Composite application reloaded

As I was starting a new project I was looking for a library that will help me write a composite application, i.e. an application with a main shell (window) and pluggable extensions (DLLs / modules) to be added to it dynamically; A library like Prism but hopefully much simpler.

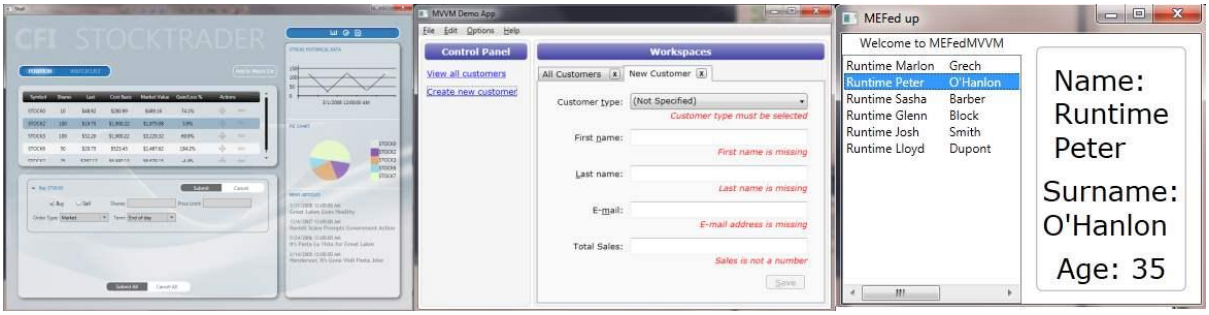
Many pieces of the puzzle could already be found elsewhere. The application had to have a clear separation between data and view, i.e. an MVVM approach. Services had to be linked automatically with something like MEF. Data validation should be automatic (thanks to `ValidationAttribute`(s)).

But improvement had to be made regarding disconnected messaging and view resolution. About the view resolution, i.e. the process of finding the appropriate view to represent a given business data object, I wanted to just tag the view with an attribute, such as `DataView(typeof(BusinessData1))` and let the library take care of the rest. This is where this library came out.

TODO Download link here

Contents

Samples



To test if my library was up to its goal I have ported three samples to it. In all case I was able to reduce the application size and maintain functionality.

- Josh Smith MVVM Demo. This is the best sample, as it is small and simple yet it covers almost all features of the library (after some modifications) and is a real composite application. I was able to get rid of the hand written validation code and use `ValidationAttribute` instead. And I tweaked the `MainWindow` and `App` class to make it a composite application, and use the `DataControl` in the `TabItem` to bind multiple controls to the same model with different view.
- Prism's main sample, the StockTraderApp project (huge sample).
  I removed the presenters (code which were used to bind views and view models, now replaced with call to `Composition.GetView()` and `DataControl`), the `EventAggregator` and custom prism events (replaced by `Notifications` static method). The most challenging and interesting part was to get rid of the `RegionManager` and replace it with the `IShellView` which explicitly expose the area of the shell that can be used and get rid of the `RegionManager`'s magic string approach.
- There is the MEFedMVVM library demo. The application is relatively simple but it makes extensive usage of design time support, and the design time experience is a joy to behold.

The unit tests illustrate how the most important features are working (i.e. Composition, Notification, ViewModelBase and Command).

About MEF and MVVM

Josh Smith talked about MVVM extensively on MSDN already. But, to summarize, MVVM is a View Model approach where all the logic and information is in the models. And by all I mean all, to the extent of including the selected element of a list, or the position of the caret, if need be.

In MVVM the view is nothing more than some declarative XAML (and possibly some UI specific code if need be, with no business code at all just pure UI logic). And because business data might not express all the information present in a view (such as selected area, incorrect value in a text box, etc...), business models might be wrapped in view models. View models are business models wrappers with a few extra, view-friendly, properties. This offers various advantages including

increased testability, better separation of concerns, possibility to have independent team for business data and UI.

MEF, or Manage Extensibility Framework, solves the problem of passing services and other shared objects around in a very neat way. It also enables to find the interface implementations easily.

Basically "consumer" objects declare what they need with the import attribute, like so:

```
[Import(typeof(ISomeInterface)]
public ISomeInterface MySome { get; set; }
```

Somewhere else in the code the exporting object are declared with export attribute.

```
[Export(typeof(ISomeInterface))]
public class SomeInterfaceImplentation : ISomeInterface
{ … }
```

Remark property and even method can be exported. Import can be single object (`Import`) or many (`ImportMany`). I strongly recommend that you read the MEF documentation.

To find the implementation for all imports needed by your objects there are 2 actions to be done:

1. At the start of the program, a "catalog" of types for MEF should be initialized from a list of types, assemblies and / or directories, which will be where MEF will look for locating exports. It's where you opt-in for the modules of interest. With this library you'll call the method `Composition.Register()`, as shown below.
2. You "compose" the objects that need to be resolved (i.e. which contains imports). With this library you'll use the method `Composition.Compose()`.

MEF contains various tags to control whether instances are shared or not, whether multiple implementations of an export is valid or not. Again this is covered in the documentation.

What does a composite application look like

A composite application is an application where there is a well-known top level UI element, typically a window for a desktop application or a Page for Silverlight. This top level UI element is called the "**Shell**".

The shell contains multiple areas where pluggable content will be hosted. Content that is not defined by the shell but in modules that are loaded dynamically (with MEF for example).

For example in the example below a shell is defined with two areas:

- A list box, showing a single list of documents.
- An `ItemsControl` (a `TabControl`) which can contains numerous items.

```
<Window
      x:Class="DemoApp.MainWindow"
```

```
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
      <Grid>
            <Grid.ColumnDefinitions>
                  <ColumnDefinition Width="Auto" />
                  <ColumnDefinition Width="*" />
            </Grid.ColumnDefinitions>
            <ListBox
                  Grid.Column="0"
                  Items="{Binding Documents}"
                  Header="Documents"
                  />
            <TabControl
                  Grid.Column="1"
                  IsSynchronizedWithCurrentItem="True"
                  ItemsSource="{Binding Workspaces}"
                  ItemContainerStyle="{StaticResource ClosableTabItemStyle}"
                  />
      </ Grid >
</Window>
```

Remark Don't worry too much about the shell when you create it, it is relatively easy to add or move areas later in the development, if need be. But it's harder to remove an area in use!

Once a shell has been defined, an interface to it should be exposed via a common library. It could be a "`IShellInterface`" (see the `IShellView` in the StockTraderApp for example) or its view model (see `MainViewModel` in DemoApp for example), or a combination of the two!

As an example here is the `IShellView` interface from the StockTraderApp sample:

```
public enum ShellRegion
{
      MainToolBar,
      Secondary,
      Action,
      Research,
      // this will set the currently selected item
      Main,
}
public interface IShellView
{
      void ShowShell(); // this will show the shell window
      void Show(ShellRegion region, object data);
      ObservableCollection<object> MainItems { get; }
}
```

Once a shell has been defined the composite application can be written. Four short steps are involved:

1. Define the DLL that are to be dynamically loaded
2. Create the shell (or skip and import it in 3.)
3. Export the shell and import the modules
4. Start the app / initialize the modules

For example, here is what StockTraderApp simplified App code could look like

```
public partial class App
{
	public App()
	{
		// 1. Opt-in for the DLL of interest (for import-export resolution)
		Composition.Register(
			typeof(Shell).Assembly
			, typeof(IShellView).Assembly
			, typeof(MarketModule).Assembly
			, typeof(PositionModule).Assembly
			, typeof(WatchModule).Assembly
			, typeof(NewsModule).Assembly
			);
		// 2. Create the shell
		Logger = new TraceLogger();
		Shell = new Shell();
	}

	[Export(typeof(IShellView))]
	public IShellView Shell { get; internal set; }

	[Export(typeof(ILoggerFacade))]
	public ILoggerFacade Logger { get; private set; }

	[ImportMany(typeof(IShellModule))]
	public IShellModule[] Modules { get; internal set; }

	public void Run()
	{
		// 3. export the shell, import the modules
		Composition.Compose(this);
		Shell.ShowShell();

		// 4. Start the modules, they would export the shell
		// and use it to appears on the UI
		foreach (var m in Modules)
			m.Init();
	}
}
```

Main Features

The library grew quite a lot from its humble starts. It consists out of two main parts. Feature which are critical to MVMM and composite development and optional features which were a useful additions.

The central class for most features of this library is the `Composition` class. It also contains two important properties, `Catalog` and `Container`, which are the used for MEF to resolve imports and exports. You need to fill the `Catalog` at the start of the application with `Composition.Register()`, for example:

```
static App() // init catalog in App's static constructor
{
	Composition.Register(
		typeof(MapPage).Assembly
		, typeof(TitleData).Assembly
		, typeof(SessionInfo).Assembly
	);
}
```

Later service imports and exports can be solved with MEF by calling `Composition.Compose()`.

Composition GetView

When an MVVM development pattern is followed, one writes business model and / or view models and views for these data models. Typically this view will only consist of "XAML code", and their `DataContext` property will be the business model. Often MVVM helper libraries will provide some ways of finding and loading these views.

In this library the views need to be tagged with a `DataViewAttribute` which specifies for which model type this view is for:

```
[DataView(typeof(CustomerViewModel))]
public partial class CustomerView : UserControl
{
	public CustomerView()
	{
		InitializeComponent();
	}
}
```

Then, from a data model, you can automatically load the appropriate view (and set its `DataContext`) with a call to `Composition.GetView()`, for example:

```
public void ShowPopup(object message, object title)
{
	var sDialog = new MsgBox();
	sDialog.Message = Composition.GetView(message);
	sDialog.Title = Composition.GetView(title);
	sDialog.Show();
}
```

Often models are not displayed as a result of some method call but simply because they are an item in an `ItemsControl` or the content of a `ContentControl`. In this case the `DataControl` control can be used in XAML to display the item by calling `Composition.GetView()`.

Remark It also brings a `DataTemplate` like functionality to Silverlight.

Because we use a View-Model approach, the same data model can be shown in multiple places at the same time hence `Composition.GetView()`, `DataViewAttribute` and the `DataControl` have an optional `location` parameter.

In the example below, the same `UserViewModel` instance (subclass of `WorkspaceViewModel`) is used to display both the `TabItem` header and content using different location parameter (*note: location is not set, i.e.it is null, in the second template*).

```
<Style x:Key="ClosableTabItemStyle" TargetType="TabItem" BasedOn="{StaticResource
{x:Type TabItem}}">
	<Setter Property="HeaderTemplate">
		<Setter.Value>
			<DataTemplate>
				<g:DataControl Data="{Binding}" Location="header"/>
			</DataTemplate>
		</Setter.Value>
	</Setter>
	<Setter Property="ContentTemplate">
		<Setter.Value>
			<DataTemplate>
				<g:DataControl Data="{Binding}"/>
			</DataTemplate>
		</Setter.Value>
	</Setter>
</Style>
```

Both views were defined like this (*note the first view is the default view, i.e. location is not set, it is null*):

```
[DataView(typeof(CustomerViewModel))]
public partial class CustomerView : System.Windows.Controls.UserControl
{
	public CustomerView()
	{
		InitializeComponent();
	}
}

[DataView(typeof(WorkspaceViewModel), "header")]
public partial class CustomerHeaderView : UserControl
{
	public CustomerHeaderView()
	{
```

```
            InitializeComponent();
        }
}
```

Validation and ViewModelBase

Inspired by Rob Eisenberg's talk, I created a `ViewModelBase` class which implement two important interfaces for WPF development: `INotifyPropertyChanged` and `IDataErrorInfo`.

The `INotifyPropertyChanged` implementation is strongly typed (refactor friendly):

```
public class Person : ViewModelBase
{
        public string Name
        {
                get { return mName; }
                set
                {
                        if (value == mName)
                                return;
                        mName = value;
                        OnPropertyChanged(() => Name); // See, no magic string!
                }
        }
        string mName;
}
```

The `IDataErrorInfo` interface allows the WPF bindings to validate the properties they are bound to (if `NotifyOnValidationError=true`). The implementation in `ViewModelBase` validates the properties using `ValidationAttribute`(s) on the properties themselves. For example:

```
public class Person : ViewModelBase
{
        [Required]
        public string Name { get; set; }

        [Required]
        public string LastName { get; set; }

        [OpenRangeValidation(0, null)]
        public int Age { get; set; }

        [PropertyValidation("Name")]
        [PropertyValidation("LastName")]
        [DelegateValidation("InitialsError")]
        public string Initials { get; set; }

        public string InitialsError()
```

```
        {
                if (Initials == null || Initials.Length != 2)
                        return "Initials is not a 2 letter string";
                return null;
        }
}
```

The example above also illustrates some of the new `ValidationAttribute` subclasses provided in this library, in the Galador.Applications.Validation namespace, i.e.

```
ConversionValidationAttribute
DelegateValidationAttribute
OpenRangeValidationAttribute
PropertyValidationAttribute
```

A control with an invalid binding will automatically be surrounded by a red border (default style), but the error feedback can be customized as shown in this XAML fragment below, which display the error message below the validated text:

```
<!-- FIRST NAME-->
<Label
        Grid.Row="2" Grid.Column="0"
        Content="First _name:"
        HorizontalAlignment="Right"
        Target="{Binding ElementName=firstNameTxt}"
        />
<TextBox
        x:Name="firstNameTxt"
        Grid.Row="2" Grid.Column="2"
        Text="{Binding Path=Customer.FirstName, ValidatesOnDataErrors=True, UpdateS
ourceTrigger=PropertyChanged, BindingGroupName=CustomerGroup}"
        Validation.ErrorTemplate="{x:Null}"
        />
<!-- Display the error string to the user -->
<ContentPresenter
        Grid.Row="3" Grid.Column="2"
        Content="{Binding ElementName=firstNameTxt, Path=(Validation.Errors).Curren
tItem}"
        />
```

Disconnected Messaging

In a composite application there is a need for components to send messages to each other without knowing each other. The `Notifications` class and its static methods are here to solve this problem.

First a common message type should be defined in a common library, objects can

- Subscribe to messages for this type (with the static `Subscribe()` and `Register()` methods).
- Publish messages (with `Publish()`).
- Unsubscribe from messages if they are no longer interested in them (with `Unsubscribe()`).

Remark The subscription thread is an optional parameter that can be either the original thread, the UI thread or a background thread.

To illustrate these functionalities here is a snippet of code from the Notifications' unit test class.

```
public void TestSubscribe()
{
	// subscribe to a message
	Notifications.Subscribe<NotificationsTests, MessageData>(null, StaticSubscr
ibed, ThreadOption.PublisherThread);

	// publish a message
	Notifications.Publish(new MessageData { });

	// unsubscribe to a message below
	Notifications.Unsubscribe<NotificationsTests, MessageData>(null, StaticSubs
cribed);
}

static void StaticSubscribed(NotificationsTests t, MessageData md)
{
	// message handling
}
```

Arguably the `Notifications.Subscribe()` syntax is a bit cumbersome. It's why an object can also subscribe to multiple message type in one swoop by calling `Notifications.Register(this)`, which will subscribe all its methods with one argument and tagged with `NotificationHandlerAttribute`, as in

```
public void TestRegister()
{
	// register to multiple message type (1 shown below)
	Notifications.Register(this, ThreadOption.PublisherThread);

	// publish a message
	Notifications.Publish(new MessageData { Info = "h" });
}

[NotificationHandler]
public void Listen1(MessageData md)
{
	// message handling
}
```

Commands

To avoid the need for code in the UI, yet handle code triggering controls such as a `Button` or a `MenuItem`, WPF (and Silverlight 4) came up with commands (`ICommand` to be exact). When a button is clicked the control action is triggered, and if it has a Command property it will call `Command.Execute(parameter)`, where the parameter is the `Control.CommandParameter` property.

ViewModels need to expose a `Command` property whose `Execute()` method will call one of their methods. For this purpose there is the `DelegateCommand`.

A delegate command can be created by passing a method to execute and an optional method to check if the method can be executed (which will enable / disable the command source, i.e. the button). For example:

```
var p = new Person();
var save = new DelegateCommand<Person>(p, aP => { aP.Save(); }, aP => aP.CanSave);
```

Remark The command will automatically detect `INotifyPropertyChanged` properties and register to the `PropertyChanged` event to update its `CanExecute()` status.

Remark Sometimes you need commands such as “DoAll” as in “CancelAll” or “BuyAll” hence the support of `ForeachCommand` class, which is an `ICommand` itself and can watch a list of `ICommand`, set its status to CanBeExecuted if all its command can be executed.

Other Features

A few other non-essential features found their way into this library.

There is the `Invoker`, which assist in running code on the GUI thread. It can also be used in both WPF and Silverlight.

```
public class Invoker
{
      public static void BeginInvoke(Delegate method, params object[] args)
      public static void DelayInvoke(TimeSpan delay, Delegate method, params obje
ct[] args)
}
```

There is design time support for the data views. Using the attached property `Composition.DesignerDataContext` on a data view sets its `DataContext` at design time:

```
<UserControl x:Class="MEFedMVVMDemo.Views.SelectedUser"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
             xmlns:g="http://schemas.galador.net/xaml/libraries"
             xmlns:models="clr-namespace:MEFedMVVMDemo.ViewModels"
```

```
            g:Composition.DesignerDataContext="models:SelectedUserViewModel"
            >
```

These view models (i.e. DataView's `DataContext`) can compose themselves (i.e. call `Composition.Compose(this)`) to import some other services.

Remark Having a design time `DataContext` makes the experience of writing a `DataTemplate` a whole lot better.

Remark These view models can be made aware that they are in design mode if they implement the `IDesignAware` interface.

Remark The services loaded by the models can be different in runtime and design time if they are exported with `ExportService` instead of `Export`, like so

```
[ExportService(ServiceContext.DesignTime, typeof(IUsersService))]
public class DesignTimeUsersService : IUsersService
```

There are multiple variations of the `Foreach` classes which can be used to observe an `IEnumerable` or an `ObservableCollection` and take whatever action is appropriate when something in the collection changes.

Summary

Hopefully this article and the samples it contains will have shown what a composite application architecture looks like and how this library makes it easy to solve the key problems most often met by a composite application:

- Resolving services dependencies using MEF.
- Finding DataView for DataModel with `DataControl` or `Composition.GetView()`.
- Implement common MVVM pattern: the `ICommand` (with `DelegateCommand` and `ForeachCommand`) and disconnected messaging (with `Notifications`).
- Implement data binding validation with `ValidationAttribute` in a subclass of `ViewModelBase`.

Compatibility

This library will work with the Client Profile for .NET4 and Silverlight 4.

If need to be ported to .NET3.5 there are two obstacles.

- MEF, which is on CodePlex.
- And the Validator class, used in the `ViewModelBase` to validate the properties from the `ValidationAttribute`(s), i.e. implement the `IDataErrorInfo` interface. Only two methods need to be reimplemented from the `Validator`.

References

MEF on Codeplex (it's also part of .NET4 & Silverlight 4)
http://mef.codeplex.com/

Prism, aka the composite application library
http://compositewpf.codeplex.com/

Josh Smith on MVVM
http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

Rob Eisenberg on MVVM
http://devlicio.us/blogs/rob_eisenberg/archive/2010/03/16/build-your-own-mvvm-framework-is-online.aspx

The MEFedMVVM library
http://mefedmvvm.codeplex.com