**Talepapir MVVM+Observer**

Model-view-viewmodel er en måde man kan opbygge et større program, således man får opbygget en god struktur.

MVVM hjælper til at strukturere en eller flere GUI’er, og får disse til at kommunikere og samarbejde med relevant data.   
I en MVVM struktur, opdeler man ansvarsområder for alle klasserne, så man ikke ender med at have for meget og uoverskuelig funktionalitet i én samlet klasse, som også kan resultere i problematisk testning senere hen.

MVVM er også god til at overholde open/close princippet, i forhold at det er nemt at udvide programmet, uden at skulle ændre i eksisterende kode.

De forskellige dele involveret er et view, som indeholder en gui, samt en tilhørende controller.

Der er modellen, som står for at håndtere data og data manipulation

Så er der view-model, som står for user-event handling. Det vil sige, at den sørger for at håndtere data fra vores model, og formatere det til data, som kan bruges i vores view. Den er således en bindeled i mellem vores model og view.

På den måde kender view til viewmodel, og viewmodel kender til model. Kommunikationen i mellem viewmodel og model foregår i gennem metodekald, hvor viewmodel får en instans af model, og bruger metoderne herfra.

Kommunikationen mellem view og viewmodel har vi bindings. Det betyder, at de fx:id’er vi giver i vores GUI, de vil blive defineret i controlleren, hvori de i controllerens constructor bliver bundet til JavaFX understøttede attributter. Disse attributter tager derfor data fra modellen, og sørger for at det kan blive understøttet i vores GUI, f.eks. vil at bruge StringProperty.

Når der trykkes på en f.eks. en knap i gui’en, håndterer view model hvad der skal ske i eventet, og opdaterer data igennem den binding der er i mellem controller og viewmodel.

**Oberserver**

Vi kan altså opdatere vores viewmodel vha. metodekald fra model, som automatisk opdaterer værdierne i view, men hvis vi nu skal opdateres på ændringer i modellen, uden at der bliver lavet metodekald, så skal vi bruge observer pattern.

Oberserver pattern bruges, når man har en mængde forskellige klasser, som er interesseret i en anden klasses tilstand.

Løsningen på dette er så, at man laver en en-til-mange afhængighed imellem det objekt som ændres, og de klasser som er interesseret i ændringen, hvorved alle afhængigheder automatisk bliver notificeret og opdateret.

Det vil sige, at man har et objekt, som der sker ændringer hos, og dette objekt står for at opdatere de andre, når den ændres. Disse kaldes observers, og da der kan være mange observers, holder subjektet en liste, hvor de forskellige observerklasser kan skrive sig på, således objektet ved hvilke klasse, der skal notificeres.

En fordel ved dette, er at man opnår en løs kobling, idet objektet ikke behøver at kende til observers, men kun en liste.

Subjektet holder derfor også metoder, som kan tilføje og fjerne observers.

Hertil har java nogle indbyggede klasser, som fungerer præcis til dette formål.

Vi har

* PropertyChangeListener, som observer klasser kan implementere, så de får en metode kaldet PropertyChange(evt:PropertyChangeEvent), hvori opdateringer fra subjektet håndteres
* PropertyChangeSupport er en liste med listeners (samme metoder som subjekt)
* PropertyChangeEvent er det event der sker, når objektet opdateres, og denne værdi sendes ud til alle observers i listen, både med den gamle og nye værdi

Der er 3 måder at implementere det på. Fælles for alle er, at subjektet skal en instans af PropertyChangeSupport, så den kan holde listen.

1. Listener klassen implementerer PropertyChangeListener
2. Lambda notation: listener klassen instantierer subject i cuntroctoren. Herefter adder den sig selv som listener på subject, hvor man laver lambda expression (evt->metodeNavn(evt)), som kalder på metoden metodeNavn
   1. Man kan undgå at metoden bliver kaldt udefra
3. som 2, men når man tilføjer sig selv som listener, tager den argumentet (”stateName”, this::metodeNavn), som kalder metoden metodeNavn.
   1. Så kan man have forskellige metoder tilpasset alt efter stateName

* MVVM = større program = god struktur
* Hjælper til stukturere GUI’er
* Får dem til at… Kommunikation og samarbejde med relevant data
* Opdeler ansvarsområder for alle klasser
* Undgå for meget og uoverskuelig funktionalitet
  + Resultere i problematisk testning
* Open/close = udvide unden at ændre
* Dele involveret
  + View = gui + controller
  + Model = data og data manipulation
  + View-model = user-event-handling
  + Håndterer data fra model, og formaterer til relevant data til view
  + BINDELED
* På den måde
  + View => viewmodel => model
  + Viewmodel => model = direkte metodekald = instans
  + View=>viewmodel = bindings
    - FX:Id’er i GUI = atributter i controller
    - I constructor bindes de til FX understøttede attributter fra Viewmodel
      * F.eks. StringProperty
    - I attributten gemmes data fra model, som castes til FX understøttet variabler, således de kan bruges i view
* Når der trykkes på en f.eks. en knap i gui’en
  + håndterer view model hvad der skal ske i eventet
  + opdaterer data igennem den binding der er i mellem controller og viewmodel.

**OBSERVER**

* ALTSÅ: Viewmodel opdateres ved metodekald, som automatisk opdaterer view
* Ændringer i model uden metodekald = observer pattern = automatisk opdateret
* ObsPat = forskkelige mængde klasser interesseret i en anden klasses tilstand
* Løsning = EN-TIL-MANGE afhængighed imellem subjekt og interesserede klasser
  + Afhængigheder automatisk notificeret og opdateret
* Subjekt – ændringer = subjekt skal opdatere andre (OBSERVERS)
  + Subjekt holder liste
  + Notificer alle i listen
    - FORDEL: LØS KOBLING, SUBJEKT BEHØVER IKKE AT KENDE TIL OBSERVERS
  + Holder tilføj/fjern metoder
* Java indbyggede klasser
  + PropertyChangeSubject
  + PropertyChangeListener
  + PropertyChangeSupport
  + PropertyChangeEvent
* 3 måder at implementere:
  + Listener implementerer PropertyChangeListener
  + Lambda notation (evt->metodeNavn(evt))
    - Undergå metodekald udefra
  + Som 2, (”stateName”, this::metodeNavn)
    - Forskellige metoder tilpasset til stateName

UML

* SE PP