

Contents

- [The MVVM framework](#)
- [The Model](#)
- [The View](#)
- [The ViewModel](#)
- [Unit Testing](#)
- [Conclusion](#)

Windows Presentation Foundation (WPF) is Microsoft's replacement for the WinForms designer. It's a *huge* improvement over what we've been using for the past 20 or so years. It uses vector graphics to render the text-based XAML code that you build in the designer into screens that can do anything you've ever seen on a video display.

But WPF isn't just for designing beautiful user interfaces. It also permits *unit testing* of your code, which is about a hundred times faster than launching the app and drilling down to the screen you were just working on. This involves using a methodology called Model-View-ViewModel (MVVM), in which you move all code-behind to another class, called a ViewModel. You can include tests in the ViewModel classes that can be run without launching the corresponding forms. Add to that the ability to easily discover that the change that you just made broke some code elsewhere in your app, and MVVM is the *inexpensive way* to build desktop applications. And by the way, WPF apps will *always* cost less to build, and run faster, than HTML5 or any other kind of browser-based nonsense. So don't believe everything you hear.

However, MVVM is not easy. Many developers have read articles on MVVM and simply decided that it wasn't worth the trouble. Who would exchange the devil you do know for the devil you don't know?

The purpose of this article is to demonstrate a minimalist approach to MVVM that you can master and use in your next application. It doesn't have all of the bells and whistles that MVVM permits, but it's straightforward, and you absolutely can master it today. So let's get started.

About the author: I'm not an IdeaBlade employee, but when I develop, I use DevForce. - [Les Pinter](#).

Platform: WPF

Language: C#

Download: [mvvm-forms-les-pinter-610.zip](#)

The MVVM framework

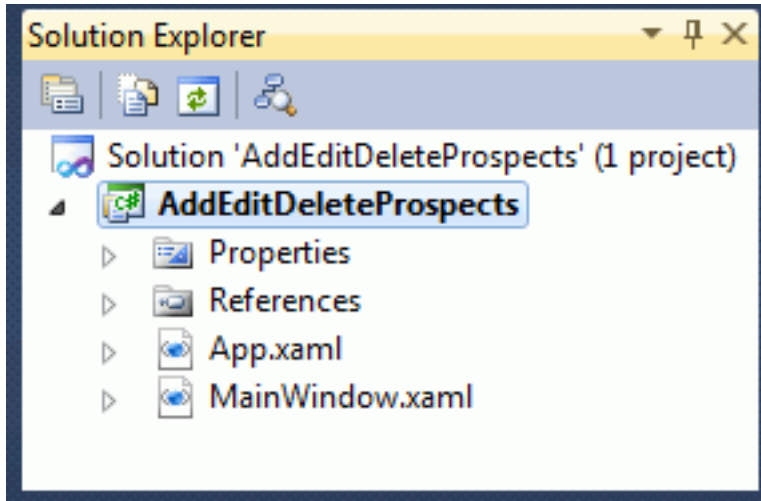
MVVM is the acronym for Model-View-ViewModel. A View is a form; a ViewModel is a class that holds the code that used to go in the form's CodeBehind file; and a Model is a class that retrieves and stores your data, and (most importantly) exposes it to your View (form) via WPF **Binding**. So the data goes from the Model into *public properties* in the ViewModel. Your View then uses binding to retrieve the data that's now in the ViewModel object, and to send changes back to it. The ViewModel then calls methods on the Model to store the data. The methods to retrieve and store data are in the ViewModel, not in the form - er, View. They're connected using something called Commanding, which requires about 4 lines of code per CommandButton.

This means that you can run unit tests against the ViewModel without launching the View, because the code that gets data and handles events like button-clicks isn't in the View's CodeBehind. And that's the whole point. You don't need to launch the form to exercise your code. So a testbench like [NUnit](#) can test your code and validate it without running the program. And did I mention that it's fast?

When I first heard about MVVM, I immediately sat down to write an Add/Edit/Delete form, the heart of all database applications. I read dozens of articles about MVVM, and surprisingly, none of them got me to the point of a working application. They went on and on about the wonders of MVVM, but they didn't show me how to add or delete a record. They reminded me of the super salesman whose new bride left him the morning after their first honeymoon night, because he sat at the end of the bed all night telling her how great it was going to be. Just get on with it, okay?

So let's get on with it.

In VS 2010, select **File, New, New Project**, and select **WPF Application**, designating a location that you'll be able to find easily later. Create a new project called **AddEditDeleteProspects** of type WPF Application. You'll see the following project in the Solution Explorer:



You'll probably want to use `MainWindow` as a splash screen and to display your main menu. You probably should change the name by renaming it, and I've done so, changing the file name to `frmStartup`. Note that if you rename it, you'll have to edit `App.xaml` and change the `StartupUri` to the new .xaml file name - it's not automatic. Note further that even though you change the file names, internally the class name and its constructor name are still `MainWindow`. So, if you also change their names internally, be sure to do so both in the .xaml and in the .xaml.cs files.

The next steps will be to add a model to describe each of the tables in your database, and then to add one View and one ViewModel for each of the tables that require a user maintenance form. You can share a ViewModel among several forms, and sometimes that's appropriate; but usually there will be exactly one ViewModel per View. A ViewModel is a class that interacts with the View, and a Model is a class that represents your data. You write the ViewModels, but IdeaBlade Devforce writes the Models. More on that later.

But before we add our Model, there's one class that we'll need; it's called `CommandMap.cs`. It's available in the download for this article. Register (it's free), and when you reload the article, the 'Log in or register' link will instead be a [download the source code](#) link. In the download, you'll find this file. This little class makes *commanding*, the process of connecting the click events of your commandbuttons (and other events) to code in the ViewModel, very easy. If I didn't include it, I'd have to go into a lengthy explanation about commanding, and that's usually where readers' eyes start to glaze over. Instead, you'll add three lines of code in each ViewModel, then add one more, plus the code for the method to execute when the button is clicked, for each command. That's simpler.

Copy `CommandMap.cs` into your project, then click on the *Show All Files* icon at the top of the Solution Explorer, highlight the new file, and then right click on it and select *Include in Project* from the resulting context menu. Look at the code if you want to, but it's not necessary, except to discover that the internal namespace is `CommandUtilities`, which you'll be using in your ViewModels. But first, let's add a Model.

The Model

I have a table called `Prospects` that's used in this project. In the download for the article, there's a SQL script that creates a database called `Tests`, then adds and populates a table called `Prospects`. I used SQL 2008 R2, which does constraints a little differently. The version shown below will also work with earlier versions of MS SQL.

Listing 1: Creating the test database and table:

```
CREATE DATABASE TESTS;
GO;
USE TESTS;
CREATE TABLE [dbo].[Prospects](
  [ID] [uniqueidentifier] NOT NULL PRIMARY KEY DEFAULT NewID(),
  [FirstName] [varchar](50) NOT NULL DEFAULT "",
  [LastName] [varchar](50) NOT NULL DEFAULT "",
  [Address] [varchar](50) NOT NULL DEFAULT "",
  [City] [varchar](50) NOT NULL DEFAULT "",
  [State] [char](2) NOT NULL DEFAULT "",
  [ZIP] [varchar](10) NOT NULL DEFAULT "");
GO;
INSERT INTO Prospects VALUES ( 'Les', 'Pinter', '34616 Highway 190', 'Springville', 'CA', '93265' );
INSERT INTO Prospects VALUES ( 'Sam', 'Schulman', '1232 Nasa Road 1', 'Webster', 'TX', '77342' );
INSERT INTO Prospects VALUES ( 'Oren', 'Springer', '22202 Westview', 'Houston', 'TX', '77024' );
INSERT INTO Prospects VALUES ( 'Venita', 'Cunningham', '1420 Columbia', 'Houston', 'TX', '77002' );
```

The choice of a GUID (a Globally Unique Identifier) is a good one, because it simplifies the insertion of a new record key. If you use an integer identity column, in a multiuser environment you're faced with reserving the next sequential number to be set aside while the user finishes entering the data for the new prospect, and then either inserting the record or canceling it. Usually a temporary negative integer is used, but it's a hassle no matter how you deal with it. GUIDs are the simplest way to supply a unique key. *NewID()* is SQL's Guid generator, and the database table creation statement shown above uses it; but in our case, we'll just assign one in code, as you'll see below.

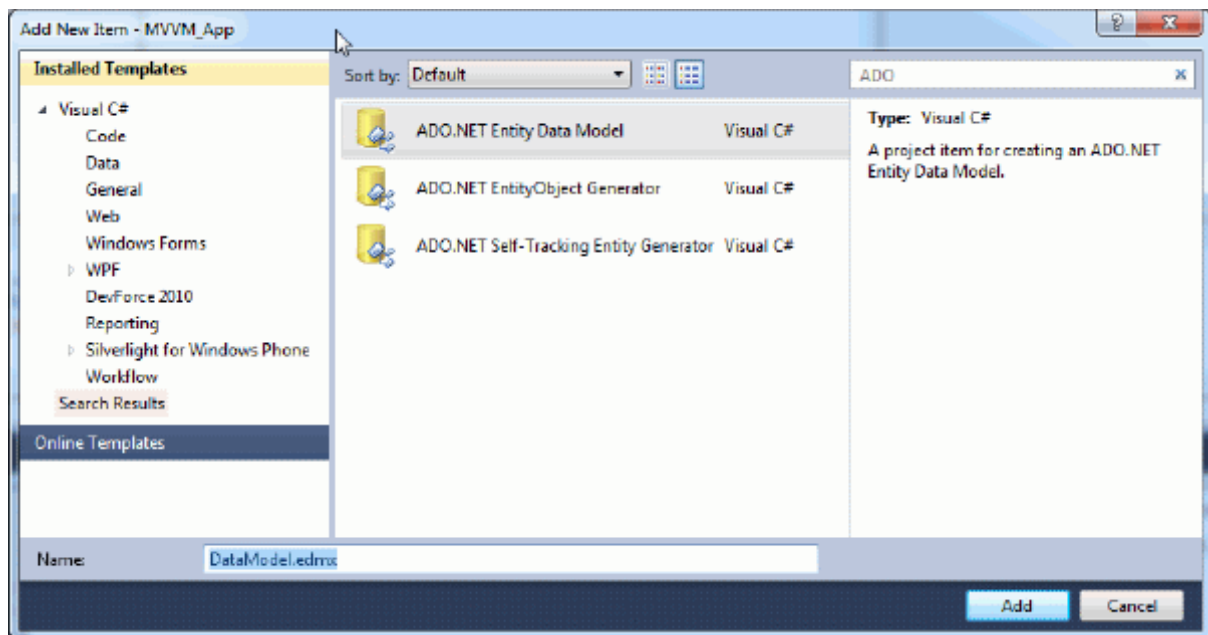
Before we go any further, at the end of this article I'm going to build a form to search for a particular Prospect and display it in the Prospects form. To do this, I've declared a public static *Guid field* in App.xaml.cs to hold the user-selected Prospect ID in *frmFindAProspect*, and then use it to retrieve the corresponding prospect when I return from the search form. Add just one line in App.xaml.cs, just after the class declaration statement (the "?" means that the field can be *null*):

```
public partial class App : Application
{
    public static Guid? ID = new Guid(); // <== add this line...
}
```

Note: You can also create a partial class for the Prospects entity, and then add this code in the Create method, and it will happen every time you create a new record. But this is an article about MVVM, not about IdeaBlade DevForce, which can do more than you'll probably ever need or even want to know about.

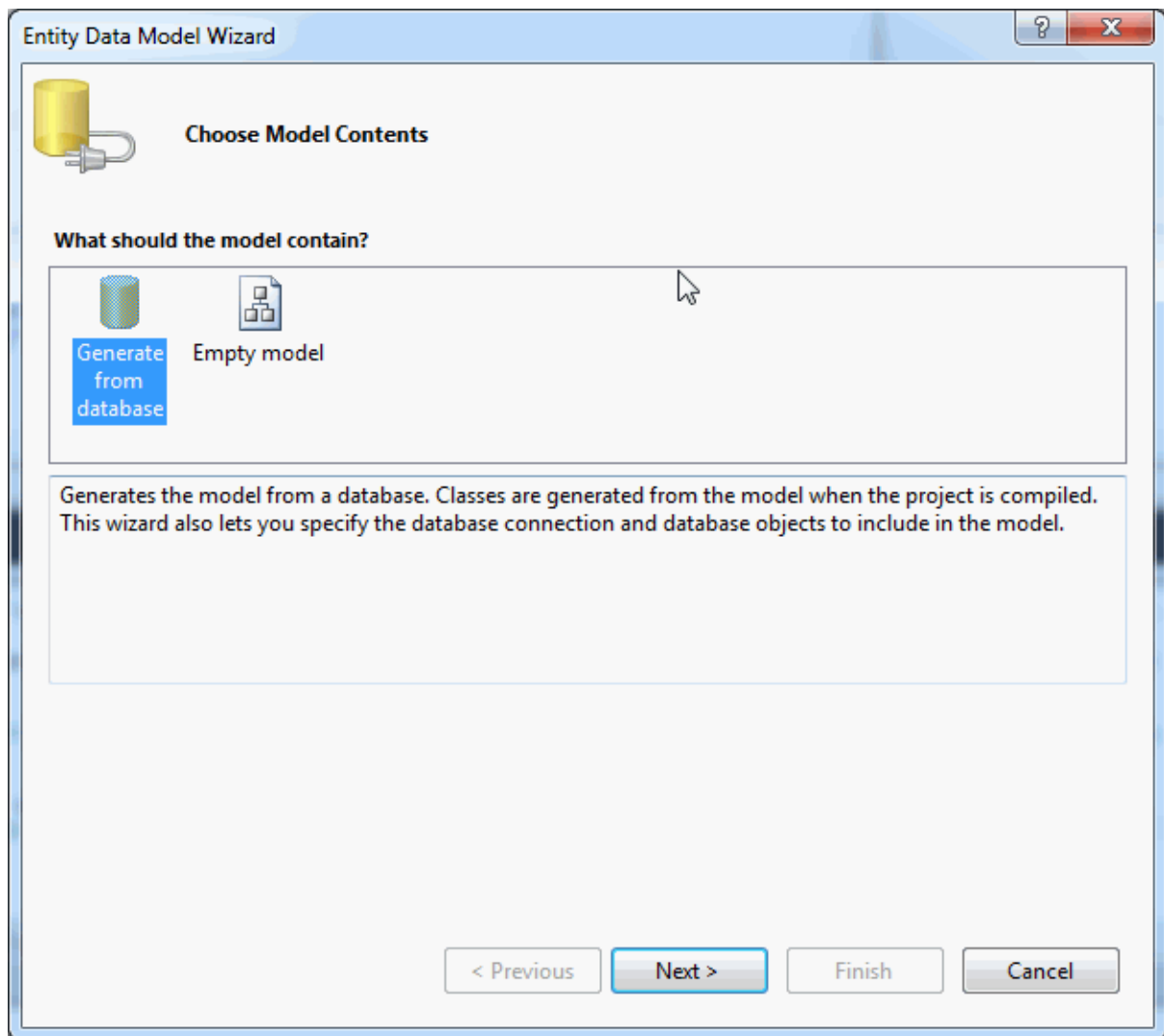
Now, if you haven't already done so, download the demonstration version of their [DevForce 2010](#) ORM tool. It doesn't time out. It will only support up to ten tables, which makes it useful only for learning purposes. If you have an application with hundreds of tables, the value of the application will justify the grand or so that it costs. But the learning version is free.

To see DevForce in action, right click on your project in the Solution Explorer and select **Add, New Item** from the context menu. Up in the **Search Installed Templates** box, type **ADO**. You'll see just a few selections.

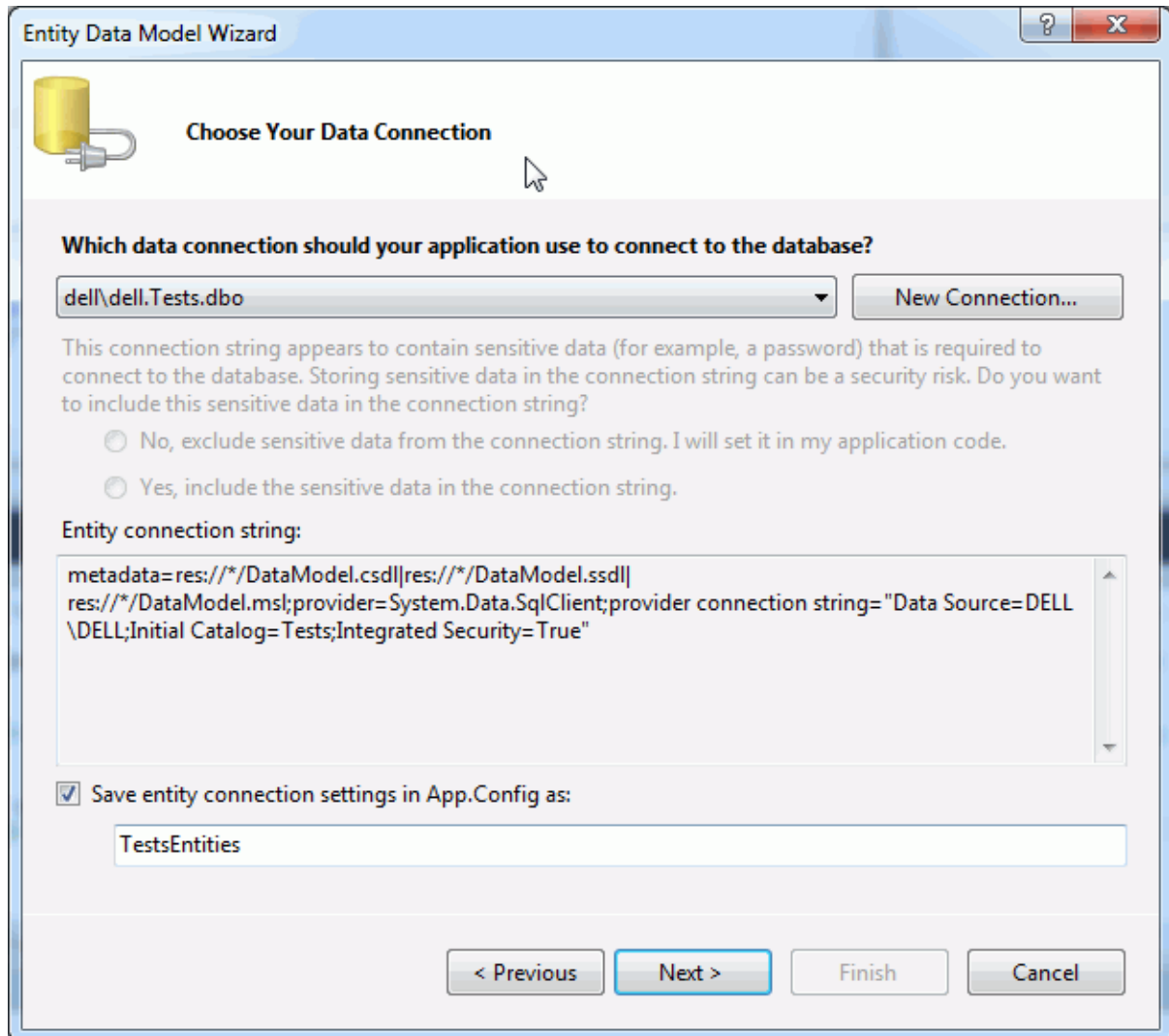


Pick the first one, **ADO.NET Entity Data Model**. Name it DataModel at the lower left of the screen and click on **Add**.

You'll be asked if you want to generate the model from the database, and you absolutely do.



You'll be asked to either select a connection to the Tests database or to create a new one. The connection string that it creates is in entity-framework format, so it might look a little strange:



The image shows a screenshot of the 'Entity Data Model Wizard' window, specifically the 'Choose Your Data Connection' step. The window has a title bar with a question mark and a close button. Below the title bar is a yellow cylinder icon with a plug. The main heading is 'Choose Your Data Connection'. Below this is a question: 'Which data connection should your application use to connect to the database?'. There is a dropdown menu showing 'dell\dell.Tests.dbo' and a 'New Connection...' button. Below the dropdown is a warning message: 'This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?'. There are two radio buttons: 'No, exclude sensitive data from the connection string. I will set it in my application code.' and 'Yes, include the sensitive data in the connection string.'. Below the radio buttons is a text box labeled 'Entity connection string:' containing the following text: 'metadata=res://*/DataModel.csd|res://*/DataModel.ssd|res://*/DataModel.msl;provider=System.Data.SqlClient;provider connection string="Data Source=DELL \DELL;Initial Catalog=Tests;Integrated Security=True"'. At the bottom, there is a checkbox labeled 'Save entity connection settings in App.Config as:' which is checked. Below the checkbox is a text box containing 'TestsEntities'. At the very bottom are four buttons: '< Previous', 'Next >', 'Finish', and 'Cancel'.

Entity Data Model Wizard

Choose Your Data Connection

Which data connection should your application use to connect to the database?

dell\dell.Tests.dbo New Connection...

This connection string appears to contain sensitive data (for example, a password) that is required to connect to the database. Storing sensitive data in the connection string can be a security risk. Do you want to include this sensitive data in the connection string?

☐ No, exclude sensitive data from the connection string. I will set it in my application code.

☐ Yes, include the sensitive data in the connection string.

Entity connection string:

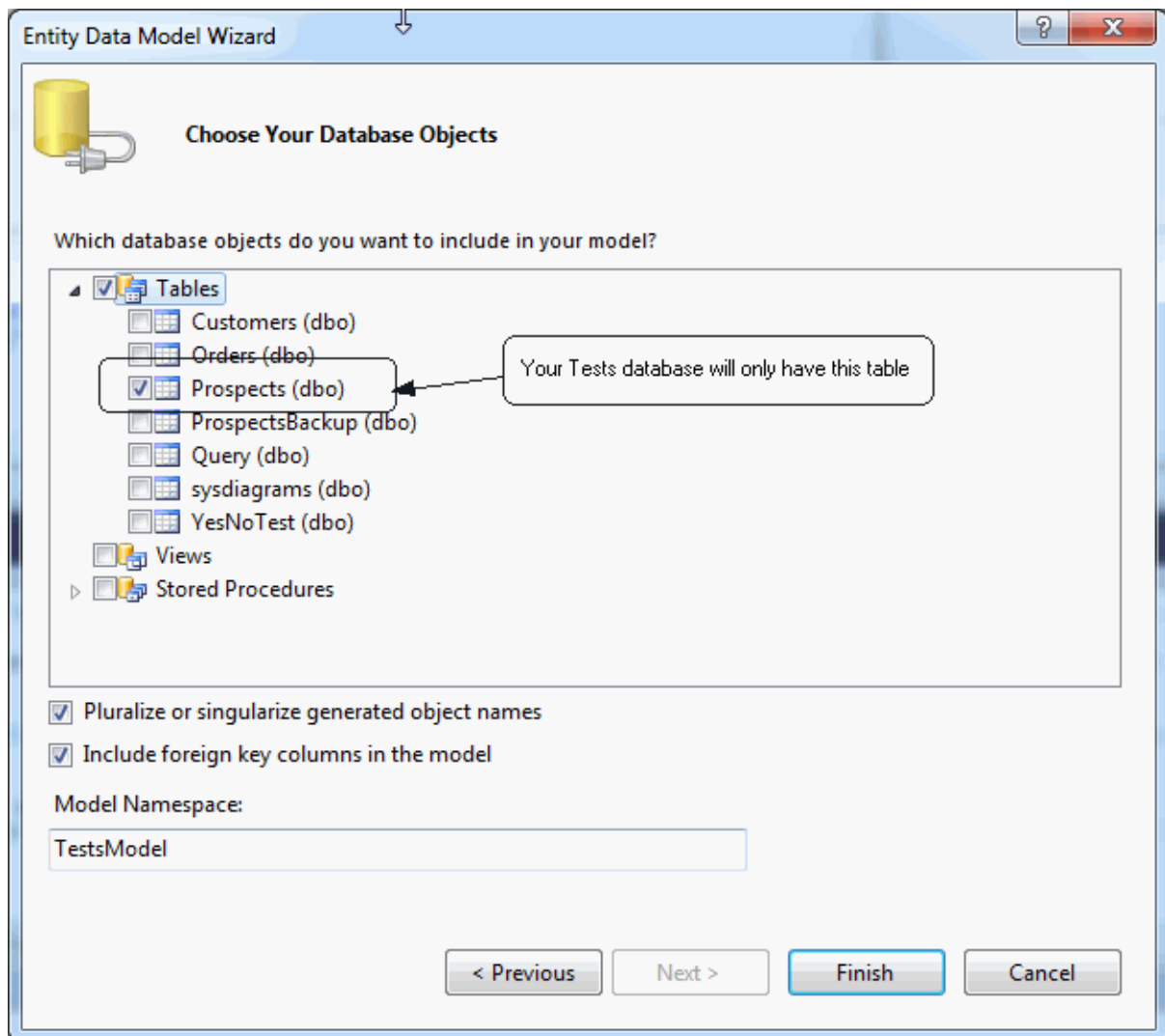
```
metadata=res://*/DataModel.csd|res://*/DataModel.ssd|
res://*/DataModel.msl;provider=System.Data.SqlClient;provider connection string="Data Source=DELL
\DELL;Initial Catalog=Tests;Integrated Security=True"
```

☒ Save entity connection settings in App.Config as:

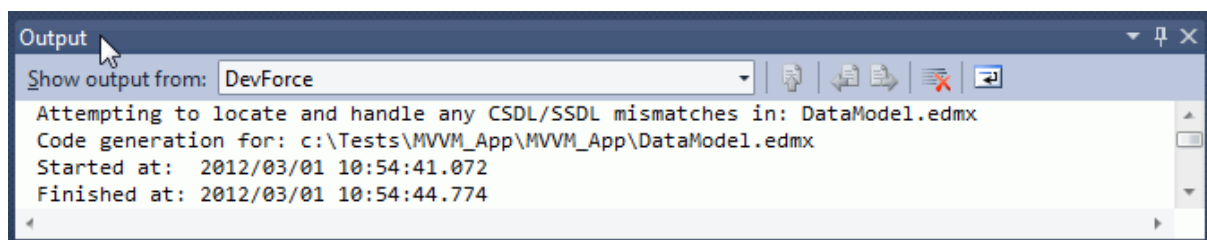
TestsEntities

< Previous Next > Finish Cancel

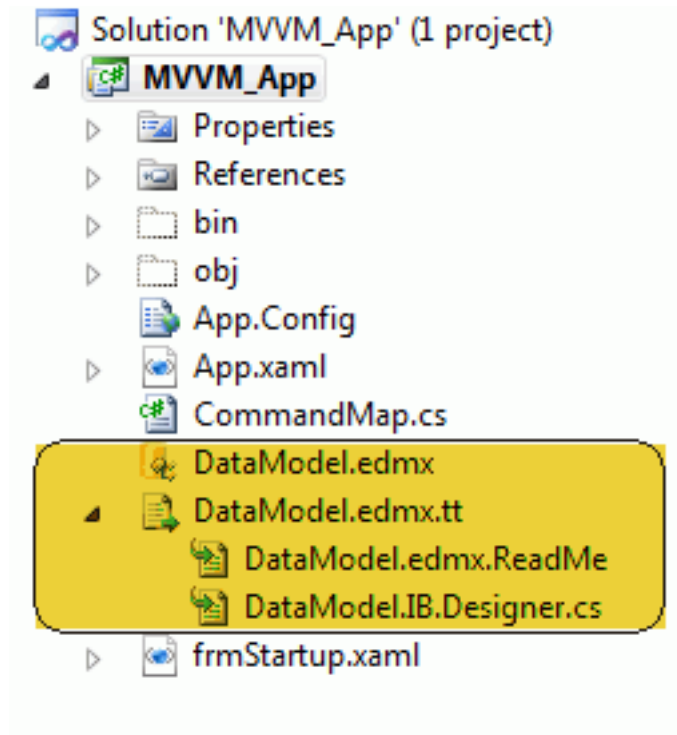
Next, you'll pick your table or tables and/or views and/or stored procedures. We just have one table, Prospects, so check it and click **Ok**.



If you look at the output window, you'll see that it says something about code generation. That's what the ORM tool does, and it's **fast**. I've seen it generate a hundred thousand lines of code in sixty seconds. You don't touch this code. If you need to add something else to one or more tables, there's an app for that; more on that later.



Your Project in the Solution Explorer will now look like:



The highlighted files represent the Model of the data. The .edmx file contains the map of the table(s) in the Model; the .tt file points to the domain model template use by the IdeaBlade ORM tool to generate the code; and the .cs file contains the generated code, partly shown below.

Code Listing 2: The DevForce-generated Model code (partial listing):

```
public partial class TestsEntities : IbEm.EntityManager {
Constructors (region collapsed)
#region EntityQueries
public IbEm.EntityQuery Prospects
{
get { return new IbEm.EntityQuery("Prospects", this); }
}
#endregion EntityQueries
#region Prospect class
/// The auto-generated Prospect class.
[DataContract(IsReference=true)]
[IbEm.DataSourceKeyName(@"TestsEntities")]
[IbEm.DefaultEntitySetName(@"TestsEntities.Prospects")]
public partial class Prospect : IbEm.Entity { ...
```

There are a half-dozen classes in this generated code, but two of them are most important to us:

- An **EntityManager** class called "TestEntities" that you use to get and save data; this class contains an **EntityQuery** called **Prospects** that can be used to return an *ObservableCollection* of Prospect entities;
- The **Prospect** class that holds a prospect table row; for each column in the table there's a corresponding public property with *property change notification* built in. This is important for WPF, as we'll see shortly.

Those constructs are the scaffolding that we'll use to build our application.

The View

Back in frmStartup, I added one line to the <Window declaration at the top of the file:

```
WindowStartupLocation="CenterScreen"
```

I've also added a menu with the following structure:

```
File  Tables
Exit  Prospects
```

We need to add *event handlers* for the Exit and Prospects MenuItems. Since their names will be used to generate event names, add "Name=" clauses to each. I used *Name=mnuExit* and *Name=mnuProspects* respectively. Now, select the **Exit** MenuItem, open the Properties Window using F4, select the **Events** tab (the one with the little lightning bolt icon), and double click on the **Click** event. In the resulting *mnuExit_Click* event handler method, add this:

```
App.Current.ShutDown();
```

But before we write the code to launch the Prospects form, we need to create said form. The WPF form designer is, to say the least, *a little different* from the WinForms designer. Every traditional developer I've ever worked with to build their first WPF form has asked the same question: "Why isn't the layout toolbar enabled?" That little question actually sheds a lot of light on the WPF paradigm.

In WPF, you use containers to align and line up controls. For example, if you want to stack five controls left to right, you use a stackpanel. You usually start with a *grid*, which has nothing to do with datagrids or gridviews or datagridviews. In WPF, a grid is just a rectangle. You can add row and column dividers, and then position a control within a particular "cell" of this grid by supplying coordinates, e.g. `<TextBox Grid.Row="2", Grid.Column="4" />`. A StackPanel stacks with either a vertical or horizontal orientation. A UniformGrid spaces things equidistantly. A Canvas is pretty much what you're used to working with in WinForms, and just to give you a flavor of this brave new world, it's absolutely the lamest of the bunch.

Containers are meant to be nested, so if you end up with a stackpanel within a grid within another grid within a stackpanel, that's probably just about right; I've had containers nested eight deep. That's why you don't have a Left and a Top setting for every control. There is, however, a Margin attribute, with Left, Top, Right and Bottom spacing parameters that occasionally comes in handy. I have a couple of articles coming up on xaml design, and I hope to convince you that it's actually pretty easy to use, even without using Blend. For now, though, try just using containers and the judicious application of *Margin*, *Padding*, *VerticalAlignment* and *HorizontalAlignment*, and that's usually enough.

In the Solution Explorer, highlight the project name, right click to open the context menu, and select **Add, New Item, Window**. Enter **frmProspects** as the file name. The Designer will open the new window (form) in a Designer window. Add the xaml shown here:

```
<Window
  x:Class="AddEditDeleteProspects.frmProspects"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  WindowStartupLocation="CenterScreen"
  Title="Prospects"
  Height="260" Width="500" MaxWidth="500" MaxHeight="260" >
  <StackPanel>
    <StackPanel Orientation="Horizontal" >
      <Label
        Content="Prospects"
        BorderBrush="Blue" BorderThickness="1"
        HorizontalAlignment="Left" VerticalAlignment="Top"
        FontSize="24" FontFamily="Comic Sans MS"
        Padding="13,3,13,9" Margin="5"
        Foreground="Purple" Background="LemonChiffon" />
    </StackPanel>
    <Grid
      HorizontalAlignment="Left" VerticalAlignment="Top"
      Height="120" Width="475" >
      <Grid.RowDefinitions>
        <RowDefinition Height="25*" />
        <RowDefinition Height="25*" />
        <RowDefinition Height="25*" />
        <RowDefinition Height="25*" />
      </Grid.RowDefinitions>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="90*" />
        <ColumnDefinition Width="135*" />
        <ColumnDefinition Width="45*" />
        <ColumnDefinition Width="32*" />
        <ColumnDefinition Width="57*" />
        <ColumnDefinition Width="118*" />
      </Grid.ColumnDefinitions>
      <Label
        Content="First name"
        Grid.Row="0" Grid.Column="0" Margin="0,0,5,0"
        HorizontalAlignment="Right" VerticalAlignment="Center" />
      <TextBox Name="txtFirstName"
        Grid.Column="1" />
```



```

        HorizontalAlignment="Left" VerticalAlignment="Center"
        Width="130" />
<Label
    Content="Last name"
    Grid.Row="1" Grid.Column="0" Margin="0,0,5,0"
    HorizontalAlignment="Right" VerticalAlignment="Center" />
<TextBox Name="txtLastName"
    Grid.Row="1" Grid.Column="1"
    HorizontalAlignment="Left" VerticalAlignment="Center"
    Width="130" />
<Label
    Content="Address"
    Grid.Row="2" Grid.Column="0" Margin="0,0,5,0"
    HorizontalAlignment="Right" VerticalAlignment="Center" />
<TextBox Name="txtAddress"
    Grid.Row="2" Grid.Column="1"
    HorizontalAlignment="Left" VerticalAlignment="Center"
    Width="300" Grid.ColumnSpan="5" />
<Label
    Content="City"
    Grid.Row="3" Grid.Column="0" Margin="0,0,5,0"
    HorizontalAlignment="Right" VerticalAlignment="Center" />
<TextBox Name="txtCity"
    Grid.Row="3" Grid.Column="1"
    HorizontalAlignment="Left" VerticalAlignment="Center"
    Width="130" />
<Label
    Content="State"
    Grid.Row="3" Grid.Column="2" Margin="0,0,5,0"
    HorizontalAlignment="Right" VerticalAlignment="Center" />
<TextBox Name="txtState"
    Grid.Row="3" Grid.Column="3" Width="30" MaxLength="2" CharacterCasing="Upper"
    HorizontalAlignment="Left" VerticalAlignment="Center" />
<Label
    Content="ZIP code"
    Grid.Row="3" Grid.Column="4" Margin="0,0,5,0"
    HorizontalAlignment="Right" VerticalAlignment="Center" />
<TextBox Name="txtZIP"
    Grid.Row="3" Grid.Column="5" MaxLength="10"
    HorizontalAlignment="Left" VerticalAlignment="Center"
    Width="90" />
</Grid>
<StackPanel Orientation="Horizontal" Margin="0,10,0,0">
    <Button Name="btnFind"
        Content="_ Find"
        Width="auto" Margin="5,0,5,0" Padding="10,0,10,0" />
    <Button Name="btnAdd"
        Content="_ Add"
        Width="auto" Margin="5,0,5,0" Padding="10,0,10,0" />
    <Button Name="btnEdit"
        Content="_ Edit"
        Width="auto" Margin="5,0,5,0" Padding="10,0,10,0" />
    <Button Name="btnDelete"
        Content="_ Delete"
        Width="auto" Margin="5,0,5,0" Padding="10,0,10,0" />
    <Button Name="btnSave"
        Content="_ Save"
        Width="auto" Margin="5,0,5,0" Padding="10,0,10,0" />
    <Button Name="btnCancel"
        Content="_ Cancel"
        Width="auto" Margin="5,0,5,0" Padding="10,0,10,0" />
    <Button Name="btnClose"
        Content="Cl_ose"
        Width="auto" Margin="5,0,5,0" Padding="10,0,10,0"
        Click="btnClose_Click" />
</StackPanel>
</StackPanel>
</Window>

```

To spice up the form's appearance, I've added a few styles to my *App.xaml*, as shown below:

```
<Application
x:Class="AddEditDeleteProspects.App"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
StartupUri="MainWindow.xaml">
<Application.Resources>
<SolidColorBrush x:Key="BG">Linen</SolidColorBrush>
<SolidColorBrush x:Key="TC">Green</SolidColorBrush>
<FontFamily x:Key="FF">Trebuchet MS</FontFamily>
<Style TargetType="Grid">
<Setter Property="Background" Value="{ StaticResource BG}" />
</Style>
<Style TargetType="StackPanel">
<Setter Property="Background" Value="{ StaticResource BG}" />
</Style>
<Style TargetType="Window">
<Setter Property="Background" Value="{ StaticResource BG}" />
</Style>
<Style TargetType="Label">
<Setter Property="FontFamily" Value="{ StaticResource FF}" />
<Setter Property="FontSize" Value="11" />
<Setter Property="Padding" Value="1" />
<Setter Property="Foreground" Value="{ StaticResource TC}" />
<Setter Property="Background" Value="Transparent" />
<Setter Property="HorizontalAlignment" Value="Left" />
<Setter Property="VerticalAlignment" Value="Top" />
</Style>
<Style TargetType="TextBox">
<Setter Property="FontFamily" Value="Courier New" />
<Setter Property="FontSize" Value="11" />
<Setter Property="Foreground" Value="{ StaticResource TC}" />
<Style.Triggers>
<Trigger Property="IsMouseOver" Value="True">
<Setter Property="Background" Value="Pink" />
</Trigger>
<Trigger Property="IsFocused" Value="True">
<Setter Property="Background" Value="Blue" />
<Setter Property="Foreground" Value="White" />
</Trigger>
</Style.Triggers>
</Style>
</Application.Resources>
</Application>
```

The use of the *StaticResources* BG, TC and FF isn't required, but it makes assigning the same attributes (a color or a font) to several styles a lot easier.

The form is shown below:

Now that you have a Prospects form, you can return to `frmStartup.xaml`, select `mnuProspects`, open the Properties Window, select the Events tab and double click on the Click event, and then add this code to the resulting `mnuProspects_Click` event handler:

```
frmProspects fP = new frmProspects(); fP.Show(); // or fP.ShowDialog();
```

Press F5 to run the application and launch the window.

At this point, the Window contains no references to the data, and the code-behind consists of exactly one line of code: *InitializeComponent*; That's where the ViewModel comes in.

The ViewModel

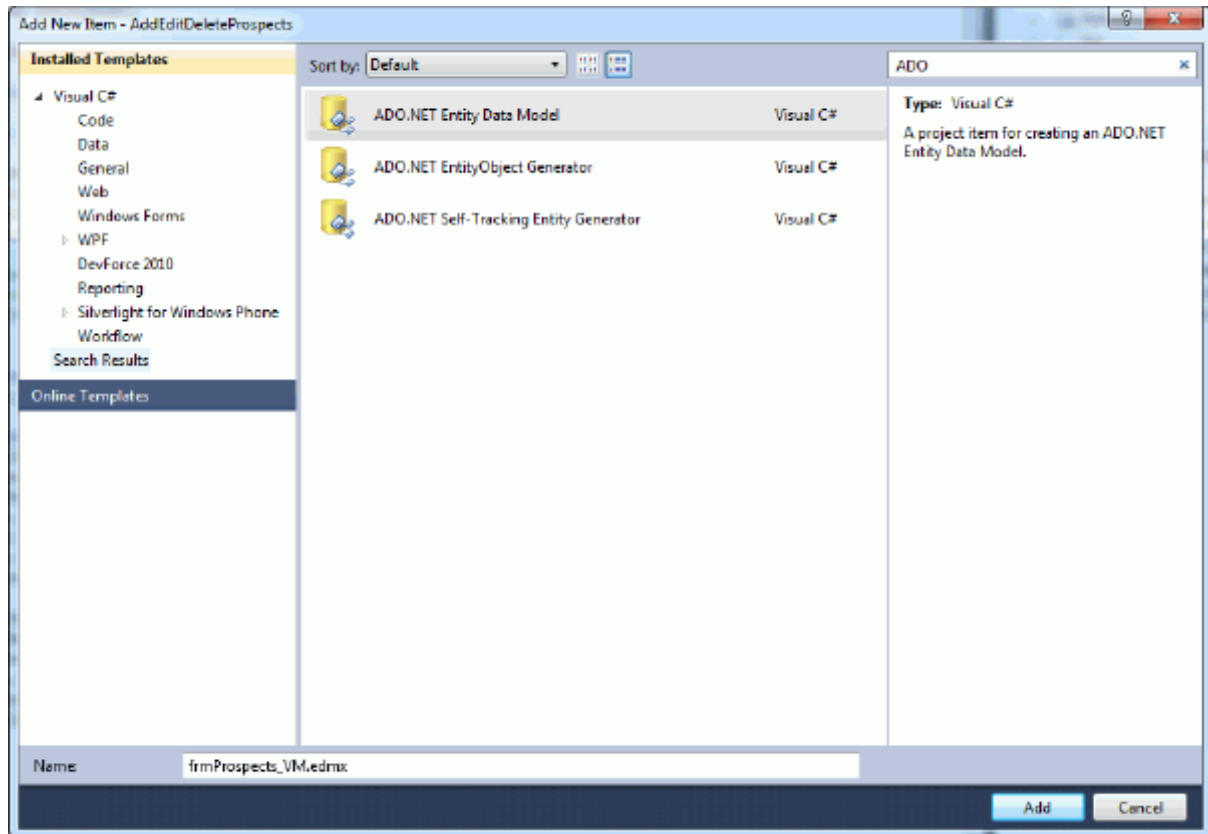
For each View you'll create a ViewModel. The ViewModel is a class that's instantiated in the Load event of the corresponding form - er, View. Sorry, it's a hard habit to break. As soon as the ViewModel object is created, it's assigned to the DataContext of the View. Thereafter, any Binding reference is assumed to be a reference to a public property in the ViewModel. So, for example, if we have a *prospect* property that contains the currently selected record from the Prospects table, a binding to `FirstName` in a View control will look for it in the public property *prospect* in the ViewModel object, or in a public Model object inside the ViewModel. These properties *must* be public, because binding uses *reflection* to find them, and they have to be public, and they have to be properties - that is, they have to have *accessors* - a getter and a setter.

The values stored in public properties may change during program execution. If they do, the ViewModel class needs to implement the *INotifyPropertyChanged* interface, so that you can *RaisePropertyChanged*(**PropertyName**) to let the View know that it needs to refresh the data in the corresponding binding. To do that, you add : ***INotifyPropertyChanged*** at the end of the ViewModel Class declaration, and add the two boldfaced lines of code at the end of the ViewModel class as shown earlier. Then, include *RaisePropertyChanged*("PropertyName") as the last line in the *getter* of each property. I've hidden an undocumented Easter Egg in the app's code to demonstrate it. It's hidden in plain sight ...

In order to link ViewModel methods to events (particularly the Button Click event) in your View, you need to implement the *ICommand* interface. That's what *CommandUtilities.cs* is for. It gives you an easy way to add command delegates (sorry, that's what they're called) to your code and attach them to a particular button. You can either click on a button to execute the corresponding command, or you can just call the method in a *TestFixture* or from another method in your code. And, if you write a second method that returns true if the button should be enabled, or false if it should be disabled, you can pass it to the Commands constructor as the second (optional) parameter to automate that aspect of button control. See the *Delete* command for an example.

That's a lot of explanation, but if you study the code using the line number references in the explanations that follow it, I think you'll find that most things are harder to explain than they are to do. Both binding and commanding generally take just a couple of lines of code.

Right click on the project name in the Solution Explorer and select **Add, Class** from the context menu. Assign the name *frmProspects_VM*, as shown:



Then, add the code shown below:

```

01 using IdeaBlade.Core;
02 using IdeaBlade.EntityModel;
03 using CommandUtilities;
04 using System; using System.Windows;
05 using System.ComponentModel;
06 using System.Collections.ObjectModel;
07
08 namespace AddEditDeleteProspects
09 {
10     class frmProspects_VM : INotifyPropertyChanged
11     {
12         TestsEntities mgr = new TestsEntities();
13
14         public ObservableCollection<Prospect> prospects { get; set; }
15
16         private Prospect _prospect;
17         public Prospect prospect { get { return _prospect; }
18             set { _prospect = value; RaisePropertyChanged("prospect"); } }
19
20         Prospect newprospect;
21
22         private CommandMap _commands;
23         public CommandMap Commands { get { return _commands; } }
24
25         public frmProspects_VM()
26         {
27             mgr = new TestsEntities();
28
29             prospects = new ObservableCollection<Prospect>();
30
31             _commands = new CommandMap();
32             _commands.AddCommand("Find", x => FindaP() );
33             _commands.AddCommand("Add", x => Add() );
34             _commands.AddCommand("Edit", x => Edit() );
35             _commands.AddCommand("Delete", x => Delete(), x => CanDelete() );
36             _commands.AddCommand("Save", x => Save() );
37         }
38     }
39 }

```

```

36 _commands.AddCommand("Cancel", x => Cancel() );
37 }
38
39 public void FindaP()
40 {
41     FindAProspect fap = new FindAProspect(); // This form will be added shortly.
42     fap.ShowDialog();
43     if (App.ID != null)
44     { prospects.Clear();
45       var q = mgr.Prospects.Where(x => x.ID == App.ID);
46       q.Execute().ForEach(prospects.Add);
47       prospect = prospects[0];
48     }
49 }
50
51 private void GetFirstRecord()
52 { if (prospects != null) prospects.Clear();
53   var query = mgr.Prospects; query.Execute().ForEach(prospects.Add);
54   if (prospects.Count > 0) { prospect = prospects[0]; }
55   else { prospect = null; MessageBox.Show("No data"); }
56 }
57
58 public void ManageControls(bool OnOff)
59 { for (int i = 0; i < App.Current.Windows.Count; i++)
60   { Window w = App.Current.Windows[i]; if (w.Title == "Prospects") {(w as frmProspects).Enabler(OnOff);}}
61 }
62
63 public void Add()
64 { newprospect = new Prospect(); newprospect.ID = Guid.NewGuid();
65   prospect = newprospect;
66   mgr.AddEntity(newprospect);
67   ManageControls(false);
68 }
69
70 public void Edit()
71 { ManageControls(false); }
72
73 public void Delete()
74 { prospects.RemoveAt(0); // Because there's always only one Prospect that was loaded
75   prospect.EntityAspect.Delete();
76   Save();
77   prospect = null;
78 }
79
80 public bool CanDelete()
81 { return (prospect != null); }
82
83 public void Save()
84 { mgr.SaveChanges(); ManageControls(true); }
85
86 public void Cancel()
87 { mgr.RejectChanges(); ManageControls(true); }
88
89 public event PropertyChangedEventHandler PropertyChanged = delegate { };
90 private void RaisePropertyChanged(string property)
91 { PropertyChanged(this, new PropertyChangedEventArgs(property)); }
92 }

```

I definitely have some 'splaining to do, so let's get started.

- 01-07: The IdeaBlade references eliminate the need for these prefixes before IdeaBlade classes: CommandUtilities is the NameSpace for the CommandMap class used to implement commanding; and the four Windows references are for dealing with Guids and collections;
- 10, 89-90: Used to implement property change notification, which sends changed values back to the View;
- 12: Creates an IdeaBlade EntityManager object, which keeps track of changes and sends them back to the database; It also contains the generated query class used to return Prospects in the Find form that will be defined below;
- 14: This is the collection where records - er, entities - returned by a query are stored; **Prospect** is the generated class that defines a record - er, an entity as well as the query for that table.

- 16-17: prospect is the object that holds the current record from the Prospects table. Prospects form controls are bound to this object;
- 19: This holds a new prospect that is created and then added to the *prospects* collection;
- 21-22: This collection will store commands in a way that simplifies both command wireup and the Binding syntax;
- 24: This is the constructor for the ViewModel class; it runs when the object is created in the View's Load event, as we'll see shortly;
- 26: This instantiates the EntityManager class generated by DevForce for this database;
- 28: This instantiates the **prospects** collection of **Prospect** records - er, entities;
- 30: This instantiates the Commands collection object defined in the CommandMap class in CommandUtilities.cs;
- 31-36: This adds six delegate commands to the Commands collection; in the next Listing we'll see how they're used;
- 39-49: This shows a modal Find form to select a single prospect to be displayed in frmProspects. The key selected by the user, which is a *Guid*, is stored in App.ID;
- 51-56: Not used unless you want the form to appear with one record already loaded;
- 58-60: Uses a little trick to find the View form and call the form's *Enabler* routine to enable/disable its TextBoxes and Buttons;
- 63-68: Adds an empty record to the prospects collection, adds a GUID, and assigns it to the prospect object for display on the View;
- 70-71: Enables data entry controls and disables all buttons except Save and Cancel (see the View code-behind);
- 73-78: Deletes the selected Prospect record and removes it from the prospects collection using the mgr object;
- 80-81: Returns True if the Delete button should be enabled; or False if it shouldn't; this is a built-in behavior of ICommand (see *CommandMap*);
- 83-84: Saves any changes and disables input controls and enables all buttons except Save and Cancel;
- 86-87: Cancels any changes and disables input controls and enables all buttons except Save and Cancel;
- 89-90: Required when you implemented the INotifyPropertyChanged interface up on line 10.

The strangest code in this ViewModel is this one:

```
46 q.Execute().ForEach(prospects.Add);
```

This means

"Execute the query; then, **Add** each row in the result to the *prospects* ObservableCollection,"

Binding the Click event on Buttons in a View to the corresponding ViewModel requires implementing the *ICommand* interface, which CommandMap does thanks to the *DelegateCommand* class in CommandUtilities.cs. Binding to data requires *public properties*, one for each column in each table, and these properties must have Property Change Notification built in. IdeaBlade entities do. Our *prospect* object, representing the prospect that's displayed on the screen, doesn't, and that's why we added it above on line 17.

The *prospects* collection is a public property (note the { get; set; } that implements the default backing store.) That's *required*. Remove the { get; set; } at the end, and it will compile just fine, but it won't work. You'll forget to add that a few times before you become completely familiar with ObservableCollection and binding.

Probably the oddest thing in this code is the occasional use of *lambda expressions*, i.e. "x => x." (e.g. line 45). There's a reason for this syntax, but *I don't care*. Just get used to typing "x => x." (or "z => z." or whatever) any time you want IntelliSense to show you your column names, and you'll be fine.

Now we can go back and add the binding expressions that tie the View to the ViewModel:

```
<Window
  x:Class="AddEditDeleteProspects.frmProspects"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  WindowStartupLocation="CenterScreen"
  Title="Prospects"
  Height="260" Width="500" MaxWidth="500" MaxHeight="260" >
  <StackPanel>
    <StackPanel Orientation="Horizontal" >
      <Label
        Content="Prospects"
        BorderBrush="Blue" BorderThickness="1"
        HorizontalAlignment="Left" VerticalAlignment="Top"
        FontSize="24" FontFamily="Comic Sans MS"
        Padding="13,3,13,9" Margin="5"
        Foreground="Purple" Background="LemonChiffon" />
    </StackPanel>
    <Grid
      DataContext="{Binding prospect}"
      HorizontalAlignment="Left" VerticalAlignment="Top"
```

```

Height="120" Width="475" >
<Grid.RowDefinitions>
<RowDefinition Height="25*" />
<RowDefinition Height="25*" />
<RowDefinition Height="25*" />
<RowDefinition Height="25*" />
</Grid.RowDefinitions>
<Grid.ColumnDefinitions>
<ColumnDefinition Width="90*" />
<ColumnDefinition Width="135*" />
<ColumnDefinition Width="45*" />
<ColumnDefinition Width="32*" />
<ColumnDefinition Width="57*" />
<ColumnDefinition Width="118*" />
</Grid.ColumnDefinitions>
<Label
Content="First name"
Grid.Row="0" Grid.Column="0" Margin="0,0,5,0"
HorizontalAlignment="Right" VerticalAlignment="Center" />
<TextBox Name="txtFirstName"
Text="{Binding Path=FirstName}"
Grid.Column="1"
HorizontalAlignment="Left" VerticalAlignment="Center"
Width="130" />
<Label
Content="Last name"
Grid.Row="1" Grid.Column="0" Margin="0,0,5,0"
HorizontalAlignment="Right" VerticalAlignment="Center" />
<TextBox Name="txtLastName"
Text="{Binding Path=LastName}"
Grid.Row="1" Grid.Column="1"
HorizontalAlignment="Left" VerticalAlignment="Center"
Width="130" />
<Label
Content="Address"
Grid.Row="2" Grid.Column="0" Margin="0,0,5,0"
HorizontalAlignment="Right" VerticalAlignment="Center" />
<TextBox Name="txtAddress"
Text="{Binding Path=Address}"
Grid.Row="2" Grid.Column="1"
HorizontalAlignment="Left" VerticalAlignment="Center"
Width="300" Grid.ColumnSpan="5" />
<Label
Content="City"
Grid.Row="3" Grid.Column="0" Margin="0,0,5,0"
HorizontalAlignment="Right" VerticalAlignment="Center" />
<TextBox Name="txtCity"
Text="{Binding Path=City}"
Grid.Row="3" Grid.Column="1"
HorizontalAlignment="Left" VerticalAlignment="Center"
Width="130" />
<Label
Content="State"
Grid.Row="3" Grid.Column="2" Margin="0,0,5,0"
HorizontalAlignment="Right" VerticalAlignment="Center" />
<TextBox Name="txtState"
Text="{Binding Path=State}"
Grid.Row="3" Grid.Column="3" Width="30" MaxLength="2" CharacterCasing="Upper"
HorizontalAlignment="Left" VerticalAlignment="Center" />
<Label
Content="ZIP code"
Grid.Row="3" Grid.Column="4" Margin="0,0,5,0"
HorizontalAlignment="Right" VerticalAlignment="Center" />
<TextBox Name="txtZIP"
Text="{Binding Path=ZIP}"
Grid.Row="3" Grid.Column="5" MaxLength="10"
HorizontalAlignment="Left" VerticalAlignment="Center"
Width="90" />
</Grid>
<StackPanel Orientation="Horizontal" Margin="0,10,0,0">
<Button Name="btnFind"

```

```

        Content="_Find"
        Command="{Binding Commands.Find}"
        Width="auto" Margin="5,0,5,0" Padding="10,0,10,0" />
<Button Name="btnAdd"
        Content="_Add"
        Command="{Binding Commands.Add}"
        Width="auto" Margin="5,0,5,0" Padding="10,0,10,0" />
<Button Name="btnEdit"
        Content="_Edit"
        Command="{Binding Commands.Edit}"
        Width="auto" Margin="5,0,5,0" Padding="10,0,10,0" />
<Button Name="btnDelete"
        Content="_Delete"
        Command="{Binding Commands.Delete}"
        Width="auto" Margin="5,0,5,0" Padding="10,0,10,0" />
<Button Name="btnSave"
        Content="_Save"
        Command="{Binding Commands.Save}"
        Width="auto" Margin="5,0,5,0" Padding="10,0,10,0" />
<Button Name="btnCancel"
        Content="_Cancel"
        Command="{Binding Commands.Cancel}"
        Width="auto" Margin="5,0,5,0" Padding="10,0,10,0" />
<Button Name="btnClose"
        Content="Close"
        Width="auto" Margin="5,0,5,0" Padding="10,0,10,0"
        Click="btnClose_Click" />
</StackPanel>
</StackPanel>
</Window>

```

The addition of the attribute **Text={Binding Path=FirstName}** causes WPF to search its current *DataContext* for a public property named *FirstName*. Since the current *DataContext* is an instance of the *frmProspects_VM* ViewModel class, and the Grid's *DataContext* is bound to *prospect*, it looks in *frmProspects_VM.prospect*, a public property of type *Prospect* (a single record - er, entity), where it will find a public property named *LastName* thanks to the *Prospects* class generated by IdeaBlade and declared in our ViewModel. Neat, eh?

Back in *frmProspects*, we need a little code to give life to our app:

```

using System.Windows;
using System.Collections.ObjectModel;
using IdeaBlade.Core;
using IdeaBlade.EntityModel;
namespace AddEditDeleteProspects
{
    public partial class frmProspects : Window
    {
        frmProspects_VM ViewModel;
        /* Constructor */
        public frmProspects()
        { InitializeComponent();
            ViewModel = new frmProspects_VM();
            DataContext = ViewModel;
            // You can also use this one-line "improvement" instead:
            // Loaded += delegate { DataContext = new Prospects_VM(); }; }
            // However, the explicit declaration permits calling the Enabler method from the ViewModel.
            Enabler(true);
        }
        private void btnClose_Click(object sender, RoutedEventArgs e)
        { Close(); }
        public void Enabler ( bool OnOff) // called from within the ViewModel!
        { btnAdd.IsEnabled = OnOff;
            btnClose.IsEnabled = OnOff;
            btnDelete.IsEnabled = OnOff;
            btnEdit.IsEnabled = OnOff;
            btnFind.IsEnabled = OnOff;
            btnSave.IsEnabled = !OnOff;
            btnCancel.IsEnabled = !OnOff;
            txtFirstName.IsEnabled = !OnOff;
            txtLastName.IsEnabled = !OnOff;
            txtAddress.IsEnabled = !OnOff;
        }
    }
}

```



```

    txtCity.IsEnabled = !OnOff;
    txtState.IsEnabled = !OnOff;
    txtZIP.IsEnabled = !OnOff;
  }
}
}

```

The reason that the ViewModel object is available to the View is because we declare and instantiate it in the constructor of *frmProspects*. The declaration is scoped to the class. In the constructor, the ViewModel object is assigned to the Form's *DataContext*. That's the reason that the Binding code in Listing 5 works.

Enabler is a method to enable/disable TextBoxes and Buttons in the View. There are two reasons why I made *Enabler()* a public method in the form's CodeBehind: First, I won't need to test that logic in the unit tests; it's only meaningful in the UI, and you test the UI by running the UI. Second, since *Enabler()* is *public*, I can call it from the ViewModel (see the *ManageControls* method in the ViewModel, above.) The way that the ViewModel code is written, if the window isn't found, the code doesn't run, so it's harmless during unit testing when no views are active.

The last thing I need in my project is the *FindAProspect* form (I mean View) referred to on line 41 of Listing 4. My sample only searches for last names starting with the letter(s) entered by the user, but obviously you can make the search form as fancy as you want. The methodology is the same: Filter the collection based on user-supplied criteria, and then when the user selects a Prospect, save its key (which is a Guid, if you recall) to the public static ID field in *App.xaml.cs*. I just used *LastName* because it demonstrates one way to filter the data. The boldfaced code toward the end of Listing 8 shows how LINQ is to provide IntelliSense for SQL Queries. Pretty slick, ¿no?

I'll first list the xaml, then the codebehind:

```

<Window
  x:Class="AddEditDeleteProspects.FindAProspect"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  WindowStartupLocation="CenterScreen"
  Title="Find a Prospect" Height="260" Width="500">
  <Grid>
    <Label
      Content="Show last names starting with "
      HorizontalAlignment="Left" VerticalAlignment="Top"
      Margin="4,10,0,0" Width="auto" Height="auto" />
    <TextBox CharacterCasing="Upper"
      Height="23" HorizontalAlignment="Right" Margin="0,6,217,0"
      Name="textBox1" VerticalAlignment="Top" Width="101" />
    <DataGrid
      ItemsSource="{ Binding Path=prospects }"
      AutoGenerateColumns="False" HorizontalAlignment="Left"
      Margin="0,36,0,36" Name="dataGrid1" Width="478" >
      <DataGrid.Columns>
        <DataGridTextColumn Header="Last name" Binding="{ Binding LastName }" Width="230" />
        <DataGridTextColumn Header="First name" Binding="{ Binding FirstName }" Width="230" />
      </DataGrid.Columns>
    </DataGrid>
    <Button
      Content="Select" Margin="179,0,244,7" Name="btnSelect"
      Padding="10,0,10,0" Height="23" VerticalAlignment="Bottom" Click="btnSelect_Click" />
    <Button
      Content="Cancel" Margin="243,0,176,7" Name="btnCancel" Padding="10,0,10,0"
      Height="23" VerticalAlignment="Bottom" Click="btnCancel_Click" />
    <Button
      Content="Show matching prospects" Height="23" Margin="305,6,12,0"
      Name="btnShow" VerticalAlignment="Top" Padding="10,0,10,0" Click="btnShow_Click" />
  </Grid>
</Window>

```

```

using System;
using System.Windows;
using System.Collections.ObjectModel;
using System.ComponentModel;
using IdeaBlade.Core;
using IdeaBlade.EntityModel;
namespace AddEditDeleteProspects
{
  public partial class FindAProspect : Window
  {

```

```

public ObservableCollection<Prospect> prospects { get; set; }
TestsEntities mgr = new TestsEntities();
public FindAProspect()
{ InitializeComponent();
  prospects = new ObservableCollection<Prospect>();
}
private void btnShow_Click(object sender, RoutedEventArgs e)
{ if (prospects != null) prospects.Clear();
  string ln = textBox1.Text.ToString().TrimEnd();
  var query = mgr.Prospects
    .Where(x => x.LastName.StartsWith(ln))
    .OrderBy(x=>x.LastName).ThenBy(x=>x.FirstName);
  query.Execute().ForEach(prospects.Add);
  dataGrid1.ItemsSource = prospects;
}
private void btnSelect_Click(object sender, RoutedEventArgs e)
{ App.ID = prospects[dataGrid1.SelectedIndex].ID; Close(); }
private void btnCancel_Click(object sender, RoutedEventArgs e)
{ App.ID = null; Close(); }
}

```

If you've never used LINQ, the code in `btnShow_Click` will be somewhat puzzling. *Prospects* is actually an *EntityQuery*, which means it's the basis of a SELECT statement. LINQ exposes SQL with IntelliSense; so, each time you type a period, the PEMs (properties, events and methods) available at that juncture appear in a dropdown list. C# ignores whitespace, including spaces and carriage returns, so you can drop down to the next line before you enter the period for enhanced readability (or if you're a publisher and you have more vertical than horizontal space; welcome to my world). Finally, you execute the query, add all returned rows to the prospects collection, and assign it to the datagrid's *ItemsSource*.

The `Where`, `OrderBy` and `ThenBy` clauses, and any others that you might want to use, need a starting point for the object whose PEMs you're exposing, and "`x=>x.`" means something like "an x such that the x's.." Just type it, and you'll be supplied with the available PEMs. It looks funny, but it works. After a while you'll use it without even thinking about it. Which is better than trying to rationalize such an odd syntax. But Microsoft gets to decide how we do our work, and they're always right, even when a mutant like this is the result. But it's the best of all possible syntaxes, isn't it?

<rant>No it isn't. Just off the top of my head, this would be better:

```

var query = mgr.Prospects P
  .Where(P.LastName.StartsWith(ln))
  .OrderBy(P.LastName).ThenBy(P.FirstName);

```

It's just a thought, gods in Redmond...</rant>

`btnSelect_Click` shows what *App.ID* is for; it provides a place to store the selected Guid until we close the modal `FindAProspect` form and get back to the code that launched it, which then uses the value stored in `App.ID` to retrieve the selected *Prospects* record - er, entity and load it into the prospect object in `frmProspects_VM`, which exposes the columns as public properties (with Property Change Notification, thanks to DevForce), which allows the *Binding Path=(columnName)* attributes in the View to find and display the data. And since by default, Binding in WPF is *TwoWay*, changes on the form are pushed back to the public *prospect* object in the ViewModel for saving by the mgr object. Got it?

You can easily use a ViewModel for the search form as well, but it's not necessary. MVVM is used to enable unit testing; it's not a religion. Besides, you can add test routines in codebehind files; you just have to write separate `[test]` methods so that they don't require human input. For example, in the download, I've added a `[Test]` to the `FindAProspect` CodeBehind. It's described in the Unit Testing section below.

Speaking of unit testing, wasn't that the original objective? So let's do it.

Unit Testing

Unit testing is much of the reason for using MVVM instead of just putting all event handling in the code-behind. So it had better be pretty easy and pretty slick. It's both.

I use `nUnit`, because it's survived the test of time, and has tons of features. You can download it free from `nUnit.org`. After installing, add **nUnit.exe** to your start menu (or **nUnit-x86.exe** if you're running a 64-bit processor). You'll be using it a lot. Then, add a reference to `nUnit.Framework.dll`, located in your `C:\Program Files(x86)\nUnit` folder, to your project references and Build the project.

Adding a test couldn't be simpler (well, it could, but that would require changing the way Microsoft has implemented usings, and that would add a lot of other complications. I miss FoxPro...<g>) Add using **nUnit.Framework** just **below** the namespace declaration in `frmProspects_VM.cs` file, and then add this:

```
[TestFixture, Description("Unit tests")]
[RequiresSTA]
class frmProspects_VM : INotifyPropertyChanged
```

If you add *using NUnit.Framework* before the namespace declaration, you'll have to use *NUnit.Framework.Description("...")*, because there's also a *Description* attribute associated with *System.ComponentModel*, and VS 2010 won't know which one you're referring to. I had never used usings except at the top of a code file, but it's never too late to learn something new.

Next, modify the *GetFirstRecord* method to look like this:

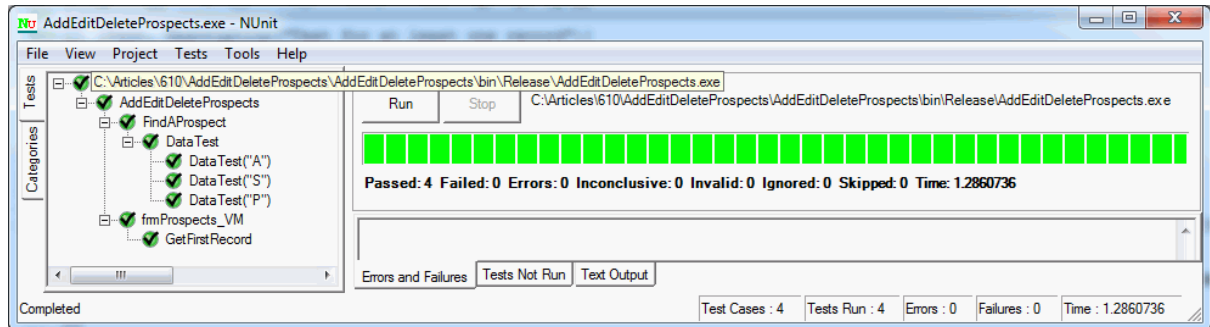
```
[Test, Description("Test for at least one record")]
[RequiresSTA]
public void GetFirstRecord()
{ if (prospects != null) prospects.Clear();
  var query = mgr.Prospects;
  query.Execute().ForEach(prospects.Add);
  Assert.Greater(prospects.Count, 0);
}
```

The *[RequiresSTA]* attribute is required. I had never seen this documented in IdeaBlade code, but trust me, it won't run without it.

I've also added a test in the CodeBehind for the search form, just to show that it works. I've got last names starting with "P" and "S", but not with "A". So I'll test whether a corresponding record is returned for each case, using the totally cool *Values parameter*:

```
[Test, Description("Returns at least one LastName starting with 'S' or 'P'; fails matching on 'A'")]
[RequiresSTA]
public void DataTest([Values("A", "S", "P")] string ln)
{ if (prospects != null) prospects.Clear();
  var query = mgr.Prospects.Where(x => x.LastName.StartsWith(ln)).OrderBy(x => x.LastName).ThenBy(x => x.FirstName);
  query.Execute().ForEach(prospects.Add);
  if (ln == "A") { Assert.AreEqual(prospects.Count, 0); }
  else { Assert.Greater(prospects.Count, 0); }
}
```

Now, launch nUnit (or nUnit-x86) and use File/Open to point to your executable, in either the bin/Debug or bin/Release folder, and click on the GO button. If you have the test data from the SQL script in SQL Listing 1 loaded, you'll see the result shown:



Unit testing can be as simple or as robust as you want it to be. Some developers write the unit tests before they build the forms. That's a little hard core for me, but whatever floats your boat.

Conclusion

WPF is a giant step forward in desktop application design, and MVVM is a good way to make full use of its powerful binding capabilities. Hopefully, this article has given you confidence to dive in and try it on one of your tables. Email me if you can't make something work. I've been answering all of my emails for 25 years at Les@Pinter.com; no reason to stop now.

See 'ya 😊