MVVM

- Når klasser og kollektioner implementerer INotifyPropertyChanged og InotifyCollectionChanged kan UI synkroniseres med data
- Med brug af et Observable mønster kan validering adderes
- Med WPF kommando system kan man skabe egne kommandoer til Indkapsling af program logik

Model

- Objektrepræsentation af data
- Indeholder ofte validering
- Konfigureres som Observable

View

- UI - ofte letvægt
- Indeholder ikke intelligens på nogen måde

ViewModel

- ViewModel er et single stop for al data som behøves af view.
- Det betyder ikke at ViewModel er ANSVARLIG for at hente data-
- Men kalder relevant kode (f.eks. EF)

Ansvar for validering og Observable pattern?

- Flere synspunkter
  - Nogle mente ViewModel (at bringe dette til Model er et brud på opdelingen)
  - Andre mente Model skulle have ansvaret (for at nedbringe duplikering af kode)
    - Det kommer an på!!
    - Hvis INotifyPropertyChanged, IDataErrorInfo, og
    - INotifyDataErrorInfo er implementeret i model klasser – lad det blive Model

WPF Binding notifikationssystem

- WPF binding system som bygger på XAML-baseret applikationer muliggør at binde dataobjekter I et notifikationssystem.

Observable models

- XAML

```
<Window x:Class="Notifications.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:local="clr-namespace:Notifications"
    mc:Ignorable="d"
    Title="Fun with Notifications!" Height="225" Width="325"
    WindowStartupLocation="CenterOwner">
  <Grid IsSharedSizeScope="True" Margin="5,0,5,5">
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>
    <Grid Grid.Row="0">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto" SharedSizeGroup="CarLabels"/>
        <ColumnDefinition Width="*"/>
      </Grid.ColumnDefinitions>
      <Label Grid.Column="0" Content="Vehicle"/>
      <ComboBox Name="cboCars"  Grid.Column="1" DisplayMemberPath="PetName" />
    </Grid>
```

(se komplet XAML i bogen)

- Data

```
public class Inventory
{
  public int CarId { get; set; }
  public string Make { get; set; }
  public string Color { get; set; }
  public string PetName { get; set; }
}
```

- Addering af bindings og data
- SelectedItem

```
<Grid Grid.Row="1" DataContext="{Binding ElementName=cboCars, Path=SelectedItem}">
```

- Husk controls navigerer op indtil DataContext findes

```
<TextBox Grid.Column="1" Grid.Row="0" Text="{Binding Path=Make}" />
<TextBox Grid.Column="1" Grid.Row="1" Text="{Binding Path=Color}" />
<TextBox Grid.Column="1" Grid.Row="2" Text="{Binding Path=PetName}" />
```

- Addering af data til ComboBox

```
using Notifications.Models;
public class MainWindow : Window
{
  readonly IList<Inventory> _cars;
  public MainWindow()
  {
    InitializeComponent();
    _cars = new List<Inventory>
    {
      new Inventory {CarId=1,Color="Blue",Make="Chevy",PetName="Kit" },
      new Inventory {CarId=2,Color="Red",Make="Ford",PetName="Red Rider" },
    };
    cboCars.ItemsSource = _cars;
    }
}
```

Data opdatering

- Knap til opdatering af color

```
<Button x:Name="btnAddCar" Content="Add Car" Margin="5,0,5,0" Padding="4, 2"
Click="btnAddCar_Click"/>
```

- Eventhandler

```
private void btnChangeColor_Click(object sender, RoutedEventArgs e)
{
  var car = _cars.FirstOrDefault(x => x.CarId == ((Inventory)cboCars.SelectedItem)?.CarId)
  if (car != null)
  {
      car.Color = "Pink";
  }
}
```

- Men ingen UI opdatering !!!
- Se bogen for indførsel af kode til skabelse af nyt objekt

UI opdatering

```
public interface INotifyPropertyChanged
{
  event PropertyChangedEventHandler PropertyChanged;
}
```

The `PropertyChanged` event takes an object reference and a new instance of the `PropertyChangedEventArgs` class, like this:

```
PropertyChanged?.Invoke(this, new PropertyChangedEventArgs("Model"));
```

- Addering af kode til klassen

into helper method, or the update would not work. Starting in .NET 4.5, you can take advantage of the `[CallerMemberName]` attribute. This attribute assigns the name of the method (your property setter) that calls into your helper method to the `propertyName` parameter. Add a method to the `Inventory` class (named `OnPropertyChanged`) and raise the `PropertyChangedEvent` like this:

```
internal void OnPropertyChanged([CallerMemberName] string propertyName = "")
{
  PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

- Opdatering af automatiske properties:

Next, update each of the automatic properties in the `Inventory` class to have a full getter and setter with a backing field. When the value is changed, call the `OnPropertyChanged` helper method. Here is the `CarId` property updated:

```
private int _carId;
public int CarId
{
  get { return _carId; }
  set
  {
    if (value == _carId) return;
    _carId = value;
    OnPropertyChanged();
  }
}
```

- Ændringer – totalt – i klassen

```
using System.ComponentModel;
using System.Runtime.CompilerServices;
namespace Notifications.Models
{
  public class Inventory : INotifyPropertyChanged
  {
    private int _carId;
    public int CarId
    {
      get { return _carId; }
      set
      {
        if (value == _carId) return;
        _carId = value;
        OnPropertyChanged();
      }
    }

   private string _make;
   public string Make
   {
     get { return _make; }
     set
     {
       if (value == _make) return;
       _make = value;
       OnPropertyChanged();
     }
   }
```

```
private string _color;
public string Color
{
  get { return _color; }
  set
  {
    if (value == _color) return;
    _color = value;
    OnPropertyChanged();
  }
}

private string _petName;
public string PetName
{
  get { return _petName; }
  set
  {
    if (value == _petName) return;
    _petName = value;
    OnPropertyChanged();
  }
}
```

```
    public event PropertyChangedEventHandler PropertyChanged;

    internal void OnPropertyChanged([CallerMemberName] string propertyName = "")
    {
      PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
  }
}
```

- Nu reflekteres ændringerne!

nameof operator

Using nameof

A new feature in C# 6 is the `nameof` operator, which provides the string name of the item passed into the `nameof` method. You can use this in the calls to `OnPropertyChanged` in your setters, like this:

```
private string _color;
public string Color
{
  get { return _color; }
  set
  {
    if (value == _color) return;
    _color = value;
    OnPropertyChanged(nameof(Color));
  }
}
```

1338

Observable Collections

Observable Collections

The next problem to resolve is updating the UI when the contents of a collection change. This is done by implementing the `INotifyCollectionChanged` interface. Like the `INotifyPropertyChanged` interface, this interface exposes one event, the `CollectionChanged` event. Unlike the `INotifyPropertyChanged` event, implementing this interface by hand is more than just calling a method in the setter. You need to create a full `List` implementation and raise the `CollectionChanged` event any time your list changes.

The `CollectionChanged` event takes one parameter, a new instance of the `CollectionChangedEventArgs`. The `CollectionChangedEventArgs` takes one or more parameters in its constructor, based on the operation. The first parameter is always one of the `NotifyCollectionChangedAction enum` values, which informs the binding engine what changed with the list. The values for the `NotifyCollectionChangedAction enum` are shown in Table 30-1.

Table 30-1. *NotifyCollectionChangedAction enum Values*

Member	Meaning in Life
`Add`	One or more times were added to the collection.
`Move`	One or more items moved in the collection.
`Remove`	One or more items were removed from the collection.
`Replace`	One or more items were replaced in the collection.
`Reset`	So much changed that the best option is to start over and rebind everything related to the collection.

Table 30-2. *NotifyCollectionChangedEventArgs Constructor Options*

Operation	Additional Parameters
`Reset`	None
`Add (single)`	Item to be added, [optional] index of location for add
`Add (List)`	Items to be added, [optional] index of location for add
`Remove (single)`	Item to be removed, [optional] index of item to be removed
`Remove (List)`	Items to be removed, [optional] start index of location for removal
`Move (Single)`	Item to be moved, original index, destination index
`Move (List)`	Items to be moved, start original index, destination index
`Replace (single)`	Item to be added, item to be removed, [optional] index of change
`Replace (List)`	Items to be added, items to be removed, [optional] starting index of change

- Se kodeeksempel (Ilist<Inventory>) s. 1340-1343

Implementering af dirty flag

- Tracker tilstandsændringer
- Et objekt har skiftet tilstand

tracking (tracking when one or more of an object's values have changed) is trivial. Add a bool property named IsChanged to the Inventory class. Make sure to call OnPropertyChanged just like the other properties in the Inventory class.

```
private bool _isChanged;
public bool IsChanged {
  get { return _isChanged;}
  set
  {
    if (value == _isChanged) return;
    _isChanged = value;
    OnPropertyChanged();
  }
}
```

At the end of that Grid, add a Label and a CheckBox, and then bind the CheckBox to the IsChanged property as follows:

```
<Label Grid.Column="0" Grid.Row="4" Content="Is Changed"/>
<CheckBox Grid.Column="1" Grid.Row="4" VerticalAlignment="Center"
    Margin="10,0,0,0" IsEnabled="False" IsChecked="{Binding Path=IsChanged}" />
```

You need to set the IsChanged property to true anytime another property changes. The best place to do this is in the OnPropertyChanged helper method, since it is called any time a property changes. You also need to make sure that you aren't setting IsChanged to true when IsChanged is changed, or you will hit a stack overflow exception! Open Inventory.cs and update the OnPropertyChanged method to the following (which uses the nameof method discussed earlier):

```
protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = "")
{
  if (propertyName != nameof(IsChanged))
  {
    IsChanged = true;
  }
  PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
```

If you were run the app now, you would see that every single record shows up as changed, even though you haven't changed anything! This is because object creation sets property values, and setting any values calls `OnPropertyChanged`. This sets the object's `IsChanged` property. To correct this, set the `IsChanged` property to `false` as the last property in the object initialization code. Open `MainWindow.xaml.cs` and change code to create the list to the following:

```
_cars = new ObservableCollection<Inventory>
{
  //IsChanged must be last in the list
  new Inventory {CarId=1,Color="Blue",Make="Chevy",PetName="Kit", IsChanged = false},
  new Inventory {CarId=2,Color="Red",Make="Ford",PetName="Red Rider", IsChanged = false },
};
```

Opdateringstidspunkt (UI interaction)

- UpdateSourceTrigger bestemmer hvornår der opdateres
- Default er LostFocus

Table 30-3. *UpdateSourceTrigger Values*

Member	Meaning in Life
`Default`	Set to the default for the control (e.g. `LostFocus` for `TextBoxes`).
`Explicit`	Updates source object only when the `UpdateSource` method is called.
`LostFocus`	Updates when the control loses focus. Default for `TextBoxes`.
`PropertyChanged`	Updates as soon as the property changes. Default for `CheckBoxes`.

Validering

- Validation klassen

Table 30-4. *Key Members of the `Validation` Class*

Member	Meaning in Life
`HasError`	Attached property indicating that a validation rule failed somewhere in the process
`Errors`	Collection of all active `ValidationError` objects
`ErrorTemplate`	Control template that becomes visible and adorns the bound element when `HasError` is set to true

- Eksempel (CarID = int og skal være der)

To test this, run the app, select a record from the ComboBox, and clear out the `Id` value. Recall from the definition of the `CarId` property that you added earlier, that it is defined as an `int` (not a `nullable int`), so a numeric value is required. When you tab out of the Id field, an empty string is sent to the `CarId` property by the binding framework, and since an empty string can't be converted to an `int`, an exception in thrown in the setter. However, there isn't any indication to the user since you haven't yet opted in to displaying exceptions.

Doing so is easy; all you need to do is add `ValidatesOnExceptions = true` to the binding statements. Update the binding statements in `MainWindow.xaml` to include `ValidatesOnExceptions = true`, as shown here:

```
<TextBox Grid.Column="1" Grid.Row="0"
  Text="{Binding Path=CarId, ValidatesOnExceptions=True}" />
<TextBox Grid.Column="1" Grid.Row="1"
  Text="{Binding Path=Make, ValidatesOnExceptions=True}" />
<TextBox Grid.Column="1" Grid.Row="2"
  Text="{Binding Path=Color, ValidatesOnExceptions=True}" />
<TextBox Grid.Column="1" Grid.Row="3"
  Text="{Binding Path=PetName, ValidatesOnExceptions=True}" />
```

IDataErrorInfo

- Adderer validering til Model klasser
- Interfacet har en indexer og en streng Error

```
public interface IDataErrorInfo
{
  string this[string columnName] { get; }
  string Error { get; }
}
```

- Implementering

```
public partial class Inventory : IDataErrorInfo
{
  public string this[string columnName]
  {
    get { return string.Empty; }
  }

  public string Error { get; }
}
```

- Indexer kaldes hver gang PropertyChanged eventet sker
- Property navnet benyttes som ColumnName parameter i Indexer
- Hvis Indexer returnerer String.Empty er det gået godt

Next, you will add some simple validation logic to the indexer in `InventoryPartial.cs`. The validation rules are simple:

- If `Make` equals ModelT, set the error equal to "Too Old".
- If `Make` equals Chevy and `Color` equals Pink, set the error equal to $"{Make}'s don't come in {Color}".

statements, you will again use the `nameof` method. If the code falls through the switch statement, return `string.Empty`. The code is shown here:

```
public string this[string columnName]
{
  get
  {
    switch (columnName)
    {
      case nameof(CarId):
        break;
      case nameof(Make):
        break;
      case nameof(Color):
        break;
      case nameof(PetName):
        break;
    }
    return string.Empty;
  }
}
```

- Adder valideringsregler

```
public string this[string columnName]
{
  get
  {
    switch (columnName)
    {
      case nameof(CarId):
        break;
      case nameof(Make):
        if (Make == "ModelT")
        {
          return "Too Old";
        }
        return CheckMakeAndColor();
      case nameof(Color):
        return CheckMakeAndColor();
```

- Og

```
            case nameof(PetName):
                break;
        }
        return string.Empty;
    }
}

internal string CheckMakeAndColor()
{
    if (Make == "Chevy" && Color == "Pink")
    {
        return $"{Make}'s don't come in {Color}";
        //AddError(nameof(Color), $"{Make}'s don't come in {Color}");
        //hasError = true;
    }
    return string.Empty;
}
```

- Sidste trin:

```
<TextBox Grid.Column="1" Grid.Row="0"
    Text="{Binding Path=CarId, ValidatesOnExceptions=True, ValidatesOnDataErrors=True}" />
<TextBox Grid.Column="1" Grid.Row="1"
    Text="{Binding Path=Make, ValidatesOnExceptions=True, ValidatesOnDataErrors=True}" />
<TextBox Grid.Column="1" Grid.Row="2"
    Text="{Binding Path=Color, ValidatesOnExceptions=True, ValidatesOnDataErrors=True}" />
<TextBox Grid.Column="1" Grid.Row="3"
    Text="{Binding Path=PetName, ValidatesOnExceptions=True, ValidatesOnDataErrors=True}" />
```

- Problem: Røde adorners om TextBox ved fejl forsvinder ikke
- En måde at løse det på

There are two ways to fix this. The first is to change the `PropertyChangedEventArgs` to update every bound property by passing in `string.Empty` instead of a field name. As discussed, this causes the binding engine to update *every* property on that instance. Update the `OnPropertyChanged` method to this:

```
protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = "")
{
  if (propertyName != nameof(IsChanged))
  {
    IsChanged = true;
  }

   //PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
   PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(string.Empty));
 }
```

- Andre interfaces

INotifyDataErrorInfo (se side 1354- 1363)

Data Annotations (attributprogammering)

- Reference til: System.ComponentModel.DataAnnotations

```
[Required]
public int CarId

[Required, StringLength(50)]
public string Make

[Required, StringLength(50)]
public string Color

[StringLength(50)]
public string PetName
```

- Checking: Se side 1363 -1365

Egne Commands

an event handler directly, the `Execute` method of the command is executed when the event fires. The `CanExecute` method is used to enable or disable the control based on your custom code. In addition to the built-in commands you used in Chapter 27, you can create your own custom commands by implementing the `ICommand` interface. By using commands instead of event handlers, you gain the benefit of encapsulating application code, as well as automatically enabling and disabling controls based on business logic.

Implementing the ICommand Interface

As a quick review from Chapter 27, the `ICommand` interface is listed here:

```
public interface ICommand
{
  event EventHandler CanExecuteChanged;
  bool CanExecute(object parameter);
  void Execute(object parameter);
}
```

Eksempel

```
public class ChangeColorCommand : ICommand
{
  public bool CanExecute(object parameter)
  {
    throw new NotImplementedException();
  }
  public void Execute(object parameter)
  {
    throw new NotImplementedException();
  }
  public event EventHandler CanExecuteChanged;
}
```

Implementation

```
public override bool CanExecute(object parameter) => (parameter as Inventory) != null;
```

Og

```
public override void Execute(object parameter)
{
  ((Inventory)parameter).Color="Pink";
}
```

Opdatering af MainWindow.xaml.cs

The next change is to create an instance of this class that the `Button` can access. For now, you will place this in the code-behind file for the `MainWindow` (later in this chapter you will move this into a `ViewModel`). Open `MainWindow.xaml.cs` and delete the click event handler for the Change Color button, since you will replace this functionality with your command implementation.

Next, add a public property named `ChangeColorCmd` of type `ICommand` with a backing field. In the expression body for the property, return the backing property (make sure to instantiate a new instance of the `ChangeColorCommand` if the backing field is null).

```
private ICommand _changeColorCommand = null;
public ICommand ChangeColorCmd => _changeColorCommand ?? (_changeColorCommand = new
ChangeColorCommand());
```

Opdatering af MainWindow.xaml

```
<Button x:Name="btnhangeColor" Content="Change Color" Margin="5,0,5,0" Padding="4,2"
   Command="{Binding Path=ChangeColorCmd,
   RelativeSource={RelativeSource Mode=FindAncestor, AncestorType={x:Type Window}}}"
   CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}"/>
```

- Koden virker umiddelbart ikke da (selectedItem i ComboBox er null)

property is no longer null. However, you see that the button is still disabled.

This is because the `CanExecute` method fires when the Window first loads and then when the command manager instructs it to fire. Each command class has to opt in to the command manager. This is done with the `CanExecuteChanged` event, and is as simple as adding the following code to your `ChangeColorCommand.cs` class:

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

- Da koden skal bruges flere steder skabes en CommandBase klasse

This code needs to be in every custom command that you build, so it's best to create an abstract base class to hold it. Create a new class in the `Cmds` folder named `CommandBase`, set the class to abstract, add the `ICommand` interface, and implement the interface. Add a `using` for the `System.Windows.Input` namespace, and change the `Execute` and `CanExecute` methods to abstract. Finally, add in the `CanExecuteChanged` you just wrote. The full implementation is listed here:

```
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; }
  }
}
```

- Opdatering af ChangeColorCommand klassen

```
internal class ChangeColorCommand : CommandBase
{
    public override void Execute(object parameter)
    {
        ((Inventory)parameter).Color="Pink";
    }

    public override bool CanExecute(object parameter) =>
        (parameter as Inventory) != null;
}
```

- Se implementering af resterende Commands (s. 1372 – 1374)

MVVM eksempel 1

If you recall from the explanation of the MVVM pattern, the only code in the code-behind should be directly related to the UI. Any data needed by the View should be exposed to the View from the ViewModel (and optimally brought to the ViewModel from a repository). In your current project, the data is hard-coded in the code-behind, so the first step is to move the `Cars` collection from the code-behind to the View Model.

Start by adding a public property of type `IList<Inventory>` named `Cars`. In the constructor for the `ViewModel`, set the `Cars` property to a new `ObservableCollection<Inventory>`.

Your class should look like this:

```
public class MainWindowViewModel
{
  public IList<Inventory> Cars { get; set; }
  public MainWindowViewModel()
  {
    Cars = new ObservableCollection<Inventory>
    {
      new Inventory {CarId=1,Color="Blue",Make="Chevy",PetName="Kit", IsChanged = false},
      new Inventory {CarId=2,Color="Red",Make="Ford",PetName="Red Rider", IsChanged = false },
    };
  }
}
```

- MainWindow.xaml.cs (renses)

In MainWindow.xaml.cs, delete the old creation of the list (in the constructor), and the line setting the ItemSource for the ComboBox to the list. Leave the backing field (_cars) for now; you don't need it, but two of the commands require it, and deleting it now would make the compile fail. The only code left in your constructor should be InitializeComponent, like this:

```
public MainWindow()
{
  InitializeComponent();
}
```

- Data context sættes

finds a data context. In MVVM, the ViewModel class serves as the data context for the entire Window, so set the Window's data context to the ViewModel in the constructor. Add a using for MVVM.ViewModels, like this:

```
using MVVM.ViewModels;

public partial class MainWindow:Window
{
  public MainWindow()
  {
    InitializeComponent();
    this.DataContext = new MainWindowViewModel();
  }
```

The final change to make to the Window is to add the ItemSource back to the ComboBox. Open MainWindow.xaml, add the ItemsSource attribute to the ComboBox, and bind it to the Cars property on the ViewModel. You don't have to specify the data source since the ViewModel is the data context for the Window. Your markup should look like this:

```
<ComboBox Name="cboCars" Grid.Column="1" DisplayMemberPath="PetName"
    ItemsSource="{Binding Path=Cars}"/>
```

- Flytte Commands til ViewModel

_cars field. Add a using statement for MVVM.Cmds. Your code in the MainWindowViewModel will look like this:

```
private ICommand _changeColorCommand = null;
public ICommand ChangeColorCmd =>
    _changeColorCommand ?? (_changeColorCommand = new ChangeColorCommand());

private ICommand _addCarCommand = null;
public ICommand AddCarCmd =>
    _addCarCommand ?? (_addCarCommand = new AddCarCommand(Cars));
```

```
private ICommand _removeCarCommand = null;
public ICommand RemoveCarCmd =>
    _removeCarCommand ?? (_removeCarCommand = new RemoveCarCommand(Cars));
private bool CanAddCar() => Cars != null;
```

- Sidste trin

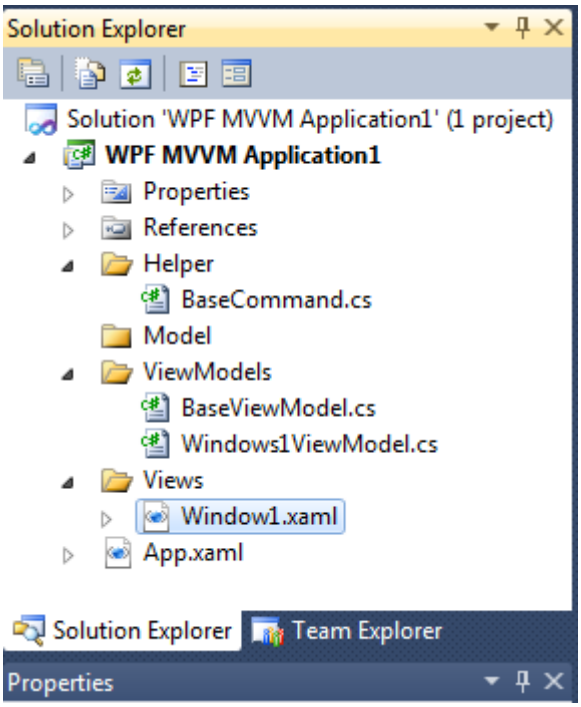
Finally, open `MainWindow.xaml`, and add `DataContext` to the Path for each of the `Button Command` binding statements.

```
<Button x:Name="cmdAddCar" Content="Add Car" Margin="5,0,5,0" Padding="4,2"
    Command="{Binding Path=DataContext.AddCarCmd,
    RelativeSource={RelativeSource Mode=FindAncestor, AncestorType={x:Type Window}}}"/>
<Button x:Name="cmdChangeColor" Content="Change Color" Margin="5,0,5,0"  Padding="4,2"
    Command="{Binding Path=DataContext.ChangeColorCmd,
    RelativeSource={RelativeSource Mode=FindAncestor, AncestorType={x:Type Window}}}"
    CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}"/>
<Button x:Name="btnRemoveCar" Content="Remove Car" Margin="5,0,5,0"  Padding="4,2"
    Command="{Binding Path=DataContext.RemoveCarCmd,
    RelativeSource={RelativeSource Mode=FindAncestor, AncestorType={x:Type Window}}}"
        CommandParameter="{Binding ElementName=cboCars, Path=SelectedItem}"/>
```

- Yderligere implementering af IDataErrorInfo og INotifyDataErrorInfo
- Se side 1376 – 1378
- (NuGet har pakker, der letter arbejdet (InotifyPropertyChanged implementation kræver mange tilpasninger i koden) . Se side 1378

MVVM eksempel 2

- Filstruktur



- Helper – BaseCommand.cs

```
using System;
using System.Windows.Input;

namespace WPF_MVVM_Application1.Helper
{
    internal class BaseCommand : ICommand
    {
        private readonly Action command;
        private readonly Func<bool> canExecute;

        public BaseCommand(Action command, Func<bool> canExecute = null)
        {
            if (command == null)
                throw new ArgumentNullException("command");
            this.canExecute = canExecute;
            this.command = command;
        }

        public void Execute(object parameter)
        {
            command();
        }

        public bool CanExecute(object parameter)
        {
            if (canExecute == null)
                return true;
            return canExecute();
        }

        public event EventHandler CanExecuteChanged;
    }
}
```

- ViewModels – BaseViewModel.cs

```
using System;
using System.ComponentModel;
using System.Linq.Expressions;
using System.Windows;
using System.Windows.Input;
using WPF_MVVM_Application1.Helper;

namespace WPF_MVVM_Application1.ViewModels
{
    public class BaseViewModel : INotifyPropertyChanged
    {
        #region WindowPropertys

        public void ShowMessageBox(string message)
        {
            MessageBox.Show(message, "", MessageBoxButton.OK, MessageBoxImage.Error);
        }

        public ICommand Close
        {
            get { return new BaseCommand(CloseApplication); }
        }

        public ICommand Maximice
        {
            get { return new BaseCommand(MaximiceApplication); }
        }

        public ICommand Minimice
        {
            get { return new BaseCommand(MinimiceApplication); }
        }

        public ICommand DragMove
        {
            get { return new BaseCommand(DragMoveCommand); }
        }

        public ICommand Restart
        {
            get { return new BaseCommand(RestartCommand); }
```

```
        }

        private static void RestartCommand()
        {
            Application.Current.Shutdown();
        }

        private static void DragMoveCommand()
        {
            Application.Current.MainWindow.DragMove();
        }

        private static void CloseApplication()
        {
            Application.Current.Shutdown();
        }

        private static void MaximiceApplication()
        {
            if (Application.Current.MainWindow.WindowState == WindowState.Maximized)
                Application.Current.MainWindow.WindowState = WindowState.Normal;
            else
                Application.Current.MainWindow.WindowState = WindowState.Maximized;
        }

        private static void MinimiceApplication()
        {
            if (Application.Current.MainWindow.WindowState == WindowState.Minimized)
            {
                Application.Current.MainWindow.Opacity = 1;
                Application.Current.MainWindow.WindowState = WindowState.Normal;
            }
            else
            {
                Application.Current.MainWindow.Opacity = 0;
                Application.Current.MainWindow.WindowState = WindowState.Minimized;
            }
        }

        #endregion

        #region Propertychanged

        protected void OnPropertyChanged<T>(Expression<Func<T>> action)
        {
            var propertyName = GetPropertyName(action);
            OnPropertyChanged(propertyName);
        }

        private static string GetPropertyName<T>(Expression<Func<T>> action)
        {
            var expression = (MemberExpression)action.Body;
            var propertyName = expression.Member.Name;
            return propertyName;
        }

        private void OnPropertyChanged(string propertyName)
        {
            PropertyChangedEventHandler handler = PropertyChanged;
            if (handler != null)
```

```
            {
                var e = new PropertyChangedEventArgs(propertyName);
                handler(this, e);
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;

        #endregion
    }
}
```

- ViewModels – Windows1ViewModel.c

```
using System;
using System.Windows.Input;
using WPF_MVVM_Application1.Helper;

namespace WPF_MVVM_Application1.ViewModels
{
    public class Windows1ViewModel : BaseViewModel
    {
        private DateTime textBoxTime;
        public ICommand ClickCommand { get { return new BaseCommand(Click); } }

        private void Click()
        {
            TextBoxTime = DateTime.Now;
        }

        public DateTime TextBoxTime
        {
            get { return textBoxTime; }
            set
            {
                textBoxTime = value;
                OnPropertyChanged(() => TextBoxTime);
            }
        }
    }
}
```

- Views - Window1.xaml

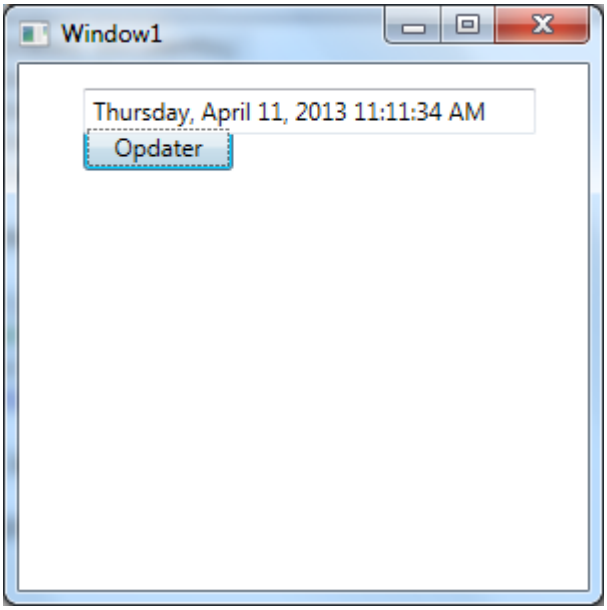
```
<Window x:Class="WPF_MVVM_Application1.Views.Window1"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" xmlns:me="clr-
namespace:WPF_MVVM_Application1.ViewModels" Title="Window1" Height="300" Width="300">
    <Window.Resources>
        <me:Windows1ViewModel x:Key="viewModel" />
    </Window.Resources>
```

```
    <Grid>
        <Button Content="Opdater" Height="23" HorizontalAlignment="Left" Margin="32,30,0,0"
Name="button1" VerticalAlignment="Top" Width="75" Command="{Binding Source={StaticResource
viewModel}, Path=ClickCommand}" />
        <TextBox Height="23" HorizontalAlignment="Left" Margin="32,12,0,0" Name="textBox1"
VerticalAlignment="Top" Width="226" Text="{Binding Mode=TwoWay,
UpdateSourceTrigger=PropertyChanged, Source={StaticResource viewModel}, Path=TextBoxTime,
ValidatesOnExceptions=True, StringFormat=\{0:F\}, NotifyOnValidationError=True,
ValidatesOnDataErrors=True}" />
    </Grid>
</Window>
```

- Views - Window1.xaml.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Shapes;

namespace WPF_MVVM_Application1.Views
{
    /// <summary>
    /// Interaktionslogik for Window1.xaml
    /// </summary>
    public partial class Window1 : Window
    {
        public Window1()
        {
            InitializeComponent();
        }
    }
}
```


Window1
Thursday, April 11, 2013 11:11:34 AM
Opdater