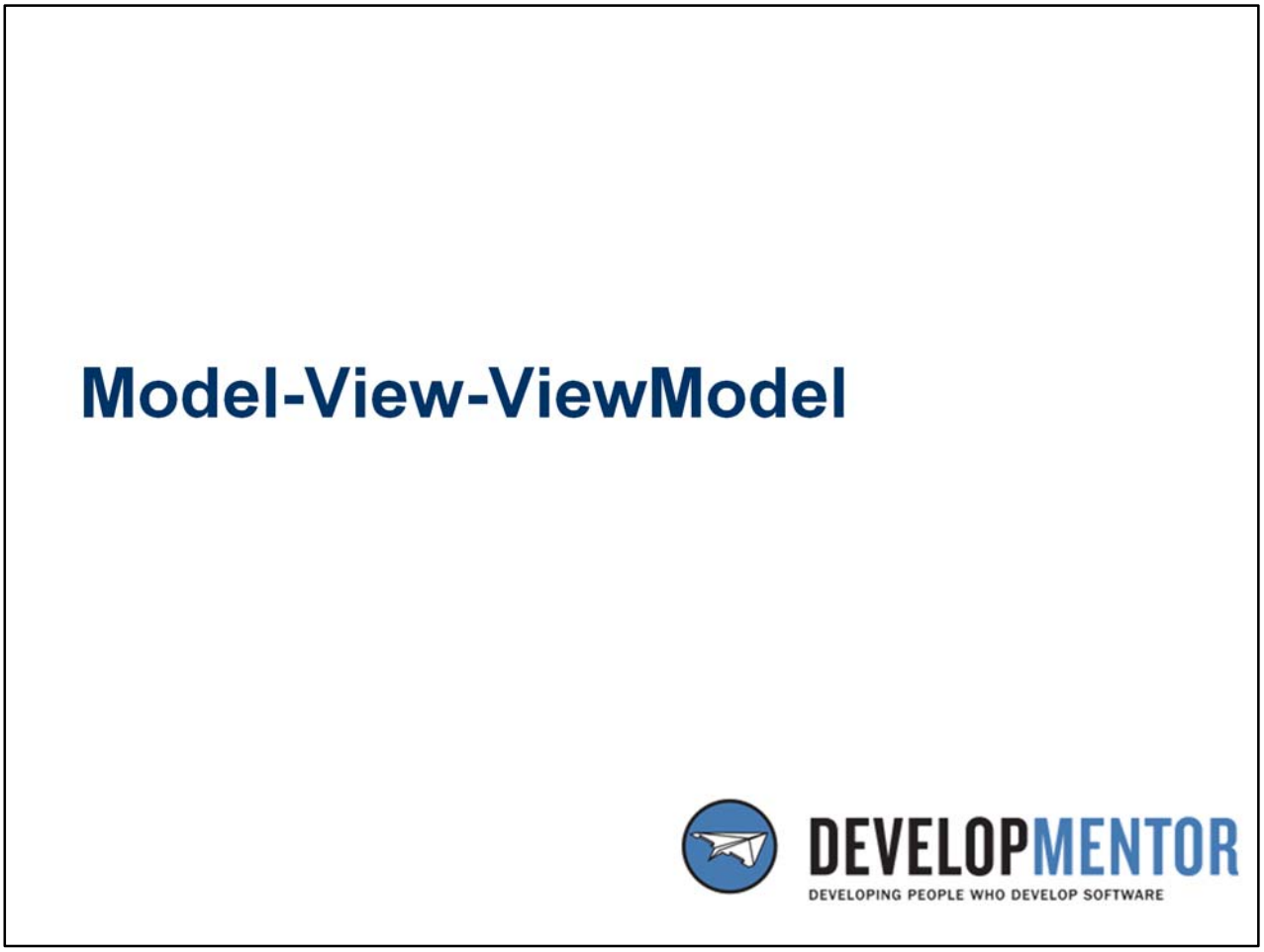
9: Model-View-ViewModel



9: Model-View-ViewModel

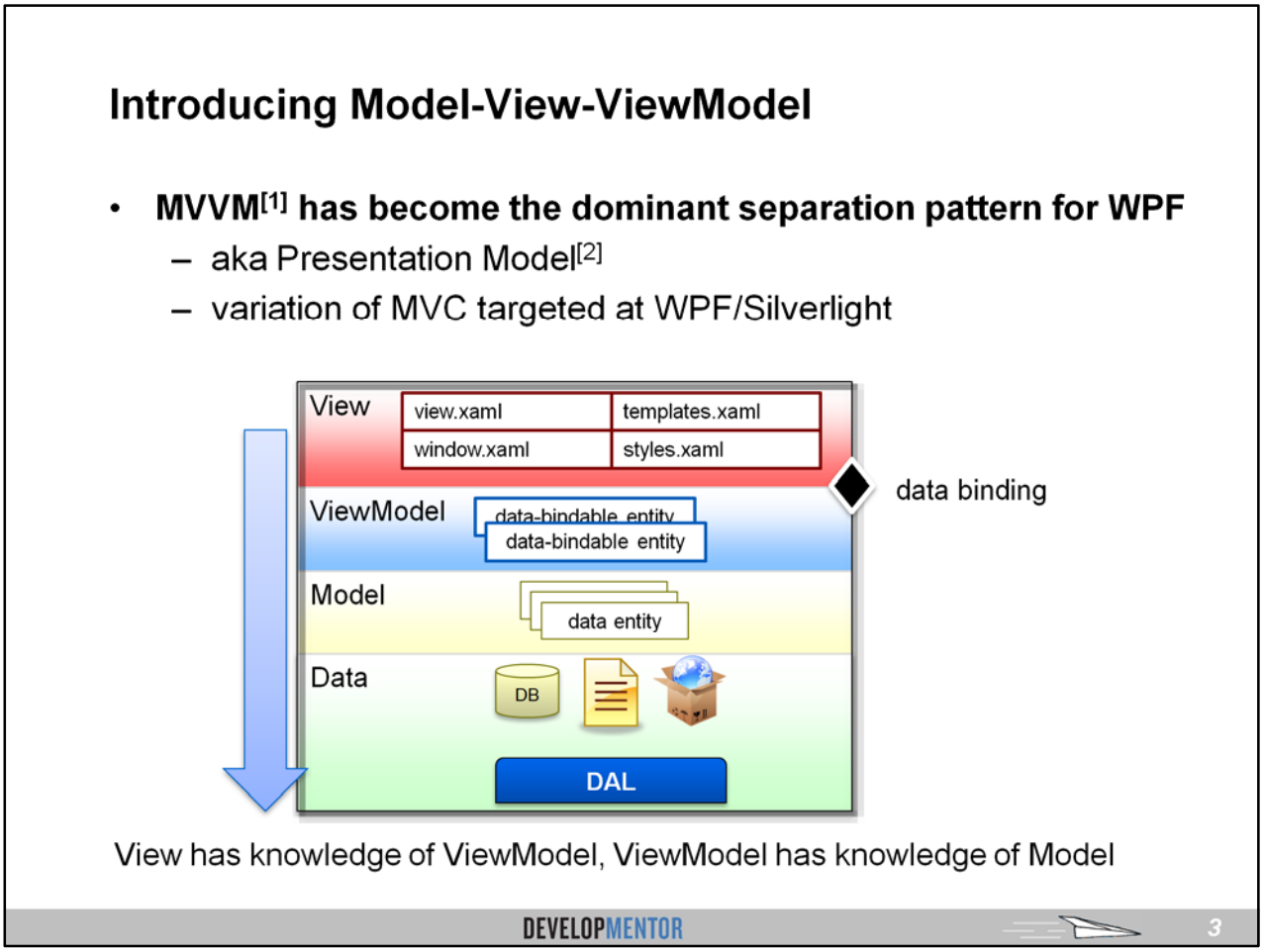
Formalizing the separation between code and UI

- **Good to decouple interaction between UI and code behind**
  - designer works on UI (XAML), developer works on code
  - enables unit testing and reduces contention between roles

goal is to avoid the spaghetti code monster!

DEVELOPMENTOR 2

9: Model-View-ViewModel



[1] MVVM was coined by John Gossman (WPF/Silverlight architect, originally part of the Blend team). You can read his take on MVVM at http://blogs.msdn.com/johngossman/archive/2005/10/08/478683.aspx

[2] http://martinfowler.com/eaaDev/PresentationModel.html

9: Model-View-ViewModel

What is the Model?

- **Model manages the application data and underlying state**
  - often persisted somewhere (DB, web service, etc.)
- **Represented as .NET objects**
  - should be reusable in other domains (not tied to WPF)

```
public class Employee
{
  public int Id { get; set; }
  public string Name { get; set; }
  public DateTime? TerminationDate { get; set; }
  public int Supervisor { get; set; }
  public double GetSalary();

  public static Employee GetById(int id);
  public void Update();
}
```


9: Model-View-ViewModel

What is the View?

- **View presents the information on the screen**
  - defined in terms of user interface elements
  - does not store data itself, only presents the data
  - should not contain any testable code

View is implement in XAML by designer role

everything "visual" should be defined at this level (colors, etc.)

can use `DataTemplates` or `UserControls` to generate visuals

DEVELOPMENTOR 5

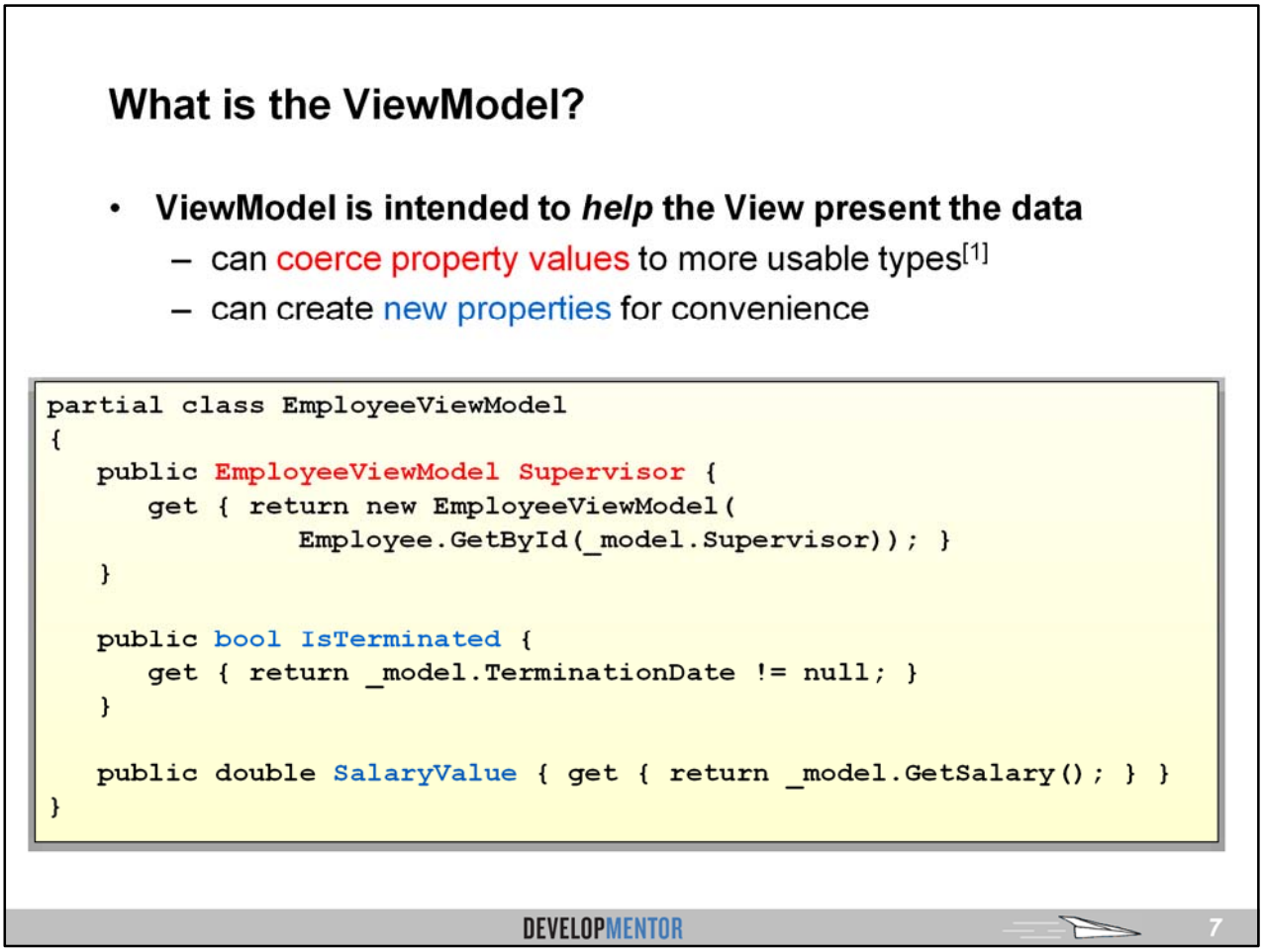
9: Model-View-ViewModel

What is the ViewModel?

- **ViewModel is intermediary between model and view**
  - "wraps" model object(s) and exposes properties
  - implements change notification for data binding use

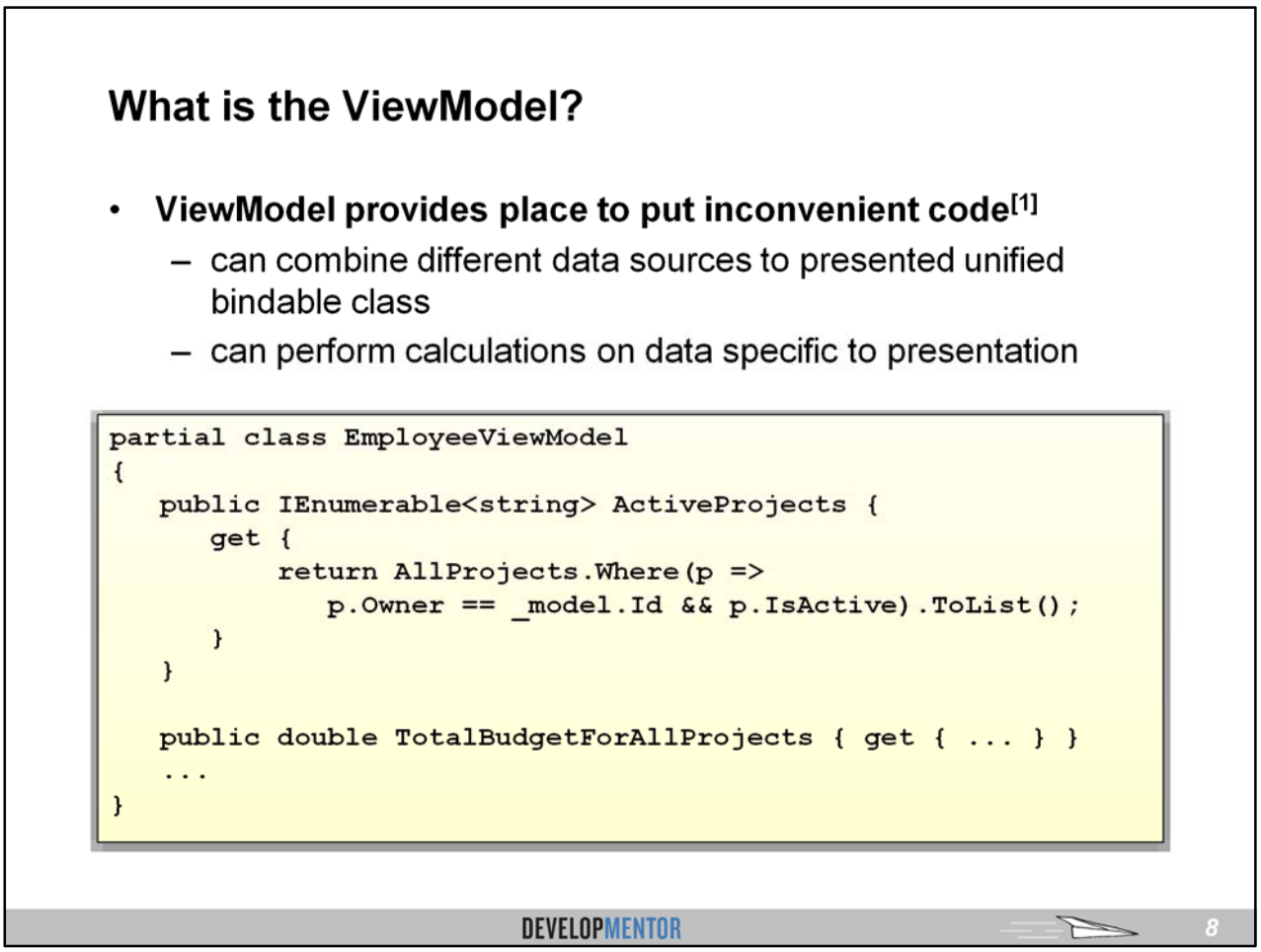
```
public class EmployeeViewModel : INotifyPropertyChanged
{
   private Employee _model;
   ...
   public string Name {
      get { return _model.Name; }
      set { _model.Name = value; OnPropertyChanged("Name"); }
   }
   ...
   public EmployeeViewModel(Employee model) {
       _model = model;
   }
}
```


9: Model-View-ViewModel



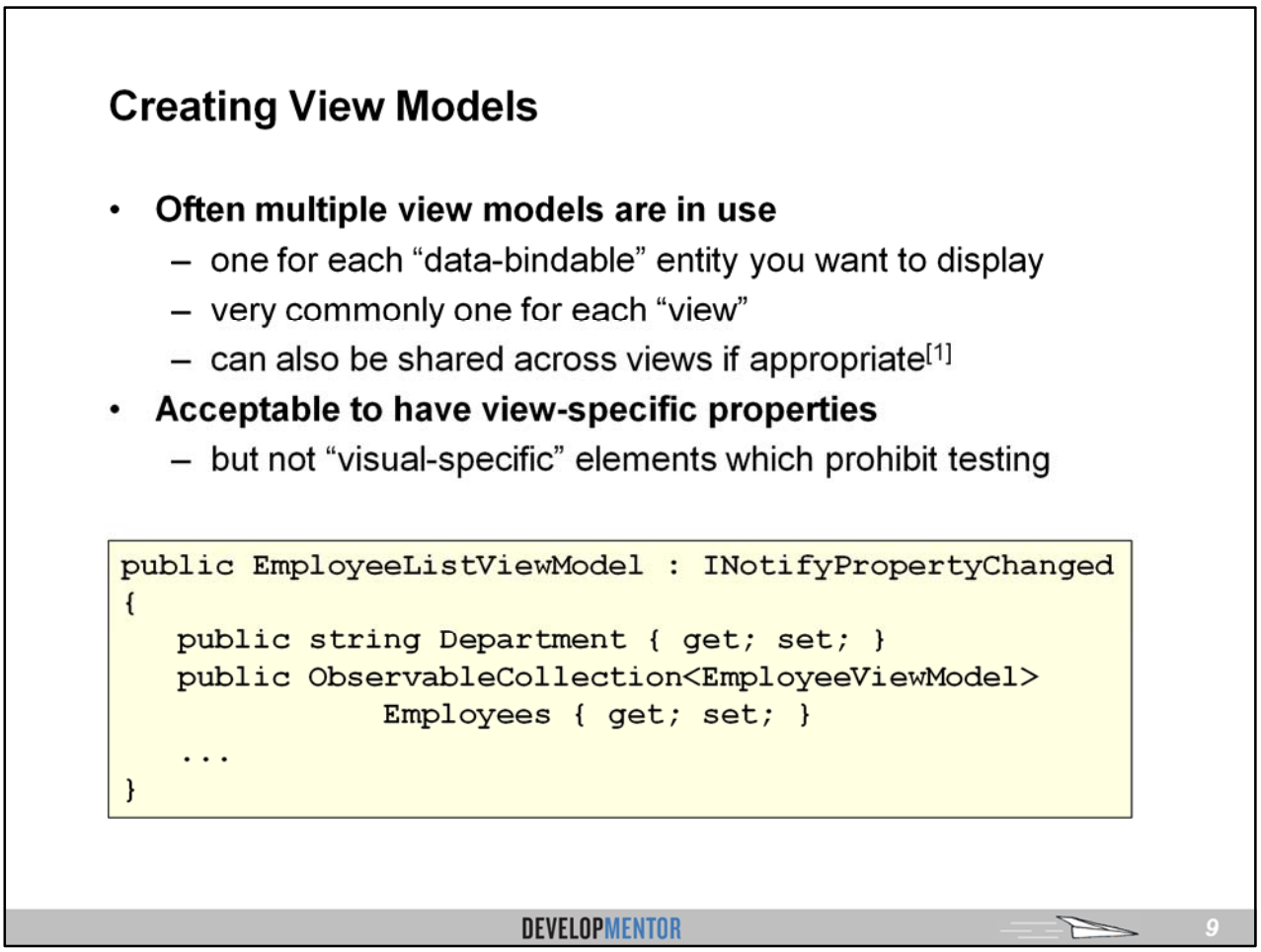
[1] An interesting thing comes out of this architecture – you will find that ValueConverters are almost never used in a full MVVM application. The ViewModel can expose the proper types to the View to make it natively bindable making the value conversion unnecessary in most cases.

9: Model-View-ViewModel



[1] The wall that most WPF programmers run into is trying to figure out where to put code that isn't visual, but isn't really business logic either. Things like the above TotalBudgetForAllProjects property are displayed, but aren't a candidate to place into the domain model objects because it requires dependencies across objects. Invariably this code ends up in the code behind and you end up using converters or some other "trick" to try to get the result you want. The ViewModel provides an elegant solution to this problem because it can contain these types of functions and properties.

9: Model-View-ViewModel



[1] As an example, consider MFC's document-view model where the document was often shared with multiple views.

Providing data to the view

- View model often set as data context for view

```
<Window ...>
   <Window.DataContext>
      <local:EmployeeListViewModel />
   </Window.DataContext>
...
```

```
partial class MainWindow
{
    public MainWindow()
    {
       DataContext = new EmployeeListViewModel();
       InitializeComponent();
    }
}
```

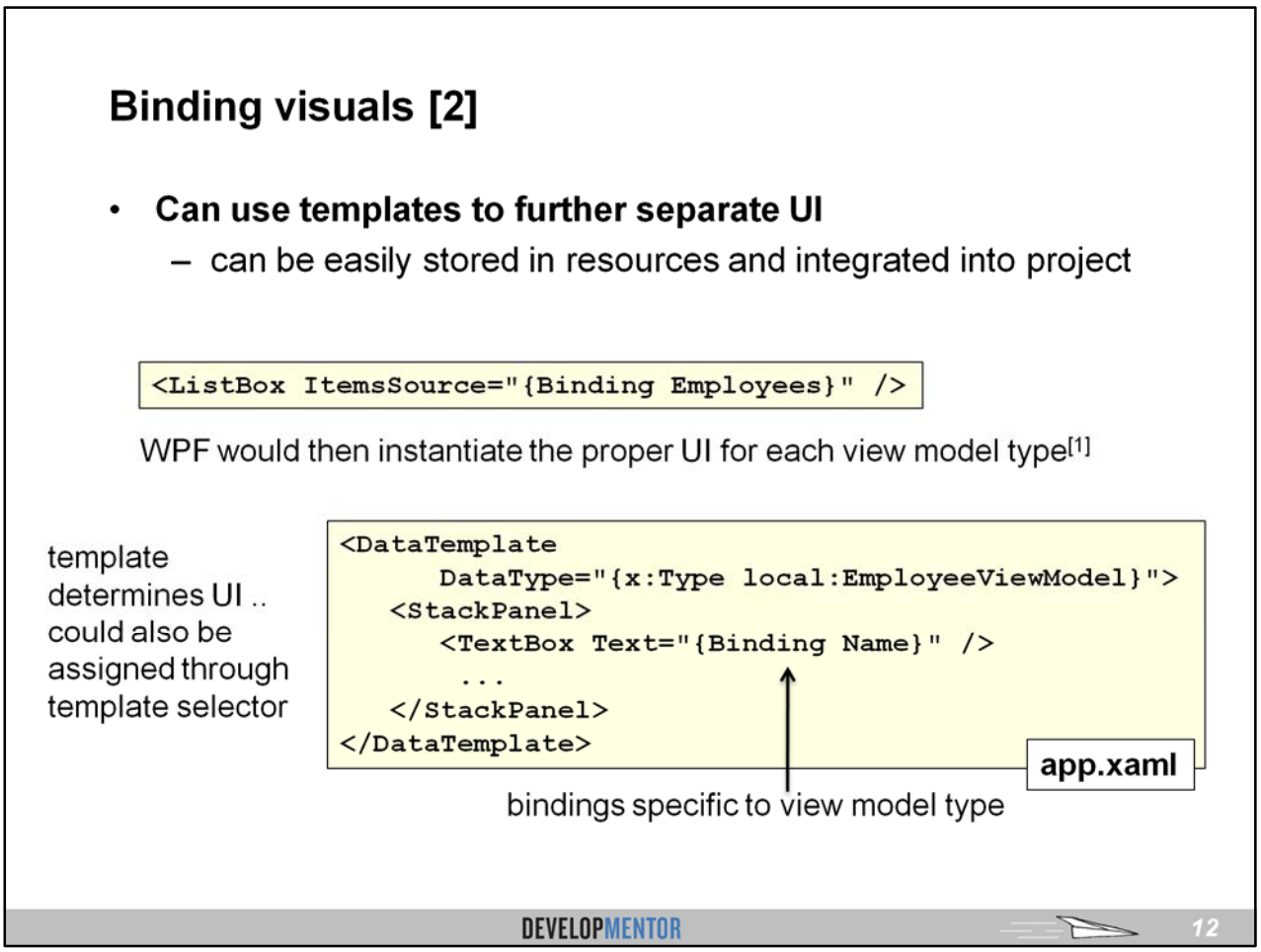

Binding visuals

- **View binds to properties of view model to present data**
  - two are "loosely coupled"

```
<UserControl>
    ...
   <TextBlock Text="{Binding Name}" />
    ...
</UserControl>
```

```
<Window>
    ...
   <dg:DataGrid
     ItemsSource="{Binding Employees}" />
    ...
</Window>
```


9: Model-View-ViewModel



[1] This is especially useful if you have multiple object types being rendered. For example:

<TabControl ItemsSource="{Binding ViewModels}" />

Could have different object types in the bound collection which generate a specific UI for each viewmodel type through a default data template:

<DataTemplate DataType="{x:Type local:ViewModel1}" />
<DataTemplate DataType="{x:Type local:ViewModel2}" />
<DataTemplate DataType="{x:Type local:ViewModel3}" />

View vs. ViewModel – what goes where?

- **General rule is to keep UI things in the xaml or xaml.cs file**
  - but do not be afraid to use the code behind when necessary!

```
<Window ...>
    ...
   <Button Content="Close Window" Click="OnClose" />

</Window>
```

  - can also forward notifications from View to ViewModel

```
private void OnClose(object sender, RoutedEventArgs e)
{
  this.Close();
  var viewModel = (EmployeeListViewModel) DataContext;
  viewModel.NotifyViewClose(this);
}
```

DEVELOPMENTOR  13

9: Model-View-ViewModel

View vs. ViewModel – visual behavior

- **ViewModel is tied to the View by design**
  - but is intended to be unit testable so avoid WPF elements

```
partial class EmployeeViewModel
{
   public Brush NameColor { get; }
}
```

Don't do this!
`Brushes` are WPF elements..

… this is better but still not ideal – colors should be determined by the designer..

```
partial class EmployeeViewModel
{
   public string TitleColor { get; }
}
```

… a better approach would be to provide a property allowing the view to decide the appropriate color based on whatever "triggers" the change (`IsSupervisor`, etc.)

9: Model-View-ViewModel

Coordinating visual behavior

- **Visuals often change based on property changes in VM**
  - triggers are ideal for this but only work with **Dependency Properties**

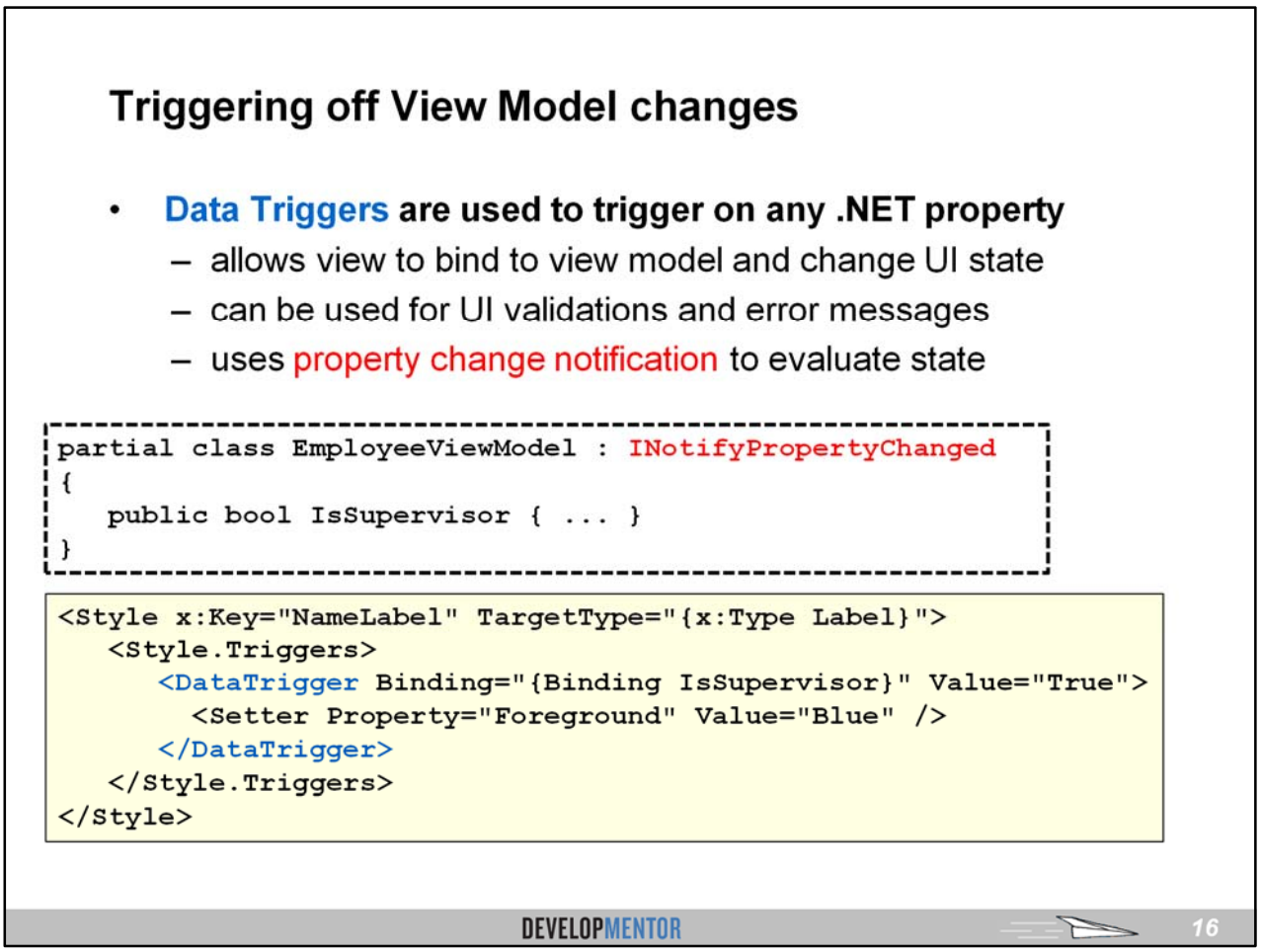
```
partial class EmployeeViewModel
{
   public bool IsSupervisor { get; }
}
```

want the **Name** label to be blue when employee is a supervisor

```
<Style x:Key="NameLabel" TargetType="{x:Type Label}">
   <Style.Triggers>
      <Trigger Property="IsSupervisor" Value="True">
        <Setter Property="Foreground" Value="Blue" />
      </Trigger>
   </Style.Triggers>
</Style>
```

DEVELOPMENTOR 15

9: Model-View-ViewModel



Expressing AND data trigger relationships

- `MultiDataTriggers` **allow multiple conditions to be applied**

.. want the background to be pink if the name is invalid and `TextBox` is focused


```
<Style TargetType="{x:Type TextBox}">
  <Style.Triggers>
    <MultiDataTrigger>
      <MultiDataTrigger.Conditions>
        <Condition
          Binding="{Binding IsNameValid}" Value="False" />
        <Condition
          Binding="{Binding IsKeyboardFocusWithin,
          RelativeSource={RelativeSource Self}}" Value="True" />
      </MultiDataTrigger.Conditions>
      <Setter Property="Background" Value="Pink" />
    </MultiDataTrigger>
  </Style.Triggers>
</Style>
```


Dealing with behavior

- **Need view model to handle behavior based on UI activity**
  - user clicks on button
  - user selects item in list
  - user slides slider
  - etc.

9: Model-View-ViewModel

Implementing behavior directly in view model

- **ViewModel can expose `ICommand` as property**
  - enables data binding

```
partial class EmployeeListViewModel
{
   public ICommand AddCommand { get; private set; }

   public EmployeeListViewModel() {
       AddCommand = new MyAddCommandLogic(this);
   }
}
```

view model can provide specific `ICommand` implementation for each action

```
<Button Content="_Add New"
        Command="{Binding AddCommand}" />
```

view can data bind to property to initiate behavior

Associating commands to actions

- **Reusable solution is to bind delegate shim to commands**
  - implements `ICommand` and forwards `Execute` to View Model

```
class DelegatingCommand : ICommand
{
   Action<object> _function;

   public void Execute(object parameter) {
      _function.Invoke(parameter);
   }

   public bool CanExecute(object parameter)  {...}
   public event EventHandler CanExecuteChanged;
}
```


There are multiple implementations of this pattern out in the wild. The lab has one, Prism (from P&P) has another. Pick your poison, or even roll your own.

Using DelegatingCommand in View Model

- Delegate forwards to methods on View Model

```
partial class EmployeeListViewModel
{
   public ICommand AddCommand { get; private set; }
   public ICommand FireCommand { get; private set; }

   public EmployeeListViewModel()
   {
        AddCommand = new DelegatingCommand(OnAddCommand);
        FireCommand = new DelegatingCommand(OnFireCommand);
        ...
   }

   private void OnAddCommand(object parameter)
   {
       ...
   }
}
```


Using Commands

- **WPF provides a limited set of `ICommandSource` elements**
  - `Button`
  - `Menu`
  - `Hyperlink`

```
<DataTemplate DataType="{x:Type local:EmployeeViewModel}">

   <Image Source="{Binding EmployeePicture}"
          Command="{Binding ShowDetailsCommand}" />
    ...
</DataTemplate>
```

`Image` does not support commands..
could wrap in a `Button` but is there a better way?

Altering behavior with attached properties

- **Dependency Properties can be "attached" to elements**
  - provides ability to alter behavior .. or raise commands!

```
<Window ...>

   <Image Source="{Binding EmployeePicture}"
     MouseBehavior.Click="{Binding ShowDetailsCommand}"
   />

</Window>
```

```
partial class MouseBehavior
{
   public static DependencyProperty ClickProperty = ...;
}
```


Attaching command behaviors

- **Attached property used to associate `ICommand` to event**
  - callback manages event registration

```
partial class MouseBehavior
{
   public static DependencyProperty ClickProperty =
       DependencyProperty.RegisterAttached("Click",
           typeof(ICommand), typeof(MouseBehavior),
               new FrameworkPropertyMetadata(null,
                  new PropertyChangedCallback(ClickChanged)));

   public ICommand GetClick(DependencyObject dpo) ...
   public void SetClick(DependencyObject dpo, ICommand command) ...
}
```


Attaching command behaviors [2]

```
partial class MouseBehavior
{
   static void ClickChanged(DependencyObject target,
                            DependencyPropertyChangedEventArgs e)
   {
      UIElement element = target as UIElement;
      if (element != null)
      {
         element.MouseButtonUp -= OnClick;
         if (e.NewValue != null)
            element.MouseButtonUp += OnClick;
      }
  }

  static void OnClick(object sender, EventArgs e)
  {
     var uie = (DependencyObject)sender;
     ICommand cmd = GetClick(uie);
     cmd.Execute(null);
  }
}
```


Managing selection with ICollectionView

- **Can use `ICollectionView` to manage selection**
  - but makes testing much harder ..

```
public EmployeeListViewModel : INotifyPropertyChanged
{
   EmployeeListViewModel()
   {
      ...
      ICollectionView cv = CollectionViewSource.
                           GetDefaultView(Employees);
      cv.CurrentPosition = 0;
   }
   ...
}
```

```
<ListBox ItemsSource="{Binding Employees}"
         IsSynchronizedWithCurrentItem="true"
         ItemTemplate="{StaticResource EmployeeTempl}" .. />
```

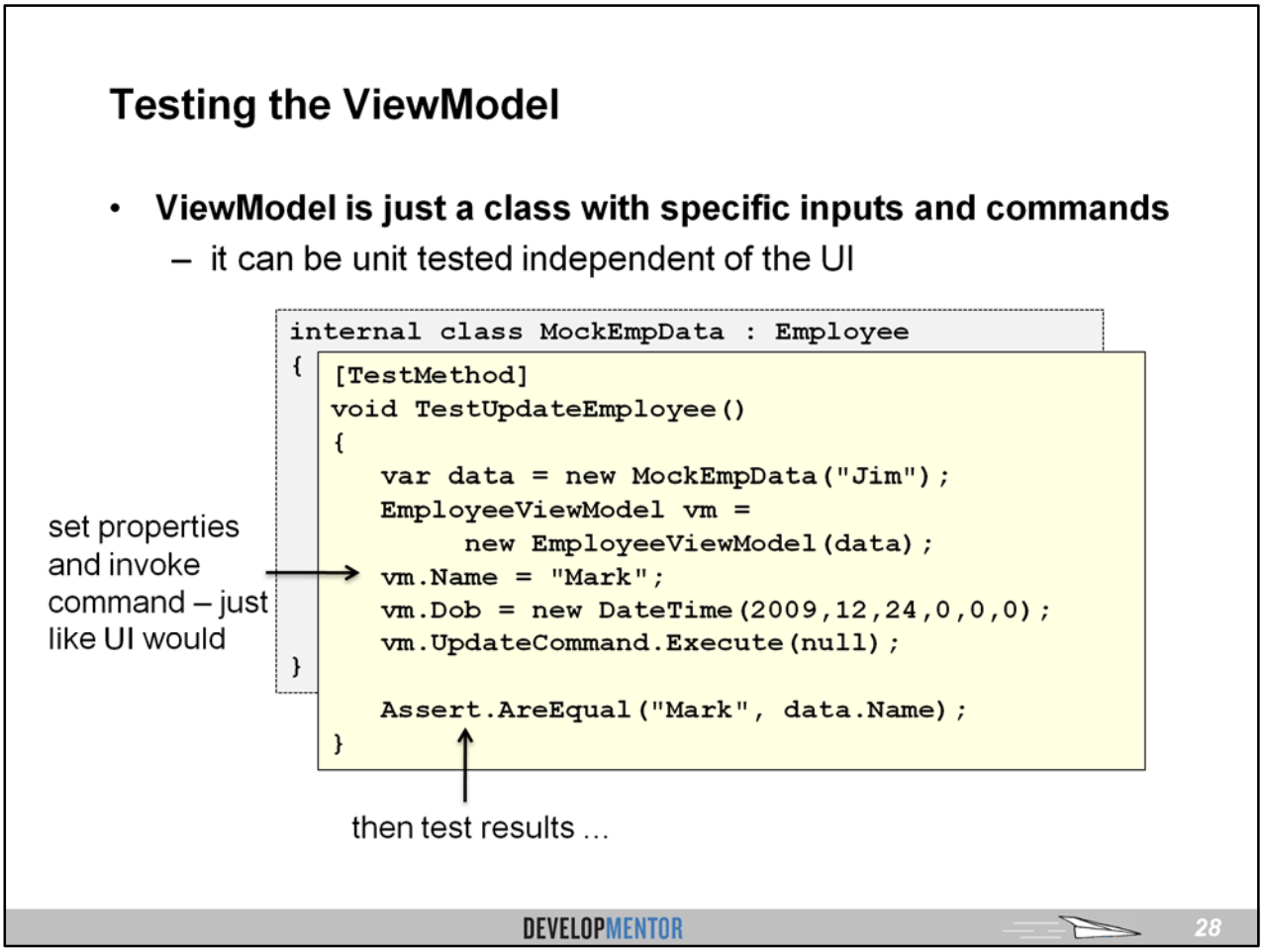

Managing selection directly

- View model can manage current item selection
  - breaks tie with `ICollectionView`

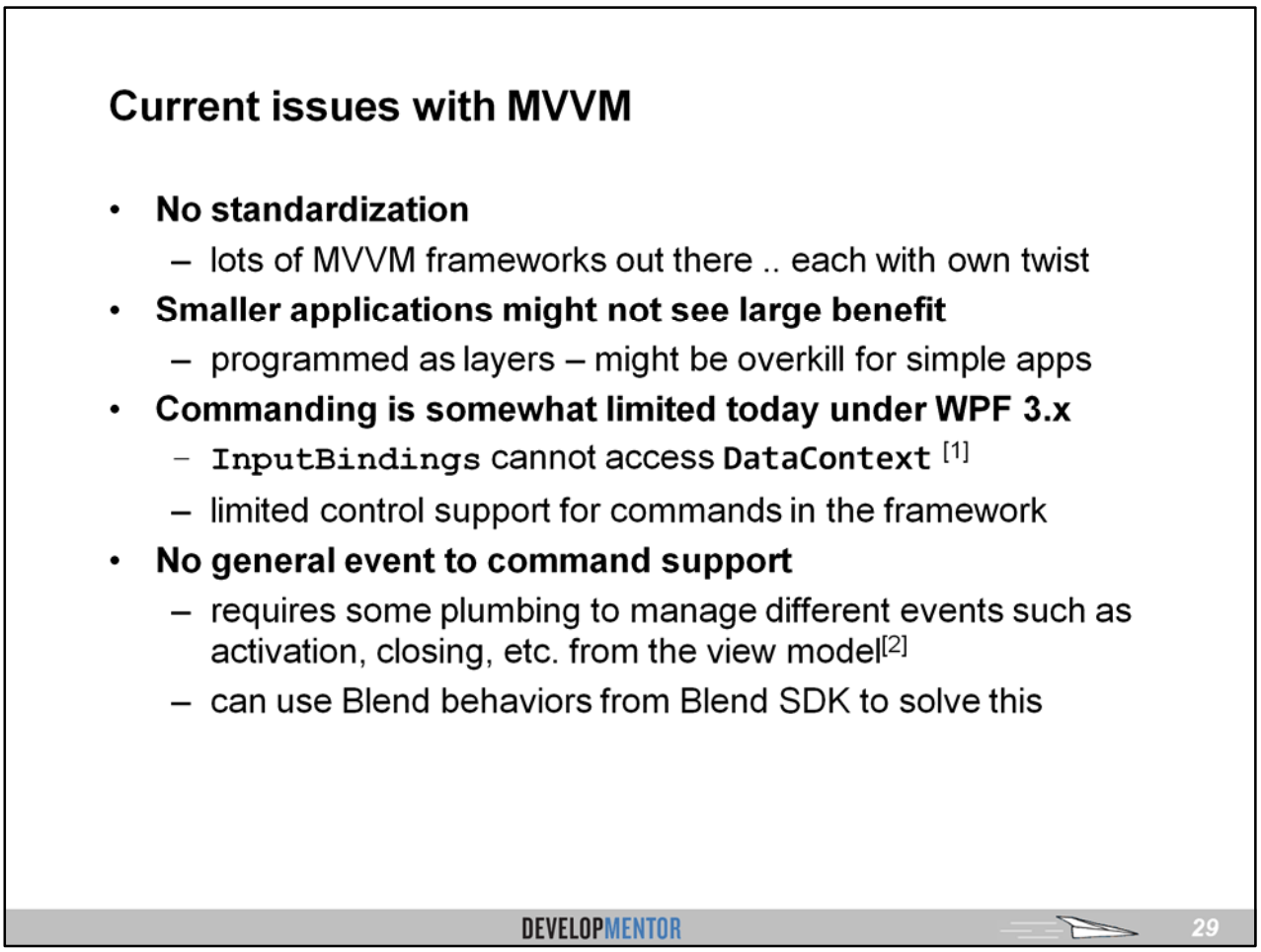
```
public EmployeeListViewModel : INotifyPropertyChanged
{
   private EmployeeViewModel _semp;
   public EmployeeViewModel SelectedEmployee
   {
      get { return _semp; }
      set { _semp = value; OnPropertyChanged("SelectedEmployee"); }
   }
   public EmployeeListViewModel()
   {
      SelectedEmployee = Employees[0];
   }
}
```

```
<ListBox ItemsSource="{Binding Employees}"
         SelectedItem="{Binding SelectedEmployee}"
         ItemTemplate="{StaticResource EmployeeTempl}" .. />
```


9: Model-View-ViewModel



Note that in order for this testing to be valid, the UI must not do any significant logic on it's own – everything must be performed by the ViewModel. This also does not reduce the need for functional end-to-end testing as the UI may not be invoking the logic properly and that can only be found through pushing real buttons.



[1] The WPF team has addressed this in WPF 4.0

[2] JulMar has a CommandEvent mapping class – see http://www.julmar.com/blog/mark. Alternatively, consider the InvokeCommand Blend Behavior, or CallMethod behavior also in Blend.

Summary

- **MVVM is a great way to segregate "developer" vs. "designer" responsibilities**
  - provides separation of concerns
  - provides testability of core logic
- **Lots of example frameworks out there**
  - http://wpf.codeplex.com (Microsoft's bare bones version)
  - http://mvvmhelpers.codeplex.com
  - http://mvvmlight.codeplex.com
  - http://caliburn.codeplex.com
  - http://compositewpf.codeplex.com
  - …
  - also "Data" project templates in Blend 4

