**5**

**Implementing the MVVM Pattern**

The Model-View-ViewModel (MVVM) pattern helps you to cleanly separate the business and presentation logic of your application from its user interface (UI). Maintaining a clean separation between application logic and UI helps to address numerous development and design issues and can make your application much easier to test, maintain, and evolve. It can also greatly improve code re-use opportunities and allows developers and UI designers to more easily collaborate when developing their respective parts of the application.

Using the MVVM pattern, the UI of the application and the underlying presentation and business logic is separated into three separate classes: the view, which encapsulates the UI and UI logic; the view model, which encapsulates presentation logic and state; and the model, which encapsulates the application's business logic and data.

Crystal includes samples and reference implementations that show how to implement the MVVM pattern in a Windows Presentation Foundation (WPF) application. The Crystal Library also provides features that can help you implement the pattern in your own applications. These features embody the most common practices for implementing the MVVM pattern and are designed to support testability and to work well with Expression Blend and Visual Studio.

# Class Responsibility and Characteristics

The MVVM pattern is a close variant of the Presentation Model pattern, optimized to leverage some of the core capabilities of WPF, such as data binding, data templates, commands, and behaviors.

In the MVVM pattern, the view encapsulates the UI and any UI logic, the view model encapsulates presentation logic and state, and the model encapsulates business logic and data. The view interacts with the view model through data binding, commands, and change notification events. The view model queries, observes, and coordinates updates to the model, converting, validating, and aggregating data as necessary for display in the view.

The following illustration shows the three MVVM classes and their interaction.



**Figure 5.1 The MVVM classes and their interactions**

Like with all separated presentation patterns, the key to using the MVVM pattern effectively lies in understanding the appropriate way to factor your application's code into the correct classes, and in understanding the ways in which these classes interact in various scenarios. The following sections describe the responsibilities and characteristics of each of the classes in the MVVM pattern.

# The View Class

The view's responsibility is to define the structure and appearance of what the user sees on the screen. Ideally, the code-behind of a view contains only a constructor that calls the InitializeComponent method. In some cases, the code-behind may contain UI logic code that implements visual behavior that is difficult or inefficient to express in Extensible Application Markup Language (XAML), such as complex animations, or when the code needs to directly manipulate visual elements that are part of the view. You should not put any logic code in the view that you need to unit test. Typically, logic code in the view's code-behind will be tested via a UI automation testing approach.

In WPF, data binding expressions in the view are evaluated against its data context. In MVVM, the view's data context is set to the view model. The view model implements properties and commands to which the view can bind and notifies the view of any changes in state through change notification events. There is typically a one-to-one relationship between a view and its view model.

Typically, views are Control-derived or UserControl-derived classes. However, in some cases, the view may be represented by a data template, which specifies the UI elements to be used to visually represent an object when it is displayed. Using data templates, a visual designer can easily define how a view model will be rendered or can modify its default visual representation without changing the underlying object itself or the behavior of the control that is used to display it.

Data templates can be thought of as views that do not have any code-behind. They are designed to bind to a specific view model type whenever one is required to be displayed in the UI. At run time, the view, as defined by the data template, will be automatically instantiated and its data context set to the corresponding view model.

In WPF, you can associate a data template with a view model type at the application level. WPF will then automatically apply the data template to any view model objects of the specified type whenever they are displayed in the UI. This is known as implicit data templating. The data template can be defined in-line with the control that uses it or in a resource dictionary outside the parent view and declaratively merged into the view's resource dictionary.

To summarize, the view has the following key characteristics:

* The view is a visual element, such as a window, page, user control, or data template. The view defines the controls contained in the view and their visual layout and styling.
* The view references the view model through its **DataContext** property. The controls in the view are data bound to the properties and commands exposed by the view model.
* The view may customize the data binding behavior between the view and the view model. For example, the view may use value converters to format the data to be displayed in the UI, or it may use validation rules to provide additional input data validation to the user.
* The view defines and handles UI visual behavior, such as animations or transitions that may be triggered from a state change in the view model or via the user's interaction with the UI.
* The view's code-behind may define UI logic to implement visual behavior that is difficult to express in XAML or that requires direct references to the specific UI controls defined in the view.

# The ViewModel Class

The view model in the MVVM pattern encapsulates the presentation logic and data for the view. It has no direct reference to the view or any knowledge about the view's specific implementation or type. The view model implements properties and commands to which the view can data bind and notifies the view of any state changes through change notification events. The properties and commands that the view model provides define the functionality to be offered by the UI, but the view determines how that functionality is to be rendered.

The view model is responsible for coordinating the view's interaction with any model classes that are required. Typically, there is a one-to many-relationship between the view model and the model classes. The view model may choose to expose model classes directly to the view so that controls in the view can data bind directly to them. In this case, the model classes will need to be designed to support data binding and the relevant change notification events. For more information about this scenario, see the section, Data Binding, later in this topic.

The view model may convert or manipulate model data so that it can be easily consumed by the view. The view model may define additional properties to specifically support the view; these properties would not normally be part of (or cannot be added to) the model. For example, the view model may combine the value of two fields to make it easier for the view to present, or it may calculate the number of characters remaining for input for fields with a maximum length. The view model may also implement data validation logic to ensure data consistency.

The view model may also define logical states the view can use to provide visual changes in the UI. The view may define layout or styling changes that reflect the state of the view model. For example, the view model may define a state that indicates that data is being submitted asynchronously to a web service. The view can display an animation during this state to provide visual feedback to the user.

Typically, the view model will define commands or actions that can be represented in the UI and that the user can invoke. A common example is when the view model provides a Submit command that allows the user submit data to a web service or to a data repository. The view may choose to represent that command with a button so that the user can click the button to submit the data. Typically, when the command becomes unavailable, its associated UI representation becomes disabled. Commands provide a way to encapsulate user actions and to cleanly separate them from their visual representation in the UI.

To summarize, the view model has the following key characteristics:

* The view model is a non-visual class and does not derive from any WPF base class. It encapsulates the presentation logic required to support a use case or user task in the application. The view model is testable independently of the view and the model.
* The view model typically does not directly reference the view. It implements properties and commands to which the view can data bind. It notifies the view of any state changes via change notification events via the **INotifyPropertyChanged** and **INotifyCollectionChanged** interfaces.
* The view model coordinates the view's interaction with the model. It may convert or manipulate data so that it can be easily consumed by the view and may implement additional properties that may not be present on the model. It may also implement data validation via the **IDataErrorInfo** or **INotifyDataErrorInfo** interfaces.
* The view model may define logical states that the view can represent visually to the user.

View or ViewModel?

Many times, determining where certain functionality should be implemented is not obvious. The general rule of thumb is: Anything concerned with the specific visual appearance of the UI on the screen and that could be re-styled later (even if you are not currently planning to re-style it) should go into the view; anything that is important to the logical behavior of the application should go into the view model. In addition, because the view model should have no explicit knowledge of the specific visual elements in the view, code to programmatically manipulate visual elements within the view should reside in the view's code-behind or be encapsulated in a behavior. Similarly, code to retrieve or manipulate data items that are to be displayed in the view through data binding should reside in the view model.

For example, the highlight color of the selected item in a list box should be defined in the view, but the list of items to display, and the reference to the selected item itself, should be defined by the view model.

# The Model Class

The model in the MVVM pattern encapsulates business logic and data. Business logic is defined as any application logic that is concerned with the retrieval and management of application data and for making sure that any business rules that ensure data consistency and validity are imposed. To maximize re-use opportunities, models should not contain any use case–specific or user task–specific behavior or application logic.

Typically, the model represents the client-side domain model for the application. It can define data structures based on the application's data model and any supporting business and validation logic. The model may also include the code to support data access and caching, though typically a separate data repository or service is employed for this. Often, the model and data access layer are generated as part of a data access or service strategy, such as the ADO.NET Entity Framework, WCF Data Services, or WCF RIA Services.

Typically, the model implements the facilities that make it easy to bind to the view. This usually means it supports property and collection changed notification through the **INotifyPropertyChanged** and **INotifyCollectionChanged** interfaces. Model classes that represent collections of objects typically derive from the **ObservableCollection<T>** class, which provides an implementation of the **INotifyCollectionChanged** interface.

The model may also support data validation and error reporting through the **IDataErrorInfo** (or **INotifyDataErrorInfo**) interfaces. The **IDataErrorInfo** and **INotifyDataErrorInfo** interfaces allow WPF data binding to be notified when values change so that the UI can be updated. They also enable support for data validation and error reporting in the UI layer.

What if your model classes do not implement the required interfaces?

Sometimes you will need to work with model objects that do not implement the **INotifyPropertyChanged**, **INotifyCollectionChanged**, **IDataErrorInfo**, or **INotifyDataErrorInfo** interfaces. In those cases, the view model may need to wrap the model objects and expose the required properties to the view. The values for these properties will be provided directly by the model objects. The view model will implement the required interfaces for the properties it exposes so that the view can easily data bind to them.

The model has the following key characteristics:

* Model classes are non-visual classes that encapsulate the application's data and business logic. They are responsible for managing the application's data and for ensuring its consistency and validity by encapsulating the required business rules and data validation logic.
* The model classes do not directly reference the view or view model classes and have no dependency on how they are implemented.
* The model classes typically provide property and collection change notification events through the **INotifyPropertyChanged** and **INotifyCollectionChanged** interfaces. This allows them to be easily data bound in the view. Model classes that represent collections of objects typically derive from the **ObservableCollection<T>** class.
* The model classes typically provide data validation and error reporting through either the
* **IDataErrorInfo** or **INotifyDataErrorInfo** interfaces.
* The model classes are typically used in conjunction with a service or repository that encapsulates data access and caching.

# Class Interactions

The MVVM pattern provides a clean separation between your application's user interface, its presentation logic, and its business logic and data by separating each into separate classes. Therefore, when you implement MVVM, it is important to factor in your application's code to the correct classes, as described in the previous section.

Well-designed view, view model, and model classes will not only encapsulate the correct type of code and behavior; they will also be designed so that they can easily interact with each other via data binding, commands, and data validation interfaces.

The interactions between the view and its view model are perhaps the most important to consider, but the interactions between the model classes and the view model are also important. The following sections describe the various patterns for these interactions and describe how to design for them when implementing the MVVM pattern in your applications.

1. Data Binding

Data binding plays a very important role in the MVVM pattern. WPF provides powerful data binding capabilities. Your view model and (ideally) your model classes should be designed to support data binding so that they can take advantage of these capabilities. Typically, this means that they must implement the correct interfaces.

WPF data binding supports multiple data binding modes. With one-way data binding, UI controls can be bound to a view model so that they reflect the value of the underlying data when the display is rendered. Two-way data binding will also automatically update the underlying data when the user modifies it in the UI.

To ensure that the UI is kept up to date when the data changes in the view model, it should implement the appropriate change notification interface. If it defines properties that can be data bound, it should implement the **INotifyPropertyChanged** interface. If the view model represents a collection, it should implement the **INotifyCollectionChanged** interface or derive from the **ObservableCollection<T>** class that provides an implementation of this interface. Both of these interfaces define an event that is raised whenever the underlying data is changed. Any data bound controls will be automatically updated when these events are raised.

In many cases, a view model will define properties that return objects (and which, in turn, may define properties that return additional objects). WPF data binding supports binding to nested properties via the Path property. Therefore, it is very common for a view's view model to return references to other view model or model classes. All view model and model classes accessible to the view should implement the **INotifyPropertyChanged** or **INotifyCollectionChanged** interfaces, as appropriate.

The following sections describe how to implement the required interfaces in order to support data binding within the MVVM pattern.

1. Implementing INotifyPropertyChanged

Implementing the **INotifyPropertyChanged** interface in your view model or model classes allows them to provide change notifications to any data-bound controls in the view when the underlying property value changes. Implementing this interface is straightforward, as shown in the following code example.

public class Questionnaire : INotifyPropertyChanged

{

private string \_favoriteColor;

public string FavoriteColor

{

get => \_favoriteColor;

set

{

if (\_favoriteColor != value)

{

\_favoriteColor = value;

OnPropertyChanged();

}

}

}

public event PropertyChangedEventHandler PropertyChanged;

protected void OnPropertyChanged([CallerMemberName] string propertyName = "")

=> PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));

}

Implementing the **INotifyPropertyChanged** interface on many view model classes can be repetitive and error-prone because of the need to specify the property name in the event argument. The Crystal Library provides the **BindableBase** base class from which you can derive your view model classes that implements the **INotifyPropertyChanged** interface in a type-safe manner, as shown here.

public abstract class BindableBase : INotifyPropertyChanged

{

public event PropertyChangedEventHandler PropertyChanged;

protected virtual bool SetProperty<T>(ref T storage, T value,

[CallerMemberName] string propertyName = null)

{

if (EqualityComparer<T>.Default.Equals(storage, value))

{

return false;

}

storage = value;

OnPropertyChanged(propertyName);

return true;

}

protected virtual void OnPropertyChanged(PropertyChangedEventArgs args)

=> PropertyChanged?.Invoke(this, args);

}

A derived view model class can raise the property change event in the setter by calling the **SetProperty** method. The **SetProperty** method checks whether the backing field is different from the value being set. If different, the backing field is updated and the **PropertyChanged** event is raised.

The following code example shows how to set the property and simultaneously signal the change of another property by using a lambda expression in the **OnPropertyChanged** method. This example comes from the Stock Trader RI. The **TransactionInfo** and **TickerSymbol** properties are related. If the **TransactionInfo** property changes, the **TickerSymbol** will also likely be updated. By calling **OnPropertyChanged** for the **TickerSymbol** property in the setter of the **TransactionInfo** property, two **PropertyChanged** events will be raised, one for **TransactionInfo** and one for **TickerSymbol**.

**Note**: Using a lambda expression in this way involves a small performance cost because the lambda expression has to be evaluated for each call. The benefit is that this approach provides compile-time type safety and refactoring support if you rename a property. Although the performance cost is small and would not normally impact your application, the costs can accrue if you have many change notifications. In this case, you should consider using the non-lambda method overload.

Often, your model or view model will include properties whose values are calculated from other properties in the model or view model. When handling changes to properties, be sure to also raise notification events for any calculated properties.

1. Implementing INotifyCollectionChanged

Your view model or model class may represent a collection of items, or it may define one or more properties that return a collection of items. In either case, it is likely that you will want to display the collection in an **ItemsControl**, such as a **ListBox**, or in a **DataGrid** control in the view. These controls can be data bound to a view model that represents a collection or to a property that returns a collection via the **ItemSource** property.

<DataGrid ItemsSource="{Binding Path=LineItems}"/>

To properly support change notification requests, the view model or model class, if it represents a collection, should implement the **INotifyCollectionChanged** interface (in addition to the **INotifyPropertyChanged** interface). If the view model or model class defines a property that returns a reference to a collection, the collection class returned should implement the **INotifyCollectionChanged** interface.

However, implementing the **INotifyCollectionChanged** interface can be challenging because it has to provide notifications when items are added, removed, or changed within the collection. Instead of directly implementing the interface, it is often easier to use or derive from a collection class that already implements it. The **ObservableCollection<T>** class provides an implementation of this interface and is commonly used as either a base class or to implement properties that represent a collection of items.

If you need to provide a collection to the view for data binding, and you do not need to track the user's selection or to support filtering, sorting, or grouping of the items in the collection, you can simply define a property on your view model that returns a reference to the **ObservableCollection<T>** instance.

public class OrderViewModel : BindableBase

{

public OrderViewModel(IOrderService orderService)

{

LineItems = new ObservableCollection<OrderLineItem>(

orderService.GetLineItemList());

}

public ObservableCollection<OrderLineItem> LineItems { get; private set; }

}

If you obtain a reference to a collection class (for example, from another component or service that does not implement **INotifyCollectionChanged**), you can often wrap that collection in an **ObservableCollection<T>** instance using one of the constructors that take an **IEnumerable<T>** or **List<T>** parameter.

1. Implementing ICollectionView

The preceding code example shows how to implement a simple view model property that returns a collection of items that can be displayed via data bound controls in the view. Because the **ObservableCollection<T>** class implements the **INotifyCollectionChanged** interface, the controls in the view will be automatically updated to reflect the current list of items in the collection as items are added or removed.

However, you will often need to more finely control how the collection of items is displayed in the view, or track the user's interaction with the displayed collection of items, from within the view model itself. For example, you may need to allow the collection of items to be filtered or sorted according to presentation logic implemented in the view model, or you may need to keep track of the currently selected item in the view so that commands implemented in the view model can act on the currently selected item.

WPF supports these scenarios by providing various classes that implement the **ICollectionView** interface. This interface provides properties and methods to allow a collection to be filtered, sorted, or grouped, and allow the currently selected item to be tracked or changed. WPF provides an implementation of this interface using the **ListCollectionView** class.

Collection view classes work by wrapping an underlying collection of items so that they can provide automatic selection tracking and sorting, filtering, and paging for them. An instance of these classes can be created programmatically or declaratively in XAML using the **CollectionViewSource** class.

Note: In WPF, a default collection view will actually be automatically created whenever a control is bound to a collection.

Collection view classes can be used by the view model to keep track of important state information for the underlying collection, while maintaining a clean separation of concerns between the UI in the view and the underlying data in the model. In effect, **CollectionViews** are view models that are designed specifically to support collections.

Therefore, if you need to implement filtering, sorting, grouping, or selection tracking of items in the collection from within your view model, your view model should create an instance of a collection view class for each collection to be exposed to the view. You can then subscribe to selection changed events, such as the **CurrentChanged** event, or control filtering, sorting, or grouping using the methods provided by the collection view class from within your view model.

The view model should implement a read-only property that returns an **ICollectionView** reference so that controls in the view can data bind to the collection view object and interact with it. All WPF controls that derive from the ItemsControl base class can automatically interact with **ICollectionView** classes.

The following code example shows the use of the **ListCollectionView** in WPF to keep track of the currently selected customer.

1. Commands

In addition to providing access to the data to be displayed or edited in the view, the view model will likely define one or more actions or operations that can be performed by the user. In WPF, actions or operations that the user can perform through the UI are typically defined as commands. Commands provide a convenient way to represent actions or operations that can be easily bound to controls in the UI. They encapsulate the actual code that implements the action or operation and help to keep it decoupled from its actual visual representation in the view.

Commands can be visually represented and invoked in many different ways by the user as they interact with the view. In most cases, they are invoked as a result of a mouse click, but they can also be invoked as a result of shortcut key presses, touch gestures, or any other input events. Controls in the view are data bound to the view model's commands so that the user can invoke them using whatever input event or gesture the control defines. Interaction between the UI controls in the view and the command can be two-way. In this case, the command can be invoked as the user interacts with the UI, and the UI can be automatically enabled or disabled as the underlying command becomes enabled or disabled.

The view model can implement commands as either a **Command Method** or as a **Command Object** (an object that implements the **ICommand** interface). In either case, the view's interaction with the command can be defined declaratively without requiring complex event handling code in the view's code-behind file. For example, certain controls in WPF inherently support commands and provide a **Command** property that can be data bound to an **ICommand** object provided by the view model. In other cases, a command behavior can be used to associate a control with a command method or command object provided by the view model.

**Note**: Behaviors are a powerful and flexible extensibility mechanism that can be used to encapsulate interaction logic and behavior that can then be declaratively associated with controls in the view.

Command behaviors can be used to associate command objects or methods with controls that were not specifically designed to interact with commands.

The following sections describe how to implement commands in your view, as command methods or as command objects, and how to associate them with controls in the view.

1. Implementing a Task-Based Delegate Command

There are a number of scenarios where the command calls code with long running transactions that cannot block the UI thread. For these scenarios you should use the **FromAsyncHandler** method of the **DelegateCommand** class, which creates a new instance of the **DelegateCommand** from an async handler method.

1. Data Validation and Error Reporting

Your view model or model will often be required to perform data validation and to signal any data validation errors to the view so that the user can act to correct them.

WPF provides support for managing data validation errors that occur when changing individual properties that are bound to controls in the view. For single properties that are data-bound to a control, the view model or model can signal a data validation error within the property setter by rejecting an incoming bad value and throwing an exception. If the **ValidatesOnExceptions** property on the data binding is **true**, the data binding engine in WPF will handle the exception and display a visual cue to the user that there is a data validation error.

However, throwing exceptions with properties in this way should be avoided where possible. An alternative approach is to implement the **IDataErrorInfo** or **INotifyDataErrorInfo** interfaces on your view model or model classes. These interfaces allow your view model or model to perform data validation for one or more property values and to return an error message to the view so that the user can be notified of the error.

1. Implementing Exceptions
2. Implementing IDataErrorInfo

The IDataErrorInfo interface provides basic support for property data validation and error reporting. It defines two read-only properties: an indexer property, with the property name as the indexer argument, and an Error property. Both properties return a string value.

The indexer property allows the view model or model class to provide an error message specific to the named property. An empty string or null return value indicates to the view that the changed property value is valid. The Error property allows the view model or model class to provide an error message for the entire object. Note, however, that this property is not currently called by the WPF data binding engine.

The IDataErrorInfo indexer property is accessed when a data-bound property is first displayed, and whenever it is subsequently changed. Because the indexer property is called for all properties that change, you should be careful to ensure that data validation is as fast and as efficient as possible.

When binding controls in the view to properties you want to validate through the IDataErrorInfo interface, set the ValidatesOnDataErrors property on the data binding to true. This will ensure that the data binding engine will request error information for the data-bound property

1. Implementing INotifyDataErrorInfo

The **INotifyDataErrorInfo** interface is more flexible than the **IDataErrorInfo** interface. It supports multiple errors for a property, asynchronous data validation, and the ability to notify the view if the error state changes for an object.

The **INotifyDataErrorInfo** interface defines a **HasErrors** property, which allows the view model to indicate whether an error (or multiple errors) for any properties exist, and a **GetErrors** method, which allows the view model to return a list of error messages for a particular property.

The **INotifyDataErrorInfo** interface also defines an **ErrorsChanged** event. This supports asynchronous validation scenarios by allowing the view or view model to signal a change in error state for a particular property through the **ErrorsChanged** event. Property values can be changed in a number of ways, and not just via data binding—for example, as a result of a web service call or background calculation. The **ErrorsChanged** event allows the view model to inform the view of an error once a data validation error has been identified.

Ds

1. Implementing ValidationRules

Ds

1. Implementing Data Annotation

Sd

# Construction and Configuration

The MVVM pattern helps you to cleanly separate your UI from your presentation and business logic and data, so implementing the right code in the right class is an important first step in using the MVVM pattern effectively. Managing the interactions between the view and view model classes through data binding and commands are also important aspects to consider. The next step is to consider how the view, view model, and model classes are instantiated and associated with each other at run time.

**Note**: Choosing an appropriate strategy to manage this step is especially important if you are using a dependency injection container in your application. The Managed Extensibility Framework (MEF) and the Unity Application Block (Unity) both provide the ability to specify dependencies between the view, view model, and model classes and to have them fulfilled by the container. For more advanced scenarios, see Advanced MVVM Scenarios.

Typically, there is a one-to-one relationship between a view and its view model. The view and view model are loosely coupled via the view's data context property; this allows visual elements and behaviors in the view to be data bound to properties, commands, and methods on the view model. You will need to decide how to manage the instantiation of the view and view model classes and their association via the DataContext property at run time.

Care must also be taken when constructing and connecting the view and view model to ensure that loose coupling is maintained. As noted in the previous section, the view model should ideally not depend on any specific implementation of a view. Similarly, the view should ideally not depend on any specific implementation of a view model.

**Note**: However, it should be noted that the view will implicitly depend on specific properties, commands, and methods on the view model because of the data bindings it defines. If the view model does not implement the required property, command, or method, a run-time exception will be generated by the data binding engine, which will be displayed in the Visual Studio output window during debugging.

There are multiple ways the view and the view model can be constructed and associated at run time. The most appropriate approach for your application will largely depend on whether you create the view or the view model first, and whether you do this programmatically or declaratively. The following sections describe common ways in which the view and view model classes can be created and associated with each other at run time.

1. Creating the View Model Using XAML

Perhaps the simplest approach is for the view to declaratively instantiate its corresponding view model in XAML. When the view is constructed, the corresponding view model object will also be constructed. You can also specify in XAML that the view model be set as the view's data context.

<UserControl.DataContext>

<my:MyViewModel/>

</UserControl.DataContext>

When this view is created, an instance of the **MyViewModel** is automatically constructed and set as the view's data context. This approach requires your view model to have a default (parameter-less) constructor.

The declarative construction and assignment of the view model by the view has the advantage that it is simple and works well in design-time tools such as Microsoft Expression Blend or Microsoft Visual Studio. The disadvantage of this approach is that the view has knowledge of the corresponding view model type and that the view model type must have a default constructor.

# Key Decisions

When you choose to use the MVVM pattern to construct your application, you will have to make certain design decisions that will be difficult to change later on. Generally, these decisions are application-wide and their consistent use throughout the application will improve developer and designer productivity.

The following summarizes the most important decisions when implementing the MVVM pattern:

* Decide on the approach to view and view model construction you will use. You need to decide if your application constructs the views or the view models first and whether to use a dependency injection container, such as Unity or MEF. You will usually want this to be consistent application- wide. For more information, see the section, Construction and Wire-Up, in this topic and the section Advanced Construction and Wire-Up, in Advanced MVVM Scenarios.
* Decide if you will expose commands from your view models as command methods or command objects. Command methods are simple to expose and can be invoked through behaviors in the view. Command objects can neatly encapsulate the command and enabled/disabled logic and can be invoked through behaviors or via the Command property on **ButtonBase**-derived controls. To make it easier on your developers and designers, it is a good idea to make this an application-wide choice. For more information, see the section, Commands, in this topic.

* Decide how your view models and models will report errors to the view. Your models can either support **IDataErrorInfo** or **INotifyDataErrorInfo**. Not all models may need to report error information, but for those that do, it is preferable to have a consistent approach for your developers. For more information, see the section, Data Validation and Error Reporting, in this topic.
* Decide whether Microsoft Blend for Visual Studio design-time data support is important to your team. If you will use Blend to design and maintain your UI and want to see design time data, make sure that your views and view models offer constructors that do not have parameters and that your views provide a design-time data context. Alternatively, consider using the design-time features provided by Microsoft Blend for Visual Studio using design- time attributes such as **d:DataContext** and **d:DesignSource**.