An Introduction to the Windows Presentation Foundation with the Model-View-ViewModel

Part 1

Paul Grenyer

After three wonderful years working with Java I am back in the C# arena and amazed by how things have changed. When I was working with C# previously it was with .Net 1.1 and as I return .Net 4 is ready to go. I started a new contract and my client suggested that to get ahead of the game I should learn Windows Presentation Foundation (WPF), the latest Microsoft framework for creating Windows desktop (and web) applications. It replaces the likes of Windows Forms on the desktop. Two of the major features of WPF are that it is rendered entirely on a computer's graphics card and separates presentation from presentation logic.

Manning is my preferred technical book publisher, so I bought the PDF version of WPF In Action with Visual Studio 2008 [WPFInAction] and read it on my Kindle. It is a great introduction to producing Graphical User Interfaces (GUIs) with WPF, but I later discovered that although Model-View-ViewModel (MVVM) is covered, the detail is not great. The MVVM pattern is similar to Martin Fowler's Presentation Model [Presentation model], but where the presentation model is a means of creating a UI platform-independent abstraction of a view, MVVM is a standardised way to leverage core features of WPF to simplify the creation of user interfaces. Fortunately there is a great MSDN Magazine article called WPF Apps With The Model-View-ViewModel Design Pattern [MVVM] that explains it simply and in a fair amount of detail.

Canon

Canon - Any comprehensive list of books within a field.

- dictionary.com

To demonstrate WPF with MVVM I am going to incrementally develop a small application which allows the user to search an archive of books. The application is called Canon and the source code [SourceCode] is available for download from my website. I developed Canon using Visual Studio 2010 and .Net 4, but WPF applications can also be created with Visual Studio 2008 and .Net 3.5. I have assumed that the reader is following along.

Fire up Visual Studio and create a new WPF Application called Canon. Build and run the application to make sure everything works correctly, and you should see a very simple window like the one shown in figure 1:

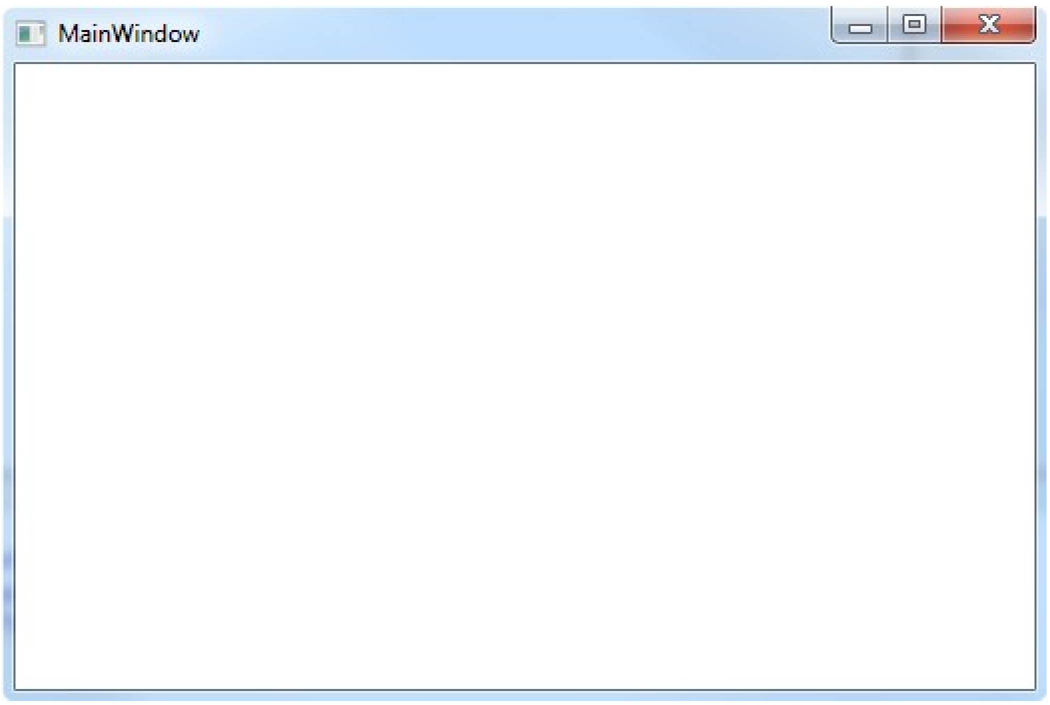


Figure 1: Default WPF Application Window

As with any normal window you should be able to minimise, maximise, resize and close it.

If you take a look at the project structure in Visual studio you'll see there appear to be just two source files, `App.xaml` and `MainWindow.xaml`. Actually there are four source files. If you use the arrow next to each file to expand it you will see that each `.xaml` file has a corresponding `.cs` file: `App.xaml.cs` and `MainWindow.xaml.cs`. I'll explain the relationship between all four files shortly, but first I want to put all views into a `View` folder and the `view` namespace. In Visual Studio, create a project level folder called `View` and move `MainWindow.xaml` into it. `MainWindow.xaml.cs` will come along with it. Then go into `MainWindow.xaml.cs` and change the namespace from `Canon` to `Canon.View`. Then go into `MainWindow.xaml` and modify the `x:Class` attribute of the `Window` element so that it reads:

```
<Window x:Class="Canon.View.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow"
        Height="350"
        Width="525">
```

Finally go into `App.xaml` and modify the `StartupUri` attribute of the `Application` element so that it reads:

```
<Application x:Class="Canon.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="View/MainWindow.xaml">
```

If you made all of these modifications correctly you should get the same window again when you build and run the application. Now is a good time to add the project to source control as we will only be adding to the structure from now on, rather than changing it.

Sidebar: Adding WPF Applications to Source Control

As with most Visual Studio solutions you need to ensure you check in all source files, and not binaries or other build artefacts. Source file include the `.xaml` and `.xaml.cs` files.

WPF Project Structure

WPF uses XAML (pronounced *zammel*), which stands for e**X**tensible **A**pplication **M**arkup **L**anguage, to layout User Interfaces (UIs). As we've seen all `.xaml` files have a corresponding `xml.cs` source file file. In *most* cases anything that can be defined in XAML can also be written in C# and vice-versa. Both files define the same class. It is not required to have both files, but in most projects there are some things you'll want to do in XAML and others in C#.

Let's start by taking a look at WPF's equivalent to `main`, `App.xml` and `App.xml.cs`, starting with `App.xml`:

```
<Application x:Class="Canon.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             StartupUri="View/MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

`App.xml` defines the WPF application with the `Application` element. The first attribute, `x:Class` specifies the namespace and name of the corresponding class, which is defined in `App.xml.cs`. The next two attributes bring in the necessary namespaces for XAML and the `StartupUri` attribute specifies the path to the main window's XAML file. The main window is the first window displayed by the application on start-up. The `Application.Resource` elements are for declaring resources for use within the application. WPF In Action contains an explanation and several examples of how and when they can be useful. We'll have a look at resources later when we want to load images into the Canon application.

```
namespace Canon
{
    public partial class App : Application
    {
    }
}
```

`App.xaml.cs` defines the C# part of the application class. As you can see the class name and namespace correspond to the name and namespace defined in the `x:Class` attribute of the `Application` element. The `App` class is partial. Part of it is defined in XAML, including the inheritance from the `Application` class, and part in C#. The `App` class does not have any fields or values as it is currently completely defined in XAML. We'll want to change this shortly when we inject a view model.

Now that we understand how a WPF application is defined let's take a look at how a window is defined by examining `MainWindow.xaml` and `MainWindow.xaml.cs`.

MainWindow.xaml is in the View folder we created earlier. Its name and location correspond to the value of the StartupUri attribute in the Application element in App.xaml. Therefore it is the first window that will be displayed.

```
<Window x:Class="Canon.View.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow"
        Height="350"
        Width="525">
    <Grid>

    </Grid>
</Window>
```

In the Window element the x:Class attribute specifies the name and namespace of the corresponding C# class and the next two elements bring in the XAML namespaces. The Title attribute specifies the title that is displayed in the window and the Height and Width attributes specify the height and width of the window. The Grid element declares the type of layout that the window will use to display its controls. I'll explain more about layouts when we create the UI controls later.

```
namespace Canon.View
{
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }
    }
}
```

MainWindow.xaml.cs defines the code behind the window. The class name and namespace correspond to the name and namespace defined in the x:Class attribute of the Window. The MainWindow class is partial as part of it is defined in XAML - including inheritance from the Window class - and part in C#. Inheriting from Window in the source file is therefore redundant and can be removed. The MainWindow class's only member is a constructor which calls InitializeComponents. InitializeComponents behaves in exactly the same way as it does in a Windows Forms application and initialises the components defined in MainWindow.xaml.

Injecting a ViewModel

Before we can inject a view model into a view we need an instance of a view to inject it into. To get the instance of the main window you can remove the StartupUri attribute:

```
<Application x:Class="Canon.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
...
</Application>
```

add a constructor to the App class and instantiate an instance of the view there instead. To actually display the main window you need to call Show on it.

```
public partial class App
{
    public App()
    {
        new MainWindow().Show();
    }
}
```

If you run the application again now (you need to add:

```
using Canon.View;
```

of course), you will see exactly the same window. All we've done is move the creation of the first window from XAML to C#. Now we have an instance of a window to inject a view model into.

A view model need be nothing more complex than a normal class. It does not require any special base class, interfaces or members. It's just about the data. Create a project level folder called `ViewModel` and create the following class in it (don't forget to add it source control):

```
namespace Canon.ViewModel
{
    public class MainWindowViewModel
    {
    }
}
```

Every WPF view has a `DataContext` property of type `object`. This property is `null` unless a view model is injected into the view. When a view model is injected WPF sees that `DataContext` is no longer null and uses it. We'll cover an example of simple binding shortly. The `DataContext` property is also available within the view. This means the view knows about the view model it has, but the view model continues to know nothing about the view that's using it. You can Inject the view model into the view by creating an instance of it and setting the `DataContext` property on the view:

```
public partial class App
{
    public App()
    {
        new MainWindow
        {
                DataContext = new MainWindowViewModel()
        }.Show();
    }
}
```

If you run the application again there will be no difference. Something in the view must be bound to a property in the model to see a difference in the UI.

A Slight Case of Over Binding

Binding is the WPF way of transferring values between a UI component and a property in a view model. It can be very simple or quite complex. Binding is explained in quite a lot of detail in WPF in Action[1].

I think the best way to demonstrate binding is with a simple example. In this one we'll bind the main window's title to a property in the view model. Let's start off by adding the property to the view model:

```
public class MainWindowViewModel
{
    public string AppTitle
    {
        get
        {
            return "Canon";
        }
    }
}
```

Once the binding is in place the main window will display the string returned by the `AppTitle` property. To bind the window title to the property we have to modify the `Title` attribute of the `Window` element in `MainWindow.xml` from:

```
<Window x:Class="Canon.View.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
```

to:

```
<Window x:Class="Canon.View.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="{Binding AppTitle}" Height="350" Width="525">
```

The curly braces tell WPF that we *do not* want to display the literal value. The key word `Binding` tells WPF we want to bind to a property in the view's view model and `AppTitle` is the name of that property. Remember that the `x:Class` attribute specifies a C# class and WPF knows it can bind to that class's `DataContext` property. If you run the application again now, you will see that the main window's title displays “Canon” instead of “MainWindow”.

The Canon Model

Now that we have a view and a view model, we need a book model for our archive:

```
namespace Canon.Model
{
    public class Book
    {
        public long? Id { get; set; }
        public string Title { get; set; }
        public string Author { get; set; }
        public string Publisher { get; set; }
        public string ISBN { get; set; }

        public Book()
        {
            Title = string.Empty;
            Author = string.Empty;
```

1 See chapter 11, Data binding with WPF

```
            Publisher = string.Empty;
            ISBN = string.Empty;
        }

        public override bool Equals(object obj)
        {
            if (ReferenceEquals(null, obj)) return false;
            if (obj.GetType() != typeof(Book)) return false;
            return Equals((Book)obj);
        }

        public bool Equals(Book other)
        {
            if (ReferenceEquals(null, other)) return false;
            return Equals(Id, other.Id);
        }

        public override int GetHashCode()
        {
            return Id.GetHashCode();
        }
    }
}
```

This simple `Book` class contains a unique nullable id for each book, its title, author, publisher and ISBN. If a `Book` instance is created with a null id it means that it is a new book. If the id has a value it means that the book has been saved before.

```
namespace Canon.Model
{
    public interface IBookRepository
    {
        Book Search(string searchTest);

        Book Save(Book book);
    }
}
```

The book repository interface, `IBookRepository`, contains two simple persistence methods, `Search` for searching for books and `Save` for saving books. Create a project level folder called `Model` and add the `Book` class and the `IBookRepository` interface to it. The view model will make use of the interface so add it as a field and a constructor parameter:

```
public class MainWindowViewModel
{
    private readonly IBookRepository repo;

    public MainWindowViewModel(IBookRepository repo)
    {
        this.repo = repo;
    }

    ...
}
```

This of course will prevent the project from building. To get it building again we need an implementing instance of `IBookRepository` to pass to `MainWindowViewModel`'s constructor. For this I knocked up a memory based mock implementation:

```
public class SimpleBookRepository : IBookRepository
{
    private readonly IList<Book> books = new List<Book>();

    public SimpleBookRepository()
    {
        Save(new Book { Title = "Redemption Ark",
                        Author = "Alistair Reynolds",
                        Publisher = "Gollancz",
                        ISBN = "978-0575083103" });

        Save(new Book { Title = "The C++ Standard Library",
                        Author = "Nico Josuttis",
                        Publisher = "Addison Wesley",
                        ISBN = "978-0201379266" });
    }

    public Book Search(string searchtext)
    {
        Book foundBook = null;
        foreach (var book in books)
        {
            if (SearchFields(book, searchtext))
            {
                foundBook = book;
                break;
            }
        }
        return foundBook;
    }

    public Book Save(Book book)
    {
        if (!book.Id.HasValue)
        {
            book.Id = getNextId();
        }
        else if (books.Contains(book))
        {
            books.Remove(book);
        }

        books.Add(book);
        return book;
    }

    private long getNextId()
    {
        long id = 0;
        foreach (var book in books)
        {
            id = Math.Max(book.Id.Value, id);
        }
        return id + 1;
    }

    private static bool SearchFields(Book book, string searchText)
    {
        searchText = searchText.ToLower();

        return  book.Title.ToLower().Contains(searchText) ||
                book.Author.ToLower().Contains(searchText) ||
                book.Publisher.ToLower().Contains(searchText) ||
                book.ISBN.ToLower().Contains(searchText);
    }
}
```

Sidebar: `SimpleBookRepository`

The `SimpleBookRepository` mock object is fairly straight forward. It persists a list of books in the `books` list:

Note that if the `SearchFields` method returns `true` the `Search` method knows it's found a matching book, stops iterating through the books and returns the current book. Of course there might be multiple matches, but the `Search` method only returns the first match.

The `Save` method can both update existing books and save new ones. New books are identified as having a `null` Id. If the book being saved *has* an id and *is already* in the book list, it is removed. This may seem a little odd. However, if the existing book is just added to book list it will be in there twice. Also remember that books are compared for equality by their ids. By the time the bottom of the `Save` method is reached the book has an id and does not exist in the book list, so it can be added without fear of duplication.

A `SimpleBookRepository` instance can be injected into the main window view mode as follows:

```
public partial class App
{
    public App()
    {
        new MainWindow
        {
                DataContext = new MainWindowViewModel( new SimpleBookRepository() )
        }.Show();
    }
}
```

and the project will build again. However there's still no change in the main window when the application is run.

Building the User Interface

Next we need a UI to manipulate the model. Figure 2 shows a very simple UI that can be knocked up with a few lines of XAML:

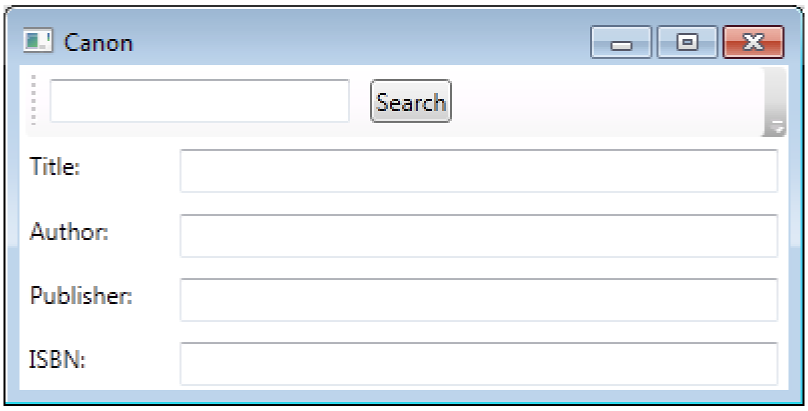


Figure 2: Canon User Interface Mk I

The first thing you might notice is that the Canon UI is smaller than the default window pictured in figure 1. This is because I modified the `Window` element in `MainWindow.xaml` to specify a starting height and width and a minimum height and width:

```
<Window x:Class="Canon.View.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="{Binding AppTitle}"
        MinHeight="200"
        Height="200"
        MinWidth="450"
        Width="450">
```

The window starts with a height of 200 and a width of 450 and can be expanded, however it cannot be contracted below 200 high and 450 wide. You're probably thinking that I could have just specified the minimums and you're right, I could have. However, the window would have started off somewhat bigger to begin with. Play around with different sizes until you get a feel for it.

WPF uses layouts for arranging controls on a UI. WPF layouts are a little bit like Java layouts. The window in figure 2 consists of a `DockPanel` layout, a `Grid` layout and a `StackPanel` layout. A `DockPanel` consists of five sections, top, bottom, left, right and centre. A section is only visible if a component is put into it. For example you could have a window with a tool bar across the top, a status bar at the bottom, an explorer view to the left, a help view to the right and a text editor in the middle. The Canon UI has a toolbar at the top that holds a `StackPanel` (another type of layout we'll look at in a minute) which in turn holds the search box and search button. The remaining space in the `DockPanel`, the centre section which gets the components added last to the `DockPanel`, holds a `Grid` layout with the rest of the UI components. To create a `DockPanel` simply declare it as a child element of the `Window` element.

```
<Window x:Class="Canon.View.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="{Binding AppTitle}"
        MinHeight="200"
        Height="200"
        MinWidth="450"
        Width="450">
    <DockPanel>

    </DockPanel>
</Window>
```

Next we want to add a tool bar and tell the `DockPanel` that we want to display it at the top:

```
<DockPanel>
    <ToolBarTray DockPanel.Dock="Top">
        <ToolBar>

        </ToolBar>
    </ToolBarTray>

</DockPanel>
```

Tool bars usually sit within a `ToolBarTray` which helps give them the usual Windows look and feel and can host multiple tool bars. To insert the `ToolBarTray` into the `DockPanel` you just make it a child element. You'll notice that the `ToolBarTray` inherits the `DockPanel.Dock` attribute from its parent and uses it to specify that the `ToolBarTray` should be displayed at the top. Child controls inheriting properties from their parents is a common occurrence throughout WPF and makes for far less verbose XAML. WPF In Action discusses this in more detail[2]. The `ToolBar` is a child of the `ToolBarTray`.

If you run the application again now you will see that the toolbar takes over the whole client area of the window. We only want it to be a thin strip across the top and we want the rest of the area to be a `Grid` layout. All we have to do is add a `Grid` to the `DockPanel`:

```
<DockPanel>
     <ToolBarTray DockPanel.Dock="Top">
        <ToolBar>

        </ToolBar>
    </ToolBarTray>
    <Grid>

    </Grid>
</DockPanel>
```

I'll discuss the `Grid` layout in more detail once we've completed the toolbar, but I wanted you to be able to run the application and see the toolbar across the top of the window and the empty `Grid` in the remaining client area. You'll notice that the dock position is not specified for the `Grid`. You can only specify a dock position of `Top`, `Bottom`, `Left` or `Right`. Any panel or control that does not have a dock position specified is placed in the centre section of the `DockPanel`. Child ordering effects positioning because the `DockPanel` iterates through its child elements in order, setting the position of each element depending on remaining space.

The toolbar consists of a text box that is used to enter a title, author, publisher or ISBN number to search for and a button to initiate the search. These can be placed directly into the `ToolBar`, but using `StackPanel` creates a better looking layout:

```
<ToolBarTray DockPanel.Dock="Top">
    <ToolBar>
        <StackPanel Orientation="Horizontal">
            <TextBox Margin="5,5,5,5" Width="150"/>
            <Button Margin="5,5,5,5" IsDefault="True">Search</Button>
        </StackPanel>
    </ToolBar>
</ToolBarTray>
```

A `StackPanel` stacks its children horizontally or vertically. This is ideal for us as we want to group the text box and button together in the tool bar. We want them horizontally, so we set the `Orientation` attribute to `Horizontal`. The `Vertical` orientation could also be used, but that would look rather odd. To add a `TextBox` and a `Button` to the `StackPanel`, just declare them as children. Both controls have a `Margin` attribute which puts a border around the *outside* of each control. Each comma delimited number specifies the spacing around the top, left, bottom and right of the control in that order. The text box's `Width` attribute speaks for itself. Without it the text box would be very narrow and would grow as content was typed into it. Setting the width gives it a sensible starting width and

2 See chapter 2, Working with layouts

maintains it. The button's IsDefault attribute is also set to true as we want the search button to be the default action. The text box's label is specified between the open and closing elements. This is also quite common for WPF controls. If you run the application you can enter text into the text box and click the button. The button does not do anything yet as it does not have a command associated, I'll discuss commands in the next section.

So far we've looked at the DockPanel and StackPanel layouts. These are two of the most important WPF layouts, but by far the most useful and therefore the most commonly used layout is the Grid layout. It has rows and columns like any other grid and allows you to to put any control in any sell or across many cells. In most cases rows and columns are defined using RowDefinition and ColumnDefinition elements:

```
<Grid IsSharedSizeScope="True">
    <Grid.RowDefinitions>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="auto"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition SharedSizeGroup="A"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>

</Grid>
```

Rows and columns can be defined just by placing the appropriate empty element (e.g. <RowDefinition/>) in the appropriate section. This would give the rows and columns a default height and width and is almost certainly *not* what you want. Setting the RowDefinition Height attribute to auto will adjust the height of the row to match the controls contained in each cell. This is ideal as all the rows in the Canon UI contain a label and a text box and are therefore all the same height.

The first column contains the labels for all of a book's fields and the second column holds the text boxes for the values of the fields. The cells in the first column should all be the same width as the longest label. To achieve this we set the Grid's IsSharedSizeScope attribute to true (the default is false) and set the first ColumnDefinition's SharedSizeGroup attribute. The name given to it is unimportant. If we had more than one column that we wanted to be the same width, we'd specify the same name in all of those columns. Without using shared size scoping we'd have to set a specific width for the column. We want the second column to take up the remainder of the UI's width, so we set its Width attribute value to an asterisk to tell it to stretch out as far as it can. All that's left is to put the controls into the cells:

```
<Grid IsSharedSizeScope="True">
    <Grid.RowDefinitions>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="auto"/>
        <RowDefinition Height="auto"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition SharedSizeGroup="A"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Label Grid.Column="0" Grid.Row="0">Title:</Label>
    <TextBox Grid.Column="1" Grid.Row="0" Margin="5,5,5,5"/>
    <Label Grid.Column="0" Grid.Row="1">Author:</Label>
    <TextBox Grid.Column="1" Grid.Row="1" Margin="5,5,5,5"/>
    <Label Grid.Column="0" Grid.Row="2">Publisher:</Label>
    <TextBox Grid.Column="1" Grid.Row="2" Margin="5,5,5,5"/>
    <Label Grid.Column="0" Grid.Row="3">ISBN:</Label>
    <TextBox Grid.Column="1" Grid.Row="3" Margin="5,5,5,5"/>
</Grid>
```

As you can see, each `Label` and `TextBox` has a `Grid.Column` and `Grid.Row` attribute that specifies its position in the `Grid`. As with a `Button`, the text for the Labels is specified between the opening and closing `Label` elements. Each of the text boxes also has a `Margin` set so that there is a reasonable gap between each of them.

Commands

The search text box and search button are closely related (but not coupled!). A user won't see the result of their search until they have typed something into the text box *and* clicked the button. The button shouldn't really be enabled unless there is content in the text box. To achieve this, we need to bind the text box to a property in the view model and bind the button to a command. I'll explain a bit more about WPF commands in a moment. To bind the search text box to a property in the view model, we first need the property:

```
public class MainWindowViewModel
{
    public string SearchText { get; set; }
    ...
}
```

and then add a bound text attribute:

```
<TextBox Margin="5,5,5,5" Width="150" Text="{Binding SearchText"/>
```

As with the `AppTitle` binding, `TextBox` binding is a simple case of using curly braces, the `Binding` keyword and the name of the property to bind too. The one difference is that the `SearchText` property has both a getter and setter. This means that as well as the value of `SearchText` being displayed in the search `TextBox`, any change to the search `TextBox` by the user is also written to the `SearchText` property. This is two way binding and is worked out by WPF automatically.

WPF has a version of the Command Pattern [CommandPattern]. WPF In Action describes WPF's implementation of the command pattern in detail and WPF Apps With The Model-View-ViewModel Design Pattern describes an `ICommand` based implementation that can be bound when using MVVM[3]. What we're interested in is binding a button to a command so that we can perform an action when that button is pressed and telling that button whether it should be enabled or not. This is the WPF Apps With The Model-View-ViewModel Design Pattern implementation:

```
public class RelayCommand : ICommand
{
    private readonly Action<object> execute;
    private readonly Predicate<object> canExecute;

    public RelayCommand(Action<object> execute)
```

3 See the section on Relaying command logic

```
        : this(execute, null)
    {}

    public RelayCommand(Action<object> execute, Predicate<object> canExecute)
    {
        if (execute == null)
        {
            throw new ArgumentNullException("execute");
        }

        this.execute = execute;
        this.canExecute = canExecute;
    }

    [DebuggerStepThrough]
    public bool CanExecute(object parameter)
    {
        return canExecute == null ? true : canExecute(parameter);
    }

    public event EventHandler CanExecuteChanged
    {
        add { CommandManager.RequerySuggested += value; }
        remove { CommandManager.RequerySuggested -= value; }
    }

    public void Execute(object parameter)
    {
        execute(parameter);
    }
}
```

Showing how it is used should provide enough explanation of it for our purposes. If you want to understand it in more detail see WPF Apps With The Model-View-ViewModel Design Pattern. Some people recommend lazy loading `RelayCommand` objects:

```
private RelayCommand _saveCommand;
public ICommand SaveCommand
{
    get
    {
        if (_saveCommand == null)
        {
            _saveCommand = new RelayCommand(...);
        }
        return _saveCommand;
    }
}
```

but I really don't see the need. It's a lot of extra code, including a `null` check and the property is accessed as soon as the window is displayed and bound anyway. So I just do this:

```
public class MainWindowViewModel
{
    ...
    public string SearchText { get; set; }
    public ICommand RunSearch{ get; private set; }

    public MainWindowViewModel(IBookRepository repo)
    {
        ...
        RunSearch = new RelayCommand(o => Search(), o => canSearch() );
    }

    private bool canSearch()
    {
        return !string.IsNullOrEmpty(SearchText);
    }

    private void Search()
    {

    }
    ...
}
```

The getter of the `RunSearch` property is public so that it can be bound to, but the setter is private so that it can only be set internally. The `RelayCommand` object itself is created in the view model constructor:

```
RunSearch = new RelayCommand( o => Search(), o => canSearch() );
```

Take another look at the `RelayCommand`'s two parameter constructor:

```
public RelayCommand(Action<object> execute, Predicate<object> canExecute)
```

The first parameter is an `Action` delegate, which encapsulates a method that has a single parameter and does not return a value. A lambda expression is used to specify the method to call when the command is executed. As it's a delegate you could do all sorts of in-line command implementations, but I find it clearer to delegate to another method. The second parameter is a `Predicate` delegate, which represents a method that defines a set of criteria and determines whether the specified object meets those criteria. A lambda expression is used to specify a method that determines whether the command should be enabled. (The `o` parameter is ignored as it is not needed in this scenario). To determine if the command should be enabled, we look to see if `SearchText` is *not* `null` or *is* empty:

```
private bool canSearch()
{
    return !string.IsNullOrEmpty(SearchText);
}
```

The next stage is to bind the command to the button. This is achieved by by adding a `Command` attribute to the search `Button` element:

```
<Button Margin="5,5,5,5" IsDefault="True" Command="{Binding RunSearch}">Search</Button>
```

If you run the application now you will see that the search button is disabled and does not enable until you enter something into the search text box *and* it loses focus. This is because, as with an edit box in a browser, the event which indicates that the contents have changed is not fired until the text box loses focus. To have the event fired every time the contents of the text box have changed, we need to modify its binding:

```
<TextBox Margin="5,5,5,5"
         Width="150"
         Text="{Binding SearchText, UpdateSourceTrigger=PropertyChanged}"/>
```

If you run the application again you will see that the button immediately enables or disables depending on whether the text box has content. However, when clicked the button still does nothing. In the next section we'll look at finishing the binding and getting books from the repository.

Before we move on to finish the binding there is an irritation about the UI we should fix. When the application starts, the focus is not on the search text box.

```
<Window x:Class="Canon.View.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="{Binding AppTitle}"
        MinHeight="200"
        Height="200"
        MinWidth="450"
        Width="450"
        FocusManager.FocusedElement="{Binding ElementName=searchBox}">
        ...
        <TextBox Margin="5,5,5,5"
                 Width="150"
                 Text="{Binding SearchText, UpdateSourceTrigger=PropertyChanged}"
                 Name="searchBox"/>
        ...
</Window>
```

As you can see, the `FocusedElement` of the `FocusManager` is bound to an `ElementName`, which must be specified. For this to work we have to set the `Name` attribute of the search `TextBox`. When you run the application the cursor will be waiting for you in the search text box.

Searching for Books

Before we can search for and display books, we need to bind the Title, Author, Publisher and ISBN text boxes:

```
public class MainWindowViewModel
{
    private readonly IBookRepository repo;

    public string SearchText { get; set; }
    public ICommand RunSearch{ get; private set; }

    public string Title { get; set; }
    public string Author { get; set; }
    public string Publisher { get; set; }
    public string ISBN { get; set; }
    ...
}
```

```
...
<Label Grid.Column="0" Grid.Row="0">Title:</Label>
<TextBox Grid.Column="1" Grid.Row="0"
       Margin="5,5,5,5" Text="{Binding Title, UpdateSourceTrigger=PropertyChanged}"/>
<Label Grid.Column="0" Grid.Row="1">Author:</Label>
<TextBox Grid.Column="1" Grid.Row="1"
       Margin="5,5,5,5" Text="{Binding Author, UpdateSourceTrigger=PropertyChanged}}"/>
<Label Grid.Column="0" Grid.Row="2">Publisher:</Label>
<TextBox Grid.Column="1" Grid.Row="2"
       Margin="5,5,5,5" Text="{Binding Publisher, UpdateSourceTrigger=PropertyChanged}}"/>
<Label Grid.Column="0" Grid.Row="3">ISBN:</Label>
<TextBox Grid.Column="1" Grid.Row="3"
       Margin="5,5,5,5" Text="{Binding ISBN, UpdateSourceTrigger=PropertyChanged}}"/>
```

```
...
```

The Title, Author, Publisher and ISBN text boxes use two way binding just like the search text box. If a book is found it is used to set the view model's properties:

```
private void Search()
{
    Book book = repo.Search(SearchText);
    if (book != null)
    {
        Title = book.Title;
        Author = book.Author;
        Publisher = book.Publisher;
        ISBN = book.ISBN;
    }
}
```

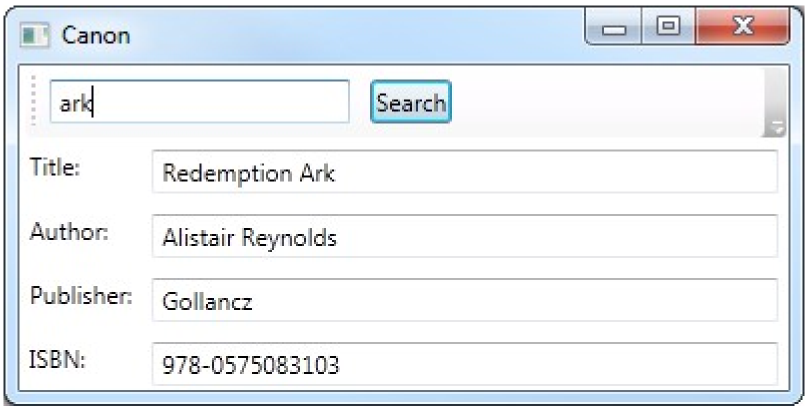
The above `Search` method uses the current value of the `SearchText` property to call `Search` on the repository. Remember that the view model's private `Search` method is called by the `RunSearch` command and the command can only be executed if the `SearchText` property is not `null` or empty. So by the time the `Search` method is called, `SearchText` is guaranteed to be valid. If a book is found a valid `Book` object is returned, otherwise `null` is returned. If a valid `Book` object is returned, its properties are used to set the view model's properties. However, if you run the application now you will be disappointed. Even if you enter a matching search criteria and click the search button, you will not see the Title, Author, Publisher or ISBN text boxes populated. This is because we haven't told WPF that the properties have changed. WPF will automatically register itself with a `ProperyChanged` event if one is provided:

```
public abstract class PropertyChangeEventBase : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

This implementation from WPF Apps With The Model-View-ViewModel Design Pattern is so useful that it's worth putting it in an abstract base class and then inheriting from it in the view model. This makes the `OnPropertyChanged` method available to the view model and when called fires an event with a `PropertyChangedEventArgs` object containing the name of the property that has changed. WPF picks this up and uses the appropriate binding to update the UI. You can see this if you modify the view model `Search` method as follows:

```
private void Search()
{
    Book book = repo.Search(SearchText);
    if (book != null)
    {
        Title = book.Title;
        Author = book.Author;
        Publisher = book.Publisher;
        ISBN = book.ISBN;

        OnPropertyChanged("Title");
        OnPropertyChanged("Author");
        OnPropertyChanged("Publisher");
        OnPropertyChanged("ISBN");
    }
}
```

Now if you run the application, enter a matching search criteria and click the search button, you will see that book details are displayed!



Title: Redemption Ark

Author: Alistair Reynolds

Publisher: Gollancz

ISBN: 978-0575083103

Figure 3: A Successful Search

Images

Currently the Canon application uses the standard icon in its title bar. It doesn't really make Canon stand out from any other Windows application. If you look on your (Windows 7 at least) task bar you'll see that all the open applications have an icon. If they all had the standard icon it would be difficult to tell them apart.

I use free icon libraries, like Silk Icon Set [SilkIcons], available on the internet for icons. I usual put images into an `Images` folder at the project level, so create one for the Canon project. Paste a suitable image (e.g. a 16x16 PNG) for the Canon icon into it and add the image to the project in the usual way. Make sure its Build Action property is set to `Resource`. Adding the image as an icon to the main window is done by setting the `Icon` attribute in the `Window` element:

```
<Window x:Class="Canon.View.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="{Binding AppTitle}"
        MinHeight="230"
        Height="230"
        MinWidth="450"
        Width="450"
        FocusManager.FocusedElement="{Binding ElementName=searchBox}"
        Icon="/Canon;component/images/lightbulb.png">
```

The format of the `Icon` attribute value is Microsoft Pack URI [PackURI] and consists of the following tokens:

> `/Canon` The name of the resource file, including its path, relative to the root of the referenced assembly's project folder.

`;component` Specifies that the assembly being referred to is referenced from the local assembly.

`/images/lightbulb.png` The relative path to the image file.

Menus and Tool Bar Icons

As it stands the Canon application is not very useful as it only allows us to search for the two preloaded books. What it needs to be able to do next is save updates to those books and create new ones. Save actions are often invoked by a menu and/or tool bar button or via a keyboard shortcut. Next I'll show you how to add a menu, with menu items bound to commands, which share an icon with a tool bar button we'll add to a new tool bar. First add a menu to the top section of the dock panel:

```
<DockPanel>
    <Menu DockPanel.Dock="Top">

    </Menu>
    <ToolBarTray DockPanel.Dock="Top">

    </ToolBarTray>

</DockPanel>
```

Menus are declared in their parent component with the `Menu` element. In the case of a `DockPanel` they also inherit the `DockPanel.Dock` attribute which is set to `Top` to put it in the same place as the tool bar tray. The order of child elements is important. If you put the menu below the tool bar tray the menu will *appear* below the tool bar tray. Add a drop down menu by adding a `MenuItem` with the `Header` attribute set:

```
<Menu DockPanel.Dock="Top">
    <MenuItem Header="_File">

    </MenuItem>
</Menu>
```

The underscore in front of the `F` in `File` specifies that `F` is the short cut key for the File menu. To add an item to the drop down menu, add a child `MenuItem` element:

```
<Menu DockPanel.Dock="Top">
    <MenuItem Header="_File">
        <MenuItem Header="_Save" Command="{Binding RunSave}"/>
    </MenuItem>
</Menu>
```

The `Header` attribute specifies the name of the item and the command binding is the same as a button command binding. We also need to add the command to the view model:

```
public class MainWindowViewModel : PropertyChangeEventBase
{
    …
    public ICommand RunSave { get; private set; }
    ...

    public MainWindowViewModel(IBookRepository repo)
    {
        ...
        RunSave = new RelayCommand(o => Save(), o => canSave());
    }

    private bool canSave()
    {
        return true;
    }

    private void Save()
    {}
}
```

The `canSave` method just returns true for the time being. We'll put it to better use later. Menu items can also have images and the same image can be used for a tool bar button too. You could repeat the location of the image for both the menu item and the tool bar button, but a better solution is to add a resource:

```
<DockPanel>
    <DockPanel.Resources>
        <BitmapImage x:Key="SaveImage" UriSource="/Canon;component/images/disk.png" />
    </DockPanel.Resources>
</DockPanel>
```

This resource is added to the dock panel. Resources can be added to most components and are in scope within that component and all of its children. Before you add the `DockPanel.Resources` element, make sure you add a suitable image, called something like `disk.png`, to the images folder the name must match the name specified in `UriSource`. You can add all sorts of resources including the `BitmapImage` shown above. The `x:Key` attribute specifies the name that the resource will be referred to by when it's used by other components. The `UriSource` attribute is the path to the resource. It also uses Pack URI.

The `MenuItem.Icon` and `Image` child elements are required to add an image to a menu item:

```
<MenuItem Header="_Save" Command="{Binding RunSave}">
    <MenuItem.Icon>
        <Image Source="{StaticResource SaveImage}"/>
    </MenuItem.Icon>
</MenuItem>
```

The image to use is specified by the `Source` attribute of the `Image` element which maps to the x:Key attribute of the resources. The image is bound to the resource, so uses curly braces. The resource is static as it is known at compile time, so uses the `StaticResource` keyword followed by the name of the resource. If you run the application now you will see the image next to the new menu item. The same image can be used as a tool bar icon. Add a new tool bar under the existing one. Add a button with a `Command` binding to the tool bar and an `Image` element that binds to the save image.

```
<ToolBarTray DockPanel.Dock="Top">
    <ToolBar>
        <StackPanel Orientation="Horizontal">
            ...
        </StackPanel>
    </ToolBar>
    <ToolBar>
```

```
        <Button Command="{Binding RunSave}">
            <Image Source="{StaticResource SaveImage}" />
        </Button>
    </ToolBar>
</ToolBarTray>
```

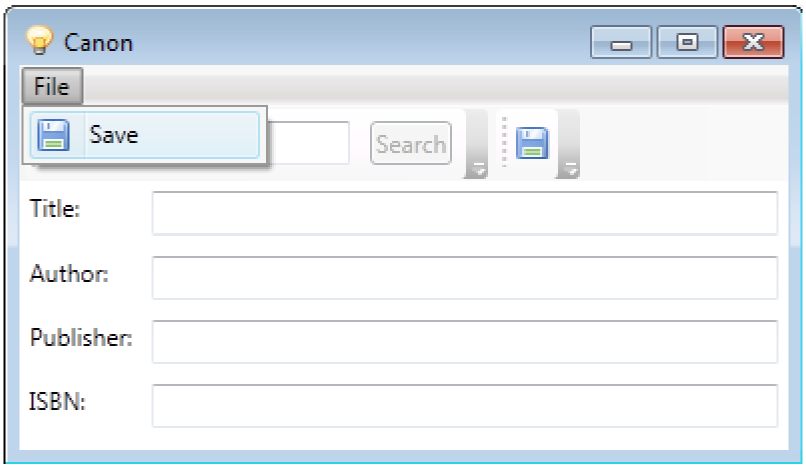


Figure 4: Menus and Icon Toolbar

The save menu item and button do not currently save. The simplest way to save a book is to create a new `Book` instance, initialise it from the UI fields and pass it to the `Save` method of the repository:

```
private void Save()
{
    repo.Save(new Book{Title = Title, Author = Author, Publisher = Publisher, ISBN = ISBN});
}
```

Can you spot the flaw? The `Id` is not set, which means every time you save a new book instance will be created, even if it has exactly the same field values as an existing one. To get around this, we need to keep a reference to the loaded book:

```
public class MainWindowViewModel : PropertyChangeEventBase
{
    ...
    private Book currentBook;
    ...

    public MainWindowViewModel(IBookRepository repo)
    {
        ...
        currentBook = new Book();
        ...
    }

    private void Search()
    {
        var book = repo.Search(SearchText);
        if (book != null)
        {
            currentBook = book;

            Title = book.Title;
            Author = book.Author;
            Publisher = book.Publisher;
            ISBN = book.ISBN;
            OnPropertyChanged("Title");
            OnPropertyChanged("Author");
            OnPropertyChanged("Publisher");
            OnPropertyChanged("ISBN");
        }
    }

    ...
    private void Save()
    {
        currentBook = repo.Save(new Book
        {
            Id = currentBook.Id,
            Title = Title,
            Author = Author,
            Publisher = Publisher,
            ISBN = ISBN
        });
    }


}
```

To hold the reference we add a book field called `currentBook` to the `MainWindowViewModel`. We default initialise it in the constructor to make sure it is valid even if a book has not been loaded yet. Then if we find a book when we search for one we set the `currentBook` reference to the new book. Finally when we save the new book we use the `Id` from `currentBook` to create a new book instance. After a successful save we set `currentBook` to the new book instance. Try it out and see if you can spot the further flaw.

The only way to create a new book is to enter values into all the fields and save before searching for a book and even then you can only do it once. What we need is a new book menu item, image and tool bar button:

```
<DockPanel>
    <DockPanel.Resources>
            <BitmapImage x:Key="SaveImage" UriSource="/Canon;component/images/disk.png" />
            <BitmapImage x:Key="NewImage" UriSource="/Canon;component/images/add.png" />
        </DockPanel.Resources>
        <Menu DockPanel.Dock="Top">
        <MenuItem Header="_File">
            <MenuItem Header="_New" Command="{Binding RunNew}">
                <MenuItem.Icon>
                    <Image Source="{StaticResource NewImage}"/>
                </MenuItem.Icon>
            </MenuItem>
                <MenuItem Header="_Save" Command="{Binding RunSave}">
                <MenuItem.Icon>
                    <Image Source="{StaticResource SaveImage}"/>
                </MenuItem.Icon>
            </MenuItem>
        </MenuItem>
    </Menu>
<ToolBarTray DockPanel.Dock="Top">
    ...
    <ToolBar>
        <Button Command="{Binding RunNew}">
             <Image Source="{StaticResource NewImage}" />
        </Button>
        <Button Command="{Binding RunSave}">
            <Image Source="{StaticResource SaveImage}" />
        </Button>
    </ToolBar>
</ToolBarTray>
```

and a new Command like `RunSave` and `RunSearch`. The difference with `RunNew` is that it does not need a `canNew` method as it is always permitted to create a new book:

```
RunNew = new RelayCommand(o => New());
```

You could create a `canNew` method hard coded to return `true` for consistency if you wanted too. The implementation of `New` looks like this:

```
private void New()
{
    Update(new Book());
}

private void Update(Book book)
{
    currentBook = book;

    Title = book.Title;
    Author = book.Author;
    Publisher = book.Publisher;
    ISBN = book.ISBN;

    OnPropertyChanged("Title");
    OnPropertyChanged("Author");
    OnPropertyChanged("Publisher");
    OnPropertyChanged("ISBN");
}
```

The Update method is duplication of the code in the `Search` method, so the `Search` method can be refactored to remove the duplication:

```
private void Search()
{
    var book = repo.Search(SearchText);
    if (book != null)
    {
        Update(book);
    }
}
```

If you run the application now you can create, save and search for new books.

System Commands

WPF supports a range of system commands for operations including cutting, copying and pasting. This means you can add standard functionality without having to implement the details. For example you can add an edit menu:

```
<MenuItem Header="_Edit">
    <MenuItem Header="Cut" Command="ApplicationCommands.Cut" />
    <MenuItem Header="Copy" Command="ApplicationCommands.Copy" />
    <MenuItem Header="Paste" Command="ApplicationCommands.Paste" />
</MenuItem>
```

You can of course add images and a corresponding tool bar in the way already described. Here we've replaced the command bindings with the system commands for cut, copy and paste. If you run the application you'll find cut, copy and paste just work as expected. WPF In Action [WPFInAction], the book in Introduced in part 1, goes into the system commands in more detail[4].

Not all system commands are as straight forward. Unfortunately if you add the system `Close` command to the file menu:

```
 <MenuItem Header="Close" Command="ApplicationCommands.Close" />
```

it is not enabled and does not close the application. What is missing is a command binding and handler methods:

```
<Window>
    <Window.CommandBindings>
        <CommandBinding Command="ApplicationCommands.Close" Executed="CloseCommandHandler"/>
    </Window.CommandBindings>
    ...
</Window>
```

The `CommandBinding` element uses its `Command` and `Executed` attributes to map the `Close` system command to the `CloseCommandHandler` handler, which is defined in the `MainMindow` class:

```
private void CloseCommandHandler(object sender, ExecutedRoutedEventArgs e)
{
    Close();
}
```

Clearly `CloseCommandHandler` just calls the `Close` method on the window to close it and consequently the application.

Detecting Changes

Do you remember that earlier on we implemented a not particularly helpful binding for the Canon window title? Do you also remember the `canSave` method that always returns `true`? It would be far more useful for the user to only be able to save when there were changes to be saved and for the window title to indicate when there are changes to be saved:

```
public string AppTitle
{
    get
    {
        return string.Format("Canon{0}", IsDirty ? " *" : "");
    }
}
...
private bool canSave()
{
    return IsDirty;
}
```

4 See chapter 10, Commands

IsDirty is a boolean property the indicates if any changes have been made. In the case of AppTitle it is used to determine whether an asterisk should be appended to the title when there are changes and in the case canSave it is just returned to indicate if the command should be enabled.

```
public bool IsDirty
{
    get
    {
        return  !currentBook.Title.Equals(Title) ||
                !currentBook.Author.Equals(Author) ||
                !currentBook.Publisher.Equals(Publisher) ||
                !currentBook.ISBN.Equals(ISBN);
    }
}
```

The IsDirty property compares the current book fields against the equivalent UI fields to determine if there are any changes. Unfortunately this leads to some more verbose changes to the UI field properties to get the title and save command to update in real time:

```
private string title = string.Empty;
public string Title
{
    get
    {
        return title;
    }
    set
    {
        title = value;
        OnPropertyChanged("Title");
        OnChange();
    }
}
...
private void OnChange()
{
    OnPropertyChanged("AppTitle");
}
```

I have only shown the changes for the Title property, but the Author, Publisher and ISBN properties must be changed in the same way. Instead of using the default property implementation we have to implement our own set method so that when the property is updated we can tell WPF to also update the window title. This means we also need to *separately* store the property value, which is initialised to an empty string to match the default Book instance, and implement a get method too. One advantage is that we can also move the WPF notification that the property has changed to the property itself so that we don't need to remember to call OnPropertyChanged anywhere else in the code where we assign the property. So the Update method is reduced to:

```
private void Update(Book book)
{
    currentBook = book;

    Title = book.Title;
    Author = book.Author;
    Publisher = book.Publisher;
    ISBN = book.ISBN;
}
```

The window title also needs to be updated when a book is saved as there are no longer any changes:

```
private void Save()
{
    currentBook = repo.Save(new Book
    {
        Id = currentBook.Id,
        Title = Title,
        Author = Author,
        Publisher = Publisher,
        ISBN = ISBN
    });
    OnChange();
}
```

Finally

This is where this article leaves the Canon application. There is more to do, but that falls outside the scope of an introductory article. Here I introduced you to simple WPF UI development and the Model-View-ViewModel pattern including simple binding and commands. Then I demonstrated how to make WPF GUIs more aesthetically pleasing with the use of images and more user friendly with the use of menus and toolbars and showed how to implement those menus and toolbars with custom and system commands.

In future articles I will cover unit testing and patterns for maintaining the separation between the view model and the view when you want to display message boxes and child windows or use custom controls.

References

[WPFInAction] WPF In Action with Visual Studio 2008 by Arlen Feldman and Maxx Daymon. Manning. ISBN: 978-1933988221

[Presentation Model] Presentation Model by Martin Fowler: http://martinfowler.com/eaaDev/PresentationModel.html

[MVVM] WPF Apps With The Model-View-ViewModel Design Pattern by Josh Smith. MSDN Magazine: http://msdn.microsoft.com/en-us/magazine/dd419663.aspx

[SourceCode] Canon 0.0.1 Source Code: http://paulgrenyer.net/dnld/Canon-0.0.1.zip

[CommandPattern] Design patterns : elements of reusable object-oriented software by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Addison Wesley. ISBN: 978-0201633610

[SilkIcons] Silk Icon Set from Mark James: http://www.famfamfam.com/lab/icons/silk/

[PackURI] Pack URIs in WPF: http://msdn.microsoft.com/en-us/library/aa970069.aspx