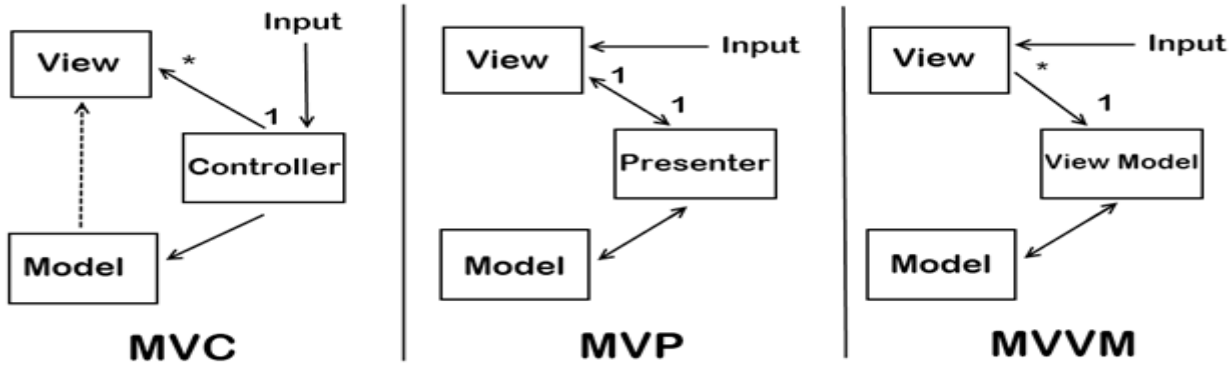
5. MVC & MVP Pattern

1. **Redegør for hvad software design pattern er**
2. **Redegør for i MVC Pattern og dets variationer**
    a. Forsøger at adskille GUI logik fra Forretningslogik
        i. Opdeler kode med forskelligt ansvar
    b. Model:
        i. Forretningslogikken
        ii. Afspejler databasen
        iii. Benytter dataannotationer som kan valideres i controller
        iv. Fjerner unødvendig forretningslogik fra controllere
        v. Ændringer i Model notificeres til modellens View
        vi. Frakoblet fra GUI, fremstiller kun data
            1. kan derfor bruges flere steder
    c. Controllers
        i. Beslutter hvilke ændringer der skal ske i Modellen
        ii. Ved et tryk på en knap, er det controlleren der fortæller View hvad der skal vises
            1. fortæller også Modellen hvad der skal ske
        iii. bruger sender et request fra view, controller tager inputtet og sender det til Modellen
            1. hvis modellen godkender input, sendes det til controller og videre til view
    d. View
        i. Præsentere Modellen for en bruger
        ii. Kan være implementeret som f.eks. et Strategy Pattern
            1. Controlleren giver viewet Strategien.
        iii. Kan være implementeret som et Composite Pattern
            1. kan bestå af flere elementer, ”Composites” indeholder ”leafs” eller andre ”Composites”
    e. Fordele/ulemper ved MVC
        i. + God til at vise en Model på flere måder
        ii. + Ingen Business Logic i UI
        iii. + Nem at Unit Teste
        iv. – Ikke skalerbart, separere UI men ikke Model
        v. – Controllere bliver ofte store (uoverskuelige)
        vi. – Overholder ikke Single Responsibility og interface Segregation principperne
3. Redegør for Model-View-Presenter og dets variationer
    a. MVP er en udvikling af MVC
    b. View får et interface som presenter benytter sig
        i. Øger testbarheden af presenter
    c. Model og View fungerer som i MVC
    d. Controller bliver en Presenter (her ligger forskellen fra MVC)
        i. Indeholder krævende business logic
    e. View indeholder kun lige præcis nok logik til at kalde funktionaliteten fra presenteren
    f. Sørger for at modtage og videgiver data korrekt fra model til view
    g. To typer:
    h. Passiv view

        i. View kender ikke til model
        ii. View har ingen logik
        iii. Ingen **code behind**
        iv. Al data binding skal skrives selv
    i. Supervising Presenter
        i. Viewet kender til modellen (modellen kan ikke opdatere view)
        ii. Presenteren interagere med Viewet med den opdaterede Model
    j. Se Eksempler i disposition
    k. For passivt view gælder at:
        i. View
            1. Består kun af en række kontroller der viser modellen, brugerinteraktion går fra view til presenter
        ii. Model
            1. Er repræsentation af data som i MVC
        iii. Presenter
            1. Manipulere modellen ud fra modtagne events fra viewet, opdaterer view gennem iView, interfacet står for formatering. GUI logik
    l. For Supervising Presenter gælder
        i. View
            1. Databinding er tilladt, ellers ingen ændringer
        ii. Presenter
            1. presenter kommunikerer med Modellen om hvad der skal ske i View
        iii. Der er ikke meget kode i presenter, men det er sværere at teste
    m. Fordele/ulemper:
        i. + komplekse opgaver deles ud i mindre opgaver
        ii. + Mindre objekter -> Færre fejl -> Nemmere at debugge
        iii. + Testbart
        iv. – Mange lag
        v. – ingen genbrug af model
        vi. – view og presenter er bundet til en specifik Model (use case)
4. Redegør for MVVM og dets variationer
    a. Specialisering af MVP, bruges med WPF Frameworket
    b. Næsten samme opbygning som MVVM
    c. Består af følgende:
        i. View
            1. Kan skrives kun i XAML for at eliminere behovet for code behind
            2. Kender sin ViewModel gennem DataContext
        ii. ViewModel
            1. Udstiller attributter i view'et
            2. INotifyPropertyChanged
            3. udstiller properties og commands
        iii. Model
            1. Domæneklasser lavet ud fra kravsspecifikationen
            2. Forretninglogik
            3. INotifyPropertyChanged
    d. Se eksempel i disposition

e. Connecting viewmodel and view
    i. To fremgangsmåder View First eller ViewModel First
    ii. View First creating ViewModel in CodeBehind
        1. Fordele/Ulemper
            a. + Parametre kan sendes med I ViewModel Ctor
            b. + Logik kan laves til vælge View
            c. – Ingen dummydata i design mode
            d. – Svært at teste Code Behind
    iii. View First creating ViewModel Declaratively
        1. Fordele/Ulemper
            a. + Dummy data kan benyttes i design mode
            b. + Eventhandler kan referere VievModel gennem dets navn i XAML
            c. – Ingen parametre i ViewModel Ctor
    iv. View First Using a ViewModel Locator
        1. Fordele/Ulemper
            a. + Parametre I ViewModel Ctor
            b. + Logik der kan bestemme hvilken ViewModel der skal bruges
            c. + Ingen Code Behind
            d. + Dummy data
            e. – Det kan være svært at åbne et nyt view fra viewmodel og sætte den nye viewmodel til at referere data i forretnings logikken
    v. ViewModel First Datatemplate as View
        1. Fordele/ulemper
            a. + ændringer i ViewModel binder automatisk til Viewet
            b. + Parametre i ViewModel Ctor
            c. + Mulighed for logik der vælger ViewModel
            d. + Ingen Code Behind i datatemplates
            e. – ingen dummy data
            f. – ingen design for datatemplates i Visual Studio
    vi. ViewModel First UserControl as View
        1. Fordele/ulemper
            a. Samme som ovenfor
            b. Undtagen at Visual Studio kan designe UserControls

5. Model-View-Controller (MVC) og Model-View-Presenter (MVP)

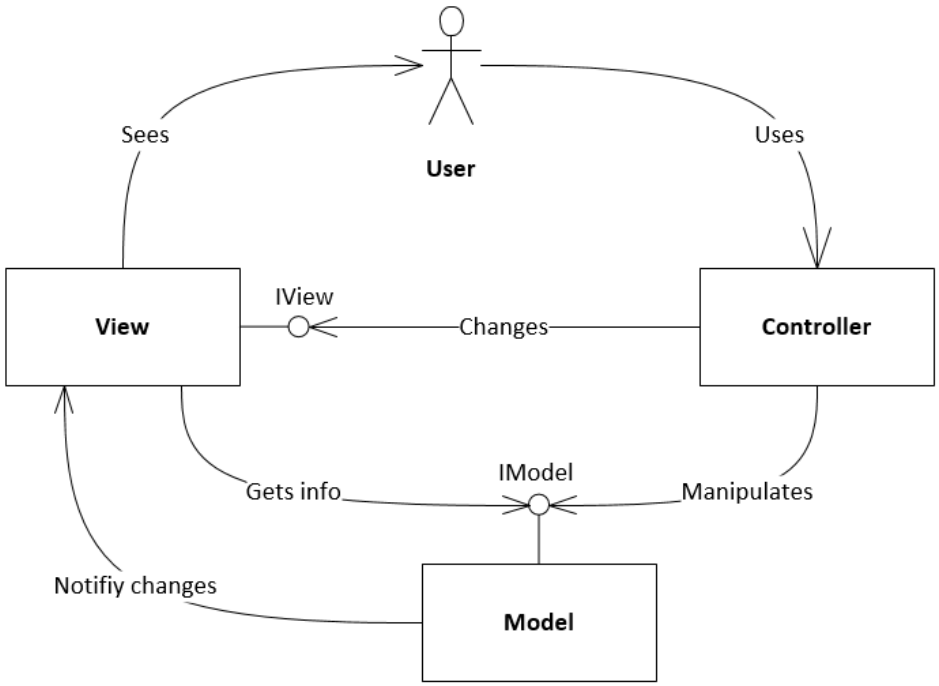


Redegør for, hvad et software design pattern er.

Et software design pattern er en general genbrugelig løsning til problemer der tit opstår i en given kontekst i software design. Det er ikke et færdigt design der kan laves direkte til kilde kode. Det er en beskrivelse eller skabelon for hvordan et problem kan løses i mange forskellige situationer. Det er formaliserede bedste praksisser som en programmør kan bruge til at løse problemer med.

Redegør for Model-View-Control mønstret og dets variationer.

Dette pattern forsøger at **adskille** GUI logik og forretnings logik ved at **opdele koden** i dele der hvert har **forskellige ansvar**.



Model-View-Controller:

- **Model** er forretningslogikken som er defineret af kravene for systemet. Modellen afspejler hvordan databasen ser ud. Man er i stand til at give modellen nogle attributter i form af dataannotations, sådan, at controlleren kan validere på data, og give bestemte fejlbeskeder tilbage.
  Det er også muligt at lægge funktioner ned i modellen, sådan at man har dele af forretningslogikken i modellen, i stedet for at have denne til at ligge i forskellige controllers, eller utility klasser.

  Når der ændres i modellen notificerer modellens view om at der er sket ændringer den er observable.

  Derved er modellen også frakoblet fra GUI'en så den kan bruges andre steder, eftersom, at modellen egentlig bare fremstiller data.

- **Controllers** opgave er at beslutte hvilke ændringer der skal ske i modellen ud fra brugerens handlinger. Hvis der trykkes på en knap kan controlleren fortælle view hvad der skal vises og fortæller modellen hvad den skal vide. Det er her en bruger kunne sende en request fra viewet hvor controlleren tager brugerinputtet. Dette input bliver sendt videre til modellen, og hvis modellen er tilfreds. Bliver det sendt tilbage til controlleren og ud til viewet eller sendt direkte til viewet fra modellen.

- **Views** opgave er at præsentere modellen for brugeren
  View kan være implementeret med strategy pattern hvor controller så vil være den der giver view'et den strategy. View kan også være Composite da det kan bestå af flere elementer som er "Composites" der indeholder "leafs" eller andre "composites"

MVC er godt til at vise den samme model på flere måder da den kan have forskellige views.

Pros:

- Ingen forretningslogik i UI
- Nemt at Unit teste

Cons:

- Skalerer ikke. Separere UI men ikke Model
- Controllere bliver ofte for store
- Bryder Single Responsibility og Interface Segregation principperne.

Redegør for Model-View-Presenter mønstret og dets variationer.

MVC vs MVP

MVC	**MVP**
Bruger Controller til at manipulere modellen med	Bruger Presenter til at manipulere Modellen med
Widgets er både Controller og View	Widgets er ikke separeret mellem controller og view
	Presenter er mere Form Level end Widget level.

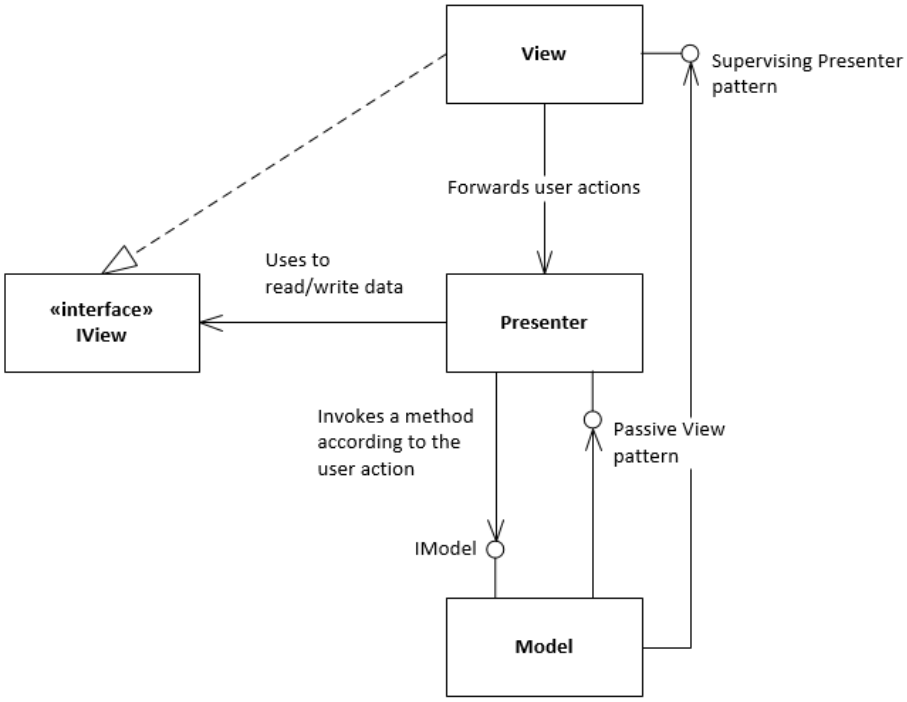
Model View Presenter er en videreudvikling af MVC. MVP tager controlleren og laver den om til en Presenter. i MVP kender viewet til presenteren igennem et interface, og kan derfor let mockes (unit test)

Modellen som i MVC indeholder alle valideringsrutiner som dataannotations og data.

Viewet er som MVC hvor brugeren kan interagere med systemet igennem en brugergrænseflade.

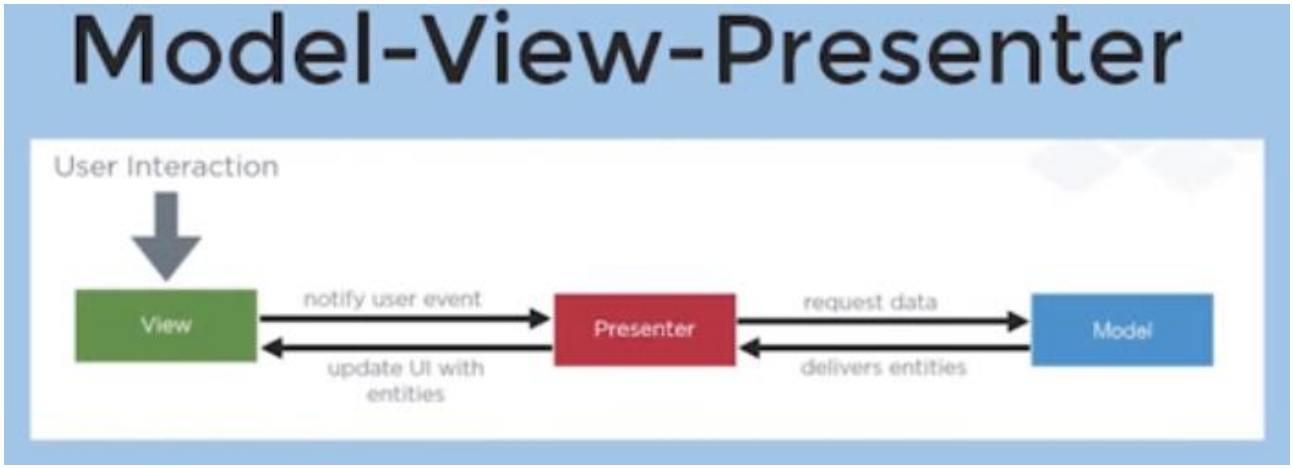
Presenteren er her forskellen fra MVC ligger. Presenteren fungere, som mellemled imellem view og modellen (Passiv View). Alt forretningslogikken som er krævende for at kunne svare tilbage på et bruger input fra viewet er skrevet i presentereren. Viewet har kun foruden brugergrænsefladen, kun event handler og logik nok til at kalde funktionaliteten i presenteren (Dette plejede at være controllerens job fra MVC). Presenteren skal også sørge for, at modtage det rigtige data fra modellen, og formaterer dataen om således, at viewet kan bruge det.

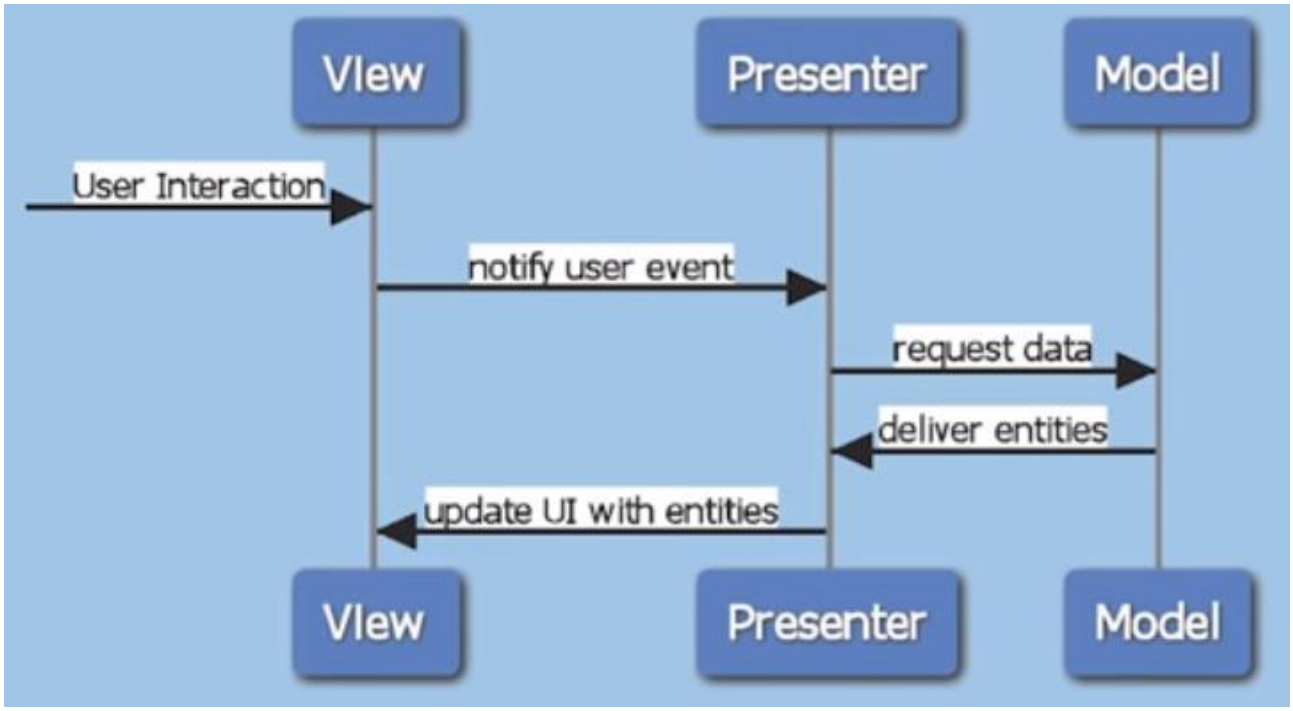
Den findes i flere varianter; Passive view og Supervising presenter.



Tager man udgangspunkt i ”Passiv View”, så kender viewet ikke til modellen. Viewet indeholder samtidig ingen logik og det er derfor presenteren, som er mellemledet imellem modellen og viewet. Dette gør at den udvikler, som arbejder på viewet skal ikke tænke på "**Code Behind**" og giver derfor en større testoverflade. Problemet er, at den databinding mellem view, presenter og model skal man typisk selv skrive.

Bruger man ”Supervising” så ved Viewet hvordan modellen ser ud, men modellen er ikke i stand til at opdatere viewet, som i MVC. Presenteren opdaterer herefter modellen, som kan intagerer med viewet.





- Øger Separation of Concerns
    - Passivt view - render logik
    - Presenter - Håndterer bruger events
    - Model - Forretningslogik

For **passive view** gælder:

- **View**
  View skal være så dumt som muligt, det består af en række kontroller som viser modellen, brugerinteraktion går fra view og ned i presenter.
- **Model**
  Er repræsentationen af data (ligesom MVC).
- **Presenter**
  Presenter manipulerer modellen ud fra de events der kommer fra view. Presenter opdaterer View igennem IView. Interfacet står også for formatering. GUI logikken bor i presenter.

For **Supervising Presenter** gælder:

- **View**
  Her tillades det at lave databinding ned i modellen. Ellers virker view som før.
- **Presenter**
  Her skal presenter stadig kommunikere med modellen om hvad der er sket i view.

Vores UI/View håndtere en simpel mapping til den underliggende model, hvor controlleren/Presenteren håndterer input og kompleks View Logik.

Der skal ikke laves så meget kode i presenteren i denne version men det er svære at teste og der er mindre indkapsling.

Pros:

- Komplekse opgaver bliver delt op i mindre opgaver
- Mindre objekter. Færre fejl. Nemmere at debugge
- Testbart

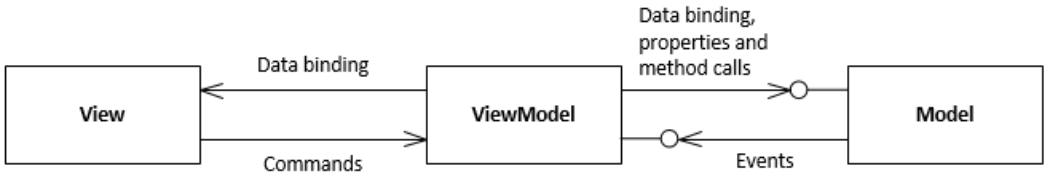
Cons:

- Man skal skrive flere lag
- Modellen kan ikke genbruges. Bundet til et specifikt use case (brug)
- View og Presenter er bundet til specifik use case (brug), siden de begge deler den samme type af objekter, med modellen

Redegør for Model-View-ViewModel mønstret og dets variationer.

MVVM er en specialicering af MVP, det er Microsofts model der bruges sammen med WPF frameworket.

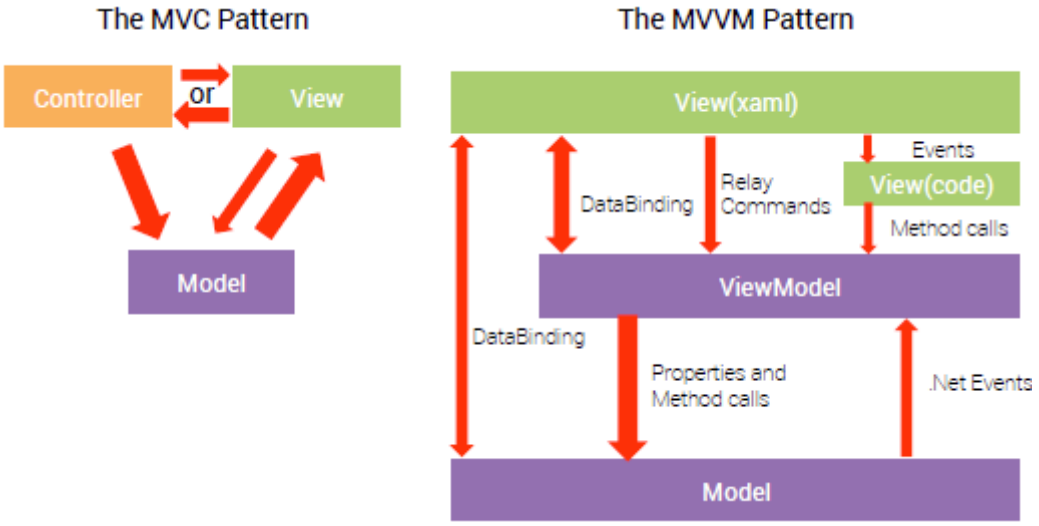
Opbygningen er således:



Opbygningen i MVVM er næsten den samme som i MVP:

- **View**
  Kan skrives i XAML og dermed eliminere code behind, det gør UI-designere i stand til at lave View'et mens programmørene kan håndtere GUI logik i ViewModel.
  Kender sin viewmodel gennem DataContext.
  Er alt hvad der er displayed i GUI. Kan repræsentere hele vinduet eller en user control eller datatemplate.
- **ViewModel**
  Står for at udstille modellens attributter til View'et, dog kan view'et også hente disse direkte i modellen. View kan også databinde til viewmodel så bliver det automatisk opdateret når der sker ændringer. Så længe INotifyPropertiesChanged er implementeret.
  View kender til viewmodel, men ViewModel kender IKKE view.
  Den udstiller properties og commands.
- **Model**
  Er de domæneklasser der er blevet lavet ud fra kravspecifikation eller use-cases
  Det er her forretningslogikken bor. Her kan INotifyPropertiesChanged implementeres så view kan opdatere.

Her ses MVVM i et mere visuelt format:



Her kan det også ses at databinding til properties er dejligt men det gælder ikke for user input!

Derfor har vi Relay Commands de hjælper os med at nå ViewModel fra View når det kommer til user input. Da Relay Commands groft sagt er properties der er knyttet til metoder i vores ViewModel.

Connecting ViewModel and View

Der er to måder at gøre det på der er "View First" og "ViewModel First"

View First Creating a ViewModel In Code Behind

Fordele:

- Der kan sendes parametre med i ViewModels constructor.
- Der kan laves logik der vælger hvilket view der skal oprettes.

```
Public MainWindow()
{
   InitializeComponent();
   DataContext = new BMIViewModel(new Model);
}
```

Ulemper:

- Kode i code behind er svært at teste og designeren kan ikke gives dummy data i design mode.

View First Creating a ViewModel declaratively

Fordele:

- Der kan bruges dummy data i design mode
- Eventhandlers kan referere viewmodel gennem det navn den har i XAML

```
<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>
```

Ulemper:

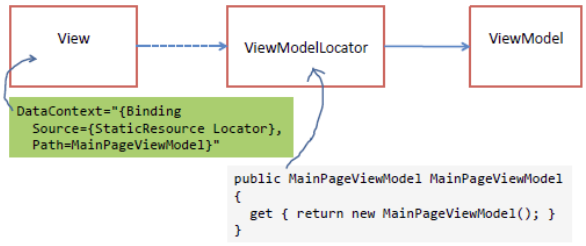
- Ingen parametre i viewmodel constructor

View First: Using a ViewModel Locator

```
public class ViewModelLocator {
   public BMIViewModel BmiViewModel
   {  get{return new BMIViewModel(new BMIModel());}}
}
```

App.xaml:

```
<Application x:Class="UsingAViewModelLocator.App"
    xmlns:local="clr-namespace:UsingAViewModelLocator"
    StartupUri="MainWindow.xaml">
  <Application.Resources>
    <local:ViewModelLocator x:Key="ViewModelLocator" />
  </Application.Resources>
</Application>
```



The View:

```
<Window x:Class="UsingAViewModelLocator.MainWindow"
        Title="BMI Calculator" Height="350" Width="525"
        DataContext="{Binding
                    Source={StaticResource ViewModelLocator},
                    Path=BmiViewModel}" >
```

Fordele:

- Der kan sendes parametre med i constructor til viewmodel
- Der kan laves logik der bestemmer hvilken viewmodel og model der skal bruges
- Ingen code behind
- Dummy data

Ulemper:

- Det kan være svært at åbne et nyt view fra viewmodel og sætte den nye viewmodel til at referere data i forretnings logikken.

Viewmodel 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>
```

Fordele:

- Når der ændres viewmodel vil kontrollen automatisk forbinde til det samhørende view.
- Der kan sendes parametre med i viewmodels constructor
- Der kan laves logik der vælger view model.
- Ingen codebehind i data templates!

Ulemper:

- Ingen dummy data og visual studio har ikke support for design af data templates!

Viewmodel First: UserControl as view

Samme fordele og ulemper som ovenfor. Her er blot mulighed for at lave code behind.

VS har support for at designe UserControls.

```
<Window …>
    <Window.Resources>
        <DataTemplate DataType="{x:Type vm:BMIViewModel}">
            <local:BmiView />
        </DataTemplate>
    </Window.Resources>

    <Grid>
        <ContentControl Content="{Binding Path=BmiViewModel}"
            />
    </Grid>
</Window>
```

```
<UserControl x:Class="UserControlAsView.BmiView"
             …
             >
    <Grid>
        <…>
        <Label Grid.Row="1"
               Grid.Column="1"
               HorizontalAlignment="Left"
               Target="{Binding ElementName=tbxWeight}"
               Content="_Weight:"
               />
```

Her ses den rigtige måde at bruge MVVM på!