Dear Sirs, can you provide a full example with detailed explanations about the use of the Mvvm framework in windows forms? I downloaded version 14.1 and I'm trying to understand the OutlookInspiredApp, in particular:  
  
- use of the different services  
- use of the BindCommand() method in the BarButtonItem (for example)  
- use of the files under the folder "Modules" (I think that it's the equivalent to Views or User Controls in the "WPF language")

Sorry, but I attempt to find information in the docs and in too many cases I found the text "to be supplied"  
  
My main goal is to build an app using the ribbon control, and some "views" to differents use cases. The items in the ribbon would be enabled or disabled depending of the user's profile. Very simple in "pure" WPF.  
  
I already read the article in "2013/12/04/devexpress-mvvm-framework-introduction-to-poco-viewmodels.aspx" and all related ones, but they're oriented to WPF and I need a similar approach to WinForms

Another thing that make me feel more "quiet" was: in the new employee view, all the required fields are with the "validation error" beside them. This's a behavior that I experimented in WPF and I was not very convinced of show it in the final application (but now I am!!), that's because the POCOs support IDataErrorInfo or some similar interface? Thus please explain also

**- use of the IDataErrorInfo ( DataAnnotation attributes)**

I appreciate and thanks your time and patience  
Best regards  
  
Enrique Thedy

# IDATAERRORINFO

>>that's because the POCOs support IDataErrorInfo or some similar interface?  
Yes, our MVVM library provides an easy way to add the IDataErrorInfo support into POCO objects. In our Outlook Inspired App demo, we have used standard data-annotation attributes (from the [System.ComponentModel.DataAnnotations](http://msdn.microsoft.com/en-us/library/system.componentmodel.dataannotations.aspx) namespace) to implement this in a declarative way:

[C#]Open in popup window

[Required, EmailAddress]

public string Email { get; set; }

The following code will add the IDataErrorInfo implementation into our classes based on data-annotation attributes:

[C#]Open in popup window

string IDataErrorInfo.Error { get { return null; } }

string IDataErrorInfo.this[string columnName] {

get { return IDataErrorInfoHelper.GetErrorText(this, columnName); }

}

You can find a detailed explanation here:

<https://community.devexpress.com/blogs/wpf/archive/2014/03/31/devexpress-mvvm-framework-using-dataannotation-attributes-and-devexpress-fluent-api.aspx>

<https://community.devexpress.com/blogs/wpf/archive/2014/03/18/devexpress-mvvm-framework-data-validation-implementing-idataerrorinfo.aspx>

# COMMAND

"Command" is a pattern that allows encapsulating a method call into a separate object stored in View Model and attach this object to any UI elements (such as buttons, bar-items or tiles).

The "Command" pattern implies that there are two methods associated with Command's Execute and CanExecute logic, which are invoked when a UI Element attached to this command is activated by a user (for example, a button is clicked). On the other hand, the CanExecuteChanged event is raised when the CanExecute logic is changed at the ViewModel level.

There are different implementations of the "Command" pattern. In our MVVM library, we have used the implementation called "Delegate Command".

Delegate command is an implementation of the System.Windows.Input.ICommand interface, and so it can be used to create commands in ViewModel.

The Delegate command implementation is described in detail in the following blog article: https://community.devexpress.com/blogs/wpf/archive/2013/08/29/getting-started-with-devexpress-mvvm-framework-commands-and-view-models.aspx.

Below is a sample that describes how easily the Command is implemented with our POCO ViewModel implementation in Office Inspired App.

Here are our View Model's logic related to showing a Map:

[C#]Open in popup window

[Command(UseCommandManager = false)]

public void ShowMap() {

ShowMapCore(SelectedEntity); // do something

}

public bool CanShowMap() {

return CanShowMapCore(SelectedEntity);

}

 The Command attribute means that after a View Model instance is created, the ShowMapCommand property of the DelegateCommand type will be automatically generated.

Since our CanExecute logic depends on a selected Entity value, we have added the following code into the OnSelectedEntityChanged method:

[C#]Open in popup window

protected override void OnSelectedEntityChanged() {

base.OnSelectedEntityChanged();

this.RaiseCanExecuteChanged(x => x.ShowMap());

}

 This code fires the CanExecuteChanged event for the DelegateCommand instance when the selected entity is changed.

The part related to the CanExecute logic can be empty - it just means that this command is always available for execution.

Starting with v 14.1, most of our WinForms controls (buttons, bar items, tiles, etc.) provide a set of BindCommand methods which allows attaching the Command instance to UI elements and automatically track the CanExecute logic changing. We will use these methods within the View module:

[C#]Open in popup window

biMap.BindCommand(() => ViewModel.ShowMap(), ViewModel);

The BindCommand method of our controls typically has the following overloads:

[C#]Open in popup window

void BindCommand(object command, Func<object> queryCommandParameter = null);

void BindCommand<T>(Expression<Action<T>> commandSelector, object source, Func<T> queryCommandParameter = null);

void BindCommand(Expression<Action> commandSelector, object source, Func<object> queryCommandParameter = null);

Thus, the following lambda-expression is used as a "command selector expression" to find a particular property in ViewModel and attach its value to the UI element (BarButtonItem in our case):

[C#]Open in popup window

() => ViewModel.ShowMap()

 The BindCommand method uses "duck-typing" to attach "something, that looks like a Command, because it has Execute and CanExecute method"  to the visual state and behavior of the target UI element. For example, all the buttons attached to the command instance will be automatically disabled when the CanExecute logic of the source command is changed. You can see this behavior in our new Outlook Inspired App while playing with different commands like New, Edit, Delete in Ribbon, in the context menu, in sub-menus or separate buttons (P.S. How many ways of creating a new Employee can you find in this demo?).

P.S. Please feel free to contact me if you need any additional assistance in this regard.

I will be happy to discuss any aspects of commands usage (or other practices related to MVVM/MVPVM in Windows Forms).

Dmitry Garavsky

DEVEXPRESS R&D

email dmitry.garavsky@devexpress.com

# SERVICES

"Service" is a kind of the IOC pattern that allows removing any references between the View Model and View layers.

In code, the service is an interface used within the ViewModel code without any assumption of "When" and "How" this interface is implemented.

The simplest scenario that Outlook Inspired App demonstrates in this regard is the use of IMessageBoxService:

[C#]Open in popup window

catch(DbException e) { MessageBoxService.Show(e.ErrorMessage, e.ErrorCaption, MessageBoxButton.OK, MessageBoxImage.Error); }

All the services are available via service-containers which exist at View Model's level and can be directly retrieved via the ViewModelBase.GetService<TService>() method (this method iterates through View Model's hierarchy and finds an appropriate service implementation):

[C#]Open in popup window

var workspaceService = GetService<Services.IWorkspaceService>();

// do something

 Our POCO-implementation for View Models provides a simplified way of using services within View Models without directly retrieving service instances from service-containers:

[C#]Open in popup window

[DevExpress.Mvvm.DataAnnotations.ServiceProperty]

protected virtual IDocumentManagerService DocumentManagerService {

get { throw new NotImplementedException(); }

}

 In this case, a specific implementation of this virtual property will be automatically generated while instantiating View Model.

The service implementation can be registered in service containers at runtime when View Model is created:

[C#]Open in popup window

this.ServiceContainer.RegisterService(new Services.WaitingService());

There are global and local services. The first kind of services are common services available from any place of an application. They are registered in the static (singleton) ServiceContainer:

[C#]Open in popup window

DevExpress.Mvvm.ServiceContainer.Default.RegisterService(new MessageBoxService());

 The other group of services only makes sense in the context of a specific module. Thus, these services are registered within specific View Model instances:

[C#]Open in popup window

serviceContainer.RegisterService("Custom Filter", new FilterDialogDocumentManagerService(ModuleType.ProductsCustomFilter));

 It is possible to override the parent's service implementation at any level of View Model's hierarchy via providing a certain service implementation at this level:

[C#]Open in popup window

serviceContainer.RegisterService(new NotImplementedDetailFormDocumentManagerService(ModuleType.CustomerEditView));

Resume:

The main goals of service usage are separate small and orthogonal interfaces, whose implementation can be created and tested independently, accessed in a centralized way and even replaced according to particular environment requirements. All this allows providing "clean" and "maintainable"  code for an entire application.

P.S. The services' main concepts and examples of usage in WPF/SL are described in the following blog-article: https://community.devexpress.com/blogs/wpf/archive/2013/09/30/devexpress-mvvm-framework-introduction-to-services-dxmessageboxservice-and-dialogservice.aspx

P.P.S. Please feel free to contact me if you need any additional assistance in this regard.

I will be happy to discuss any aspects of services usage (or other practices related to MVVM/MVPVM in Windows Forms).

Dmitry Garavsky

DEVEXPRESS R&D

email dmitry.garavsky@devexpress.com

# MODULES – USER CONTROLS – VIEWS

The Modules folder of our new Outlook Inspired App contains User Controls (equivalent to Views in MVVM pattern terminology).

Here are some simple concepts that we've using while creating our new Demos:

1) All the content of each of this modules is created at design-time according to our design requirements. So, it looks the same at run time and design-time and it s easy to customize the appearance of any elements using all the capabilities provided by the Visual Studio designer. We have used the standard design-time data source (System.Windows.Forms.BindingSource component) to customize the bindings of our list or property editors (please refer to the corresponding XtraEditors, XtraGrid and XtraDataLayout control's help articles).

 2) Inside the code behind of all of these modules, there are some simple code sections which accomplish run-time behavior of module elements:

 - a section related to command bindings (I've already described this mechanism in my previous post).

 - a section related to list or property editors binding  
 - a section related to reaction to the View Model's events

 - a section related to propagation of UI changes from the View's controls to the View Model (for example synchronization of XtraGrid focused row and the ViewModel.SelectedEntity property )

The code-implementation for each of these sections is quite trivial and based on our control's embedded capabilities. In any case, please feel free to contact us with any question related to any line of code from each of these sections.

 3) And, finally, there are some specific parts related to the MVVM infrastructure. Each module is derived from the BaseModule class that allows you to automatically create the View Model instance related to this View. The BaseModule provides some general methods to retrieve the module's own View Model and parent module View Model instances in a generic way and also contains some additional code related to resource cleanup. You can review this implementation in our Demo's sources. It is also quite trivial.

 In this concrete application, all the modules are placed onto the MainForm as child modules according to the selected item in the MainForm's navigation control. There are three types of navigation controls is used in this demo - OfficeNavigationBar, NavBarControl and in-Ribbon navigation menu. All of these controls can be discussed separately in the context of separate tickets if required. We have used [Ribbon Merging](https://documentation.devexpress.com/#WindowsForms/CustomDocument3451)  concepts to represent all the UIs related to module actions and interactivity. When changing the SelectedModuleType property in the MainViewModel via one of the mentioned navigation controls, the MainViewModel generates the ModuleAdded and the ModuleRemoved events. Within these event's handlers, the old module is removed and it's ribbon items are unmerged from the main Ribbon. The new module is determined via the ModuleLocator service and placed into the view (the module Ribbon's items are merged into main ribbon).

The module locator service provides a centralized  way to access any module by it's identifier. It also provides caching for some modules. The MainViewModel class contains all the required code related to this task. It also manages the animations related to module changing and layout persistence in the context of a concrete module. You can review all of these behavior aspects in our Demo and contact us in the case of any difficulties.