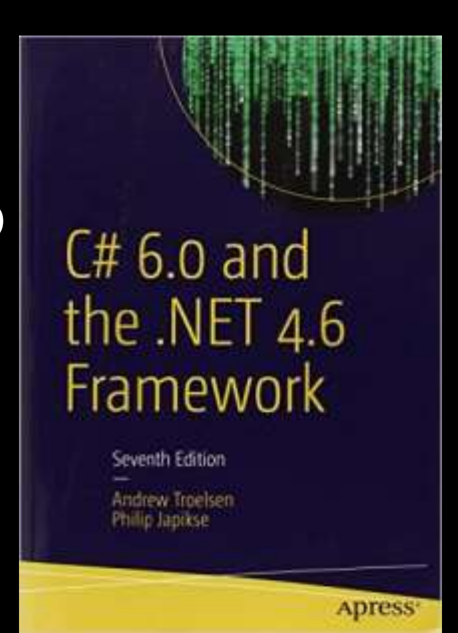
MODEL-VIEW-VIEWMODEL FOR WPF

Philip Japikse (@skimedic)
skimedic@outlook.com
www.skimedic.com/blog
Microsoft MVP, ASPInsider, MCSD, MCDBA, CSM, CSP
Principal Consultant/Architect, Strategic Data Systems

Phil.About()

- Principal Consultant/Architect, Strategic Data Systems
  - http://www.sds-consulting.com
- Developer, Coach, Author, Teacher
  - http://bit.ly/pro_csharp
- Microsoft MVP, ASPInsider, MCSD, MCDBA, CSM, CSP
- Founder, Agile Conferences, Inc.
  - http://www.dayofagile.org
- President, Cincinnati .NET User's Group



Hallway Conversations Podcast

- Hosted by Phil Japikse, Steve Bohlen, Lee Brandt, James Bender
- Website: www.hallwayconversations.com
- iTunes: http://bit.ly/hallway_convo_itunes
- Feed Burner: http://bit.ly/hallway_convo_feed
- Also available through Windows Store

THE MODEL VIEW VIEW-MODEL PATTERN

MODELS

- The data of the application
- (Can be) Optimized for storage

VIEWS

- Accepts Interactions from User
- Returns results of interactions back to user

VIEW MODEL – JOB 1

- Façade for individual models
- Transport mechanism for models

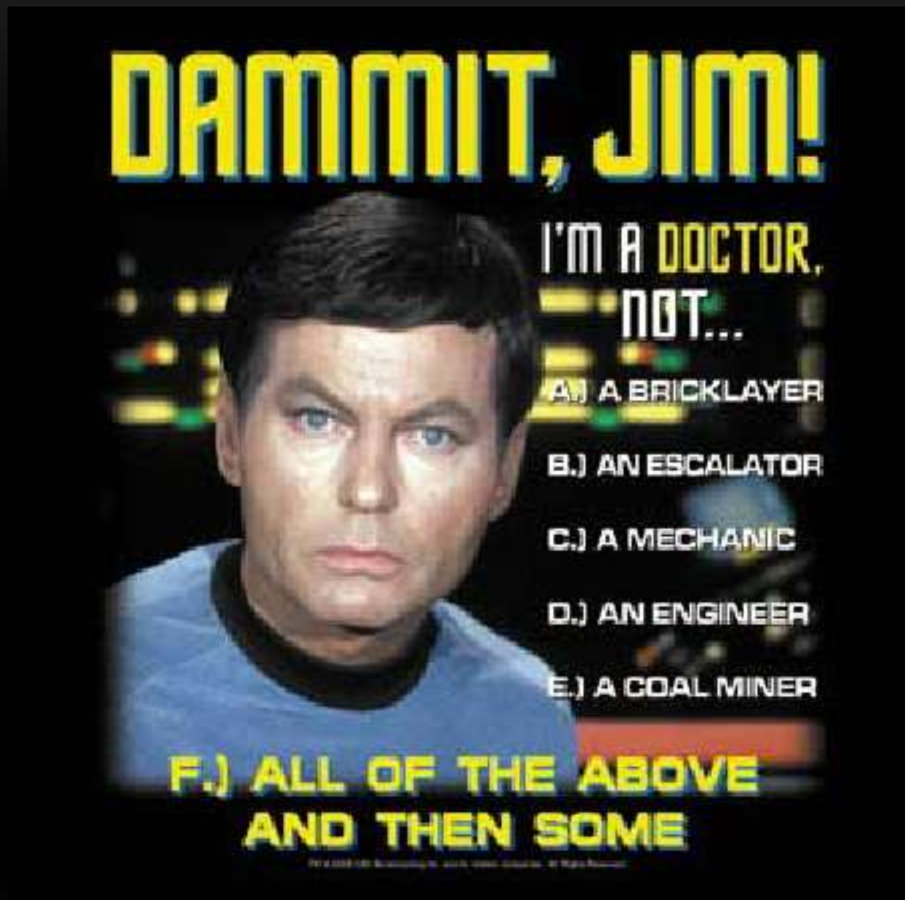
VIEW MODEL – JOB 2

- Process Incoming requests
- Perform changes to the model

WHY MVVM?

DON’T REPEAT YOURSELF

SEPARATION OF CONCERNS



MVVM != 0 CODE BEHIND

- Temper your actions with wisdom
- Code Behind is hard to test
- Complicated markup is hard to understand

IMPLEMENTING MVVM IN WPF

THE COMMAND PATTERN

ENCAPSULATING LOGIC

- Wire commands through markup
- Reuse command code (where appropriate)
- Leverage converters
  - IValueConverter
  - IMultiValueConverter

IMPLEMENTING ICOMMAND BY HAND-WPF

```
public class DoSomethingCommand : CommandBase
{
    public DoSomethingCommand()
    {
        //do something if necessary
    }
    public override void Execute(
         object parameter)
    {
        _messenger.Show("Clicked!");
    }
    public override bool CanExecute(
         object parameter)
    {
        return (parameter != null && (bool)
            parameter);
    }
}
```

```
public abstract class CommandBase : ICommand
{
 public abstract void Execute(
  object parameter);
  public abstract bool CanExecute(
  object parameter);

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

ROUTED AND DELEGATE/RELAY COMMANDS

- Both remove the need to create a new class for the command logic
- Routed Commands
  - Leverage routed events allowing handlers and invokers to be disconnected – don't need direct references to each other
  - Many implementations are provided by WPF
- Delegate/Relay Commands
  - Initially put forth by Josh Smith
  - Encapsulates ICommand interface implementation
  - Works by passing delegates in the ViewModel

RELAYCOMMANDS

```csharp
public class RelayCommand : ICommand
{
   private readonly Action _execute;
   private readonly Func<bool> _canExecute;
   public RelayCommand(Action execute) : this(execute, null)
   {
   }
   public RelayCommand(Action execute, Func<bool> canExecute)
   {
       if (execute == null)
          throw new ArgumentNullException("execute");
       _execute = execute;
       _canExecute = canExecute;
   }
   public bool CanExecute(object parameter)
   {
       return _canExecute == null || _canExecute();
   }
   public void Execute(object parameter)
   {
       _execute();
   }
   public event EventHandler CanExecuteChanged
   {
       add { CommandManager.RequerySuggested += value; }
       remove { CommandManager.RequerySuggested -= value; }
   }
}
```

```csharp
public class RelayCommand<T> : ICommand
{
   private readonly Action<T> _execute;
   private readonly Func<T, bool> _canExecute;
   public RelayCommand(Action<T> execute) : this(execute, null)
   {
   }
   public RelayCommand(Action<T> execute, Func<T, bool> canExecute)
   {
       if (execute == null)
           throw new ArgumentNullException("execute");
       _execute = execute;
       _canExecute = canExecute;
   }
   public bool CanExecute(object parameter)
   {
       return _canExecute == null || _canExecute((T)parameter);
   }
   public void Execute(object parameter)
   {
       _execute((T)parameter);
   }
   public event EventHandler CanExecuteChanged
   {
       add { CommandManager.RequerySuggested += value; }
       remove { CommandManager.RequerySuggested -= value; }
   }
}
```

VALUE CONVERTERS\COMMAND PARAMETERS

VALUE CONVERTERS

```
public class LoginMultiConverter : IMultiValueConverter
{
    public object Convert( object[] values, Type targetType,
   object parameter, System.Globalization.CultureInfo culture)
    {
        var param = new LoginParameter();
        //Omitted for brevity
        return param;
    }

    public object[] ConvertBack(object value, Type[] targetTypes,
   object parameter, System.Globalization.CultureInfo culture)
    {
        throw new NotImplementedException();
    }
}
```

PASSING IN PARAMETERS

```
<Button Command="{Binding Path=LoginCmd}">
    <Button.CommandParameter>
        <MultiBinding
     Converter="{StaticResource LoginMultiConverter}">
             <MultiBinding.Bindings>
                 <Binding Path="UserName" />
                 <Binding Path="Password" />
                 <Binding ElementName="pgLogin" />
             </MultiBinding.Bindings>
        </MultiBinding>
    </Button.CommandParameter>
</Button>
```

DEMO

The Command Pattern

OBSERVABLES

OBSERVABLES

- Models
  - Leverage INotifyPropertyChanged
    - Beware of magic strings
- Collections
  - Leverage ObservableCollections
    - Implements
      - INotifyCollectionChanged
      - INotifyPropertyChanged

INOTIFYPROPERTYCHANGED (PRIOR TO 4.5)

```
public class Product : INotifyPropertyChanged
{
    private string _modelName;
    public string ModelName
    {
        get { return _modelName; }
        set
        {
            if (_modelName == value) return;
            _modelName = value;
            onPropertyChanged(FieldNames.ModelName);
        }
    }
    public bool IsDirty { get; set; }

    public event PropertyChangedEventHandler PropertyChanged;
    private void onPropertyChanged(Enum fieldName)
    {
        IsDirty = true;
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(string.Empty));
            //PropertyChanged(this, new PropertyChangedEventArgs(fieldName.ToString()));
     }
    }
}
```

INOTIFYPROPERTYCHANGED (4.5+)

```
public class Product : INotifyPropertyChanged
{
    private string _modelName;
    public string ModelName
    {
        get { return _modelName; }
        set
        {
            if (_modelName == value) return;
            _modelName = value;
            OnPropertyChanged();
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    private void onPropertyChanged([CallerMemberName]string fieldname = "")
    {
        if (fieldname != "IsDirty") IsDirty = true;
        if (PropertyChanged != null)
        {
            PropertyChanged(this, new PropertyChangedEventArgs(string.Empty));
            //PropertyChanged(this, new PropertyChangedEventArgs(fieldName));
     }
    }
}
```

OBSERVABLECOLLECTIONS – THE HARD WAY

```
public class ProductList : IProductList
{
   private readonly IList<Product> _products;
   public ProductList(IList<Product> products)
   {
       _products = products;
   }
   public void Add(Product item)
   {
       _products.Add(item);
       notifyCollectionChanged(new
     NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction.Add, item));
   }
   public void Clear()
   {
       _products.Clear();
       notifyCollectionChanged(new
    NotifyCollectionChangedEventArgs(NotifyCollectionChangedAction.Reset));
   }
   //Ommitted for brevity
   public event NotifyCollectionChangedEventHandler CollectionChanged;
   private void notifyCollectionChanged(NotifyCollectionChangedEventArgs args)
   {
       if (CollectionChanged != null)
       {
           CollectionChanged(this, args);
       }
   }
}
```

OBSERVABLECOLLECTIONS – THE EASY WAY

- Use the built in ObservableCollection class
- Constructor takes an IEnumerable

```
Products = new
ObservableCollection<Product>(new
ProductService().GetProducts());
```

NOTIFYPROPERTYCHANGED – THE EASY WAY

- Add PropertyChanged.Fody
  - INotifyPropertyChanged is implemented for you
- Add IsChanged for dirty tracking

```
[ImplementPropertyChanged]
public abstract class ModelBase :
   INotifyDataErrorInfo, IDataErrorInfo
{
     public bool IsChanged { get; set; }
   // Omitted for brevity
}

public partial class Product : ModelBase
{
}
public partial class Product
{
    public int ID { get; set; }
    public string ModelName { get; set; }
    public string SKU { get; set; }
    public decimal Price { get; set; }
    public decimal SalePrice { get; set; }
    public int Inventory { get; set; }
}
```

WHERE TO IMPLEMENT NOTIFICATIONS?

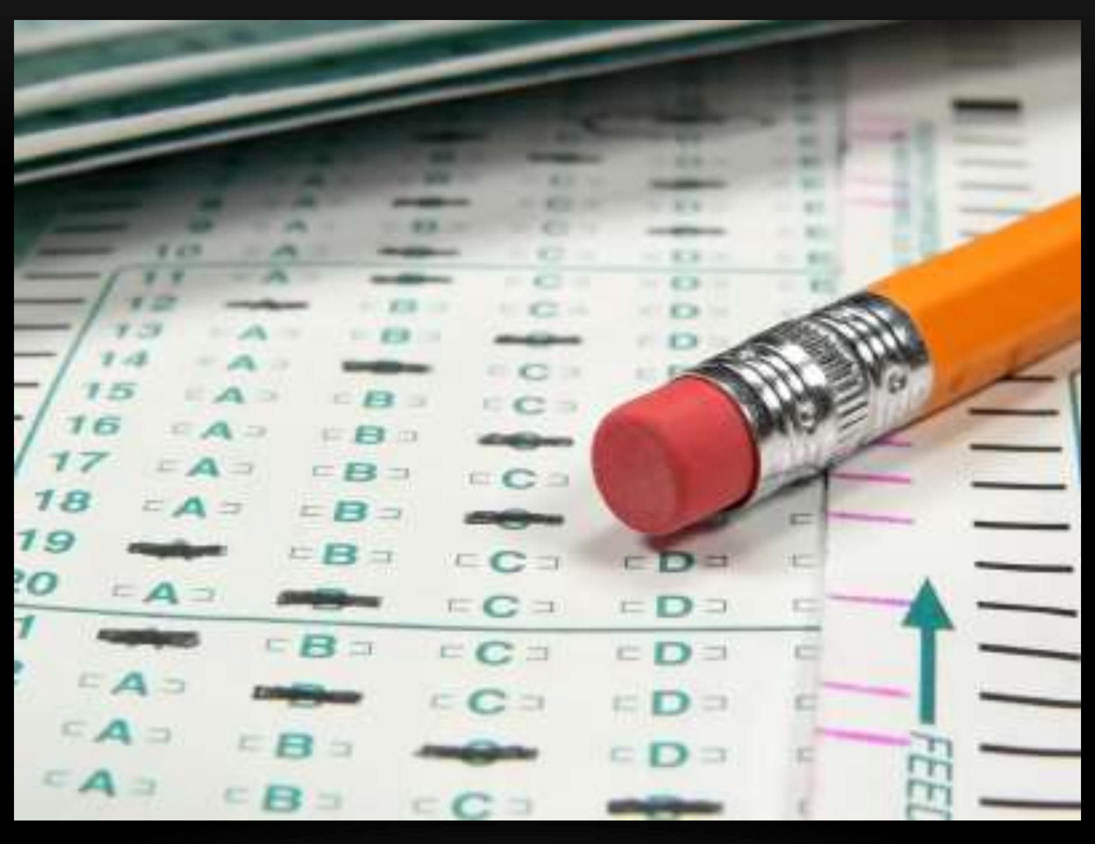
- Anemic Model
  - Implemented in ViewModel
  - “Cleaner” model
  - Lots of code duplication
- Anemic ViewModel
  - Implemented in Model
  - Less code duplication
  - Mixing of concerns?
- In general – do what makes sense

DEMO

DEMO

Observables

VALIDATION



VALIDATION METHODS

- Raise an error on your data object
- Use INotifyDataErrorInfo and/or IDataErrorInfo
  - INotifyDataErrorInfo is new in WPF 4.5
- Define validation at the binding level

- Note: Validation only occurs with TwoWay or OneWayToSource binding modes

VALIDATESONEXCEPTION

- Raises exceptions from bound property
  - Reminder: Errors are ignored on bindings, user won’t know the issue
- Sets Validation.HasError = True
- Creates ValidationError object
- If NotifyOnValidationError = true in binding
  - WPF raises Validation.Error

CUSTOM VALIDATION RULES

- Derive from ValidationRule
- Add additional properties for custom configuration
- Override Validate()
  - Return new ValidationResult(true || false, [Error Message])
- Add to Binding.ValidationRules collection

IDATAERRORINFO

- Adds an string indexer for the model
  - Error property is not used by WPF
- Relies on INotifyPropertyChanged to fire
- Requires ValidatesOnDataErrors to binding statement

IMPLEMENTING IDATAERRORINFO

```
public class Product : IDataErrorInfo
{
   public string this[string columnName]
  {
      get
   {
         var field = (FieldNames)Enum.Parse(typeof(FieldNames), columnName);
         switch (field)
         {
            case FieldNames.Inventory:
               if (Inventory < 0) { return "Inventory can not be less than zero";}
               break;
         case FieldNames.Price:
               if (Price < 0) {return "Price can not be less than zero";}
               break;
         }
         return string.Empty;
     }
  }
   public string Error { get { throw new NotImplementedException();} }
}
```

INOTIFYDATAERRORINFO

- New in WPF 4.5
- Contains
  - ErrorsChanged event
  - HasError Boolean
  - GetErrors() method to return the errors
- Relies on INotifyPropertyChanged
- ValidatesOnNotifyDataErrors is optional (set to true by default)

IMPLEMENTING INOTIFYDATAERRORINFO

```
private Dictionary<string, List<string>> errors = new Dictionary<string, List<string>>();
public event EventHandler<DataErrorsChangedEventArgs> ErrorsChanged;

public bool HasErrors { get { return (errors.Count > 0); } }
public IEnumerable GetErrors(string propertyName)
{
    if (string.IsNullOrEmpty(propertyName)) { return errors.Values; }
    else
    {
        if (errors.ContainsKey(propertyName)) { return (errors[propertyName]); }
        else { return null; }
    }
}
private void RaiseErrorsChanged(string propertyName)
{
    if (ErrorsChanged != null) {
        ErrorsChanged(this, new DataErrorsChangedEventArgs(propertyName));
    }
```

THE SUPPORT CODE

```
private void SetErrors(List<string> propertyErrors, [CallerMemberName]string propertyName = "")
{
    errors.Remove(propertyName);
    errors.Add(propertyName, propertyErrors);
    RaiseErrorsChanged(propertyName);
}
private void AddError(string error, [CallerMemberName]string propertyName = "")
{
    if (!errors.ContainsKey(propertyName)) { errors.Add(propertyName, new List<string>()); }
    if (!errors[propertyName].Contains(error))
    {
        errors[propertyName].Add(error);
        RaiseErrorsChanged(propertyName);
    }
}

private void ClearErrors([CallerMemberName]string propertyName = "")
{
    errors.Remove(propertyName);
    RaiseErrorsChanged(propertyName);
}
```

INOTIFYDATAERRORINFO VALIDATION IN THE SETTER

```
public decimal Price
{
  //Getter Omitted
  set
  {
    if (_price == value)
      return;
    _price = value;
    if (Price < 0)
    {
      var errors = new List<string>() { "Price can not be less than zero" };
      SetErrors(errors);
    }
    else
    {
      ClearErrors();
    }
    OnPropertyChanged();
  }
}
```

USING IDATAERRORINFO AND INOTIFYDATAERRORINFO

- Enables validation in generated code
- Use string indexer to call validation code
- Requires ValidatesOnDataErrors in binding statement
- ValidatesOnNotifyDataErrors is optional (set to true by default)

INOTIFYDATAERRORINFO AND IDATAERRORINFO

```
public string this[string columnName]
{
   get
   {
       var field = (FieldNames)Enum.Parse(typeof(FieldNames), columnName);
     switch (field)
     {
         case FieldNames.Inventory:
             ClearErrors(columnName);
           CheckSalePrice(columnName);
           if (Inventory < 0) { AddError("Inventory can not be less than zero", columnName); }
           CheckSalePrice(FieldNames.SalePrice.ToString());
           break;
         case FieldNames.SalePrice:
             ClearErrors(columnName);
           CheckSalePrice(columnName);
           if (SalePrice < 0) { AddError("Sale Price can not be less than zero", columnName); }
           CheckSalePrice(FieldNames.Inventory.ToString());
           break;
         default:
           break;
     }
     return string.Empty;
   }
}
```

ADDING VALIDATION RESULTS TO UI

LEVERAGING VALIDATION IN XAML

```
<TextBox Text="{Binding ElementName=ProductSelector,  Path=SelectedItem.ModelName,
 ValidatesOnExceptions=true, ValidatesOnDataErrors=true}" />

<Style.Triggers>
    <Trigger Property="Validation.HasError" Value="true">
        <Setter Property="Background" Value="Pink" />
        <Setter Property="Foreground" Value="Black" />
    </Trigger>
</Style.Triggers>
<Setter Property="Validation.ErrorTemplate">
    <Setter.Value>
        <ControlTemplate>
    <! Omitted for brevity -->
        </ControlTemplate>
    </Setter.Value>
</Setter>
```

DEMO

DEMO

Validation

BEHAVIORS

BEHAVIORS

- Introduced in Expression Blend v3
- Encapsulate functionality into reusable components
  - Drag & Drop
  - Pan and Zoom
  - Input Validation
  - Watermark Text
  - InvokeCommand
- Additional Behaviors are available from the Expression Gallery
  - http://msdn.microsoft.com/en-us/expression/jj873995

DEMO

DEMO

Behaviors

UI INTERACTION

CHALLENGES

- Challenges
  - Must keep separation of concerns
  - Must still be testable
- One suggestion
  - Create Interfaces representing UI
  - Code to Interface instead of element
- Another suggestion:
  - Create notification system
  - More complicated, more powerful

CREATE AND LEVERAGE INTERFACES

```
public interface IAnimationController
{
     void StartAnimation();
     void EndAnimation();
}


public override void Execute(object parameter)
{
       _param = parameter as LoginParameter;
       if (_param == null) { return;}
       _param.AnimationController.StartAnimation();
       _bgw.RunWorkerAsync(_param);
}
```

DEMO

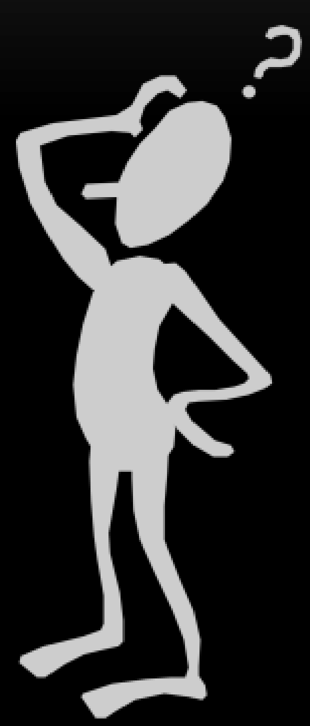
DEMO

UI Interaction

MVVM DECONSTRUCTED

- Models
  - The data of the application
  - (Can be) optimized for storage
- Views
  - Accepts interactions from user
  - Returns results of interactions
- ViewModel – Job 1
  - Façade for individual models
  - Transport mechanism for models
- ViewModel – Job 2
  - Process incoming requests
  - Perform changes to the model

Questions?



Contact Me

phil@sds-consulting.com
www.sds-consulting.com

skimedic@outlook.com
www.skimedic.com/blog
www.twitter.com/skimedic
www.hallwayconversations.com

www.about.me/skimedic