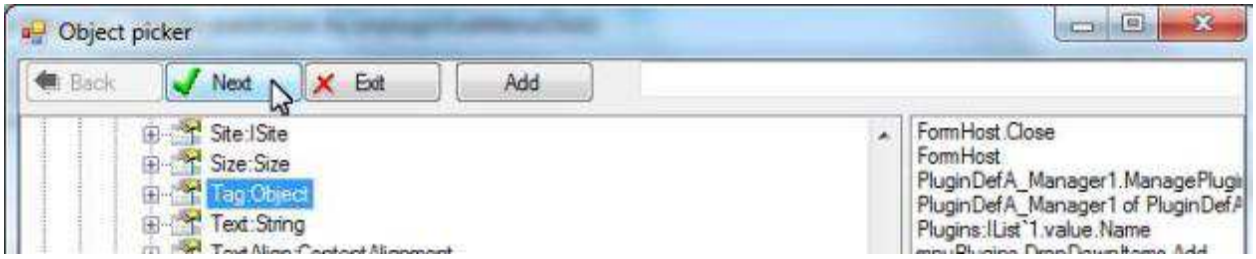


Set the variable to the Tag property of the sender:

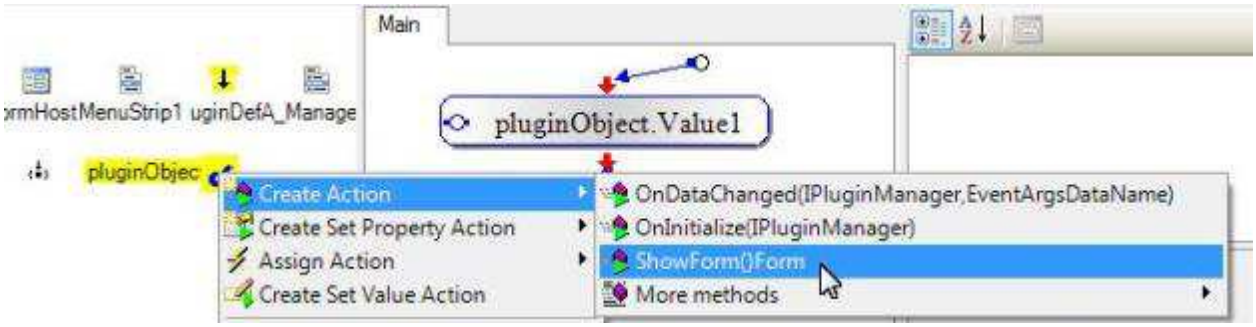




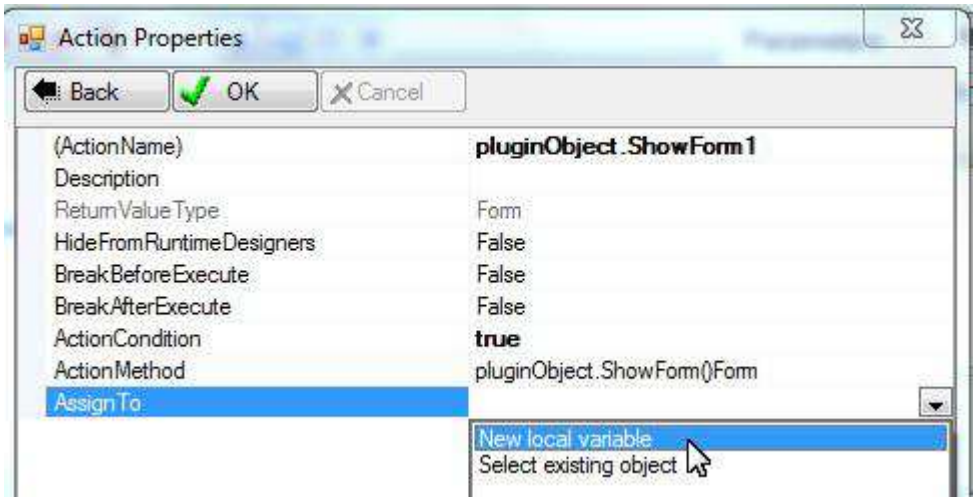


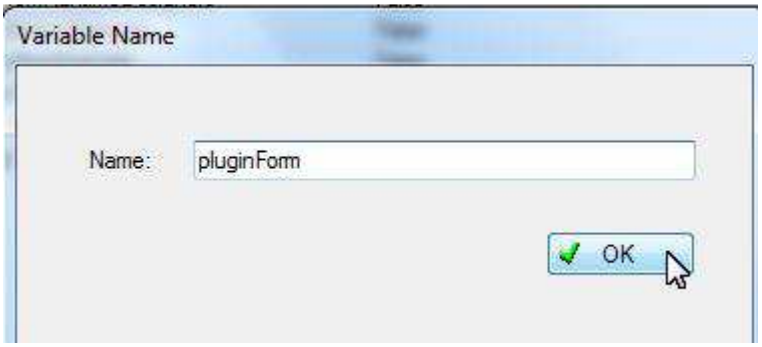


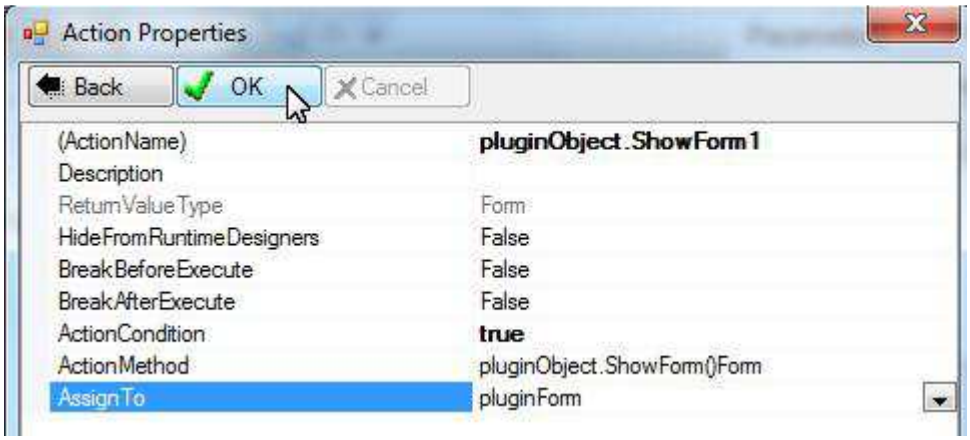
Create a ShowForm action:



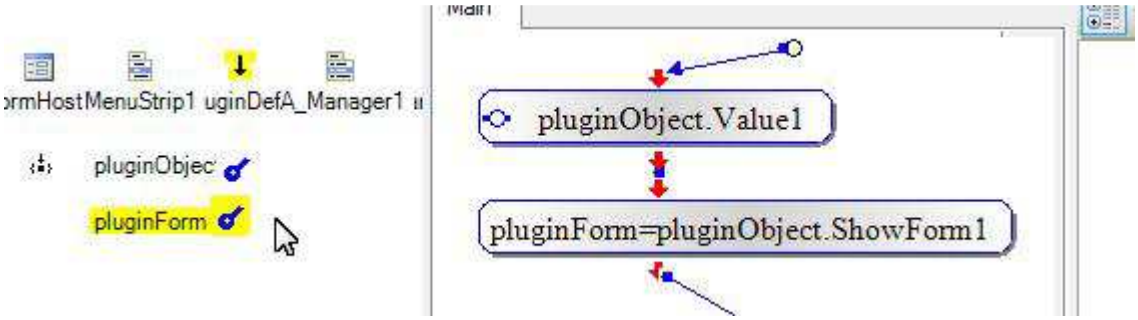
ShowForm returns a form that it shows. Assign the form to a variable so that we may set its MDI parent:



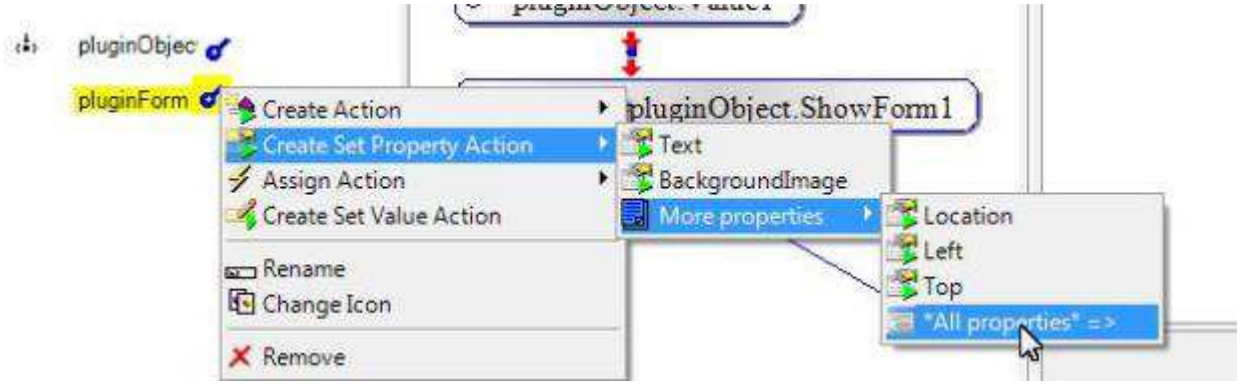


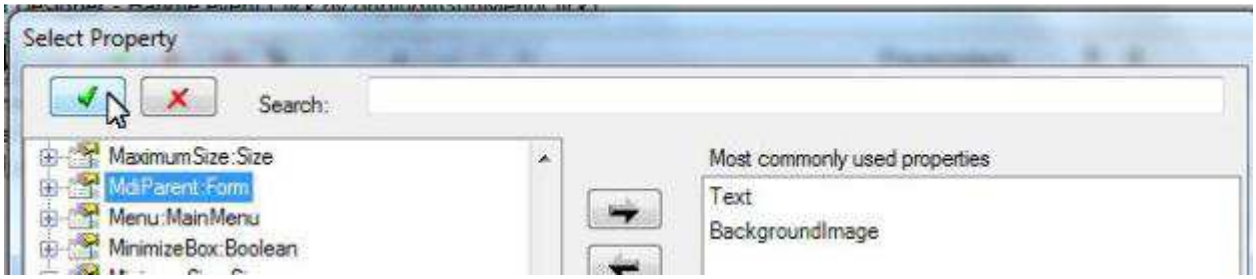


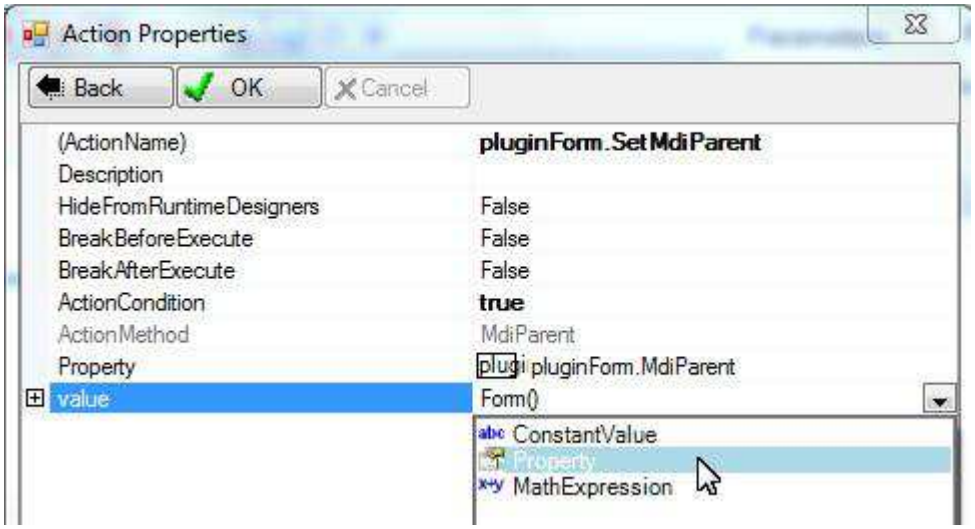
Link the actions:

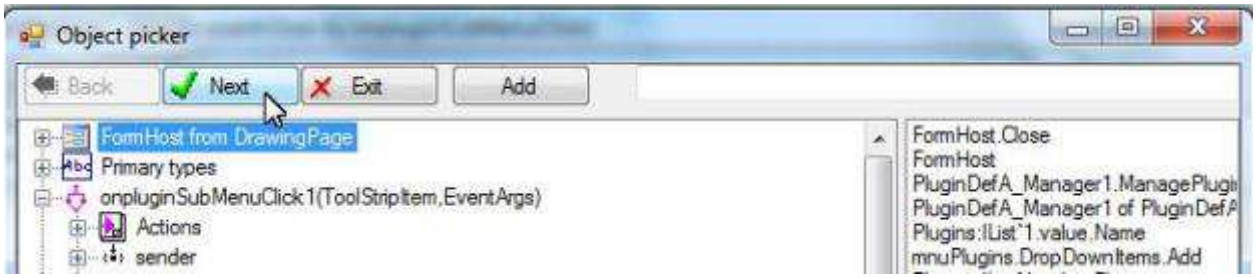


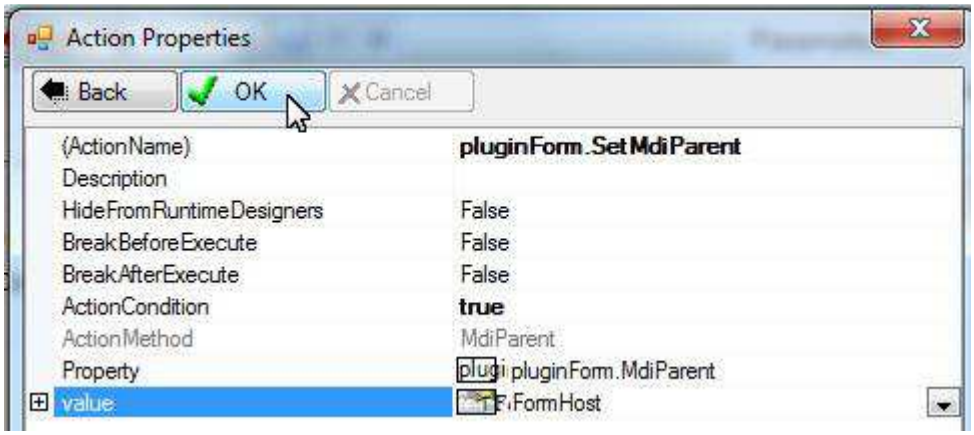
Set MdiParent property of the plug-in form to FormHost:



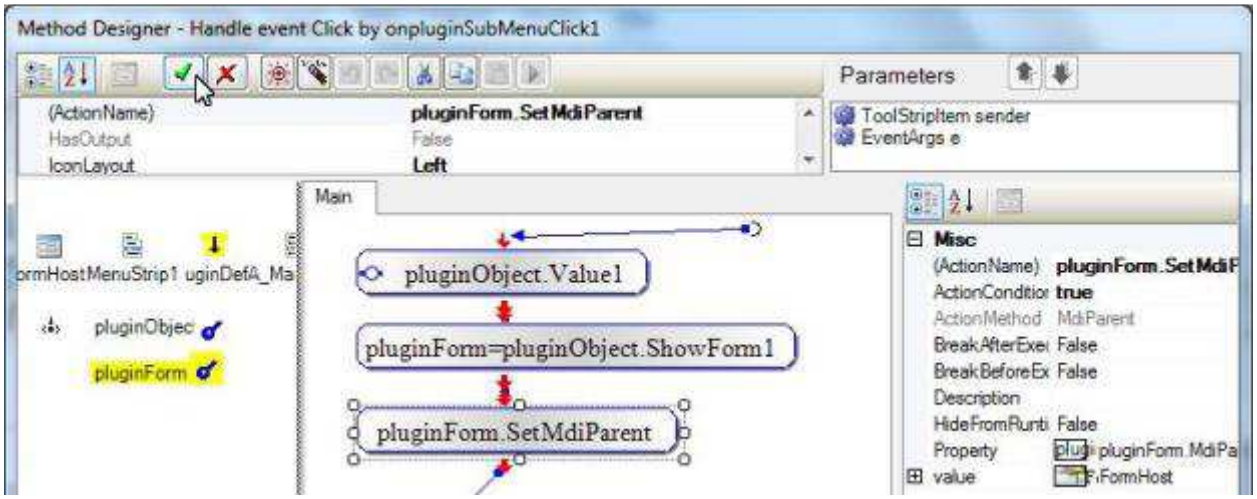




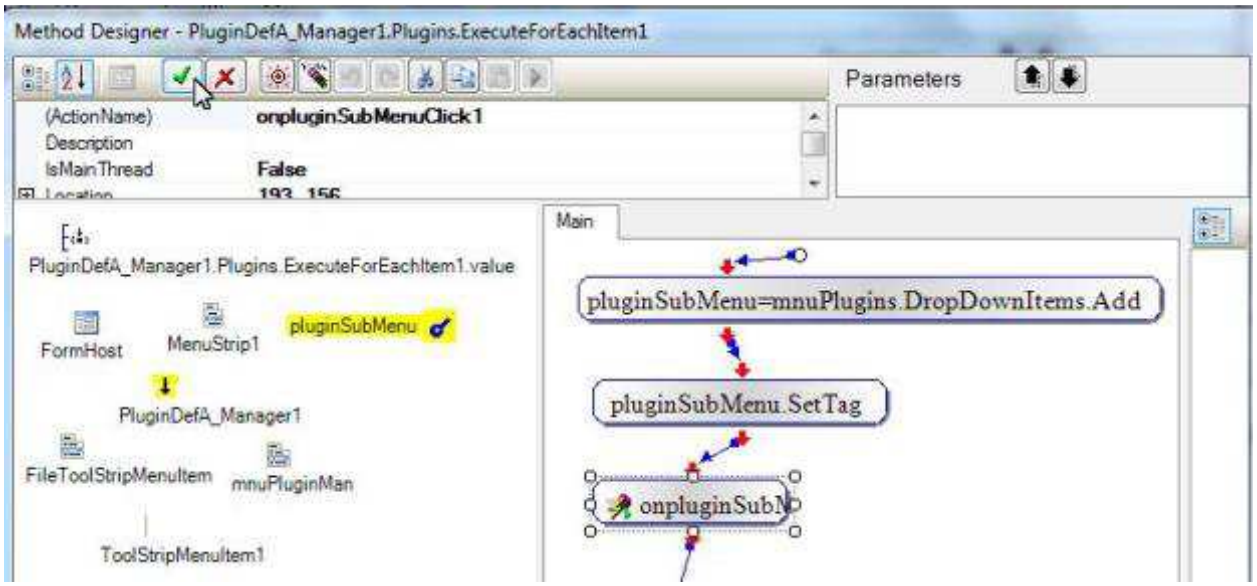




Link the action to the last action. We are done handling Click event of the new sub menu item:



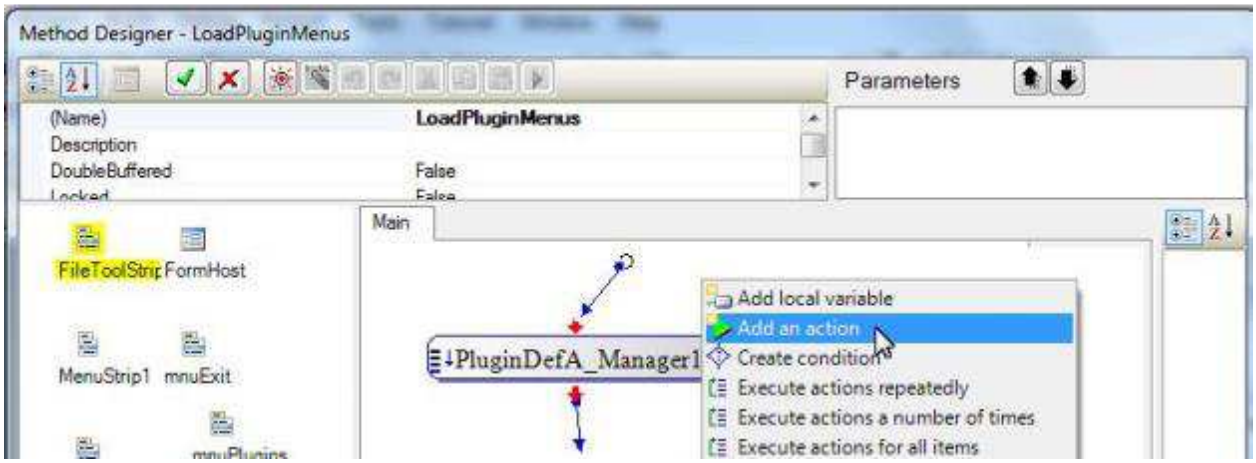
We are done processing one plug-in object.



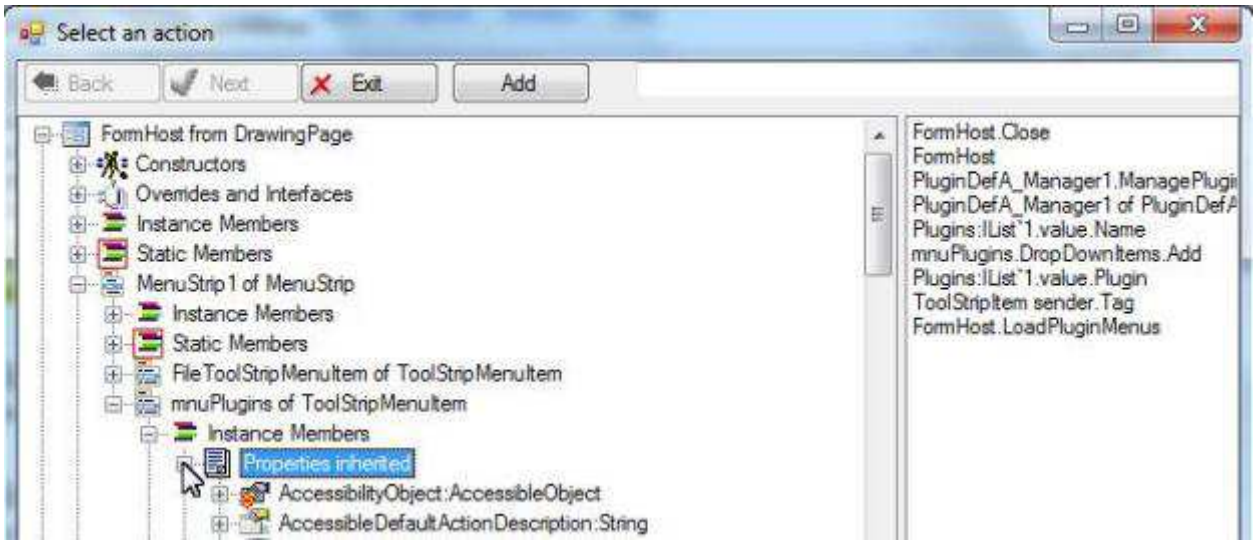
In summary, we added a new sub menu item under the plug-ins main menu; we set the Tag property of the new sub menu item to the plug-in object; we assigned actions to Click event of the new sub menu item.

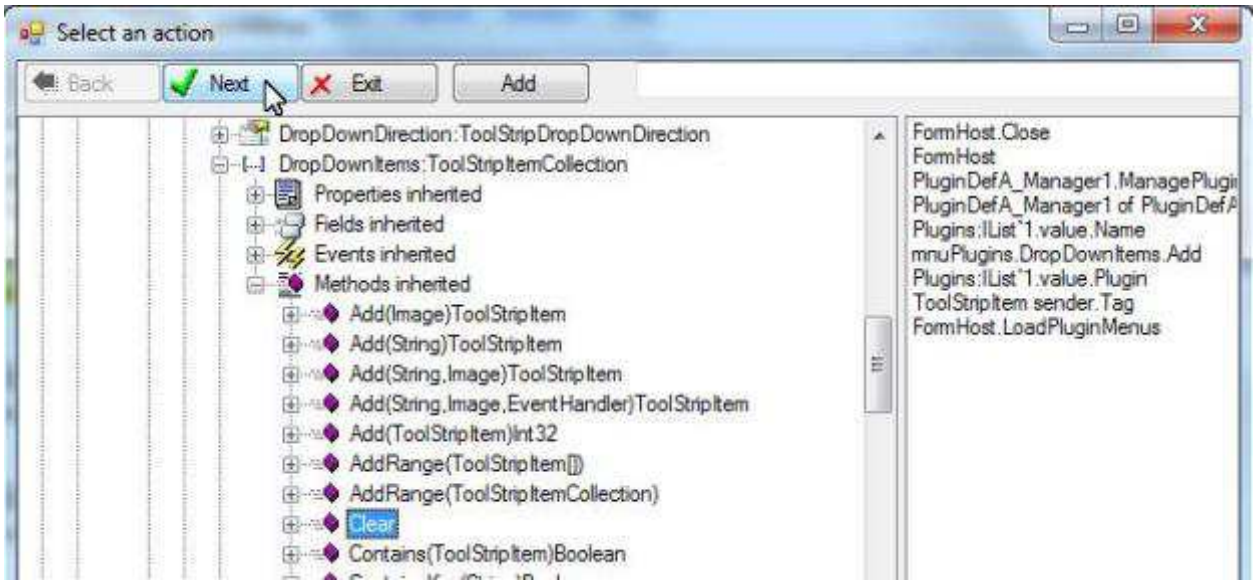
The actions we assigned to the Click event of the new sub menu item are 1) retrieving the plug-in object from the Tag property of the sender; 2) execute ShowForm on the plug-in object; 3) set the MdiParent property of the form returned from ShowForm to FormHost.

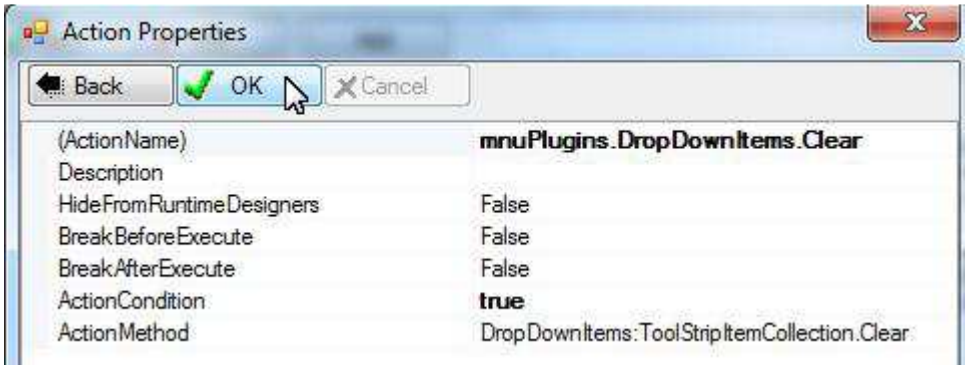
One thing we forgot to do is to remove all sub menu items from the plug-in main menu so that if this method is called more than once no duplicated sub menus will be created:



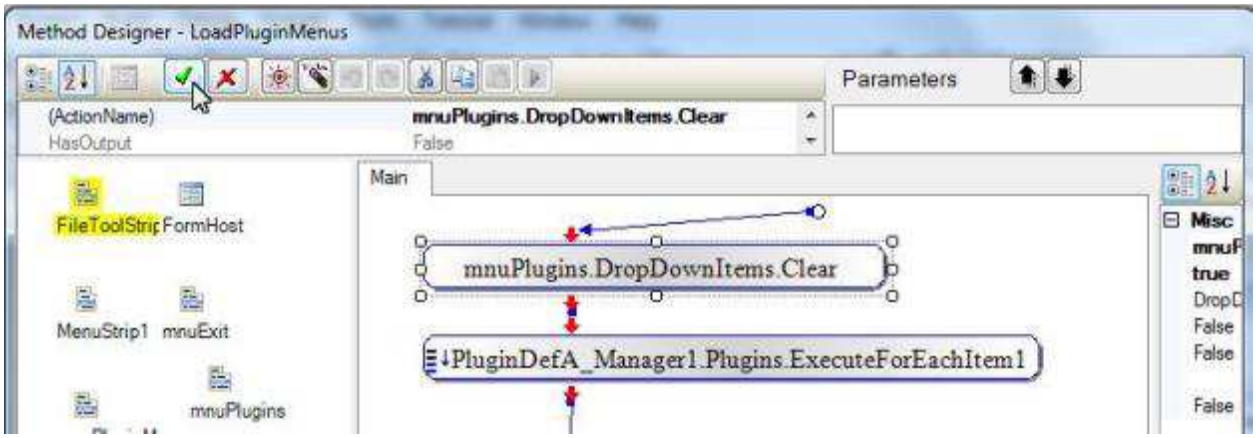
Select Clear method of DropDownItems of the plug-in main menu:



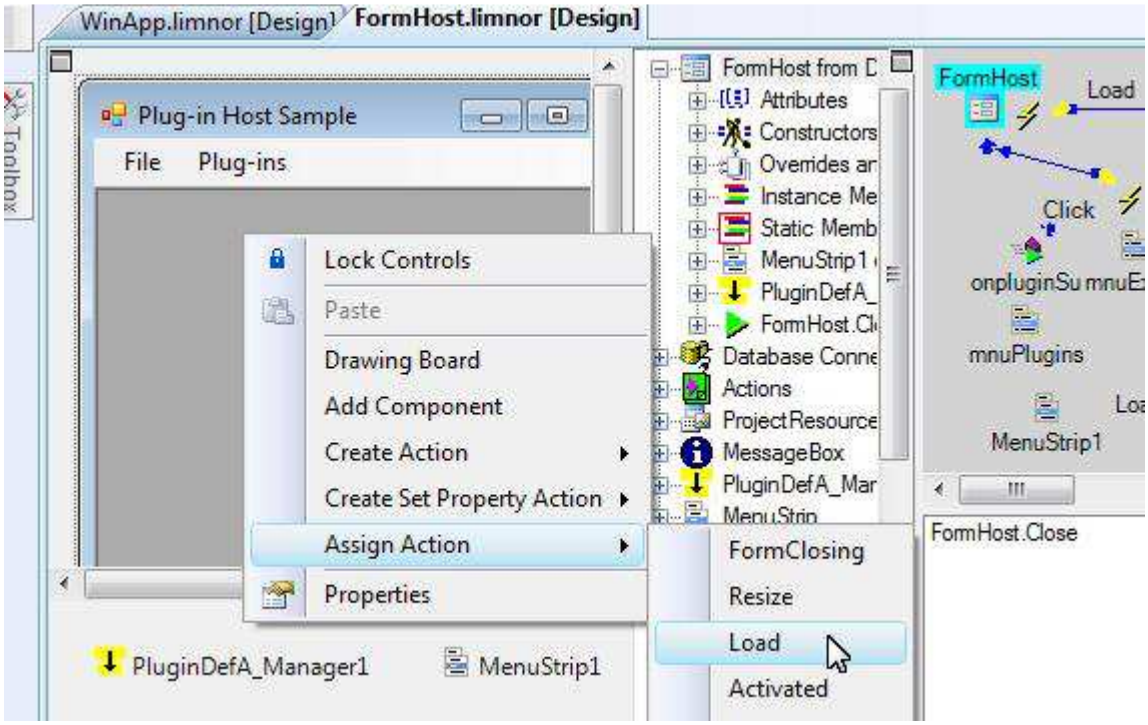


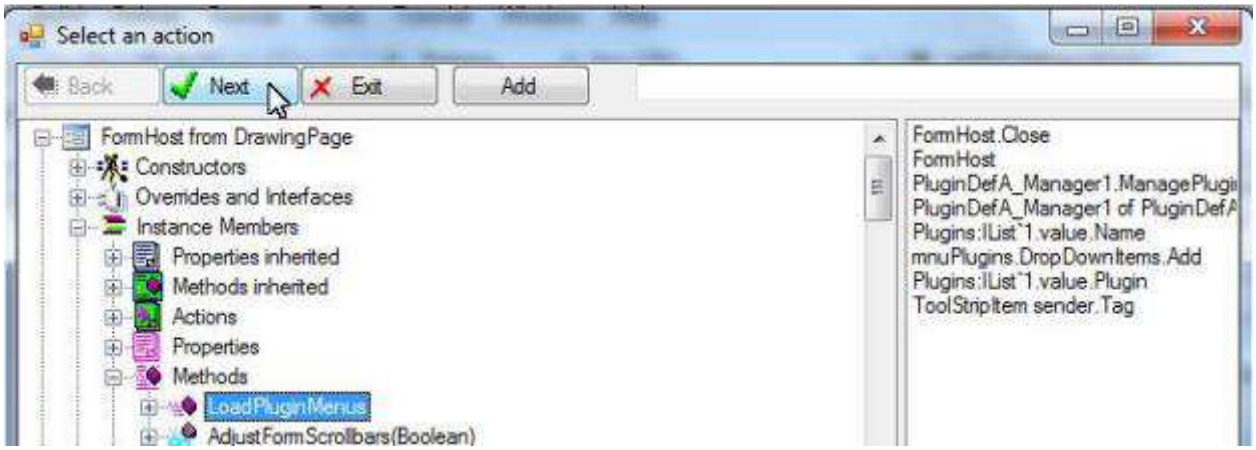


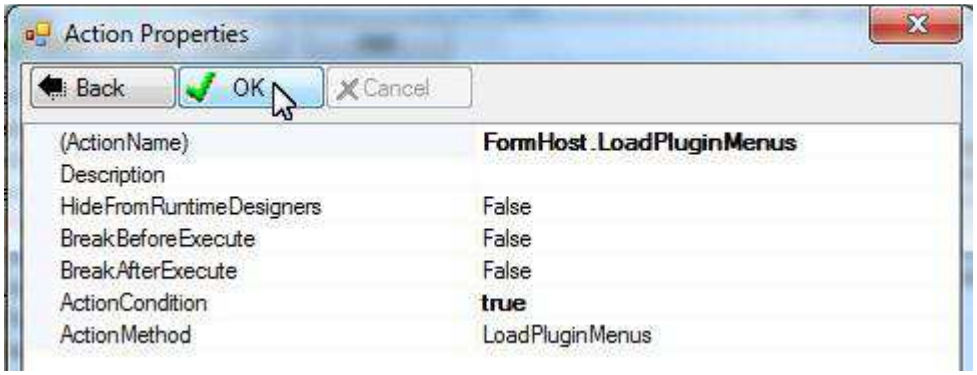
We need to link the action to be the first action. We are down processing all plug-in objects:



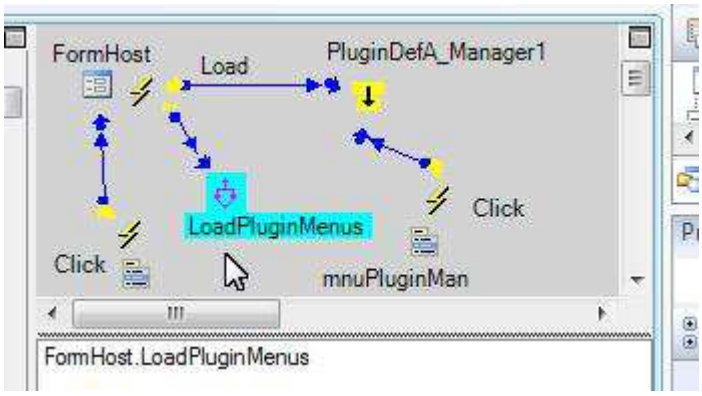
Load plug-in sub menus at form Load



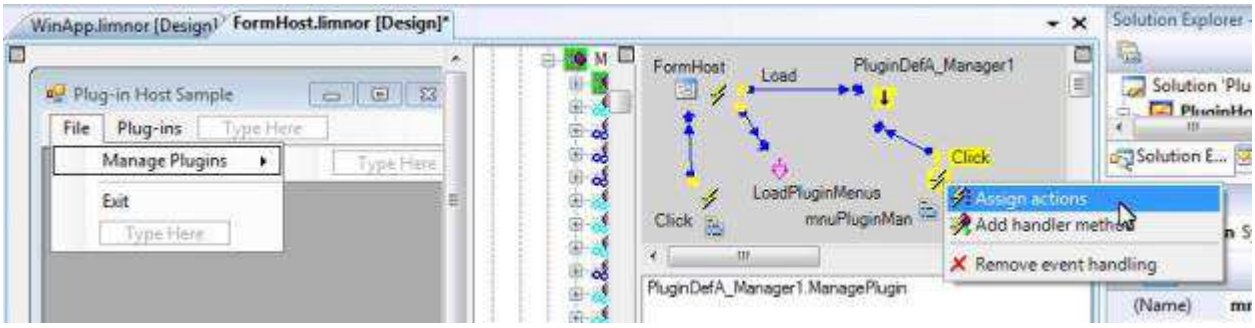




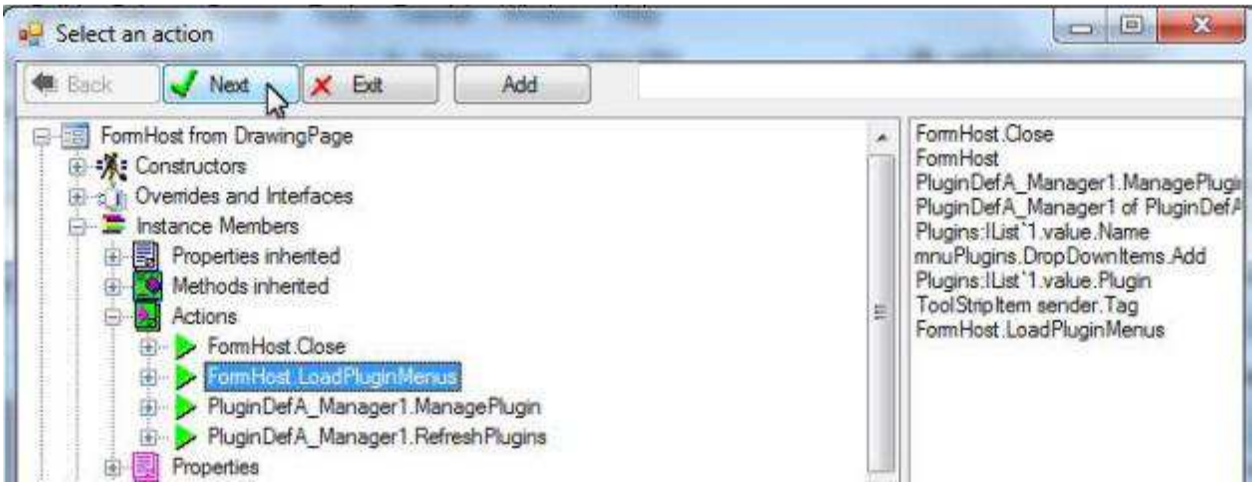
The action is created and assigned to Load event:



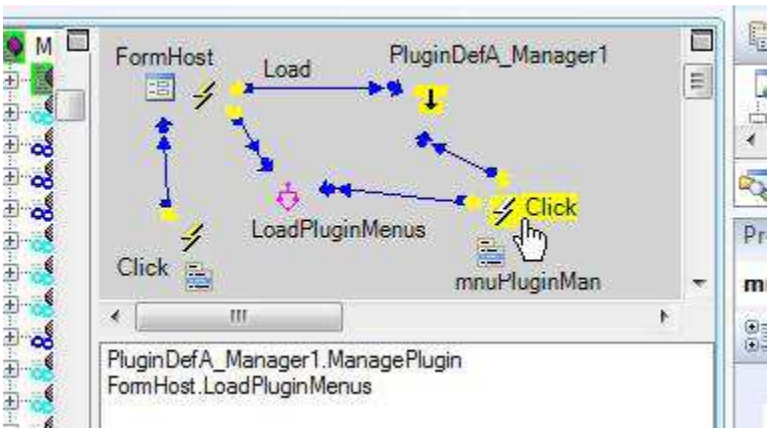
Load plug-in sub menus after managing plug-ins



Select LoadPluginMenus action:

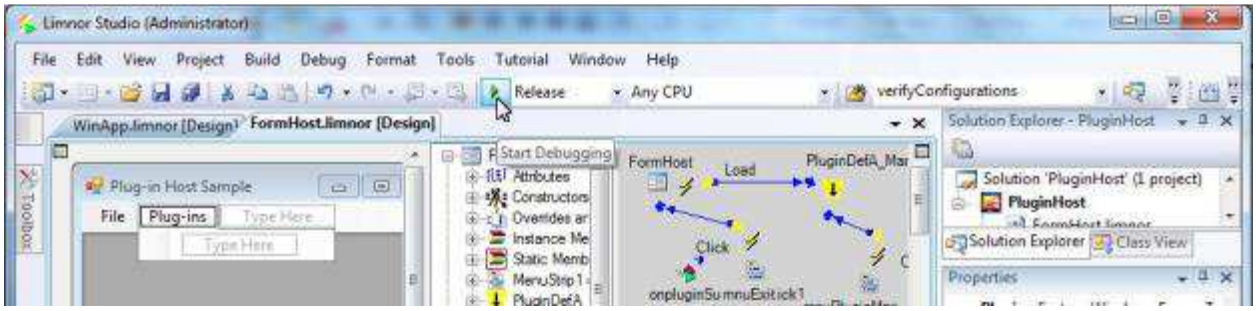


The action is assigned to the menu:

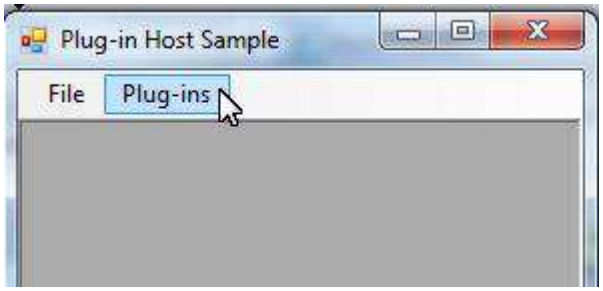


Test

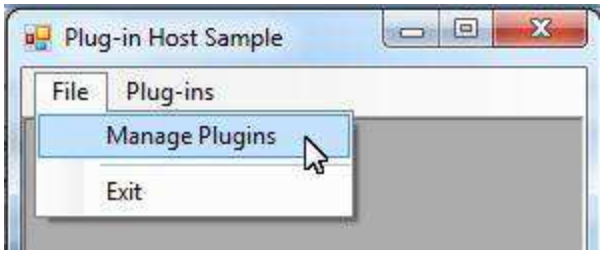
We have implemented all elements of a plug-in system. We may test drive it now.



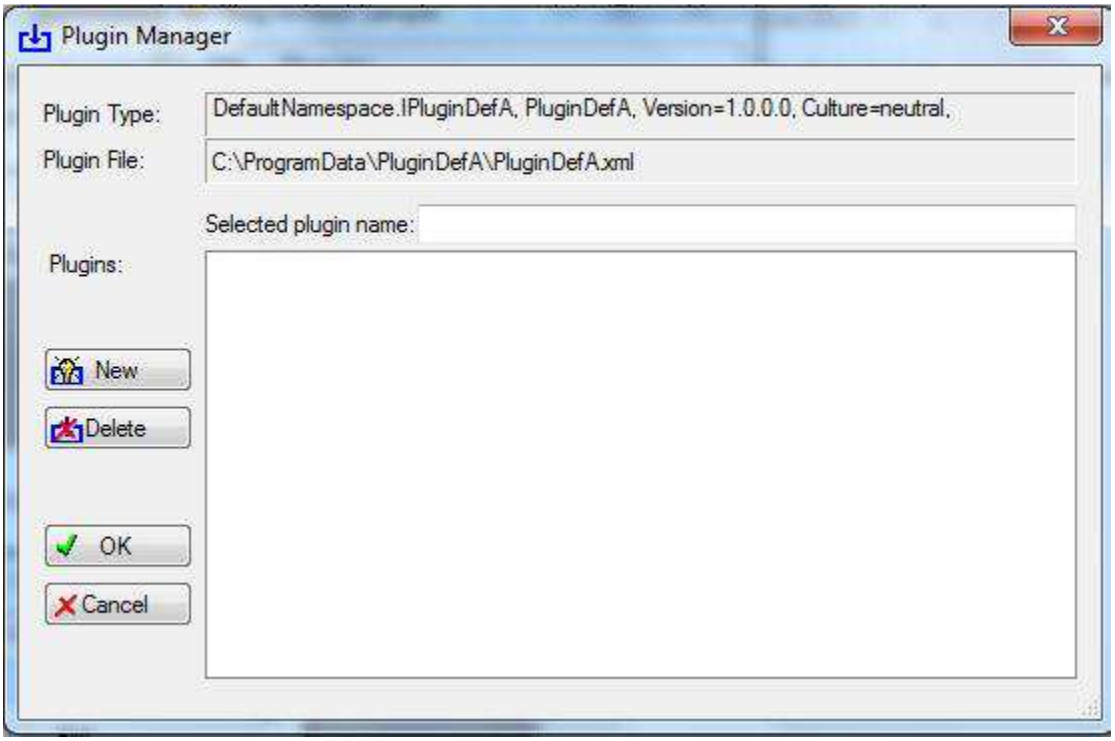
FormHost appears. We can see that there is not a sub menu under the plug-in main menu because we haven't loaded any plug-ins yet:



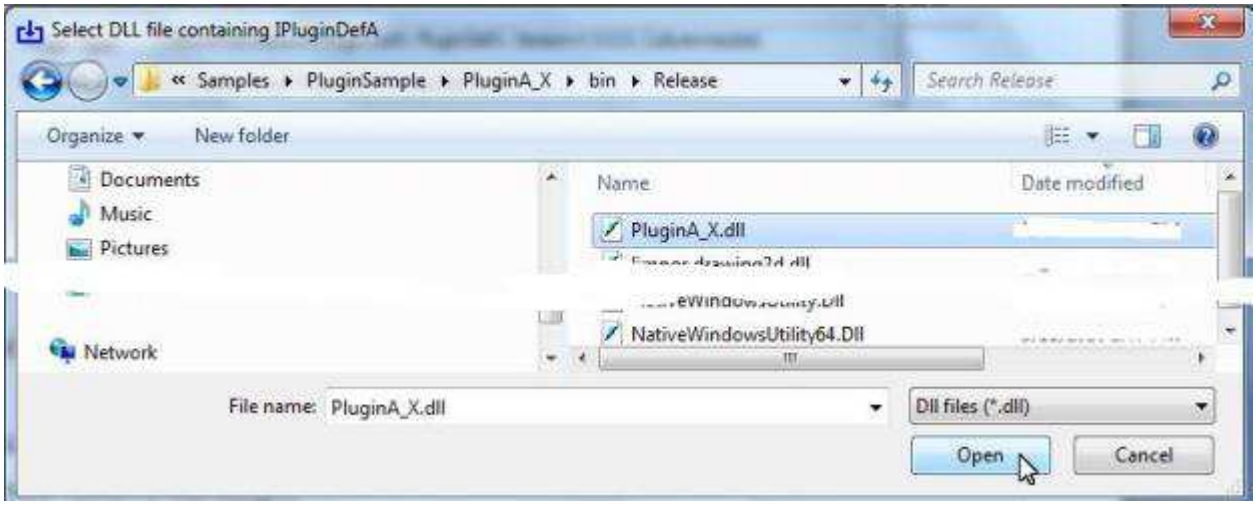
Let's select the plug-in manager menu:



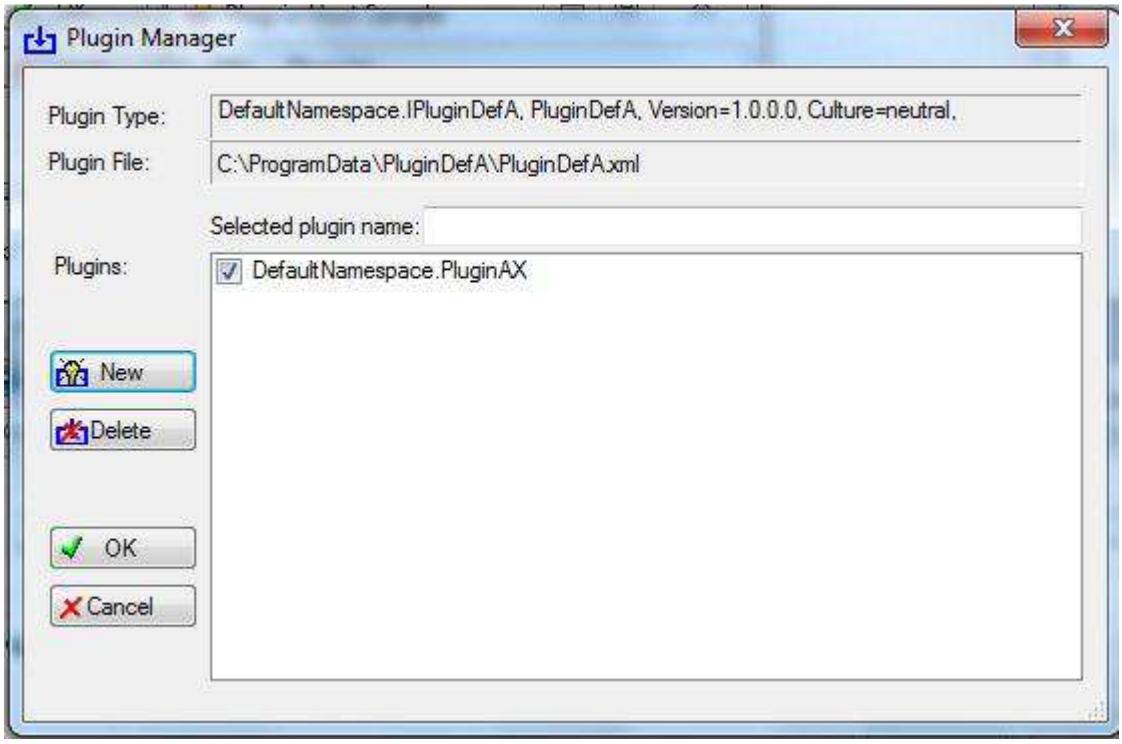
Plug-in manager appears:



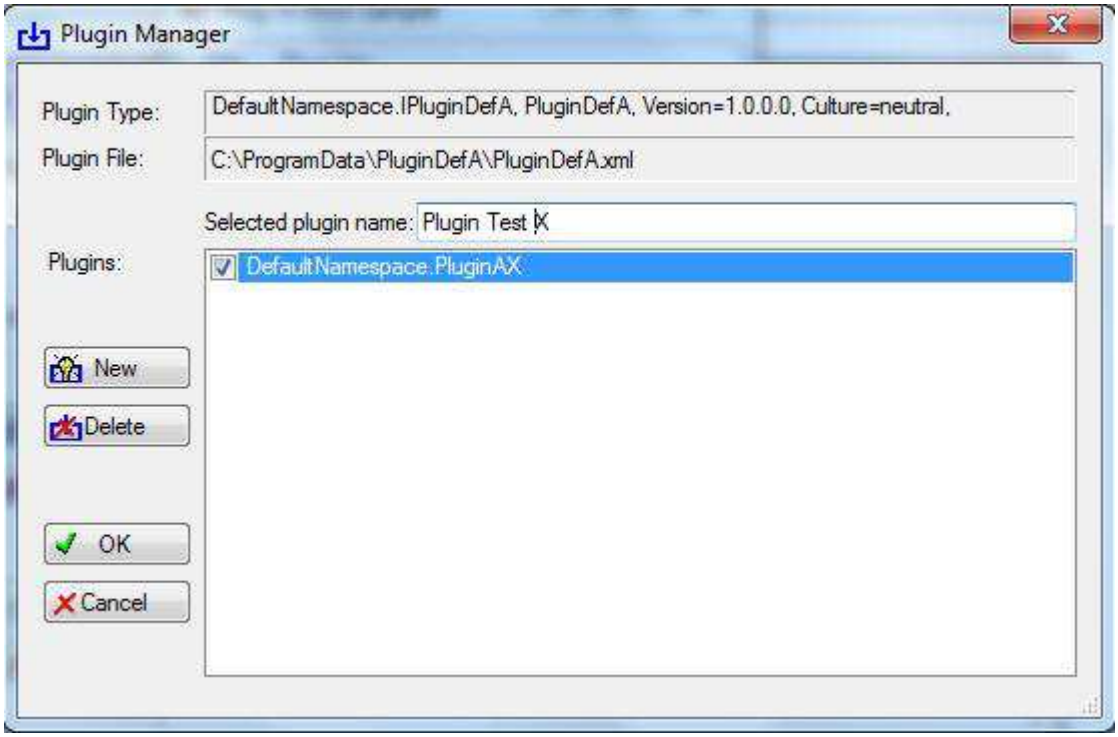
Click “New” button to select a plug-in DLL:



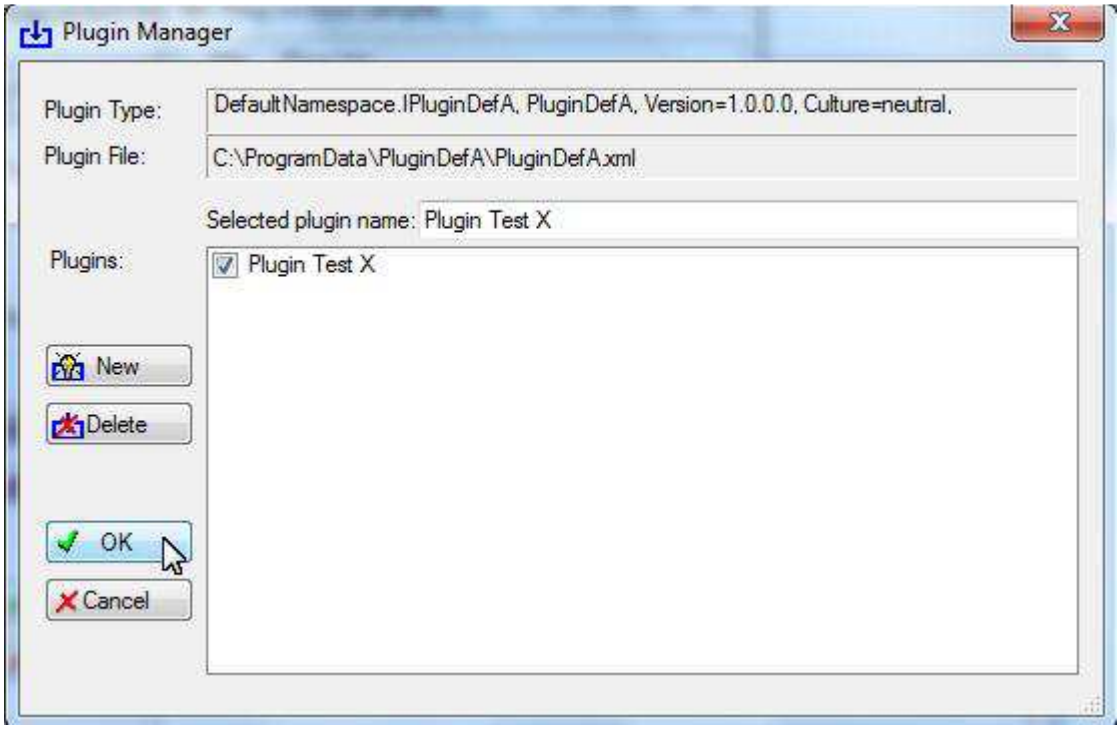
The plug-in object contained in the DLL appears in the list:



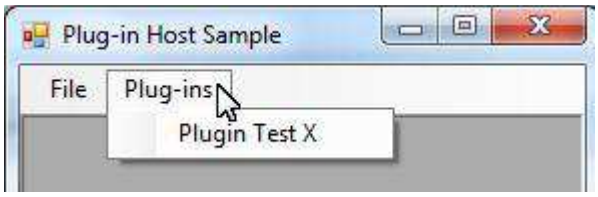
We may select the plug-in and give it a name:



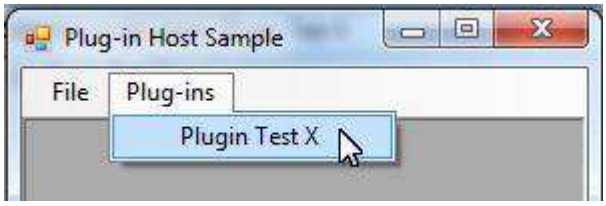
Click OK when we are done:



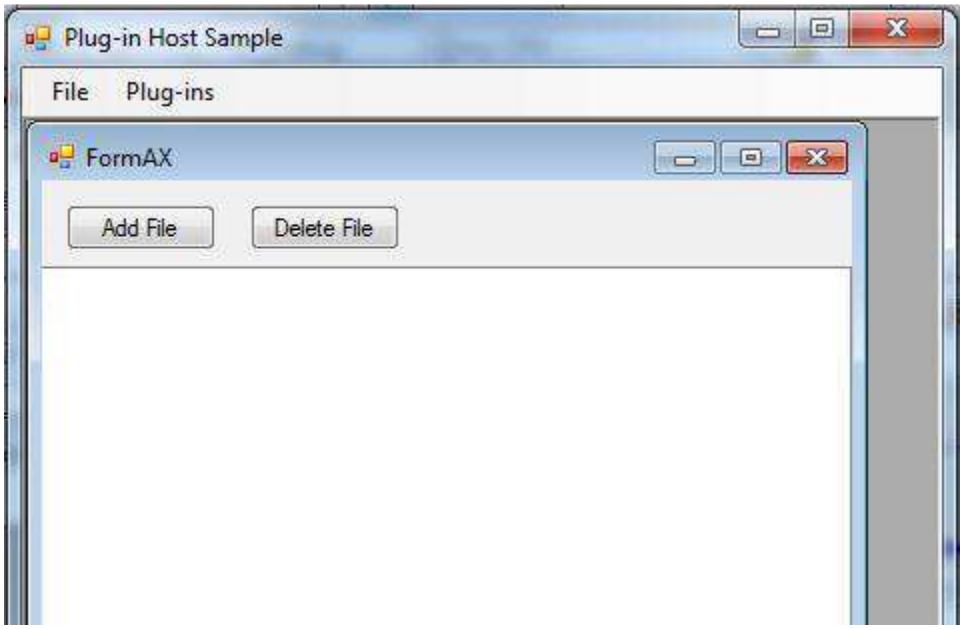
We can see that a new sub menu item appears under the plug-in main menu:



Choose the sub menu item:



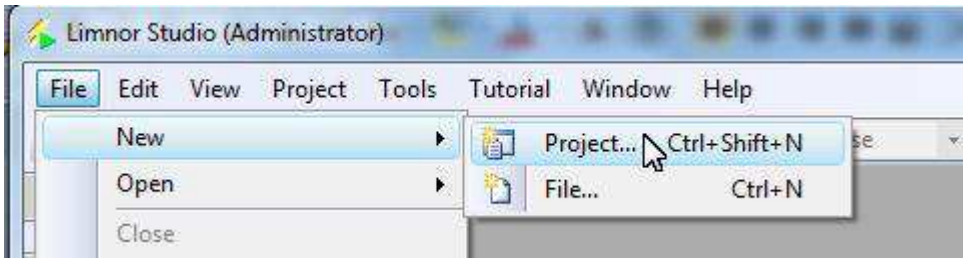
A child form appears inside MDI host:



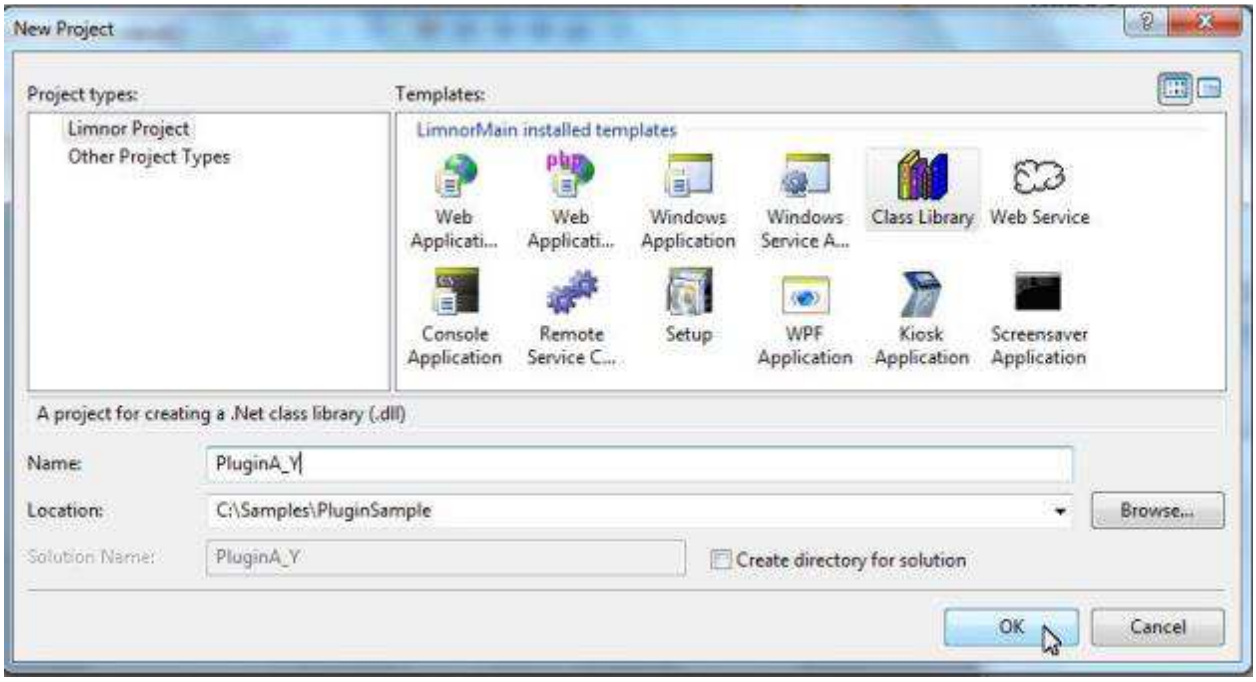
Another Implementation of Plug-in

We now develop another plug-in and load it into above host application.

Create a DLL project

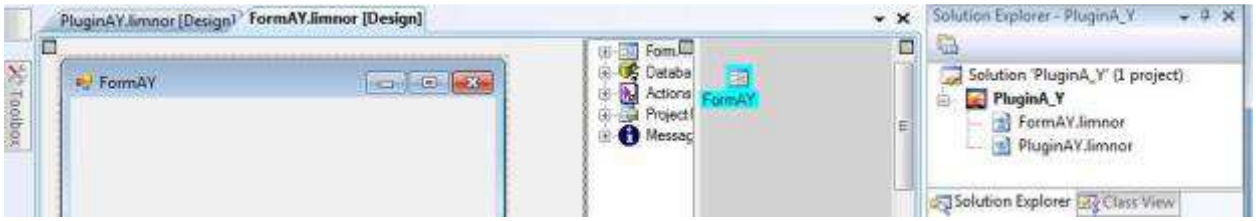


Name it PluginA_Y:

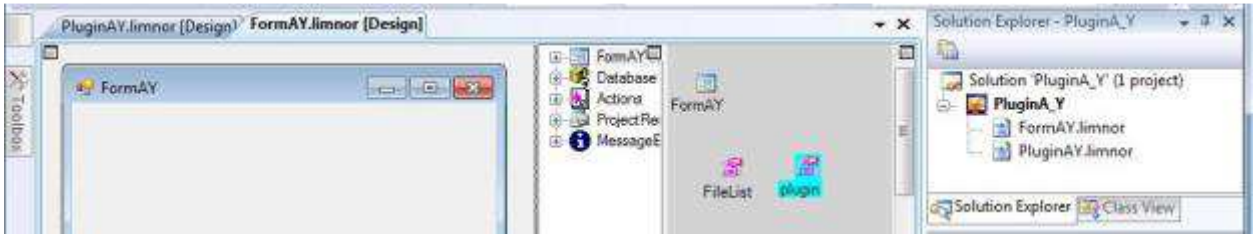


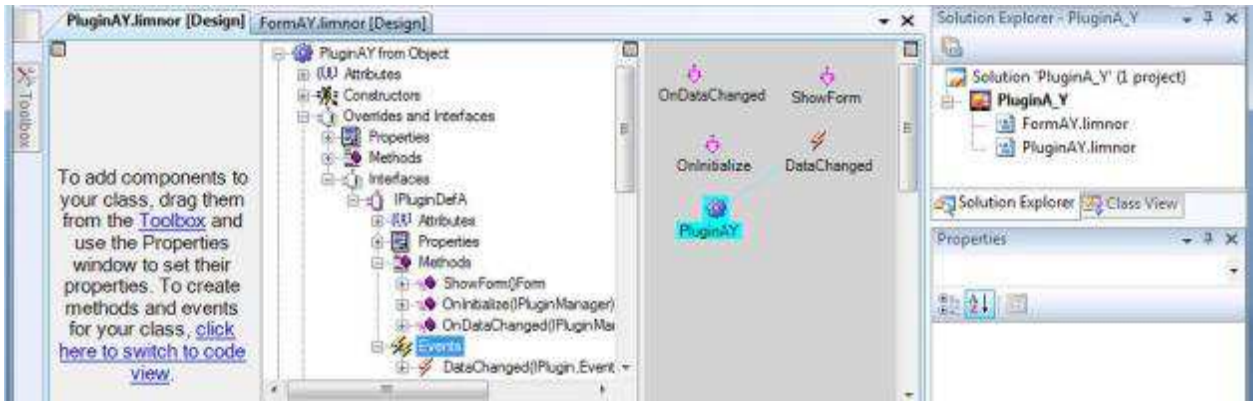
Implement Plug-in Definition

Like the first implementation of IPluginDefA, we also add a form and a class to the project:



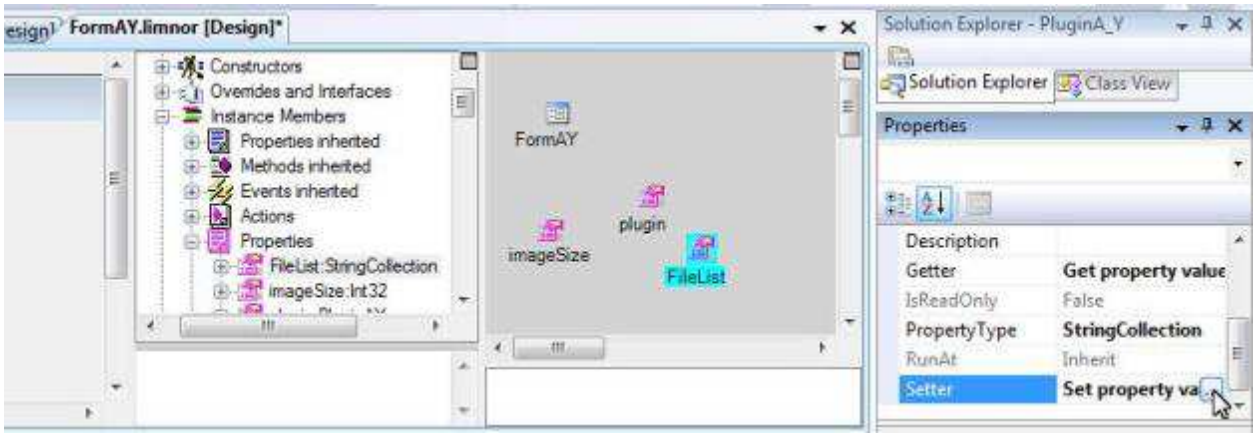
The implementation of the plug-in definition is very similar to the first implementation. The difference is in the way the Image file names are used. In this implementation, an image box control is used to display image for each image file.



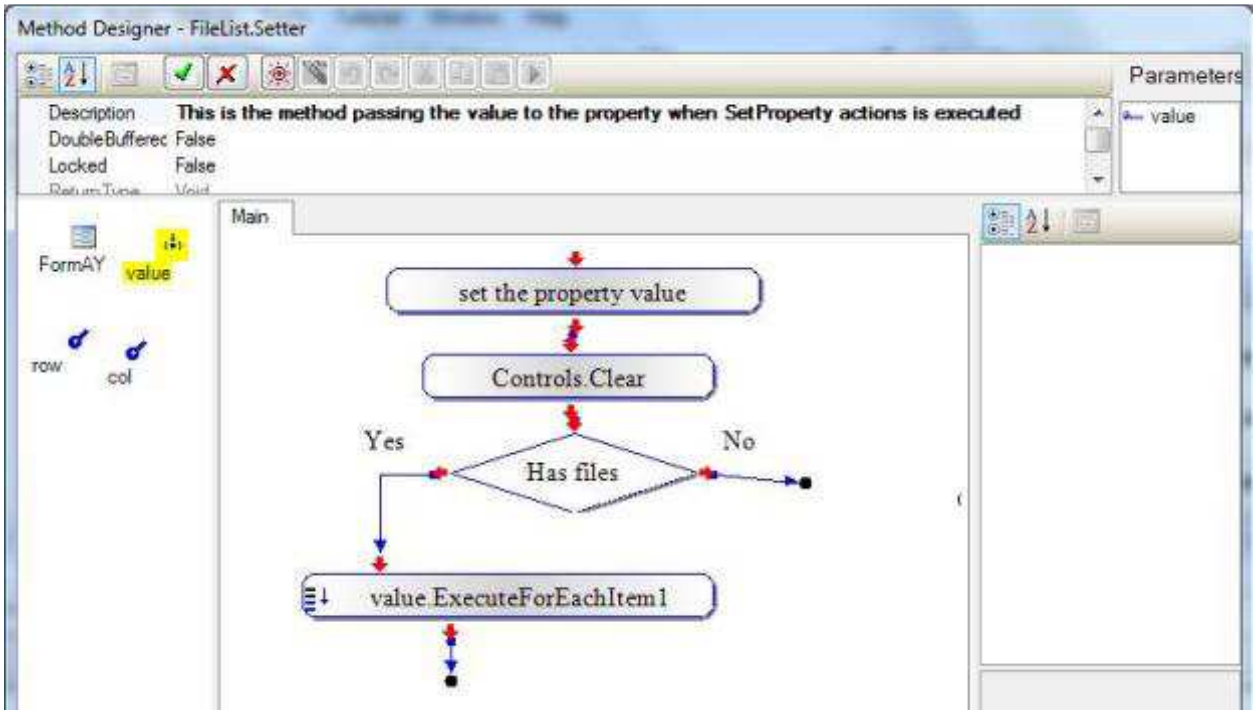


Implement Specific Features

A plug-in should have its specific features to make t worth efforts of development. For this plug-in, suppose that we want to display each image file in a picture box. We do it at the time of setting the "FileList" property of the form. Edit the Setter of "FileList" property:



We are not going to go into details of developing it because it has nothing to do with plug-in implementation. We just briefly describe the features:

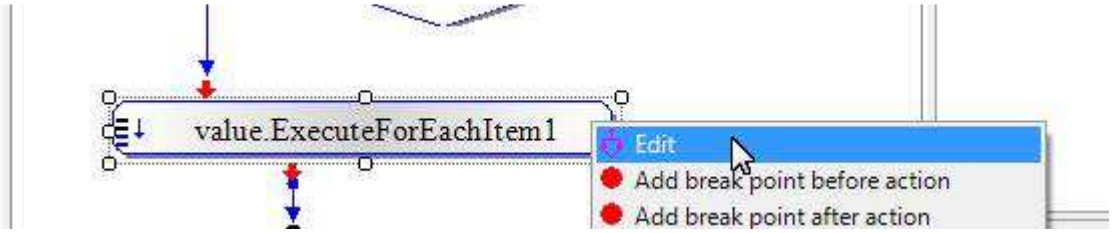


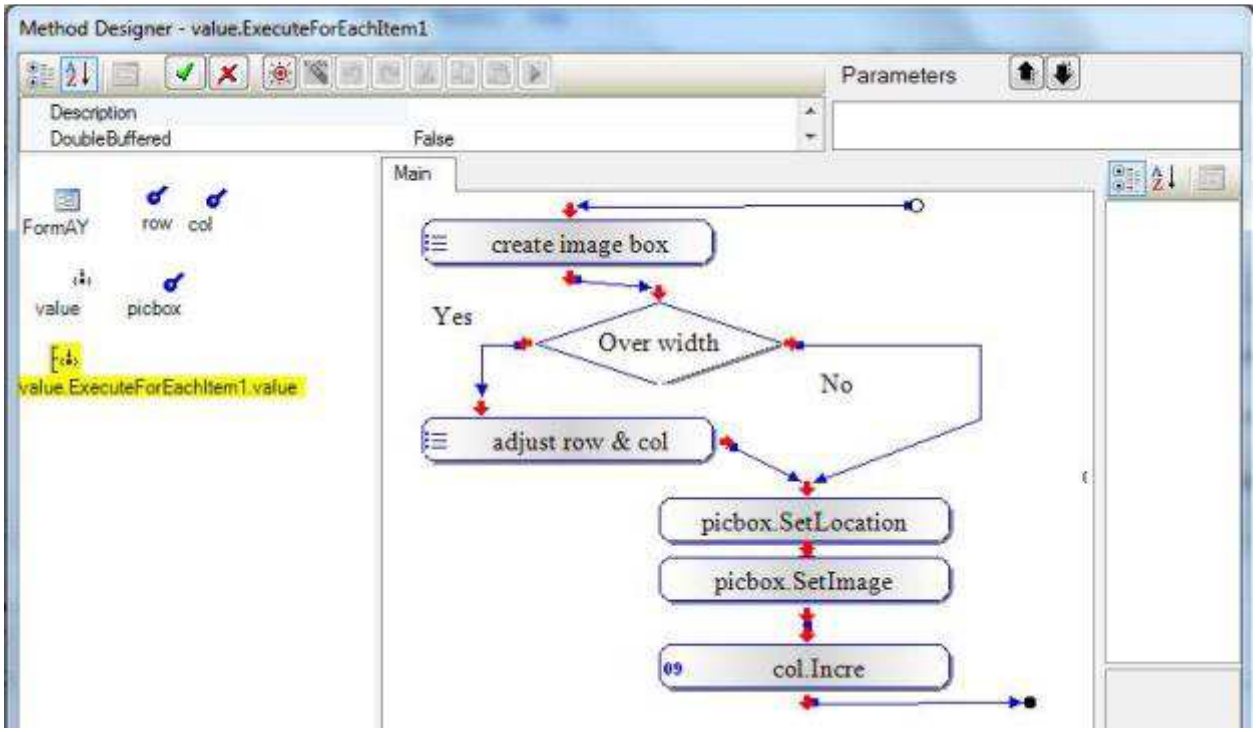
The first action is the default Setter action, which passes the new property value to the property storage.

The second action removes all controls from the form.

If the new property value is not null then the last action will go through each file name to process it.

You may open the last action to see how it processes each file name:

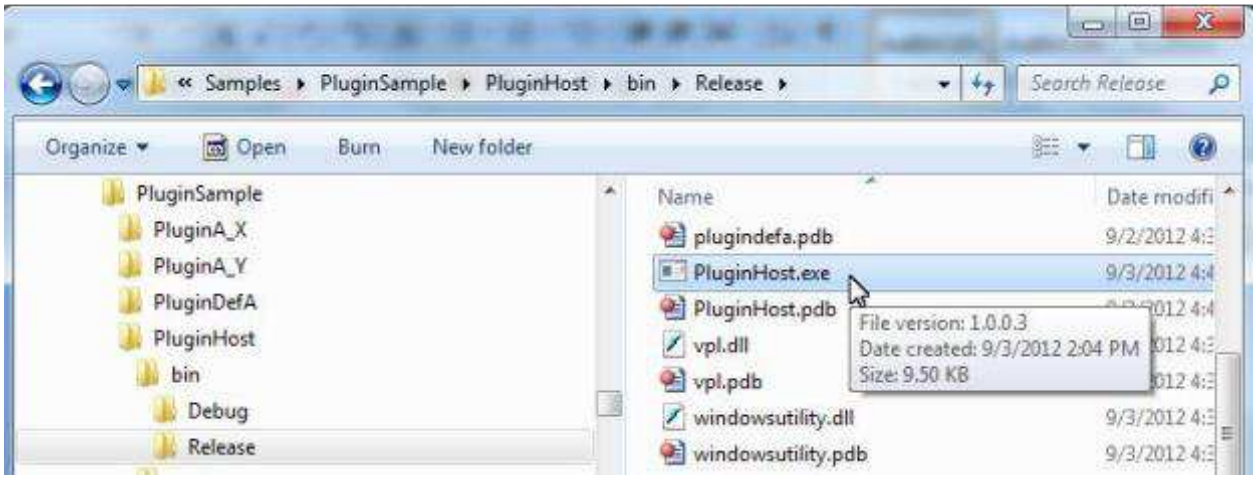




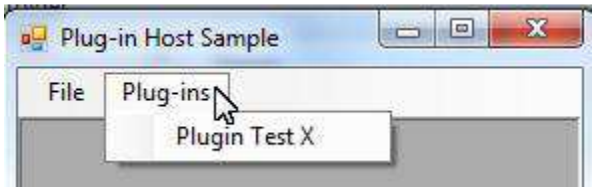
The first action list creates a picture box and adds it to the form. We arrange picture box from left to right and from top to bottom. Variables col and row represent column number and row number. They are used to calculate the picture box location.

Use the second plug-in by the host

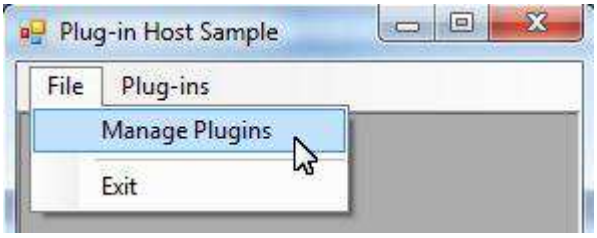
Let's start the plug-in host application again by double-clicking the exe file:



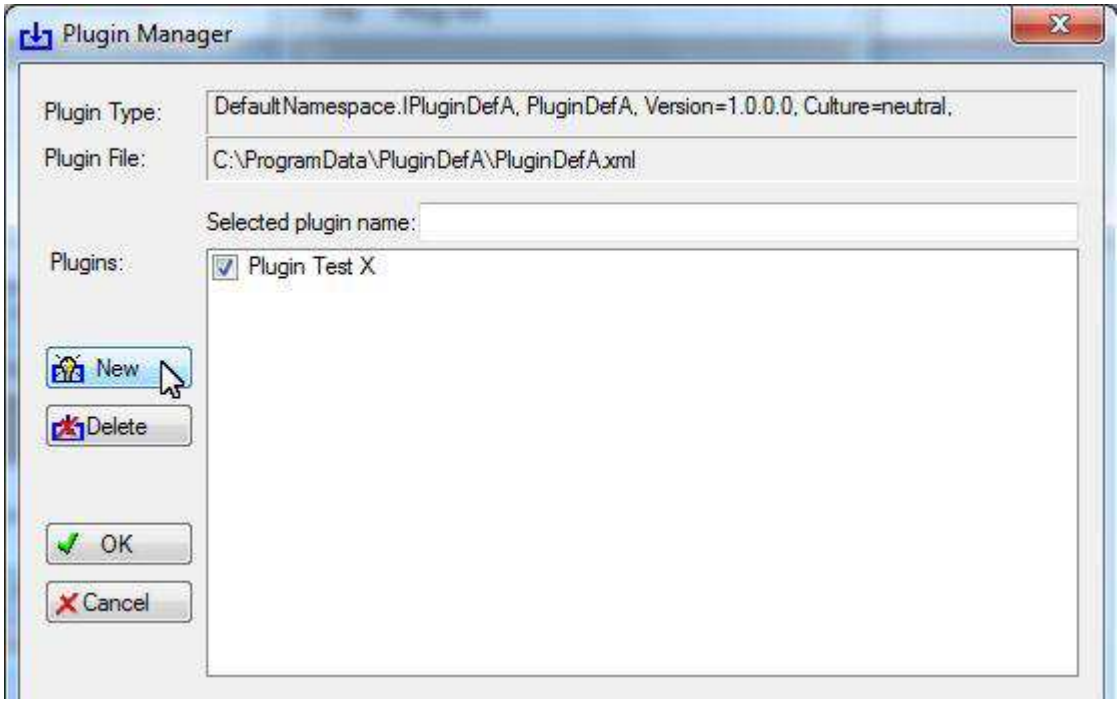
The MDI container form appears. Because we already loaded the first plug-in, we can see that there is a sub menu item under the plug-in main menu:



Open the plug-in manager to add more plug-ins:

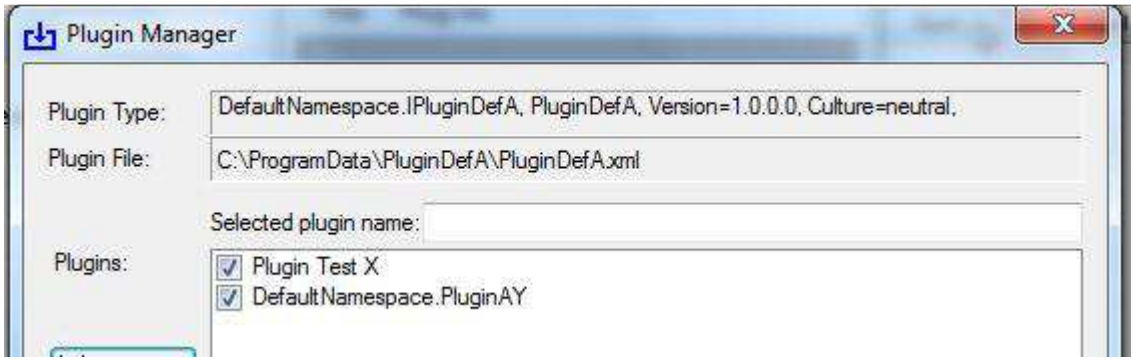


Click “New” button to add new plug-in:

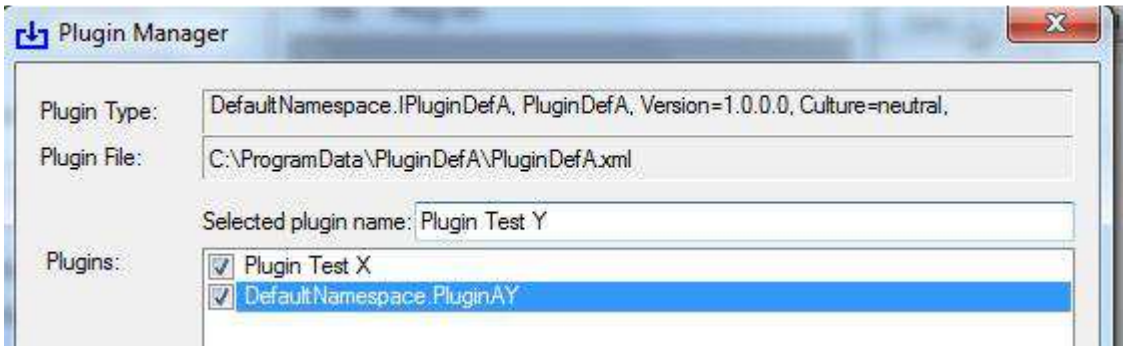


Select the DLL file for the second plug-in:

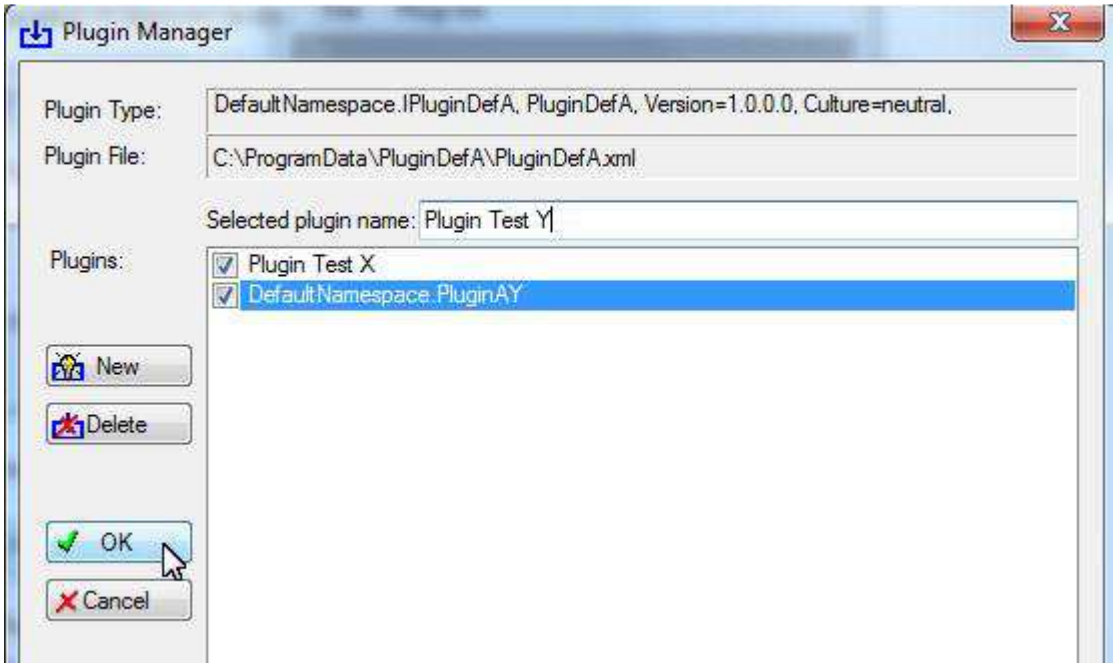
A new plug-in appears in the list:



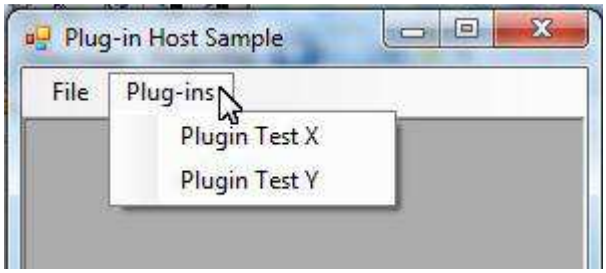
We may give it a name:



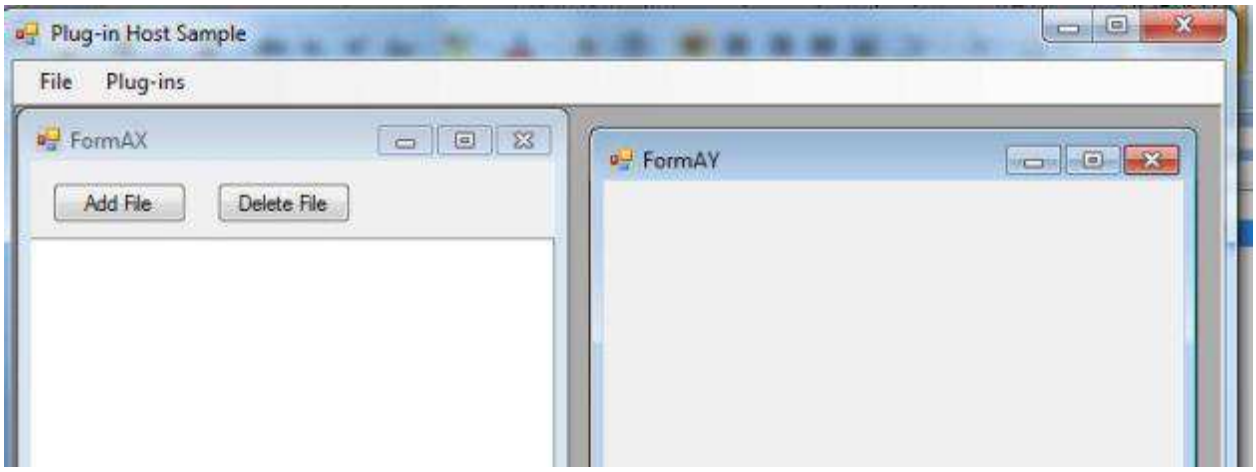
Click OK to finish managing the plug-ins:



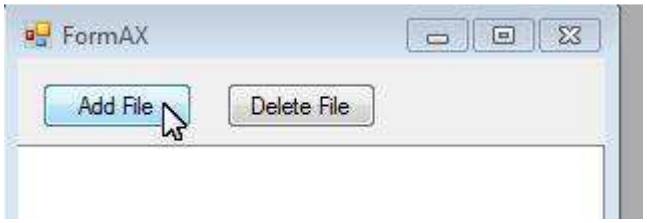
We can see that a new sub menu item appears under the plug-in main menu:



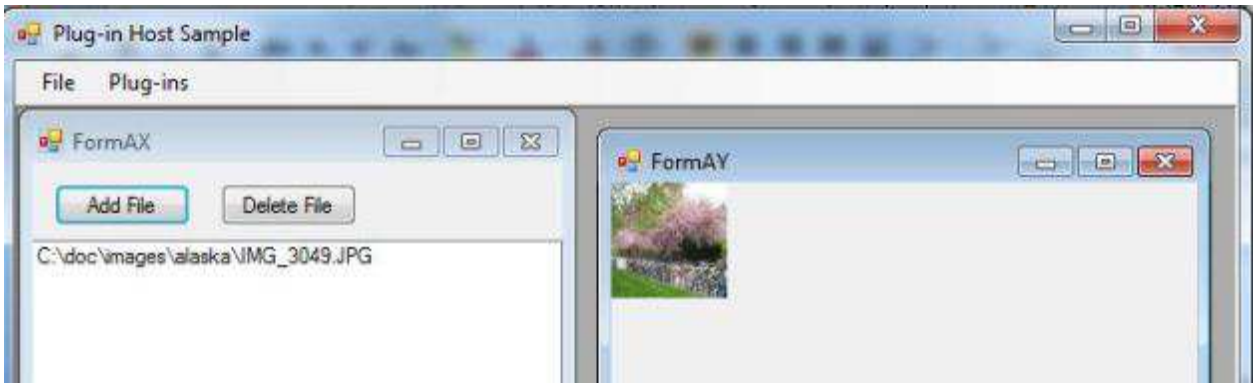
Select each menu item, a corresponding child form appears:



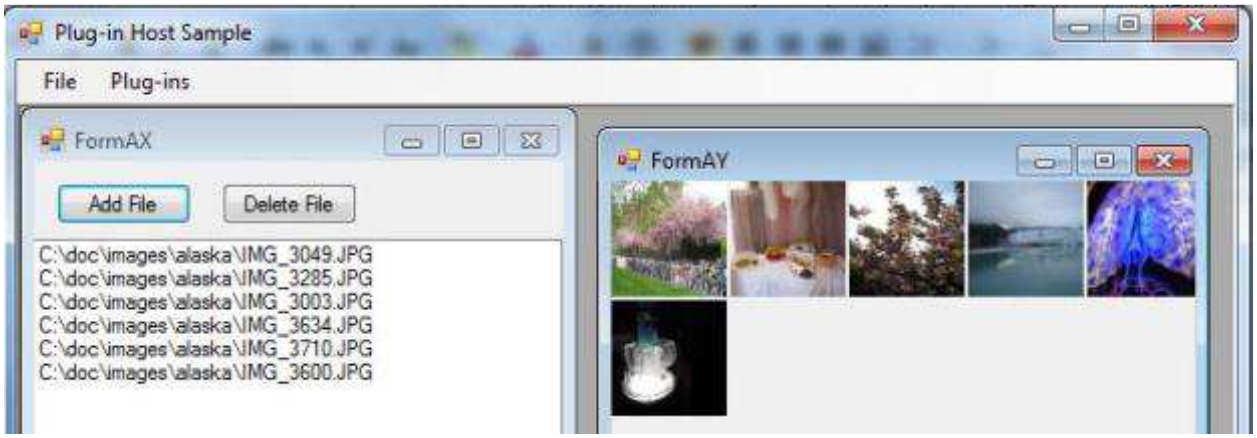
Click “Add File” button to add an image file:



We can see that a file name appears in the list. Note that the image is also displayed in the form of the second plug-in, demonstrating data-sharing between all plug-ins:

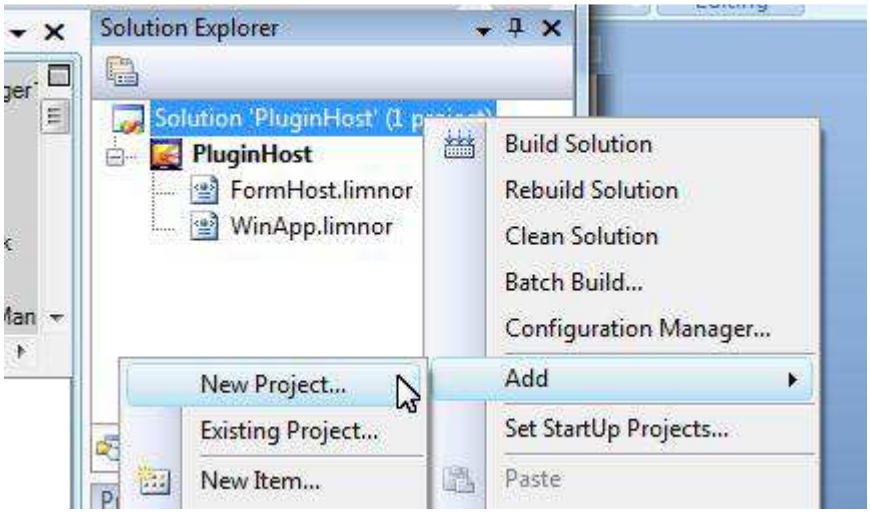


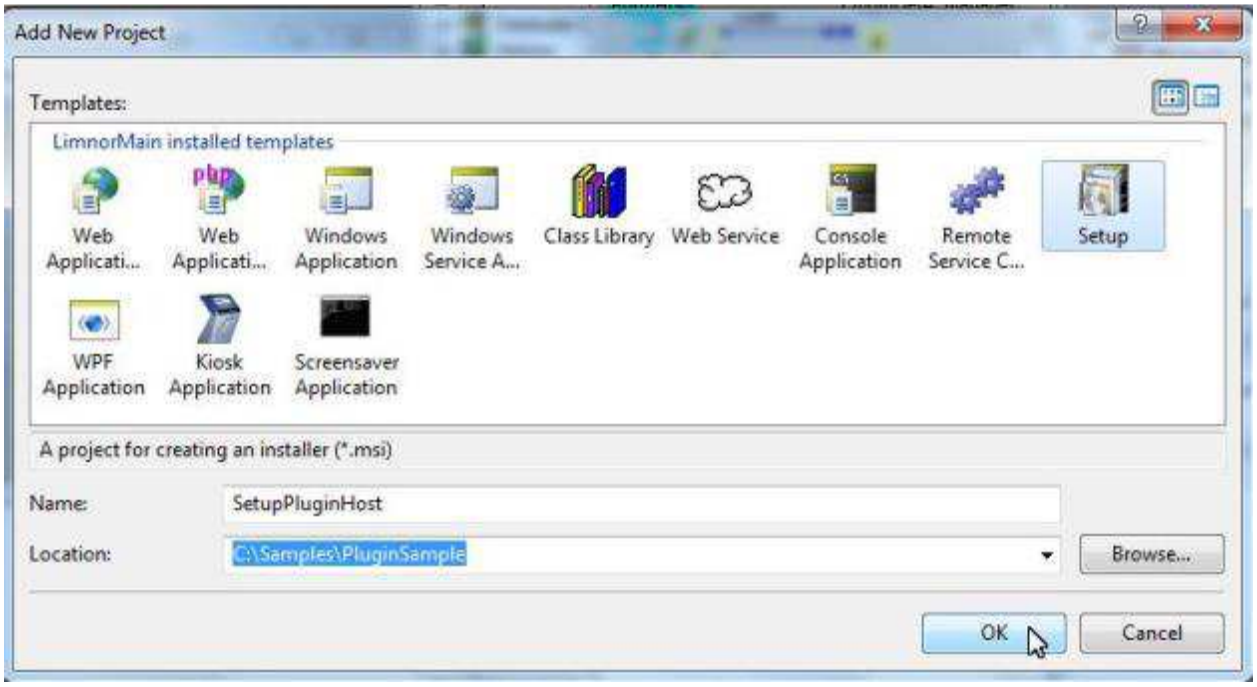
Adding more file names to show data-sharing:



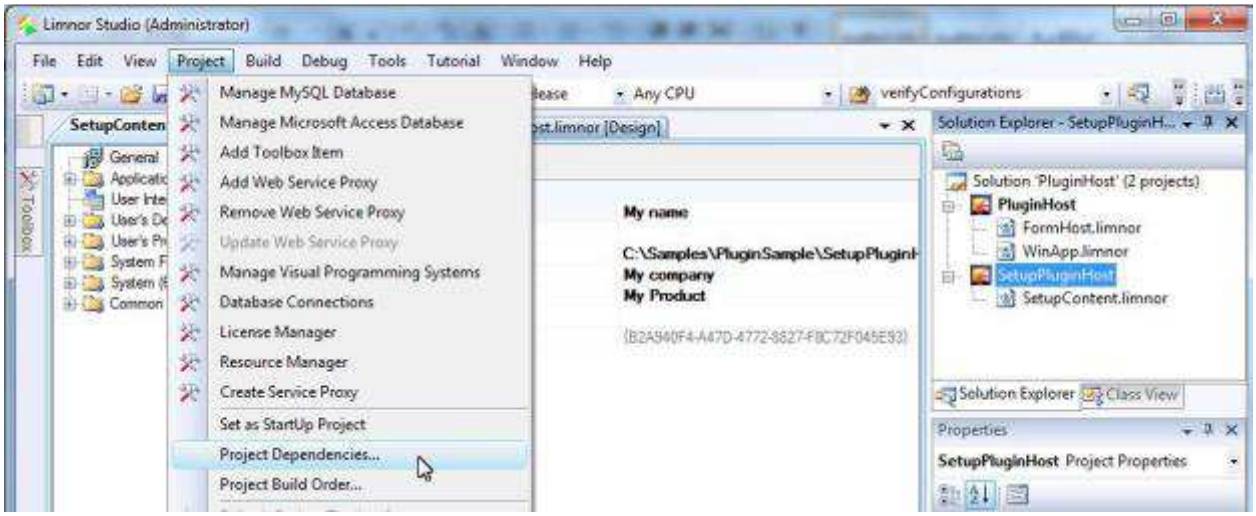
Plug-in Distribution

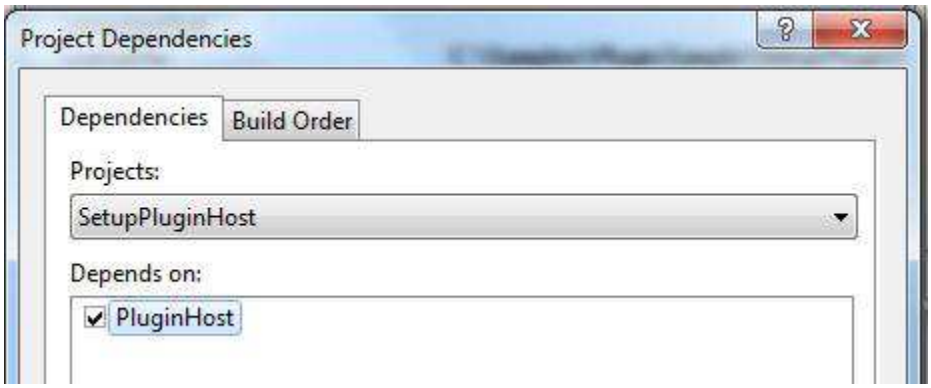
To create an installation file (*.msi) to distribute the plug-in host application, a setup project may be added to the solution:



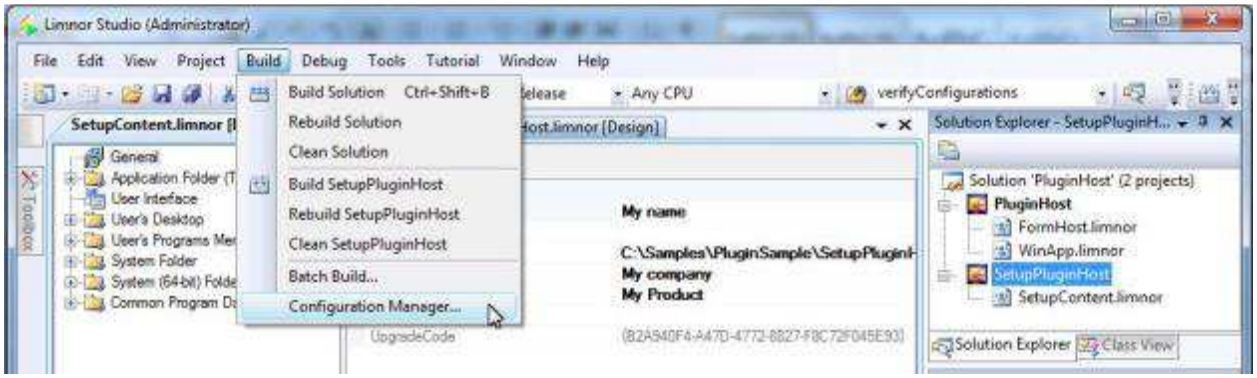


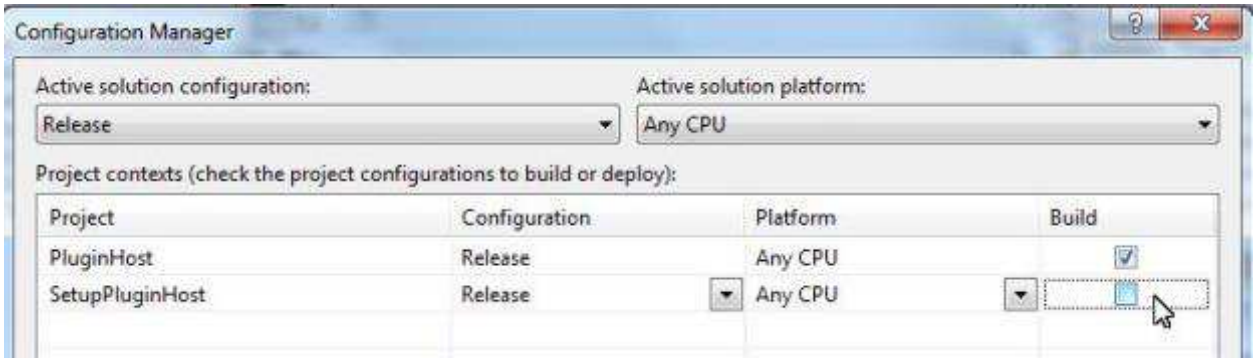
Set the dependency of the projects so that the setup project will be built the last:





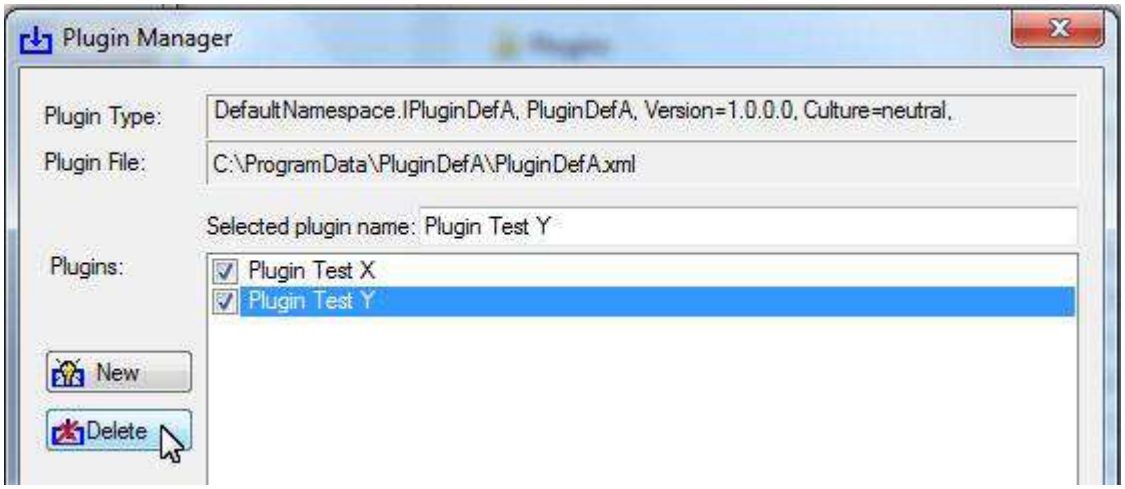
You may exclude the setup project from the build if you are not ready to make installation file.





After fully develop your application and test it, you may enable the build of the setup project to make installation file.

Note that during your testing of the host application, some plug-in objects may be loaded. For example, the testing we did previously added two plug-in objects. When you make installation, all used plug-ins will be included in the installation file and will be installed in customers' computers. If you used plug-in DLLs for testing purpose only then you need to remove such plug-ins so that they will not be distributed and installed in customers' computers:



Feedback

Please send your feedback to support@limnor.com