8. Patterns 6: Model-View-ViewModel

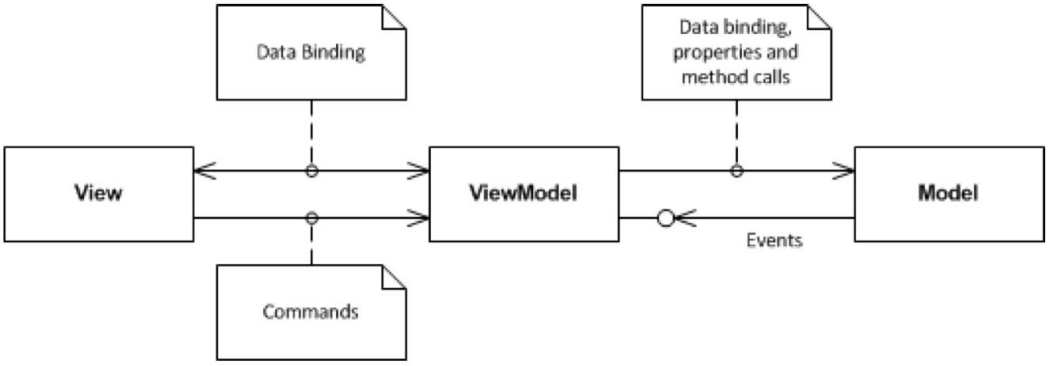
- Redegør for, hvad et software design pattern er.
- Redegør for Model-View-ViewModel mønstret og dets variationer.

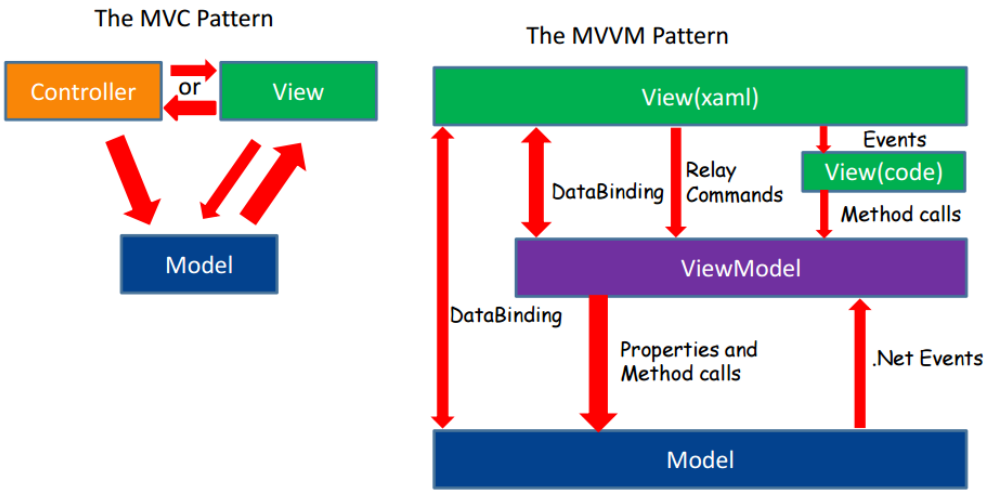
Software design pattern

- Et design pattern er en genbrugelig løsning på problemer der ofte opstår i udvikling af software
- Problemerne kan have meget forskellig karakter
  - Fx undgå kobling mellem klasser, løse problemer med tråde der skal snakke sikkert sammen osv.
- Man må selv tilpasse pattern til sin kode, det er altså ikke et færdigt stykke kode
- Giver programmører et ordforråd så vi hurtigt kan sætte os ind i hinandens kode (hvis man kender patterns)

Model-View-ViewModel

Vil gerne vise modellen på papir





- MVVM bygger på MVP, men er optimeret for WPF, Silverlight applikationer, Windows Phone ol.
  - Understøtter at viewet er skrevet i XAML og databinding
- Hjælper med at separere business logik (model) og præsentationslogik (view) i ens applikation.
  - Det bevirker bl.a. at man kan benytte samme ViewModel og Model og så ændre View'et nemt, eller lave et nyt View til en ny platform

Model:

- Er de domæne-specifikke klasser der er opstået ud fra fx navneordanalyse af kravspecifikationens use-cases eller systembeskrivelse
- Det er i modellen alt domæne specifik data ligger
    - Kan indeholde data-validering
- Modellen bør også indeholde alt applikationslogikken (men der er dog variationer hvor applog i VM)
- Modellen kan implementere INotifyPropertyChanged for at notificere ViewModellen om ændringer

View:

- Tager sig af at håndtere alt GUIen indeholder controller, styles og vinduer
    - Man ønsker som regel så lidt code-behind som muligt, og mest i XAML kode.
    - Men man kan godt have noget kode i code-behind fx animationer
- Hvert View vil have en reference til typisk 1 ViewModel som DataContext som den databinder til

ViewModel

- ViewModellen **udstiller** modellens attributter til View'et.
    - Fx kan modellen have en DateTime som kan laves om til en formatteret string i ViewModel så View'et binder til en string og ikke en DateTime
- Men ViewModel kan også blot udstille properties i modellen DIREKTE
    - Dvs. View'et kan databinde direkte ned i modellen
- Det smarte er at View'et kan **DATABINDE** ned i ViewModellen
    - Derved behøver ViewModellen ikke sørge for at opdatere View'et igennem et interface, som man skal i MVP
    - INotifyPropertyChanged skal være implementeret for de properties der skal databindes på
- Det er vigtigt at ViewModel IKKE kender til View. View kender dog til ViewModel
- På den måde er VM afkoblet View'et og nem at unit-teste

Kobling af View og ViewModel

Her går vi så også ind i **variationerne i MVVM**

- Der er forskellige måder at koble View'et sammen med ViewModel
- To metoder, View First og ViewModel First

View First:

```
Window dlg = new CreateComponent();
dlg.DataContext = locator.CreateComponentViewModel;
//Eller
dlg.DataContext = new CreateComponentViewModel(new Component());
dlg.ShowDialog();
```

- En simpel måde er at selv lave sit View og sætte DataContext'en til en ViewModel man laver
    - Man kan på denne måde selv give data med til ViewModel med oprettelse
    - Smart at man kan inject noget data med ind
- Man kan også oprette ViewModellen i XAML koden

```
<Window x:Class="CreatingAViewModelDeclaratively.MainWindow"
        xmlns:local="clr-namespace:BMICalculator"
        Title="BMI Calculator" Height="350" Width="525">
    <Window.DataContext>
        <local:BMIViewModel x:Name="viewModel"/>
    </Window.DataContext>
```

I vinduets XAML oprettes en VM

- En anden måde er at bruge en locator-pattern, dvs. en ViewModelLocator som man kan instantiere som en application ressource
    - Man kan bede locatoren om en specifik ViewModel
    - Man kan selv bestemme om der ønskes mulighed for oprettelse af en ny ViewModel
    - Eller slags "singleton" laves af hvert ViewModel (så man sikrer sig man får den samme VM)
    - Locator kan tjekke om vi er i designmode og kan derved lave noget dummy data

ViewModel First:

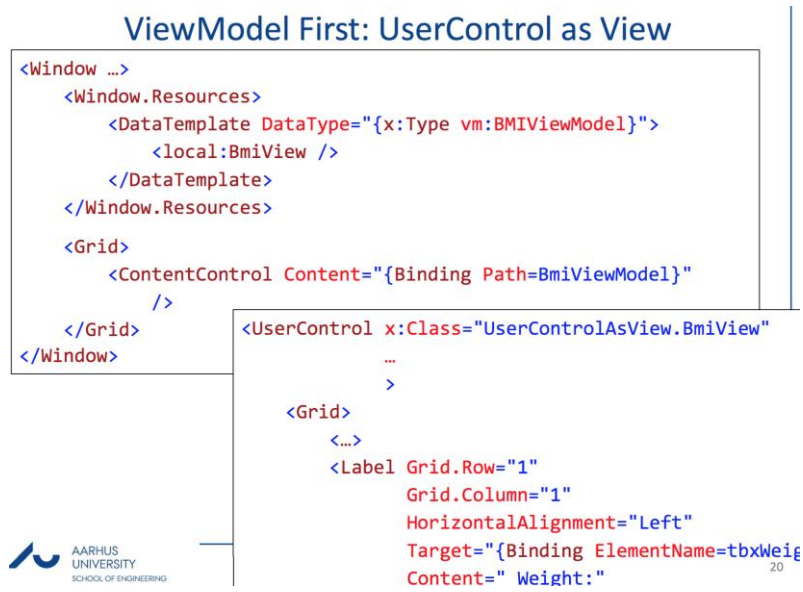
- Består i at når du laver din ViewModel så finder brugergrænsefladen selv ud af hvilket View den skal bruge til at repræsentere den ViewModel
- Kan gøre på forskellige måder:

VM-First: **Data-template as View**

```
<Window …>
    <Window.Resources>
        <DataTemplate DataType="{x:Type vm:BMIViewModel}">
            <Grid>
                <...>
            </Grid>
        </DataTemplate>
    </Window.Resources>
    <Grid>
        <ContentControl Content="{Binding Path=BmiViewModel}"
            />
    </Grid>
</Window>
```

- Viewet består af en DataTemplate som defineres for en ViewModel
- DataTemplate er en måde at definere hvordan ViewModellen skal vises
- Man kan så overskrive ContentControl på de kontroller man ønsker skal repræsentere en ViewModel

VM-First: **UserControl as View**



- UserControl er et stykke genbrugeligt XAML som kan bruges til at repræsentere en ViewModel
  - Samling af controller
  - Nederst på figuren
- Så overskriver man ContentControl til at pege på en DataTemplate som er sat til den UserControl

- Hvis man ændrer ViewModel vil View'et selv ændrer sig, da det skifter til et andet Data-template

Yderliger variationer i placering af business logik

- Nogle ligger business logikken hos ViewModel, nogle ligger hos Model
    - **MSDN artikler beskriver at business logikken bør ligge hos Model**
        - Mon ikke Microsoft har bedst styr på det?
        - Men dermed ikke sagt at det andet er forkert, da det jo er et pattern, som man må tilpasse sin aktuelle situation
    - Andre vil ligge logikken hos ViewModel som så manipulerer Model i overenstemmelse med business logikken. (Poul eksempel)