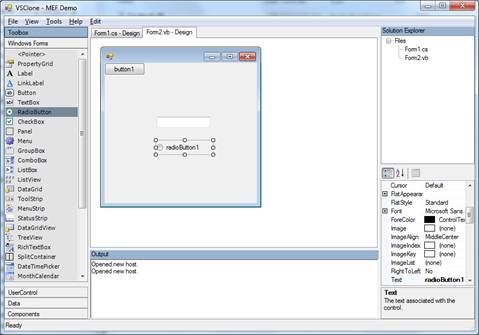
MEF Studio

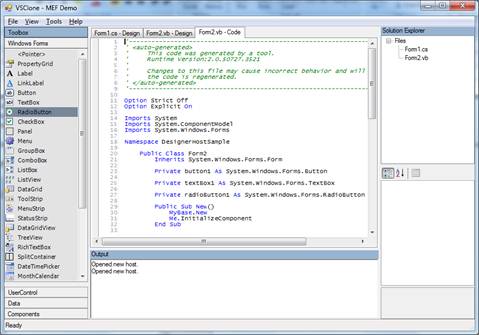
# Overview

MEF Studio is a designer hosting sample. It has an IDE that hosts various designers like Windows Forms, UserControl, etc. One can add controls on to these designers from the toolbox, set properties using the property grid, view generated code, etc. It provides design time experience for the designers and has extension points for extending the adding custom parts.

*Design View:*



*Code View:*



# How does MEF fit in?

MEF Studio is designed to be an extensible IDE, to demo how one could build such extensible applications. Almost every aspect of this sample is a MEF part. It has 3 different extension points:

1. Toolbox hosts a bunch of controls. These controls themselves are MEF parts and one can define more controls that can easily be picked up by the toolbox, thus making the toolbox easily extensible. App demos adding new controls to the toolbox.
2. By selecting ‘File | New’ option from the menu in the IDE, one can choose the designer to open. The options that appear in the file new dialog are extensible. Out of the box, the app shows items for basic Form, UserControl, Component and a custom Graph designer (these all have extension “.no” and have no code behind) and has item for C# Form (with extension “.cs” and has C# code behind). The app demos how one can extend the designers hosted in this IDE by providing extensions (i.e. MEF parts) for VB and XML Forms.
3. Each designer can support a set of commands. These show up in the menu on top of the IDE. C# Form for instance supports commands like “Cut”, “Copy” and “Paste”. Commands themselves are extensible. The application demos how one can create new commands that show up in the menu (example: “Debug | Run” option for C# Form).

# Diving deeper into the code

## Shell

Notice the MainShell itself is a part and it imports a bunch of toolwindows such as: Toolbox, PropertyGrid, Output window, Solution explorer. In addition it imports the FileNewDialog which shows the designers that can be opened. Finally, notice the import of an Action (“RefreshExtensions”). This is Action for calling Refresh() on the DirectoryCatalog (provided by the host i.e. Program.cs) to pick up new dlls that may have been dropped into the application folder (along side Shell.exe) – this action is fired when one clicks on ‘File | Refresh’ menu item in the IDE.

[Export]

public partial class MainShell : Form, IPartImportsSatisfiedNotification

{

[Import]

private IToolbox toolbox = null;

[Import]

private IOutputWindow outputWindow = null;

[Import]

private IPropertyGrid propertyGrid = null;

[Import]

private ISolutionExplorer solutionExplorer = null;

[Import]

private FileNewDialog fileNewDialog = null;

[Import("RefreshExtensions")]

private Action refresh = null;

...

}

## Toolbox

The Toolbox hosts the controls that can be added to the designers. It shows controls in separate categories by putting them in different tabs. By default there are tabs for “Windows Form”, “Data”, “Components” and “UserControl”. The controls that appear in the toolbox are MEF parts, and the Toolbox itself Imports an array of the controls. Each control has metadata that specifies which tab/category the control should appear in which is used by the toolbox to put the control in the appropriate tab. One can write more controls and drop the dll in the application folder which will then be picked up by the application and shown in the toolbox.

Interesting parts of the code are as follows:

*ToolboxUIManager.cs*

[Export]

public class ToolboxUIManager : IPartImportsSatisfiedNotification

{

[ImportMany(AllowRecomposition = true)]

private Export<Type, IToolboxItemMetadataView>[] toolboxItems = null;

...

public void OnImportsSatisfied()

{

PopulateData();

}

}

*ToolboxControls.cs (defines the controls that appear in the toolbox)*

public class ToolboxControls

{

private const string WinForms = "Windows Forms";

private const string Components = "Components";

private const string Data = "Data";

private const string UserControlTab = "UserControl";

[Export]

[ToolboxItemMetadata(Category=WinForms)]

private static Type t1 = typeof(System.Windows.Forms.PropertyGrid);

[Export]

[ToolboxItemMetadata(Category = WinForms)]

private static Type t2 = typeof(System.Windows.Forms.Label);

...

}

This app has a project ‘ToolboxControls’ under the Extensions folder that defines 3 more controls that can appear in the toolbox.

*MoreControls.cs*

public static class MoreControls

{

private const string Winforms = "Windows Forms";

[Export]

[ToolboxItemMetadata(Category=Winforms)]

private static Type t1 = typeof(System.Windows.Forms.BindingSource);

[Export]

[ToolboxItemMetadata(Category = Winforms)]

private static Type t2 = typeof(System.Windows.Forms.BindingNavigator);

[Export]

[ToolboxItemMetadata(Category = "My Custom Controls")]

private static Type t3 = typeof(System.Windows.Forms.FlowLayoutPanel);

}

To have these controls appear in the toolbox, follow these steps:

1. Open Shell.sln in VS
2. Build and run (i.e. F5)
3. Copy MoreControls.dll from bin\MEFStudio\Extensions folder and drop it in bin\MEFStudio(i.e. alongside Shell.exe)
4. Click ‘File | Refresh’ from MEF Studio shell. New controls should now appear in the toolbox!

## Designers

File New dialog is invoked by clicking on ‘File | New’ menu item from the MEF Studio shell. This dialog shows the designers that are available. This dialog imports all the available designers and shows them in the UI. The user can pick which one (s)he wants to open up. Here is the interesting piece of code:

*FileNewDialog.cs*

[Export]

public partial class FileNewDialog : Form, IPartImportsSatisfiedNotification

{

[ImportMany(AllowRecomposition=true)]

private Export<HostSurfaceFactory, IDesignerMetadataView>[] designerFactories = null;

...

}

Note that it imports an array of HostSurfaceFactory. It also has AllowRecomposition set to true so it is notified of new HostSurfaceFactories that may be available. HostSurfaceFactory is basically the one that produces the designer.

Here’s an example of C# Form factory (which inherits from HostSurfaceFactory) and produces C# Forms.

*CSharpForm.cs*

[Export(typeof(HostSurfaceFactory))]

[DesignerMetadata(Language = "C#", ItemType = "Form", FileExtension="cs")]

public class CSharpFormFactory : HostSurfaceFactory

{

public override HostSurface CreateNewCore(string name)

{

return new CSharpForm(this, name);

}

public override IEnumerable<Export<Action<HostSurface>, ICommandMetadataView>> GetCommands()

{

return Commands.Where(c => c.MetadataView.Language == "C#" && c.MetadataView.ItemType == "Form");

}

}

Notice that these also have metadata that specifies the language, item type and the file extension. These pieces of information are used by various other parts such as file new dialog, etc to show the correct UI.

The application also demos how to hook up new designers. It has a project ‘MoreDesigners’ under the Extensions folder that has designers for VB and XML. To have new designers show up in the File new dialog, follow these steps:

1. Open Shell.sln in VS
2. Build and run (i.e. F5)
3. Copy MoreDesigners.dll from bin\MEFStudio\Extensions folder and drop it in bin\MEFStudio (i.e. alongside Shell.exe)
4. Click ‘File | Refresh’ menu item from MEF Studio shell.
5. Click ‘File | New’ menu item from MEF Studio shell. The file new dialog will now show VB and XML designers!

## Commands

Each designer can support commands. These commands show up in the menu in the MEF Studio shell. Here is the interesting piece of code from HostSurfaceFactory.

*HostSurfaceFactory.cs*

public abstract class HostSurfaceFactory

{

[Import]

protected IPropertyGrid propertyGrid = null;

[ImportMany(AllowRecomposition=true)]

protected Export<Action<HostSurface>, ICommandMetadataView>[] Commands = null;

public HostSurface CreateNew(string name)

{

}

}

C# Form provides some commands like Cut, Copy and Paste. Each of these commands has metadata that specifies which menu it should appear in, in the UI. Each command is exported as a method delegated which is invoked when the corresponding menu item is clicked in the UI. Here is the commands for C# Form.

*CSharp/Commands.cs*

public static class Commands

{

[Export(typeof(Action<HostSurface>))]

[CommandMetadata(Language = "C#", ItemType = "Form", Category = "&Edit", Name = "&Cut")]

public static void Cut(HostSurface hostSurface)

{

IMenuCommandService ims = hostSurface.GetService(typeof(IMenuCommandService)) as IMenuCommandService;

if(ims!=null)

ims.GlobalInvoke(StandardCommands.Cut);

}

...

}

App also demos how to add new commands. It has project ‘MoreCommands’ under the Extensions folder that has new command such as ‘Debug | Run’ for C# Form!

*MoreCommands/Commands.cs*

public static class Commands

{

[Export(typeof(Action<HostSurface>))]

[CommandMetadata(Language = "C#", ItemType = "Form", Category = "&Debug", Name = "&Run")]

public static void Run(HostSurface hostSurface)

{

(hostSurface.Loader as CodeDomHostLoader).Run();

}

...

}

To have new commands show up in MEF Studio shell, do the following:

1. Open Shell.sln in VS
2. Build and run (i.e. F5)
3. Copy MoreCommands.dll from bin\MEFStudio\Extensions folder and drop it in bin\MEFStudio (i.e. alongside Shell.exe)
4. Click ‘File | Refresh’ menu item from MEF Studio shell.
5. When you are on the C# Form designer tab you should see the new ‘Debug | Run’ command, clicking on which should compile and run the Form!

Besides this there are other commands like ‘View | Code’ which will show the code in the appropriate language. (Note: some menu items like Help, etc don’t do anything, they are there as placeholders only ;)).

# The Entry Point

Much like other samples, the main window (shell) is a Part. This sample also shows how to invoke DirectoryCatalog.Refresh() from the shell (which is a MEF Part). To do so, the host exports a method delegate to call Refresh on the DirectoryCatalog which is imported by the Shell and hooked up to the ‘File | Refresh’ menu item. Here is the host code:

*Program.cs*

static class Program

{

/// <summary>

/// The main entry point for the application.

/// </summary>

[STAThread]

static void Main()

{

Application.EnableVisualStyles();

Application.SetCompatibleTextRenderingDefault(false);

Compose();

}

static void Compose()

{

DirectoryCatalog directoryCatalog = new DirectoryCatalog(".");

AssemblyCatalog assemblyCatalog = new AssemblyCatalog(Assembly.GetExecutingAssembly());

AggregateCatalog aggregateCatalog = new AggregateCatalog();

aggregateCatalog.Catalogs.Add(directoryCatalog);

aggregateCatalog.Catalogs.Add(assemblyCatalog);

CompositionContainer container = new CompositionContainer(aggregateCatalog);

RefreshCatalog refreshCatalog = new RefreshCatalog(directoryCatalog);

container.ComposeParts(refreshCatalog);

Application.Run(container.GetExportedObject<MainShell>());

}

}

public class RefreshCatalog

{

DirectoryCatalog directoryCatalog = null;

public RefreshCatalog(DirectoryCatalog dc)

{

this.directoryCatalog = dc;

}

[Export("RefreshExtensions", typeof(Action))]

public void RefreshExtensions()

{

this.directoryCatalog.Refresh();

}

}

We have a Compose method that defines DirectoryCatalog that looks for new dlls added to the application folder, and AssemblyCatalog for the executing assembly and an AggregateCatalog that combines the previous 2 catalogs. We also add the RefreshCatalog object to the container that has the DirectoryCatalog.Refresh() method delegate used by the MEF Studio shell. Assuming the composition succeeded then the App calls Show on the MainShell to get the application off and running.