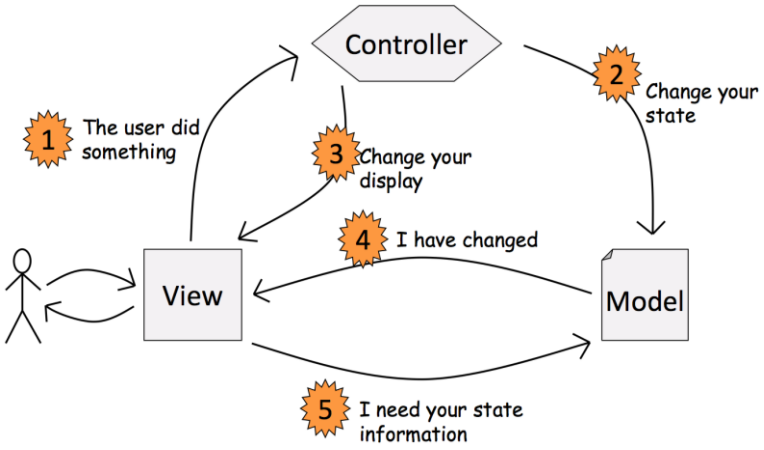
7. Patterns 5: Model-View-Controller og Model-View-Presenter

- Redegør for, hvad et software design pattern er.
- Redegør for Model-View-Control mønstret og dets variationer.
- Redegør for Model-View-Presenter 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)
- Man kan kombinere patterns, og det gør bl.a. MVC

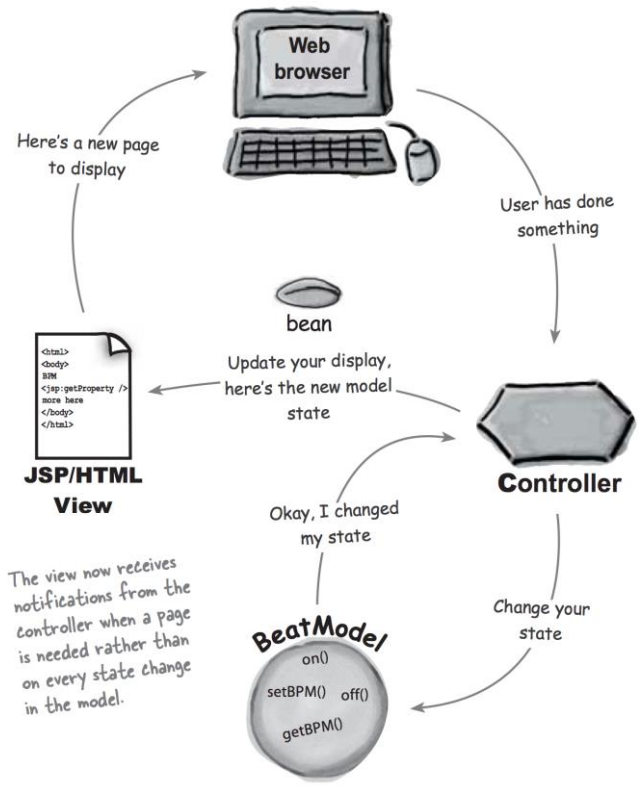


Model-View-Controller:

- MVC går ud på at opdele de domæne-specifikke klasser som beskriver problemdomænet fra præsentationen af disse objekter
- **MODEL** er de domæne-specifikke klasser der er opstået ud fra fx navneordanalyse af kravspecifikationens use-cases eller systembeskrivelse
  - Det er i modellen alt dataen og applikations-logikken befinder sig
  - Når noget ændres i model, notificerer den view'et fx vha. observer pattern eller databinding
  - Det skal være muligt at kunne genanvende logikken i modellaget andre steder.
- **CONTROLLERENS** opgave er at omforme brugerens handlinger, og finde ud af hvad der skal ske/ændres i vores model.
- Fx ved et knaptryk er det controllerens opgave at finde ud af hvad der skal ske enten:
  - Controlleren kan notificere view direkte at det skal ændre sig
  - Eller controlleren kan ændre modellen, som så notificerer view
- Det er **VIEW** som viser modellen til brugeren
  - Viewet får sine data som det skal vise fra model-laget
- MVC er godt til at vise den samme MODEL, men på forskellige måder (forskellige VIEWS)

Variationer:

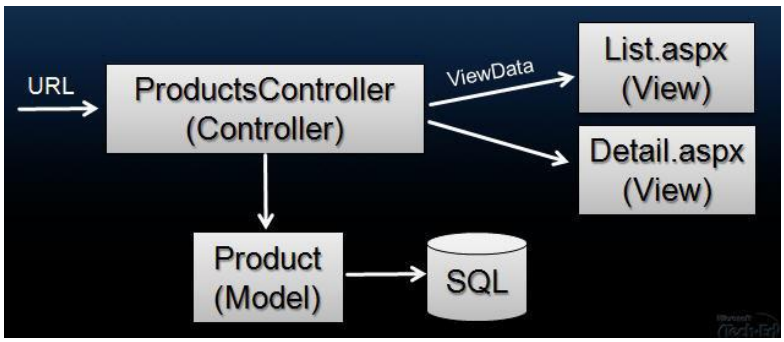
Model 2



- Er en variation af MVC som er specielt tilpasset web
- Controlleren modtager et
    - HTTP requests og ud fra disse fx ændre modellen (kunne fx være en database)
- VIEW'et er fx en javascript genereret HTML side som ændres ud fra hvilke ændringer controlleren har gjort på model
- Ændringerne kan ses ud fra en "bean" som indeholder oplysninger om hvad Controller har ændret i Model
- View/JSP siden returnerer et HTML view tilbage til browseren

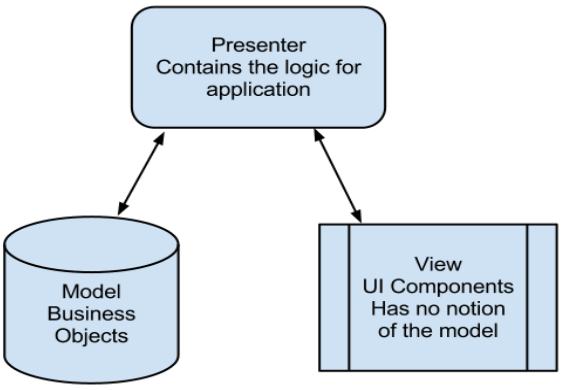
Javabean: enkapsulering af mange objekter som man kan tilgå vha. et kendt interface

ASP.NET MVC



- I denne variation mapper en controller direkte til en URL.
    - Dvs. en URL vil kalde en funktion på en Controller
    - Controlleren udfører evt. noget logik og ændrer Model
- Det er så op til Controlleren at returnere det korrekte View

Model-View-Presenter



- Jeg vil gerne starte med variationen kaldet Presentation Model Pattern (passive view)

MODEL

- Er stadig de domæne-specifikke klasser som indeholder data.

VIEW

- Er så dumt som overhovedet muligt, består af en række kontroller som viser modellen
- Bruger interaktion går fra fra View ned i Presenter (kode med som viser dette, næste side)
  - Dvs. View'et har nogle event-handlers som kalder Presenter
- Hvilket var omvendt fra MVC hvor bruger-input først blev varetaget i Controller, og som dernæst ændrede Viewet

PRESENTER

- Presenteren manipulerer så modellen ud fra events der kaldes af View
  - Den kunne fx snakke til View igennem et IView interface
- Presenteren formaterer også modellen på en sådan måde så Viewet kan vise det
  - Fx kunne den formatere en Datetime i Model til en string i en bestemt dato-format
- Presenter indeholder også vores applikations logik

Humble View

- Der er snak om humble view, når viewlaget er så dumt som muligt, da alt logik er uddelegeret væk fra viewlaget. På den måde er det muligt at unitteste uden automatiseret GUI testframework.

Supervising Presenter pattern variation:

- Her har View'et mulighed for at snakke sammen med Modellen direkte
  - Fx igennem data-binding / observer-pattern
- Presenterens job er så at notificere modellen, ud fra hvad der er sket i View
- Gør det sværere at teste, da View'et databinder ned i en model, mindre enkapsulation da den snakker direkte sammen med Model
- Nemmere da mindre kode skal skrives i Presenter-lag

Kodeeksempel MVP, her kun View:

```
public partial class Viewer : Window, IViewer
{
    private Presenter mainWindowPresenter_;
    private List<Station> StationList_ = new List<Station>();         //Model

    public Viewer()
    {
        InitializeComponent();

        //Presenter gets knowledge of the interface to the Viewer (IViewer)
        mainWindowPresenter_ = new Presenter(this);

        //This is our model being initialized:
        mainWindowPresenter_.AddStations(StationList_); //has default stations
        StationListBox.ItemsSource = StationList_;
    }

    #region Events

    private void TextBox_TextChanged(object sender, TextChangedEventArgs e)
    {
        mainWindowPresenter_.TextBoxChanged(sender);
    }

    private void TextBox_Textfocuslost(object sender, RoutedEventArgs e)
    {
        mainWindowPresenter_.TextBoxChanged(sender);
    }

    private void Actual_OnLostFocus(object sender, RoutedEventArgs e)
    {
        mainWindowPresenter_.ShowVariance();
    }

    #endregion

    //////////IMPLEMENTATION OF IVIEWER/////////
    public ListBox GetStationListBox()
    {
        return StationListBox;
    }
```

- View'et laver en ny presenter
- Presenter modtager et Interface af View'et

- Events på View'et delegeres videre til presenter, som så opdaterer View gennem det interface den har fået

MVC består i princippet af flere patterns

Observer pattern:

- Model er Subject, mens både View og Controller kan være Observer på modellen
    - På denne måde kan logik inde i Modellen som ændrer dets stadie blive videregivet ud til Controller og View
    - Ændringer Controller laver på Model vil også blive notificeret videre til VIew

Strategy pattern:

- Bruges mellem view og controller.
- View kan bruge et IController interface
    - Hver gang der sker noget på view'et delegeres opgaven ud til den controller som har med den widgets opførsel at gøre
- Afkobling mellem view og Controller

Composite:

- Viewet består af en række kontroller og widgets, som controlleren kan fortælle at opdatere ved blot at snakke med top view componenten.