Debreceni Egyetem

Informatika Kar

Using Silverlight and XAML in MVC, MVP, MVVM patterns

Külső témavezető:

Balogh Péter

MCPD Enterprise

Application Developer

MCSD For Microsoft .NET

Belső témavezető:

Dr. Juhász István

Egyetemi adjunktus

Készítette:

Fésüs Miklós

Programtervező

Informatikus

Debrecen

2009

Special Thanks To

Special Thanks to Péter Balogh, my consultant, who guided me, and managed to help me a lot, even though he was in Dublin the whole time.

István Juhász, who has been my supervisor.

Balázs Máté, who tested my application.

László Gonda, who checked this document.

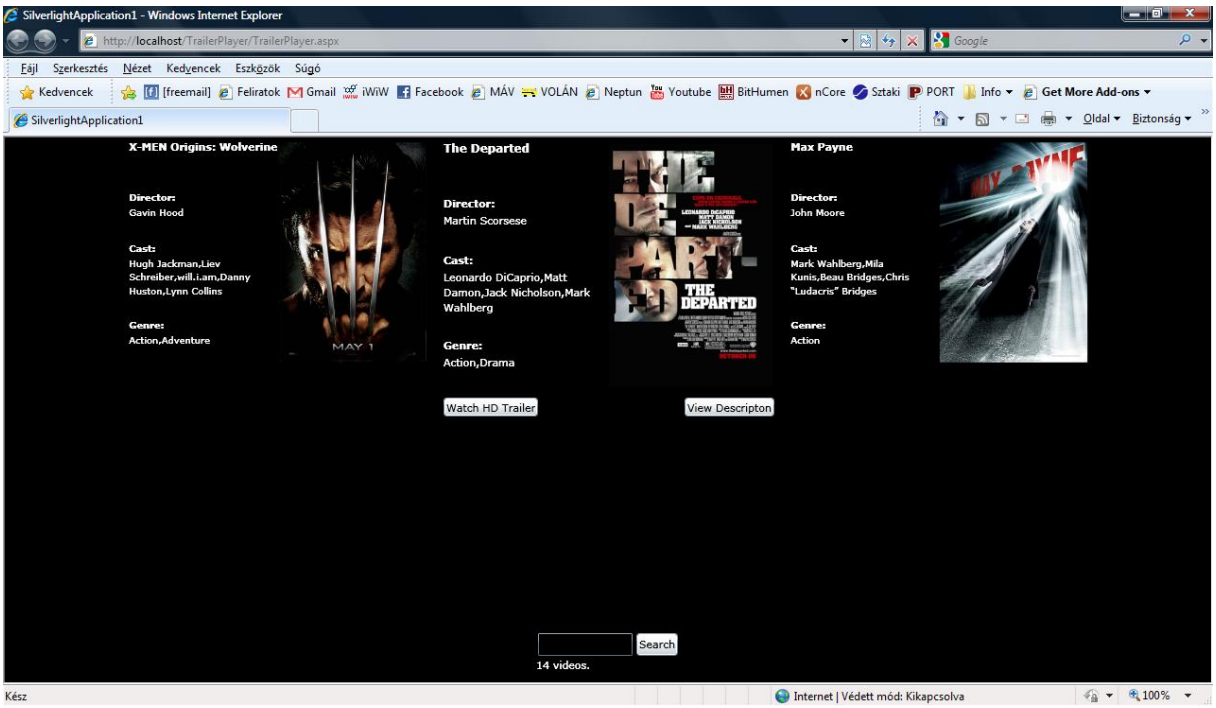
Évi, my love, for her patience.

Table of Contents

Introduction

Subject of the thesis

The subject of the thesis is to examine how the two well known design patterns, the MVC (Model-View-Controller) and the MVP (Model-View-Presenter), and a quite new pattern called MVVM (Model-View-ViewModel) can be used with Silverlight 2.0 technology. In the following chapters I intend to examine how they can be implemented using the Silverlight's markup language, the XAML (Extensible Application Markup Language), and C#. A custom sample application (Trailer Player application) is presented, that is used as a basis for the examination of how the mentioned patterns are used with this technology, and also I present a study of cases, advantages and disadvantages of using them in our application. The sample application itself is a simple web application, a video player, where the user can view the newest movie trailers, and get some related information about them, see Figure 1.



Screen Capture of the Trailer Player sample application

After a brief overview of current and past web application development solutions in the area of .NET framework, the thesis details XAML and Silverlight with more technical depth. We also lay out the minimal requirement set for the Trailer Player application in very brief use case scenarios. Then we can discuss the implementation of the Trailer Player application examining each application layer, their role and functionality beginning with the Model layer and finishing with the View where we examine how each Silverlight control is implemented.

We will see how the MVVM pattern is implemented, and further discuss how things would look like in the other two patterns. In the end of we will try to describe, how to change the application's layers, to have our application implemented in another design pattern.

Web solutions of the .NET Framework

The web applications of today's era are often commented as Web 2.0. This term mostly represents the collection of popular web applications for social networking and media consumption. Sites like FaceBook, IWIW, YouTube, are the flagships of Web 2.0 movement.

Despite the 2.0 version number in the popular term all these sites evolved from traditional web development methods and are still based on HTML and other traditional markup languages. Since the early days of HTML, the processing capacity of computers boomed, along with it the need of the users for more interactive experience on the web. As a result JavaScript and other Dynamic HTML technologies emerged. Development of these rich internet applications became increasingly difficult as the applications became richer. ASP.NET introduced the very powerful concept of code behind that enabled developers to treat markup elements as Classes and objects outside of the Browser's domain before even rendering these objects to markup. This enhanced the link between sleek UI and intelligent business logic. Since the demand from web users never ceases for fast interactive user interface usage of AJAX technologies became world-wide as link between backend processing and browser UI. On front of UI presentation technologies the Flash system of Adobe and Microsoft's Silverlight platform pave the way today.

Silverlight vs. Flash

The most successful browser plug-in is Adobe Flash, which is installed on over 90 percent of the world's web browsers. Flash has a long history that spans more than ten years, beginning

as a straightforward tool for adding animated graphics and gradually evolving into a platform for developing interactive content. It's perfectly reasonable for .NET developers to create websites that use Flash content. However, doing so requires a separate design tool, and a completely different programming language (ActionScript) and programming environment (Flex). Furthermore, there's no straightforward way to integrate Flash content with server-side .NET code. For example, creating Flash applications that call .NET components is awkward at best. Using server-side .NET code to render Flash content (for example, a custom ASP.NET control that spits out a Flash content region) is far more difficult. Silverlight aims to give .NET developers a better option for creating rich web content. Silverlight provides a browser plug-in with many similar features to Flash, but one that's designed from the ground up for .NET. Silverlight natively supports the C# language and embraces a range of .NET concepts. As a result, developers can write client-side code for Silverlight in the same language they use for server-side code (such as C# and VB), and use many of the same abstractions (including streams, controls, collections, generics, and LINQ).

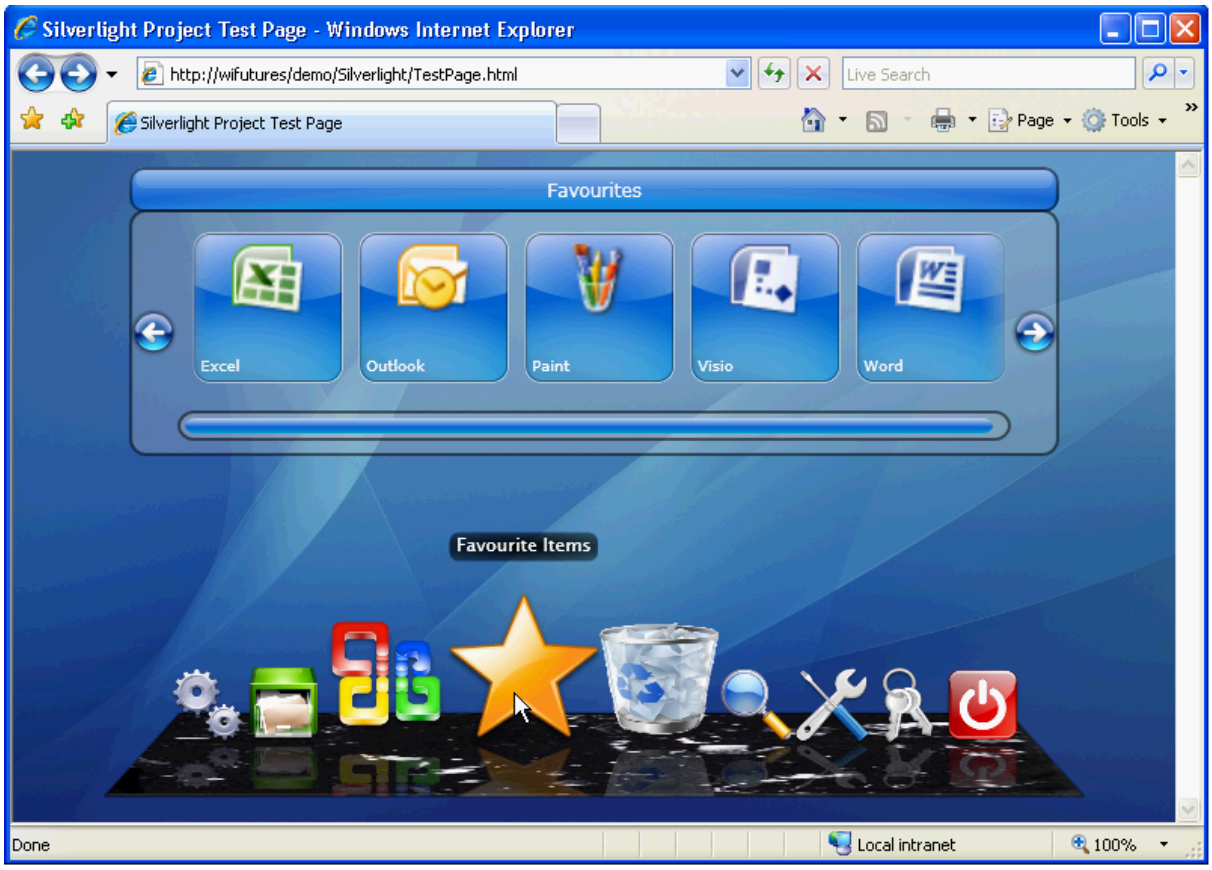
Silverlight

Silverlight is an exciting new technology from Microsoft for developing rich user experiences that are accessible on a variety of platforms. It is a cross-platform Common Language Runtime (CLR) with a strong presentation framework for compositing user interfaces and displaying images and video, making development of rich user experiences much easier than before. The purpose of the technology on one hand is to build Rich Internet Applications (RIA), on the other hand though, it can be used to build business applications as well. RIAs are web applications that have some characteristics of desktop applications, typically delivered by web browser plug-ins. At the core of Silverlight is a new markup language called Extensible Application Markup Language(XAML), which helps designers and developers work more effectively with each other since it is a declarative language with tools built around it.

The technology can be called as the web implementation of Microsoft's desktop platform the Windows Presentation Foundation (WPF), and it was named earlier as Windows Presentation Foundation/Everywhere. However the Silverlight plug-in contains only a subset of the .NET framework, while WPF can use the entire .NET framework.

Silverlight's main features:

2-D drawings: the content you draw is defined as shapes and paths, so we can manipulate this content on the client side. We can even respond to events on the graphics, which makes it easy to add interactivity to the UI.



2D drawings in a custom Silverlight Layout

Animation: Silverlight has a time-based animation model. It has key frame based animation and has a more simple animation. In the first we can add multiple key frames to describe what properties to change, when and how long does it take. In the second we just have to set the property the value and the duration.

Networking: Silverlight applications can call ASP.NET web services or WCF (Windows Communication Foundation) web services, and receive XML, JSON, or RSS data. They can also send manually created XML requests over HTTP and open socket connections for fast two-way communication.

Media: Silverlight provides playback of Windows Media Audio, Windows Media Video MP3 audio, and VC-1 (which supports high definition). Silverlight 2 has a built in control for handling videos called MediaElement.

HD video playing in Silverlight

The common language runtime. Silverlight includes a scaled-down version of the CLR, complete with an essential set of core classes, a garbage collector, a JIT (just-in-time) compiler, support for generics, threading, etc.

Data binding: Silverlight data binding grants an easy way to display large amounts of data with minimal code. One way and two way data binding are available with the help of the INotifyPropertyChanged interface.

XAML

In Silverlight applications, user interfaces are declared in XAML and programmed using a subset of the .NET Framework. XAML can be used for marking up the vector graphics and animations. Textual content created with Silverlight is searchable and index-able by search engines as it is not compiled, but represented as text.

Extensible Application Markup Language is a declarative language. It can be used to create the user interface elements in the declarative XAML markup. The advantage of this is that we can use a separate code-behind file to respond to events and manipulate the objects declared in markup.

XAML files are XML files. XAML is a case-sensitive language. The names of XAML elements and attributes are case-sensitive. The value of an attribute is potentially case-sensitive, this will depend on how the attribute value is handled for particular properties. Every element in a XAML document maps to an instance of a Silverlight class. The name of the element matches the name of the class exactly. For example, the element <Button> instructs Silverlight to create a Button object. Since XAML is an XML document, elements can be nested inside each another. Properties of a class instances can be set through attributes.

However, in some situations an attribute isn't powerful enough to handle the job, like a complex attribute, or a collection attribute. In these cases, we have to use nested tags with a special syntax.

A XAML declared object can be referenced in the code behind only if we set up its x:Name property. The set property will be the name of the object in C#.

```
<Button x:Name="button1">
```

There are two ways to declare objects in XAML:

- Indirectly, using attribute syntax: Uses an inline value to declare an object. We can use this syntax to set the value of a property. This is an indirect operation for the XAML processor, because there must be something behind the scenes that knows how to create a new object on basis of knowing what property is being set, and the supplied string value.

  ```
  <TextBlock Height="15" Text="Director" Foreground="White"/>
  ```

- Directly, using Object Element Syntax: Uses opening and closing tags to declare an object as an XML element. We can use this syntax to declare root objects or to set property values. And the two ways can be mixed.

```
<ListBox.ItemsPanel>
        <ItemsPanelTemplate>
              <StackPanel Orientation="Horizontal"/>
        </ItemsPanelTemplate>
</ListBox.ItemsPanel>
```

XAML elements map directly to Common Language Runtime object instances, while XAML attributes map to Common Language Runtime properties and events on those objects.

XAML features

The following features of XAML have been widely used in the TrailerPlayer application:

Markup Extensions

Markup extensions are a XAML language concept, used prominently in the Silverlight XAML implementation. In XAML attribute syntax, braces ({ and }) indicate a markup extension usage, as per the XAML specification. This usage directs the XAML processing to escape from the general treatment of attribute values as either a literal string or a directly string-convertible value. Instead, a parser typically calls code that backs that particular markup extension, which assists in constructing an object tree from the markup.

Silverlight supports three markup extensions that are defined under its XML namespace and understood by its XAML parser. These are: *Binding*, *StaticResource*, and *TemplateBinding*. Binding supports data binding. *StaticResource* supports referencing resources that are defined in a *ResourceDictionary*. *TemplateBinding* supports control templates in XAML that can interact with the code properties of the templated object. Silverlight also implements one markup extension that is defined in the XAML namespace, x:Null.

```
<ListBox ItemContainerStyle="{StaticResource ListBoxItemStyle}">
```

Typeconverter-Enabled Attribute Values

In attribute syntax the value of an attribute is be able to be set by a string. The basic, native handling of how strings are converted into other object types or primitive values is based on the String type itself. But many types extend the basic string attribute processing behavior, so that the instances of a more complex object type can be specified as attribute values through a string. At the code level, this processing is accomplished by specifying a CLR type converter that processes the string attribute value. That means, if we want to set for example an object's Margin property, we can do it like that:

```
<Button Margin="10,15,0,5"/>
```

Instead of something like that:

```
<Button>
<Button.Margin>
<Thickness Left="10" Top="15" Right="0" Bottom="5">
</Button.Margin>
</Button>
```

XAML Collection Properties

XAML has a language feature whereby the object element that represents a collection type can be filled with elements from markup. When a XAML processor handles a property that takes a collection type, the appropriate collection as defined by initialization of the owning class is referenced implicitly, even if the object element for that collection is not present in the markup. The relationship of parent elements and child elements in a markup page is really a single object at the root, and every object element beneath the root is either a single instance that provides a property value of the parent, or one of the items within a collection that is also a collection-type property value of the parent.

Setting a collection property of a grid:

```
<Grid x:Name="Controller" Grid.Row="1" Margin="0,-5,0,0" >
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="40"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
```

Events and XAML Code-Behind

XAML is a declarative language for objects and their properties, but it also includes syntax for attaching event handlers to objects in the markup. We can specify the name of the event as an attribute name on the object where the event is handled. For the attribute value, we specify the name of an event-handler function that we define in code. The XAML processor uses this name to create a delegate representation in the loaded object tree and adds the specified handler to an internal handler list.

Example:

In XAML: `<Button x:Name="btnSearch" Click="brnSearch_Click"  />`

Code behind: `private void brnSearch_Click(object sender, RoutedEventArgs e)`

Attached Properties

XAML specifies a language feature that enables certain properties to be specified on any element, regardless of whether the property exists in the members table for the element it is being set on. The properties version of this feature is called an attached property. Attached properties can be imagined as global members that can be set on any element/class, regardless of its class hierarchy.

Attached properties in XAML are typically used through attribute syntax. In attribute syntax, we can specify an attached property in the form OwnerType.PropertyName. OwnerType is the type that provides the accessor methods that are required by a XAML processor in order to get or set the attached property value. Attached properties have an important role in forming our applications layout. For example to place a button in the second row of the previously defined Grid we have to set the properties of the button as the following.

```
<Grid x:Name="Controller" Margin="0,-5,0,0" >
    <ToggleButton Grid.Row="1"/>
```

The row is set to 1 because it starts from 0.

Dependency properties

The purpose of dependency properties is to provide a way to compute the value of a property based on the value of other inputs. These other inputs might include external properties such

as user preference, just-in-time property determination mechanisms such as data binding and animations/storyboards, multiple-use templates such as resources and styles, or values known through parent-child relationships with other elements in the object tree. In addition, a dependency property can be implemented to provide callbacks that monitor changes to other properties. The following example sets the text for a text block, using a binding in XAML. The binding uses an inherited data context and an object data source (not shown).

```
<Canvas>
    <TextBlock Text="{Binding Team.TeamName}"/>
</Canvas>
```

Most of the properties that are exposed by Silverlight elements are dependency properties. The key feature of this system is the way that these different property providers are prioritized. To determine the current value of a property, Silverlight has to decide which one takes precedence. This process is called dynamic value resolution.

1. Animations. If an animation is currently running, and that animation is changing the property value, Silverlight uses the animated value.

2. Local value. If we explicitly set a value in XAML or in code, Silverlight uses the local value. If we set a property using a data binding or a resource, it's considered to be a locally set value.

3. Styles. Silverlight styles allow you to configure multiple controls with one rule. If we set a style that applies to this control, it will get the value from the style.

4. Property value inheritance. Silverlight uses property value inheritance with a small set of control properties, including Foreground, FontFamily, FontSize, FontStretch, FontStyle, and FontWeight. That means if we set these properties in a higher level container, they cascade down to the contained content elements (like the TextBlock that actually holds the text inside).

5. Default value. If no other property setter is at work, the dependency property gets its default value. The default value is set with the PropertyMetadata object when the dependency property is first created.

WCF

Windows Communication Foundation is a programming framework used to build applications that intercommunicate. Because WCF is part of the .NET Framework, applications that use WCF can be developed in any programming language that can target the .NET runtime. The WCF unifies the various communications programming models supported in .NET 2.0, into a single model.

Windows Communication Foundation combines and extends the capabilities of Distributed Systems, Microsoft .NET Remoting, Web Services, and Web Services Enhancements (WSE), to develop and deliver unified secured systems. The WCF framework builds loosely-coupled applications on a service oriented architecture that interoperates more securely and reliably across platforms.

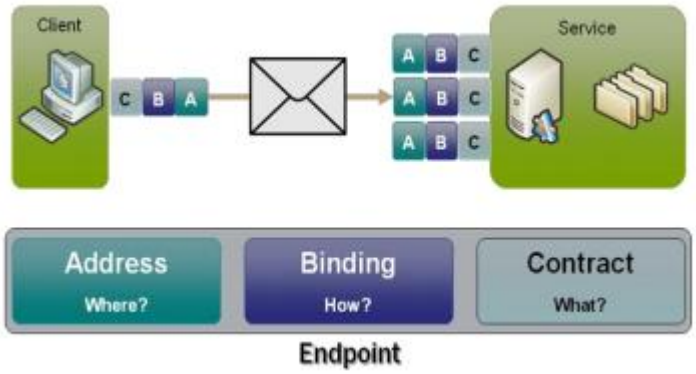
A WCF Service is composed of three parts: a Service class that implements the service to be provided, a host environment to host the service, and one or more endpoints to which clients will connect.

All communications with the WCF service will happen via the endpoints. The endpoints specify a Contract that defines which methods of the Service class will be accessible via the endpoint; each endpoint may expose a different set of methods. The endpoints also define a binding that specifies how a client will communicate with the service and the address where the endpoint is hosted.

In programming code, a WCF Service is implemented as a class. The class typically implements a Service Contract - a specially annotated interface whose methods stipulate the operations that the service performs.

Any class that is to be exposed as a WCF service must be either marked with the ServiceContract attribute, or implement an interface marked with it. All methods in the class or interface that a client can invoke using SOAP messages must be marked with OperationContract attribute. All classes or structures that define the data to be passed into or out of the operations are marked with the DataContract attribute. The attributes support the automatic generation of WSDL descriptions for the exposed methods, which can then be accessed by clients, or advertised to them.

A WCF client connects to a WCF service via an endpoint. Each Service exposes its Contract via one or more endpoints. An endpoint has an address, which is a URL specifying where the endpoint can be accessed and binding properties that specify how the data will be transferred.



The mnemonic "ABC" can be used to remember Address / Binding / Contract. Binding specifies what communication protocols are used to access the service, whether security mechanisms are to be used, and the like. WCF includes predefined bindings for most common communication protocols such as SOAP over HTTP, SOAP over TCP, and SOAP over Message Queues etc.

When a client wants to access the service via an endpoint, it not only needs to know the Contract, but it also has to adhere to the binding specified by the endpoint. Thus, both client and server must have compatible endpoints.

A client can communicate with a WCF service using any of the Remote Procedure Call-based mechanisms in which the service can be invoked as a method call. WCF supports non-blocking (asynchronous) calls between client and service

We have used WCF to implement services required in the model layer and to separate business logic from actual database functionality. It is widely accepted to use WCF in Service Oriented Applications (SOA) and approaches.

Standard Web application patterns in the . NET framework

Why do we three layered architectural patterns?

The purpose of many computer systems is to retrieve data from a data store and display it for the user. After the user changes the data, the system stores the updates in the data store. Because the key flow of information is between the data store and the user interface, it might be inclined to tie these two pieces together to reduce the amount of coding and to improve application performance. However, this seemingly natural approach has some significant problems. One problem is that the user interface tends to change much more frequently than the data storage system. Another problem with coupling the data and user interface pieces is that business applications tend to incorporate business logic that goes far beyond data transmission.

User interface logic tends to change more frequently than business logic, especially in Web-based applications. Designers like to change the look of a web application from time to time, maybe because they add a new feature, or a new skin. If presentation code and business logic are combined in a single object, we have to modify an object containing business logic every time we change the user interface. This will be resulting in errors, and we have to test the business logic again when we change the user interface.

In some cases, the application displays the same data in different ways. For example, when an analyst prefers a spreadsheet view of data whereas management prefers a pie chart of the same data. In some rich-client user interfaces, multiple views of the same data are shown at the same time. If the user changes data in one view, the system must update all other views of the data automatically.

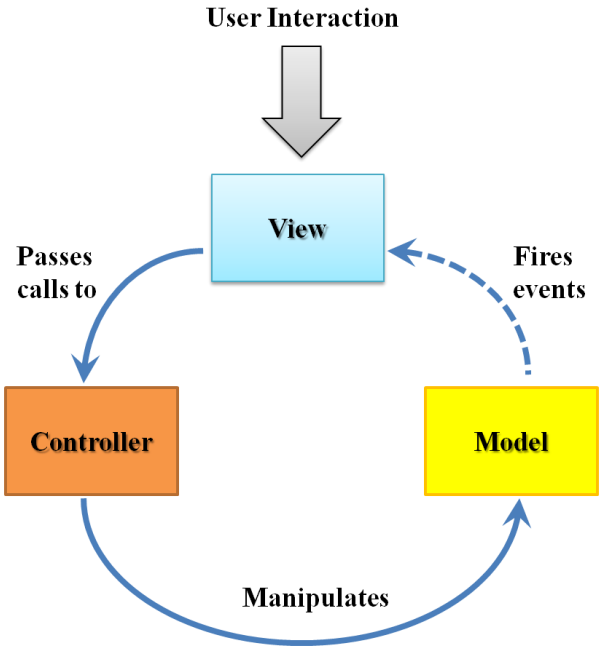
User interface code tends to be more device-dependent than business logic. For example if we want to support browsing our application from a mobile phone we must replace much of the user interface code, but the business logic may be unaffected.

If we have to change the business logic for any reason then if presentation code and business logic are combined in a single object we may have to modify the presentation code of every display.

So how do we modularize the user interface functionality of a Web application so that we can easily modify the individual parts? Luckily now we have three patterns to solve this problem, and these are the following.

MVC

Model View Controller is an architectural pattern is used in software engineering It isolates the data(model), the application/business logic (controller), and the representation of the data for the user (view), resulting an application where it is easier to modify the visual appearance or the business rule.



Model-View-Controller

Model

The model manages the behavior and data of the application. Some of its functions are query methods that permit the User to get information about the current state of the model. Others methods permit the state to be modified. A model must be able to register views and it must be able to notify all of its registered views when any of its functions cause its state to be changed.

View

View is responsible for look and feel, some custom formatting, sorting, data input validation etc. View is completely isolated from any data manipulation. View provides interface to interact with the system. The beauty of MVC approach is that it supports any kind of view, which is challenging in today's distributed and multi-platform environment.

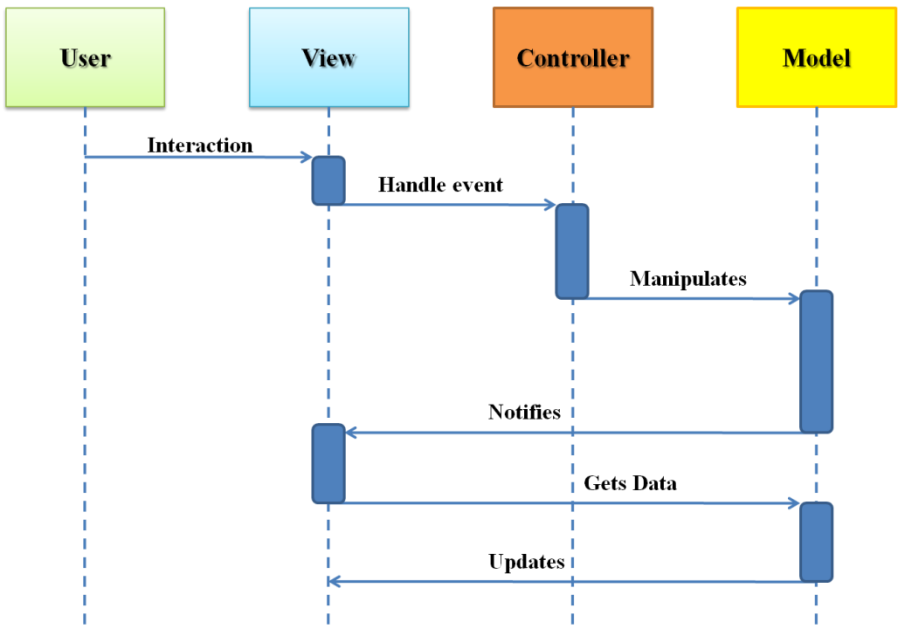
A MVC model can have multiple views, which are controlled by controller. View interface can be a web page like an aspx page, a windows forms or a WPF application, or even a Silverlight web application.

Controller

The controller interprets the mouse and keyboard inputs from the user, informing the model to change its state if necessary. It provides business functions by implementing the business logic by which the underlying model needs to be modified.

The control flow is generally as follows:

1. The user interacts with the user interface
2. The controller handles the input event from the user interface via event handlers
3. The controller modifies the model based on the user action, driven by the implemented business logic possibly resulting in a change in the model's state.
4. A view uses the model indirectly to generate an appropriate user interface. The view gets its own data from the model. The model and controller have no direct knowledge of the view.
5. The user interface waits for further user interactions, which restarts the cycle.



Sequence Diagram of MVC

When the data is changed, all Views are notified of the change and they can update themselves as necessary.

Communication of layers is often defined by Interfaces. These interfaces are used inside the Model, View and Controller components. This separation enables further decoupling by using the Factory pattern to get the actual implementation of layer components. If Factory pattern is coupled with IOC pattern, or with the usage of third party IOC containers, the actual application architecture can be declaratively configured, hence making it easy to change Views (User interfaces), Models (Database versions), and Controls (business Logic) functions.

Advantages

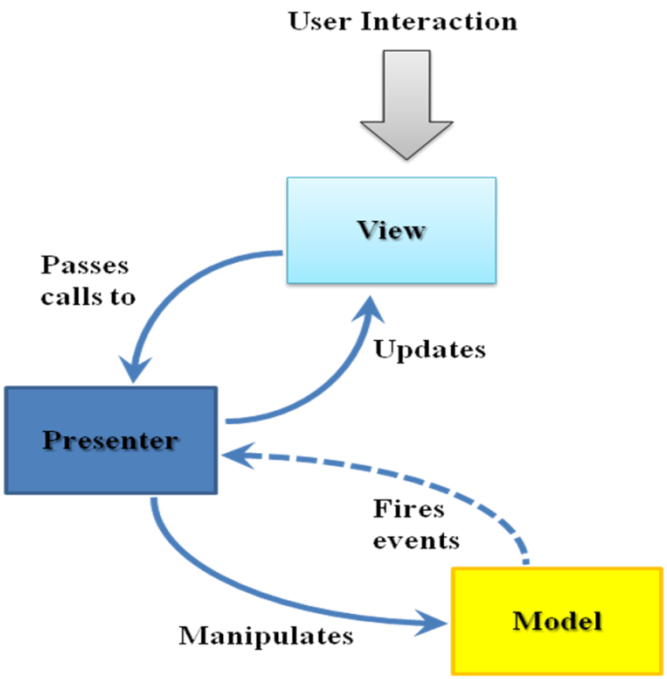
- Since MVC handles the multiple views using the same enterprise model it is easier to maintain, test and upgrade the multiple system.
- It will be easier to add new clients just by adding their views and controllers.
- Since the Model is completely decoupled from view it allows lot of flexibilities to design and implement the model considering reusability and modularity. This model also can be extended for further distributed application.
- It is possible to have development process in parallel for model, view and controller.
- This makes the application extensible and scalable.

Disadvantages

- Requires high skilled experienced professionals who can identify the requirements in depth at the front before actual design.
- It requires the significant amount of time to analyze and design.
- This design approach is not suitable for smaller applications.
- In case of complicated user interface with states of views related to each other and possible workflow models, this information though not business functionality related has to be in the Model layer. In very large systems this may be beneficial, otherwise it is usually transferred to the control layer, somewhat breaching the pattern.

MVP

Model-View-Presenter (MVP) is a variation of the Model-View-Controller (MVC) pattern but specifically geared towards a page event model. The primary differentiator of MVP is that the Presenter implements an Observer design of MVC but the basic ideas of MVC remain the same: the model stores the data, the view shows a represent-tation of the model, and the presenter coordinates communications between the layers.



Model-View-Presenter

MVP implements an Observer approach with respect to the fact that the Presenter interprets events and performs logic necessary to map those events to the proper commands to manipulate the model. It separates the responsibilities for the visual display and the event handling behavior into different classes The view class manages the controls on the user interface and forwards user events to a presenter class. The presenter contains the logic to respond to the events, update the model (business logic and data of the application) and, in turn, manipulate the state of the view.

To facilitate testing the presenter, the presenter has a reference to the view interface instead of to the concrete implementation of the view. By doing this, the real view can be replaced with a mock implementation to run tests.

Martin Fowler has suggested that MVP be split between two "new" patterns called Supervising Controller and Passive View.

Model

The model in MVP is just the same, that is described earlier in the Model-View-Controller pattern.

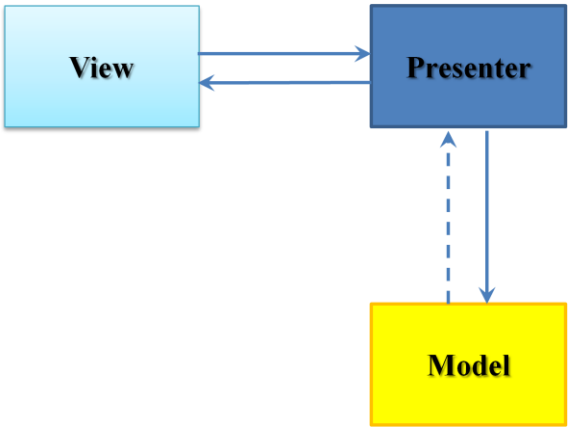
Presenter

The Presenter's responsibility is to handle user input and use this to manipulate the model data. The interactions of the User are first received by the view and then passed to the presenter for interpretation. Often, a presenter will maintain a selection that identifies a range of data in the model that is current, and the gestures received from the view will be used to act on this. Sometimes a view will adapt low-level gestures into higher-level commands before informing the presenter. The presenter is allowed to know about both its view and its model.

View

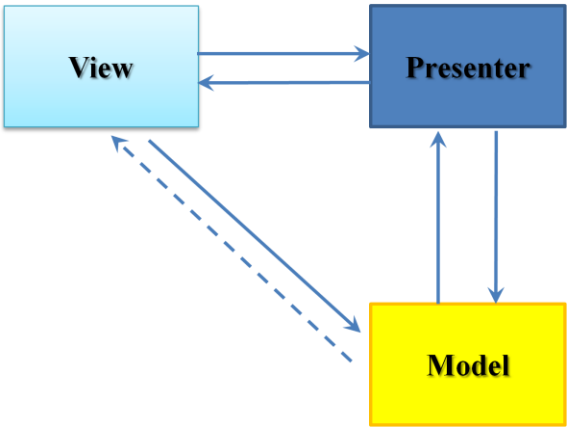
When the model is updated, the view also has to be updated to reflect the changes. View updates can be handled in several ways. The Model-View-Presenter variants, Passive View and Supervising Controller, specify different approaches to implementing view updates. In Passive View, the presenter updates the view to reflect changes in the model. The interaction with the model is handled exclusively by the presenter; the view is not aware of changes in the model.

In Supervising Controller, the view interacts directly with the model to perform simple data-binding that can be defined declaratively, without presenter intervention. The presenter updates the model; it manipulates the state of the view only in cases where complex UI logic that cannot be specified declaratively is required. .NET Databindig technologies make this approach more appealing for many cases, although if not carefully designed it may result in coupling between view and model.



Passive View:

- *The interaction with the model is handled exclusively by the presenter*
- *The View is updated exclusively by the presenter*

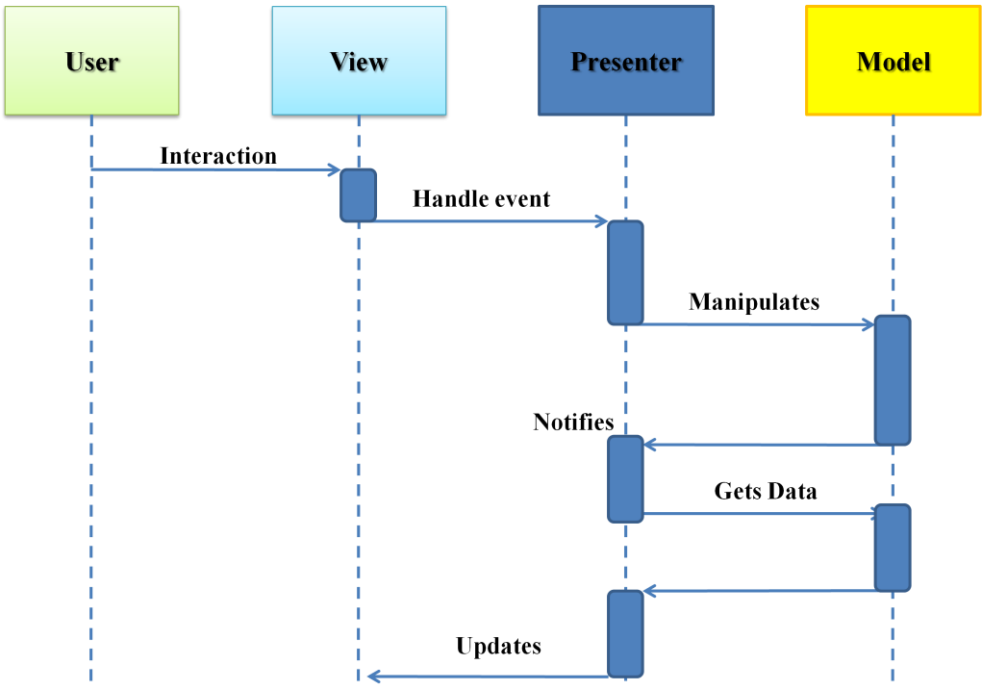


Supervising Controller:

- *The view interacts with the model for simple data binding*
- *The View is updated by the presenter and through data binding*

The control flow is generally as follows:

1. The user interacts with the user interface
2. The presenter handles the input event from the user interface via event handlers
3. The presenter performs business logic functionality on the model based on the user action, possibly resulting in a change in the model's state.
4. Update the view:
    - Passive View: The presenter is notified when the model's state changes, and then it gets the data from the model, and updates the view with it.
    - Supervising Controller: The presenter is notified when the model's state changes, and it notifies the view, which binds the data directly from the model.
5. The user interface waits for further user interactions, which restarts the cycle.



Sequence Diagram of MVP

Just like in case of MVC communication of layers is often defined by Interfaces. These interfaces are used inside the Model, View and Presenter components. This separation enables further decoupling by using the Factory pattern to get the actual implementation of layer components. If Factory pattern is coupled with IOC pattern, or with the usage of third party IOC containers, the actual application architecture can be declaratively configured, hence

making it easy to change Views (User interfaces), Models (Database versions), Presenter (business Logic) functions.

Advantages

- As with Model-View-Controller, Model-View-Presenter has the advantage that it clarifies the structure of our user interface code and can make it easier to maintain. Not only do we have our data taken out of the View and put into the Model, but also almost all complex screen logic will now reside in the Presenter.

- We have almost no code in our View apart from screen drawing code Model-View-Presenter also makes it theoretically much easier to replace user interface components, whole screens, or even the whole user interface (Windows Forms to WPF for example)). This makes the user opinion sensitive UI part perfectly separate of other parts of the application, hence it's features can be enhanced separately with more involvement from the customer.

- In addition unit testing the user interface becomes much easier as we can test the logic by just testing the Presenter.

Disadvantages

The disadvantages of Model-View-Presenter are similar to the disadvantages of Model-View-Controller:

- Requires high skilled experienced professionals who can identify the requirements in depth at the front before actual design.

- It requires the significant amount of time to analyze and design.

- This design approach is not suitable for smaller applications. It overkills the small applications.

- It can be hard to debug events being fired in active Model-View-Presenter.

- The 'Passive View' version of Model-View-Presenter can lead to a certain amount of boilerplate code having to be written to get the interface into the View to work.

- In both MVP and MVC cases the code for supporting the proper pattern implementation can be complex, hence it is not adviced to use in smaller projects.

MVVM

The Model-View-ViewModel (MVVM or ViewModel) is a pattern for separating concerns in technologies that use databinding. For Silverlight 2, it can help to make more maintainable applications by removing much of the code in the code-behind files and allowing full testing of business logic.

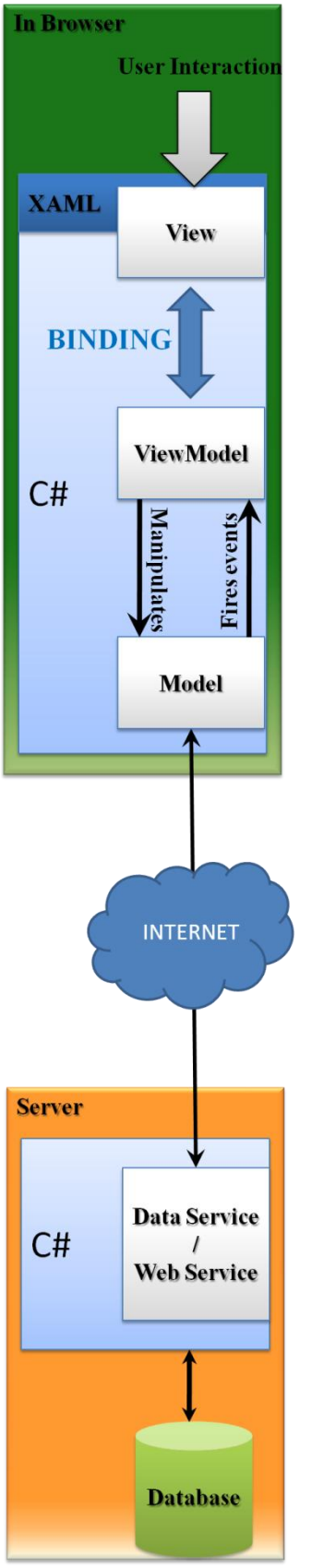
The first two pattern the MVC and MVP does not work as well as MVVM in declarative user interfaces like Windows Presentation Foundation or Silverlight because the XAML that these technologies use can define some of the interface between the input and the view (because data binding, triggers, and states can be declared in XAML).

The MVVM pattern is an adaptation of the MVC and MVP patterns in which the view model provides a data model and behavior to the view but allows the view to declaratively bind to the view model.

The view becomes a mix of XAML and C# (as Silverlight 2 controls), the model represents the data available to the application, and the view model prepares the model in order to bind it to the view.

Model

The model is especially important because it wraps the access to the data, whether access is through a set of Web services, an ADO.NET Data Service, or some other form of data retrieval. The model is separated from the view model so that the view's data (the view model) can be tested in isolation from the actual data.



Model-View-ViewModel

ViewModel

A ViewModel is a model for a view in the application as shown by its name. It has a collection which contain the data from the model currently needed for the view. Unlike the Presenter in MVP, a ViewModel does not need a reference to a view. The view binds to properties on a ViewModel, which, in turn, exposes data contained in model objects and other state specific to the view.

The bindings between view and ViewModel are simple to construct because a ViewModel object is set as the DataContext of a view. If property values in the ViewModel change, those new values automatically propagate to the view via data binding. When the user clicks a button in the View, a command on the ViewModel executes to perform the requested action. The ViewModel, never the View, performs all modifications made to the model data.

View

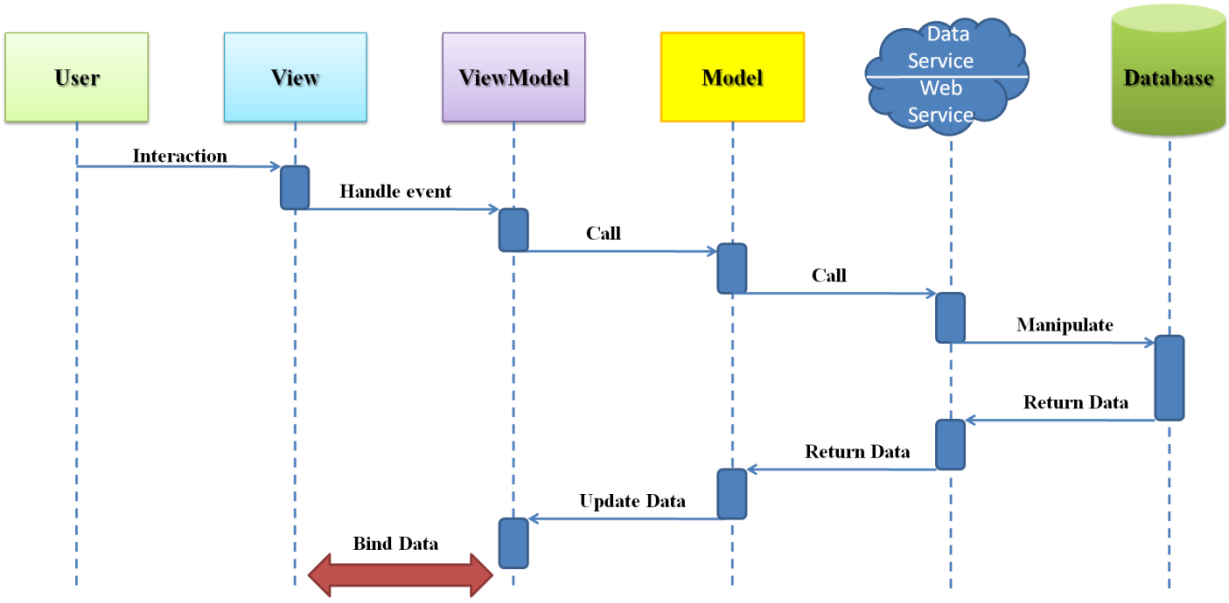
The pattern we use is to set the DataContext of a view to its ViewModel. The view classes have no idea that the model classes exist, while the ViewModel and model are unaware of the view. In fact, the model is completely oblivious to the fact that the ViewModel and view exist. A View is the actual UI behind a view in the application. The elements of the UI are implemented in XAML, and as a result of this the code behind file contains minimal code, or in some cases it doesn't contain code at all, however it is different in Silverlight and WPF

The implementation MVVM pattern in WPF projects is clearer than in Silverlight. The problem with using it in Silverlight 2 is that to make this pattern easy and seamless, Silverlight 2 really needs to support Commands and Triggers. If that were the case, we could have the XAML call the methods of the view model directly when the user interacts with the application. In Silverlight 2 this requires a bit more work, but luckily it involves writing only a little code. So finally, our View will contain some more code besides the code handling the user interface, but maybe in later versions it will be different.

The control flow is generally as follows:

1. The user interacts with the user interface
2. The ViewModel handles the event.
3. The ViewModel notifies the model of the user action.

4. The Model calls the Data/Web Service in order to manipulate the database.
5. The database changes its state and returns data
6. The model gets the data from the service, and passes it to the ViewModel
7. The View gets the data through DataBinding, and updates itself.
8. The user interface waits for further user interactions, which restarts the cycle.



Sequence Diagram of MVVM

Advantages

- Reduces the amount of code in the View's code behind file.
- UI elements can be written in XAML
- Strong Data Binding, which saves a lot of code. No need for manually refresh the view.

Disadvantages

- In bigger cases, it can be hard to design the ViewModel up front in order to get the right amount of generality.
- Data-binding for all its wonders is declarative and harder to debug than nice imperative stuff where you just set breakpoints

Implementation of a TrailerPlayer application

Application Use Case

In our application, the User can select a movie by clicking on it, and it will come to the center of the screen, and become bigger than others so the User can read the text better.

When the trailer is selected, the container of it gets two buttons on the bottom of the layout. The second button is the View Description button, and if the User clicks it, a short description about the movie fades in below the buttons. By clicking it again, the description fades out. By clicking the first button which is the Watch HD Trailer button, a media player floats in from the right before the information grid.

MediaPlayer in normal size

The media player has all the usual media player features, and even some more. There are the usual play/pause and stop buttons. It contains a time seeker, a volume slider and a mute button. There are two progress bars, one stands for the download progress, and it is behind the time seeker. The other is below it and displays the buffering status of the video in green.

The last button of the player is a full screen button, and by clicking it, the application goes to full screen, and the video can be enjoyed all over the screen, as it can be seen on the picture below. When in full screen mode, only the Player is visible and the information grid is hidden.

MediaPlayer in full screen

The controllers of the player fade in only if the mouse cursor is over them, otherwise they fade away, so they don't draw attention from the trailer. The time seeker can be dragged to any position, and this will cause the trailer to continue playing in the desired position. It takes some time though for the player to do (the) buffering.

Below the trailers the search control takes place. It is a simple text block and a search button. The search function uses a simple filter. The User can type in any text and the results will be the trailers which contain the word in the title, in the name of the director, in the cast, or in the type of the movie. There isn't any kind of list or checkbox or radio button where the User has to select the search type, like: "Title" or "Cast", because as Steven Krug said in his book "Don't make me think", so it is not a good thing if the user has to learn how to search on a page, or has to think about how the site's search engine wants him/her to phrase the question first.

By clicking "Search", the trailer list refreshes and will contain only the trailers, which are containing the search phrase, and the text below the search box will show how many trailers fulfill the search. If none of the trailers contains the search text, then the text "No Videos Found" will appear and the list won't change. If no text is entered, the list will contain all the trailers in the database.

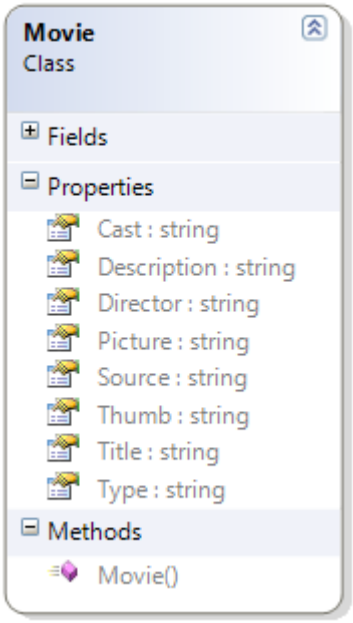
Architecture layers

In the following, we will examine the layers of the TrailerPlayer application implemented in MVVM pattern.

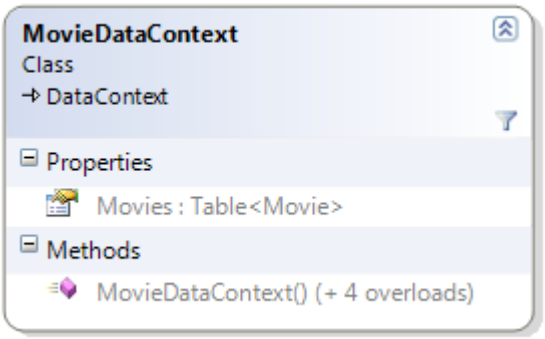
Database

The data for the application is supplied by a Microsoft SQL Server 2008 Express Edition. It contains the Movie database, which contains the Movie table. This has 8 columns:

- Title: The title of the movie
- Source: The url of the movie's HD trailer.
- Picture: The url of the movie's poster picture.
- Thumb: The url of the movie's thumb picture. This is a picture grabbed from the trailer, it can be seen when the playing of the trailer is stopped. The user can see something about the movie itself instead of a black screen.
- Director: The name of the director.
- Type: The Type/Genre of the movie.
- Cast: The names of the actors, separated by a space.
- Description: A short description of the movie.



The database is accessed from the application via LINQ to SQL classes. LINQ (Language Integrated Query) to SQL provides a runtime infrastructure for managing relational data as objects without losing the ability to query. The application is free to manipulate the objects, while LINQ to SQL stays in the background, tracking the changes automatically.



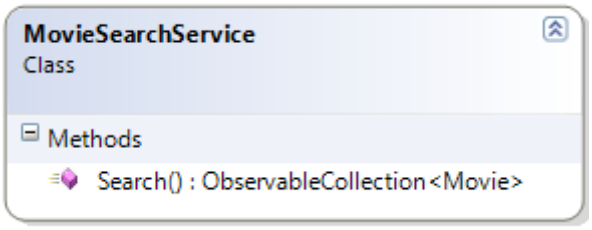
Web Service

The application is using a Windows Communication Foundation (WCF) web service to handle the communication between the Model and the Database. For Silverlight, we have a special WCF template named Silverlight-enabled WCF service.

In normal cases, the Service would contain an additional class, defining the Service interface; however in Silverlight this class is not needed.

Silverlight only supports basicHttpBinding, and the template is already configured to be used with Silverlight.

Server side



Our service is called MovieSearchService, and it contains a class which is also named MovieSearchService. The class is marked with the ServiceContract attribute. It has only one method, named Search, marked with the OperationContract attribute, which enables the method to be called remotely.

The method creates a MovieDataContext object, which handles the communication with the database. In C# 3.0 we can write the queries in C# using a little different syntax than usual, but we get full intellisense support this way.

The method has a string parameter, and depending on whether it is null or not, executes a query on the database. If it is null, the service returns all available data that from the database. If it isn't null, then it returns only the movies which contains the given string in their title, director or cast.

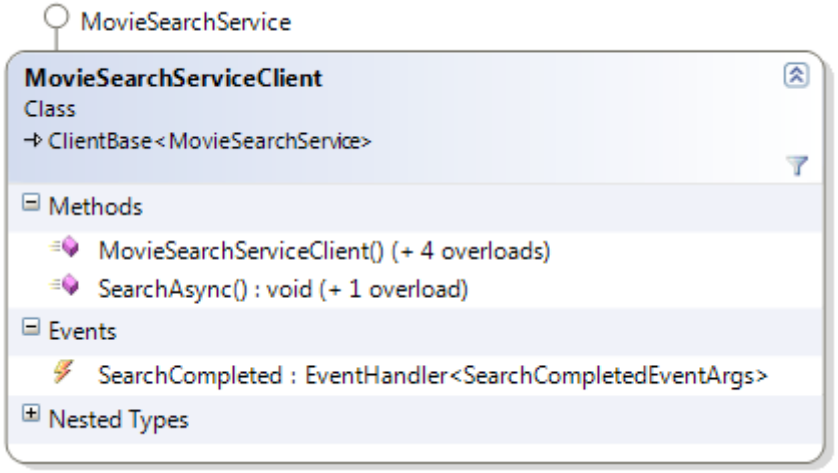
WCF services returns an ObservableCollection as default, but it can be changed in the properties of the service. ObservableCollection is a collection which implements the INotifyPropertyChanged interface.

The INotifyPropertyChanged contains an event named PropertyChanged. When used in context of databinding, the class that implements this interface can fire the event with the name of the property, which has changed as an argument, so the object which binds the property can change automatically.

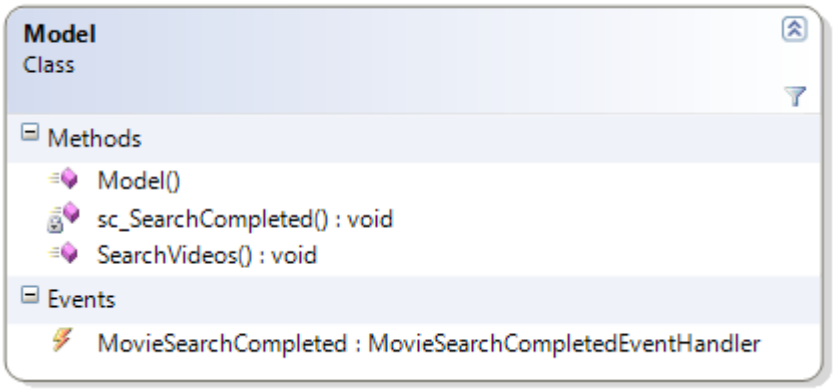
Client side

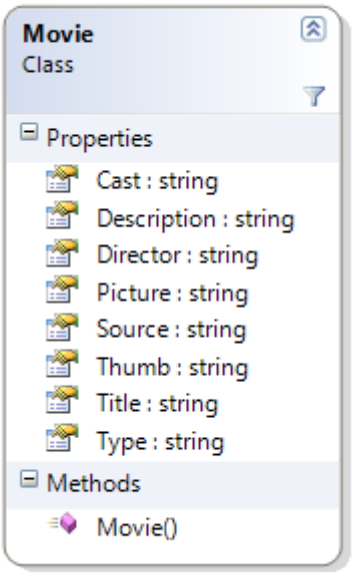
In order to use the service in the Silverlight Project, we need a service reference of the service. It can be done automatically by adding a Service Reference to the project. The only thing needed is a name for our service reference, and the URL of the service.

The Service Reference generates the code needed for the Service to be used from our Model class. We get the Movie object, synchronous and asynchronous methods of the original Search method, and events related to them.



Model



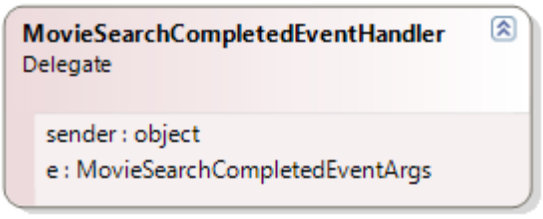


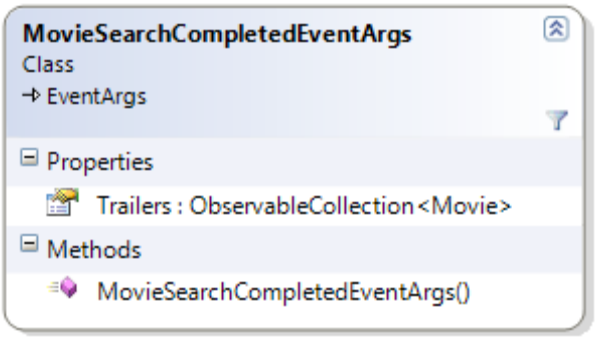
The Model layer of our application consists of three classes and a delegate, the first is a proxy class named Movie, it has all the properties that the other Movie class in our Service has, and the names are the same. This class is necessary, because as it was described before, in MVVM, only the Model communicates with the service, and if we wouldn't have a proxy class, we would have to use the other Movie class in our ViewModel through the service, which would break the architecture.

The other class named Model wraps the functionality of the MovieSearchService. It has a Search method with a string parameter, which makes an asynchronous call to the service.

When the model is created, it creates a MovieSearchServiceClient object, which handles the communication with the service. The model is subscribed for the service's search completed event, so when it is finished, the model can notify the ViewModel about it.

The MovieSearchCompletedEventHandler delegate in the model layer accepts a MovieSearchCompletedEventArgs object as a parameter, which is another class in the Model layer, and it is used to store the results of the service call. It gets an ObservableCollection as parameter.

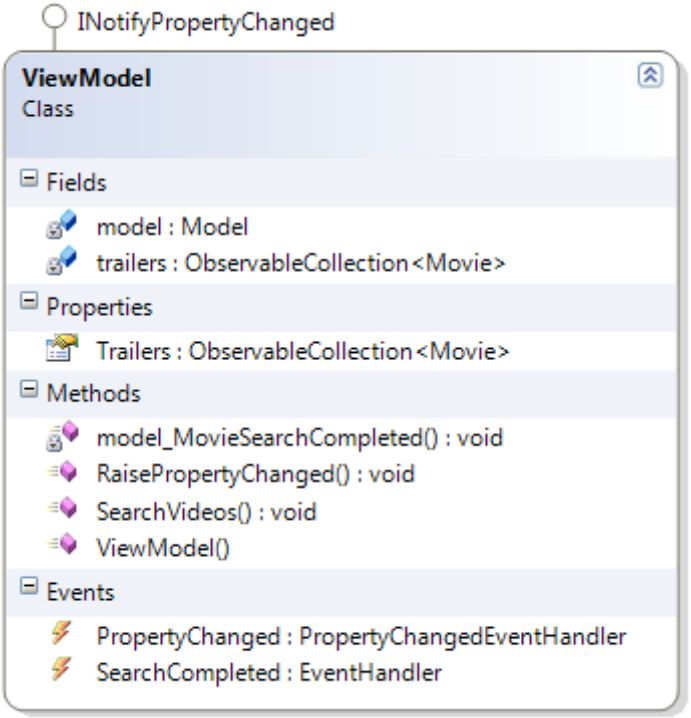




When the search is completed, the Model gets the results in an ObservableCollection which contains the service reference's Movie objects. The Model creates a new Observable-Collection filled with Movie objects of the Model layer, and then copies the results to this object, and finally fires a MovieSearchCompleted event, and passes the collection as an argument.

ViewModel

The ViewModel layer of the application contains only one class named ViewModel. It has a collection, which is an ObservableCollection containing the Model layer's Movie objects. The View will bind this collection. It implements the INotifyPropertyChanged interface. As a result of this, the ViewModel can notify the View when the Trailers collection changes.



It implements a helper method for event firing, it is the RaisePropertyChanged method. It is a preferred implementation, because we just have to pass the properties' names to this method, and then there is no need for testing whether the event is null or not every time. Moreover, any further logic can be added to the method if necessary.

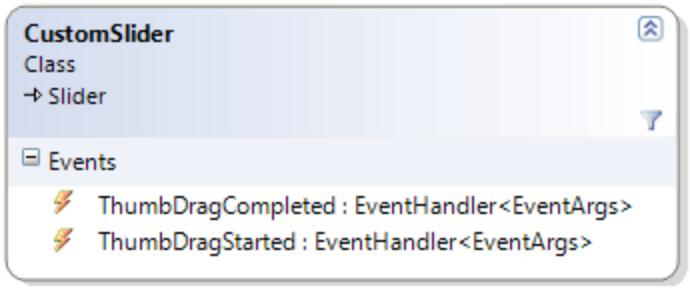
It has a Search function, which calls the model's search function. The ViewModel is subscribed for the MovieSearchCompleted event of the model, so when it changes, the ViewModel gets the results as an event argument. It changes its collection to the one got from the model, and then it calls the RaisePropertyChanged method, which raises the PropertyChanged event which notifies the View. It also fires a SearchCompleted event.

View

The View of the application consists of classes:

CustomSlider

CustomSlider is an extended slider control, its base class is the slider. The main slider does not contain events for handling the dragging of the thumb, and in the MediaPlayer we have to response to that event by positioning our video. The class captures the thumb's drag start and completed events, and fires its own events that can be captured in the player.



MediaPlayer

The MediaPlayer class is a fully implemented video player. It consist of many controls:

The first one is a MediaElement. This object is responsible for handling the video, almost every other control in this class is for manipulating the MediaElement class.

On the top of the MediaElement there is an Image control. This control displays the trailer's thumb image. When the user clicks on the image, the MediaElement gets its source, the Image fades away and the video starts.

The first button is the Play/Pause button. This is a toggle button. If the video is stopped, and the user clicks the button, the MediaElement gets its source, the Image fades away and the

video starts, or if it is already playing, then pauses it. If the user clicks the button while paused, it resumes the video.

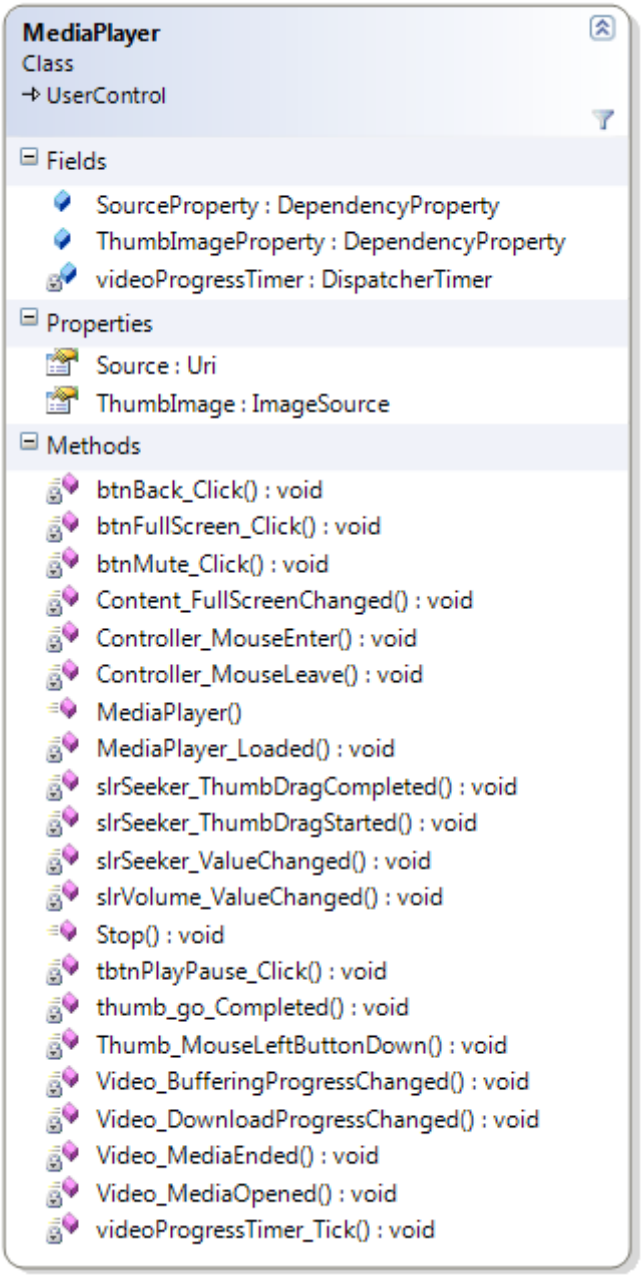
The second one is a Stop button, when it is clicked, it stops the video, clears the source, and the Thumb image fades in.

The seeker control in the middle controls the video's position. The User can drag it into any position, and the video repositions itself. The seeker follows the playback, and when it is stopped, it jumps to the beginning of the video.

The little button next to the seeker is a mute button, when toggled, it mutes the video.

The second slider is for adjusting the volume, when set to 0, it toggles the mute button as well. The default value is 50%.

The last button is the full screen button. When the user clicks it, the Silverlight plug-in goes to fullscreen, and the player adjusts its size to fit the whole screen.



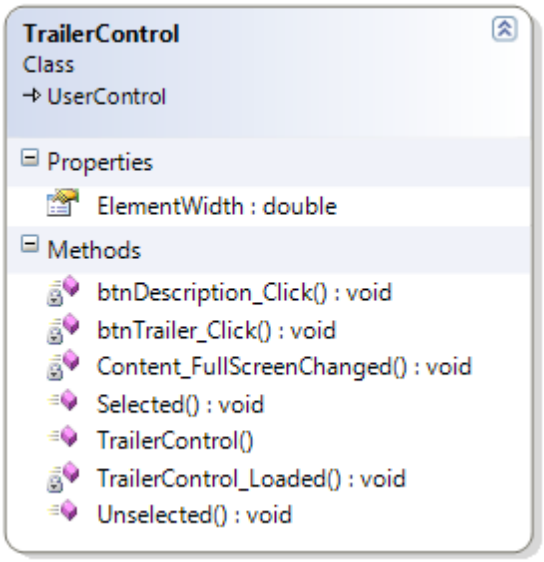
The MediaPlayer has two Dependency Propoerties. These are: Source and Thumb. They are shown in XAML, hence we can use them in declarative databind. The MediaElement's source gets the value of this source property when started, and the Thumb Image control gets the Thumb property when loaded.

TrailerControl

This control consists of an information grid, and a MediaPlayer Control. The information grid has six columns. They contain the following information: Title, Director, Cast, Type, Picture, the two control buttons and the Description.

Their content set in databinding:

`Text="{Binding Path=Title}"` etc. This kind of databinding means that whatever the DataContext of its parent will be, the control will bind the object's Title property, if there is any. This control knows nothing about what kind of object it will get. This is why the MediaPlayer has the additional Dependency Properties, because they are set just like the Title above. Moreover, if the DataContext changes, the control just rebinds it again, and changes automatically, we don't need to refresh it manually.



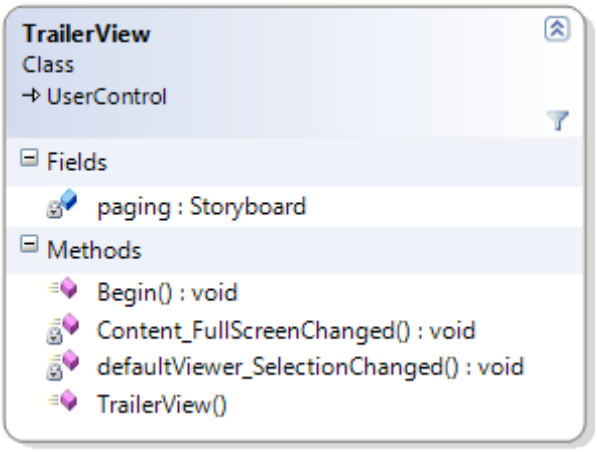
The Control has two control buttons, the first is the Watch HD trailer button. When toggled, the MediaPlayer floats in from the right, over the information grid. If the User clicks again, the MediaPlayer floats out. The second button is the View Description button, and if toggled the description text in the bottom fades in or out.

The TrailerControl has two public methods called Selected, and Unselected. Both of them play animations. The Selected runs when the User selects the trailer, and then the two Control buttons fade in and become visible.

When in fullscreen mode, the control hides its information grid, and only the MediaPlayer remains visible.

TrailerView

This control handles TrailerControl objects in a list. In Windows Forms a ListBox control contains listboxitem elements, while in XAML every container control contains whatever it gets. In this case it will contain TrailerControl elements. The DataTemplate of the ListBox is replaced with a TrailerControl object, and the



ItemsPanel template is also replaced. It contains only a horizontal StackPanel. The ConrolPanel template is bared down to a ContentPresenter object, so the default selection and other states don't work (like marking the selected item with a transparent blue layer).

The Templates in Silverlight can be replaced to almost anything, and this is a very powerful mechanism to build custom layouts in our applications. The developer and the designer has control over everything, they can replace, modify or delete elements. Every visible control can be used inside of an animation, and an animation has the highest precedence in dependency properties.

The ListBox's ItemSource property is set to this: `ItemsSource="{Binding}"` This means that we don't set exactly, what kind of objects it will get, but anything will be bound.

When the User selects a trailer from the list, it floats into the middle of the screen, and grows to its full size, while the unselected objects shrink to 80%. These are animations, one is static, the other is built-in code, depending on which trailer has been selected, and whether the plug-in is in full screen mode or not.

This would be a clear implementation of MVVM, if we would set the DataContext of this view to the ViewModel's collection, but in our case, we need to have control over the TrailerControl objects to invoke their Selected and Unselected methods. Because if we just set the DataContext, the ListBox would contain Movie objects, and not TrailerControl objects, despite that this control is the DataTemplate of the ListBox.

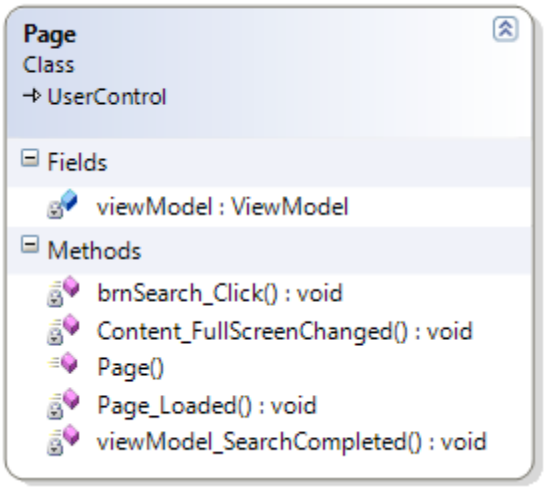
Because of this, the TrailerView has a public method called Begin, and it simply iterates through the DataContext, and creates a list of TrailerControl from it. Finally it sets the DataContext of the ListBox to this collection.

Page

The main control is the Page. This control is displayed when the application is loaded. The Page contains a TrailerView control, and a Search panel. This control is the real View in the application, it binds the ViewModel.

When the Page is loaded, it creates an instance of the ViewModel, and invokes the Search function of it with a null parameter to get a full list of the trailers, and sets its DataContext to the ViewModel. The TrailerView's DataContext is the following:

```
DataContext="{Binding Path=Trailers}"
```



This would be enough, if we wouldn't need TrailerControl objects, but since we do, we have to know when the Search is finished, so the page is subscribed to the SearchCompleted event of the ViewModel. In the handler method we set the Search panel's text to the number of trailers in the Trailers collection, and invoke the Begin method of the TrailerView.

The Search panel is for searching, the User can enter any text, and by clicking the Search button, the ViewModel's Search function is called given the text as parameter.

We don't have to reset the DataContext of the Page, nor the TrailerView, because the ViewModel is set as the DataContext of the Page, and since we implemented the INotifyPropertyChanged interface, it changes automatically. The TrailerView's DataContext changes automatically too, because DataContext inherits down in the visual tree.

Reusability of components in other patterns

MVC

The main differences between MVP and MVC patterns is, that in MVC the View gets the data indirectly from the Model, while in MVVM, the View binds the ViewModel. If we decide to use MVC instead of MVVM, we would have to do the following changes in the layers:

Model

Conclusion: As our current model layer conforms to the requirements of the MVC model, in order to use this layer using MVC, we don't have to change it.

ViewModel→Controller

We have to shift some parts of the ViewModel layer into the Controller layer, as in the MVC pattern, that layer is responsible for the responsibilities of the ViewModel. In MVC, the Controller only handles the View's events, and manipulates the model. We can do this using the following modifications:

1. We don’t need our Trailers collection.
2. We don’t have to implement the INotifyPropertyChanged interface.
3. We don’t need the SearchCompleted event either, because the View will get its data from the model. It leaves us with our Search method, where we simply call the Model’s Search method.

Conclusion: We still need a reference to the Model, but we don’t have to listen to its events in our Controller.

View

Using the MVC pattern, the View becomes a little more complex due to the fact, that we will most probably want to keep our data bindings, so we have to modify the Page and the TrailerView, as described in the following steps.

1. First we need a reference to the Model, and we have to listen to its MovieSearchCompleted event. When the event is fired, in the handler method, we get the results in a MovieSearchCompletedEventArgs class, which has a Trailers collection.
2. Since we don’t have INotifyPropertyChanged, we have to manually set the Page’s DataContext to the MovieSearchCompletedEventArgs object in our handler method.
3. We don’t have the SearchCompleted event anymore, so we have to handle the Search panel’s textblock in the MovieSearchCompleted event’s handler method too, and then finally call the TrailerView’s Begin method to refresh the View.

The data binding will work as before, because the TrailerView binds the Trailers property to whatever the DataContext of the page is.

Conclusion: The previously unbound events of the model now have to be subscribed by the view, and we have to manually reset the DataContext of the Page, and subscribe to and modify the events used by the Controller.

MVP

The main differences between MVP and MVVM patterns are, that MVVM gets the data by binding the ViewModel. In the Supervising Controller pattern, the View binds the Model, and

the Controller implements the logic only. In the Passive View pattern, the Presenter gets the data from the Model, and gives it to the View.

Passive View

Model

Conclusion: As our current model layer conforms to the requirements of the MVC model, we don't have to change this layer in order to use it with MVP.

ViewModel→Presenter

The ViewModel layer shifts some of its functionalities to the Presenter layer as described in the following steps.

1. First we need a reference of the Page, hence we need a constructor that accepts a Page as a parameter.
2. We don't need our Trailers collection, and we don't have to implement the INotifyPropertyChanged interface.
3. We don't need the SearchCompleted event, because the Presenter will give data to the View.
4. In the MovieSearchCompleted event's handler method, we have to set the Page's DataContext to the MovieSearchCompletedEventArgs object, and set the Search panel's TextBlock.
5. Finally we have to call the Begin method of the Page's TrailerView object.

Conclusion: We have to change the functionality of the ViewModel so that it corresponds to the responsibilities of the Presenter layer. This is a fairly straightforward transition by enabling it to hold a reference to the View of the View's interface. We also use a reference to the Model in the presenter layer.

View

We don't need the reference to the ViewModel, and we don't have to handle the SearchCompleted event in our Page. We only need to pass the Page to the Presenter in its constructor.

Conclusion: We have to make the View available for the Presenter layer.

Supervising Controller

Model

In the Supervising Controller, the View binds the model, which is due to the following:

1. We need a Trailer's collection, and we should implement the INotifyPropertyChanged event here.
2. We don't need the MovieSearchCompleted event nor the MovieSearchCompletedEventArgs class.
3. We need to implement the SearchCompleted event here.

Conclusion: The model has to take responsibilities of the ViewModel layer by simply transferring its functionalities.

ViewModel→Presenter

We don't need to listen to the Model's events, hence:

1. We don't need the SearchCompleted event here.
2. We have to move the Trailer's collection and the SearchCompleted event to the Model.

Conclusion: We have to disregard the events of the model in the presenter, and the ViewModel layer becomes the presenter layer.

View

The view should present and respond to changes in the Model layer directly, hence we need to do the following steps:

1. In the Page, we need a reference of the View,
2. We have to set the Page's DataContext to the Model instead of the Presenter.
3. We have to listen to the model's SearchCompleted event and handle it.

Conclusion: Using the available data binding, we have to change the binding contexts to the model layer objects, subscribe and handle events of the model.

Conclusion

In the current thesis, we have set two goals to prove. One was to demonstrate that Microsoft's Silverlight web development framework is usable with traditional standard web design patterns, especially the MVC, MVP and MVVM patterns. Our other goal was to measure and estimate, how much effort it takes to rewrite the MVVM application to MVC or MVP patterns.

We have documented and implemented the TrailerPlayer application as a proof of using the above mentioned design patterns. The code and the application are available attached to this document.

As we could see, the recommended approach we took was to implement a standard web application using the MVVM pattern. Although the application is simple and consists of only a few use cases, it is generally sufficient to establish general examinations during its transition to MVP and MVC patterns. We deliberately designed the TrailePlayer application according to Microsoft suggestions, the following attempts of transitions to other patterns out of mind. Despite all of the patterns offer separation of concerns and proper de-coupling of responsibilities using layers, most of the layers had to be rewritten. The following table summarizes the amount of changes needed in layers:

MVVM Layers	MVC	MVP (Passive View)	MVP (Supervising Controller)
Model	***None***	***None***	***Significant***
View	***Significant***	***Minor***	***Significant***
ViewModel	***Minor***	***Significant***	***Minor***

It is safe to conclude that the reusability of code written using MVVM patterns is highest in the MVC pattern. It needs somewhat more effort to reuse it in MVP when using the Passive

View approach. If requirements include it, it is possible to architect the classes of MVVM in a way that they can be reused in all three patterns.

Unfortunately, reusing the same components in MVP (Supervising Controller) pattern involves considerable rework in both the Model and the View layers. Since the ViewModel →Presenter holds the business logic, that is also prone to more frequent changes of requirements. All in all, the current MVVM Microsoft approach of Silverlight implementation is the least reusable in this pattern.

References

Silverlight 2

MATTHEW MACDONALD: PRO SILVERLIGHT 2 IN C# 2008

JEFF SCANLON: ACCELERATED SILVERLIGHT 2

http://msdn.microsoft.com/en-us/library/bb404700(VS.95).aspx

http://en.wikipedia.org/wiki/Silverlight

eXtensible Application Markup Language (XAML)

MATTHEW MACDONALD: PRO SILVERLIGHT 2 IN C# 2008

JEFF SCANLON: ACCELERATED SILVERLIGHT 2

http://msdn.microsoft.com/en-us/library/ms752059.aspx

http://en.wikipedia.org/wiki/Extensible_Application_Markup_Language

Windows Communication Foundation (WCF)

http://en.wikipedia.org/wiki/Windows_Communication_Foundation

http://msdn.microsoft.com/en-us/library/aa480210.aspx

http://www.codeproject.com/KB/WCF/WCFOverview.aspx

Model-View-Controller (MVC)

http://www.c-sharpcorner.com/UploadFile/napanchal/
MVCDesign12052005035152AM/MVCDesign.aspx

http://en.wikipedia.org/wiki/Model-view-controller

http://msdn.microsoft.com/en-us/library/ms978748.aspx

Model-View-Presenter (MVP)

http://msdn.microsoft.com/en-us/library/cc304760.aspx

http://en.wikipedia.org/wiki/Model_View_Presenter

http://martinfowler.com/eaaDev/ModelViewPresenter.html

http://www.object-arts.com/docs/index.html?modelviewpresenter.htm

http://www.codeproject.com/KB/architecture/ModelViewPresenter.aspx

Model-View-ViewModel (MVVM)

http://blogs.msdn.com/johngossman/archive/2005/10/08/478683.aspx

http://blogs.msdn.com/dancre/archive/2006/07/23/676272.aspx

http://msdn.microsoft.com/en-us/magazine/dd419663.aspx
http://weblogs.asp.net/craigshoemaker/archive/2009/02/26/hands-on-model-view-viewmodel-mvvm-for-silverlight-and-wpf.aspx
http://www.silverlightshow.net/items/Model-View-ViewModel-in-Silverlight.aspx
http://msdn.microsoft.com/en-us/magazine/dd458800.aspx

Appendices

TrailerPlayer Application Use Case Documents

UC1: Trailer Select

goal: Select an other trailer from the list of the trailers

summary: The selected trailer of the trailer list comes in the middle and additional buttons displays

actors: User

preconditions: The User clicks on another trailer in the list than the selected one.

1. The method creates the storyboard object which offsets the trailer list so that the selected trailer will be located in the middle of the screen.
2. The method invokes the Selected method of the selected Trailer's TrailerControl object, which grows the selected Trailer.
3. The method invokes the Unselected method of the deselected Trailer's TrailerControl object, which shrinks the deselected trailer, and makes the MediaPlayer and the View Description and the Wath HD Trailer buttons fade away if they are visible.
4. The View Description and the Watch HD Trailer buttons fade in on the selected Trailer's TrailerControl object.
5. The method returns the control.

UC2: Search

goal: Filter the Trailer list by Title,Director,Cast or Type

summary: The trailer list changes.

actors: User

preconditions: The User enters some text or enters nothing into the search textbox. The User clicks the Search button.

Basic Workflow:

1. The View invokes the ViewModel's Search method with the given string.
2. The ViewModel invoke's the Service's Search method with the given string.
3. The Sercvice executes the search in the database, and gets a list filtered by the given string. If the string is null the service gets all the trailers in the database.
4. The sercvice returns the list of the trailers to the ViewModel.
5. The ViewModel sets its Trailers collection to the changed list.

6. The ViewModel fires a SearchCompleted event.
7. The View is subscribed to the SearchCompleted event of the ViewModel, and when its fired, the View changes its TrailerView objects DataContext to the ViewModel's Trailers collection
8. The View invokes the TrailerView's Begin method, which iterates through its DataContext and fills its list collection with TrailerControl objects. Each TrailerControl object's DataContext gets a Movie object, and databinds it to the display.

UC3: Fullscreen

goal: Make the selected Trailer's video play change to or back from fullscreen

summary: The plugin goes to fullscreen, or exists fullscreen.

actors: User

preconditions: The User clicks on the FULL button

Basic Workflow:

1. The method examines wether the host is in fullscreen mode or not.
2. If not in fullscreen mode:
    a. The method creates the storyboard object which offsets the trailer list so that the selected trailer can be seen in fullscreen.
    b. The TrailerControl hides the DataView panel so that only the MediaPlayer is visible.
    c. The View resizez itself to be as wide and high as the screen
    d. The View hides the Search panel.
    e. The method returns the control.
3. If in fullscreen mode:
    a. The method creates the storyboard object which offsets the trailer list so that the selected trailer can be seen in the middle when the plugin exits fullscreen.
    b. The TrailerConrtrol displays the DataView panel.
    c. The View resizez itself to be as wide and high as original
    d. The View displays the Search panel.
    e. The method returns the control.