**Question 2A:** Model-View-ViewModel (MVVM)

Describe the pattern using UML diagrams.

* What is the purpose?
  + Når et program bliver større, er det godt med en god struktur.
  + Det hjælper MVVM med – den holder programmet velstruktureret
  + MVVM strukturerer GUI og Data, og får disse til at kommunikere og samarbejde
  + Man opdeler ansvarsområder for alle klasserne, så man ikke ender med at have én eller flere klasser, som laver alt for meget/laver det hele
  + Hvis én klasse har for meget ansvar, er det svært at teste ens feautures, da man heller ikke ved hvor de forskellige ting sker
  + Fordel: Man kan tilføje ny kode og ny funktionalitet (næsten)uden at ændre i eksisterende kode
  + Hvis man rør ved eksisterende kode som virker, risikerer man at ødelægge det, så det ikke længere virker
  + Vi vil gerne have vores klasser små, med et klart formål – ungå if/else 10000 gange
* What are the different parts involved and their purpose?
  + View: vores GUI view og Controller
  + View Model: User event handling (når user trykker på en knap, håndterer view Model hvad der skal ske ved det event) – view model håndtere at tage data fra modellen, og formatere det til data som View kan bruge – input validation (tjek f.eks. nummer kunn består af tal) – data validation (tjek at e-mail ikke allerede eksisterer i systemet)
  + Model: Håndterer data
  + Networking ………………. Se client/Server
* How do the different parts interact?
  + View kender til viewModel, det vil sige den har en instans af viewModel
  + ViewModel kender til Model, og har en instans af den
  + Der er bindings i mellem View og ViewModel (En slags observer Pattern, så vi binder private feltvariabler fra View med private feltvariabler fra ViewModel)
  + Der er direkte metodekald i mellem ViewModel og Model, altså vil ViewModel bruge instansen model og bruge metoder fra denne
  + For at få data fra Model til ViewModel, så skal vi bruge Observer pattern
  + Vi bruger interfaces for at få fleksibilitet (ikke nødvendig at snakke om til dette emne)

Present your own example from an assignment, sep2, or exercise

* Use relevant UML diagrams to present how your solution follows the pattern
* Show in Java code:
* How the MVVM pattern is implemented.
* Pieces that are relevant for this pattern, including the databinding between View and View-Model.

**Question 2B:** Observer design pattern

Describe the pattern using UML diagrams.

Problem som skal løses med observer design pattern: man har en mængde af (forskellige) klasser, som er interesseret i en andens klasses tilstand – det kan løses med observer!

* What is the purpose?
  + Meningen er at man kan lave en en-til-mange afhængighed, så når ét objekt ændres, så bliver all afhængigheder automatisk notificeret og opdateret.
  + Det vil sige, man har et subjekt, som er den der sker ændringer hos, og så har den tilknyttet listeners/observers, som observerer når subjektets ændringer.
  + En af fordelene ved observer design pattern er, at man opnår en løs kobling, idet at subjektet ikke behøver at kende til sine observers
  + En ulempe kan være, at der sendes unødvendige opdateringer til observeren
* What are the different parts involved? How do they interact?
  + Som sagt, så er der ét subjekt, som kan have mange observers
  + Subjektet holder en liste af dens observers, og notificerer dem automatisk, når der sker ændringer – typisk ved at bruge en af observers metoder
  + Tilføj og fjern observers fra listen burde gøres udefra (altså ikke i subjekt klassen – den skal kun holde listen, så vi opnår den løse kobling!)

JAVA support klasser:

* + - PropertyChangeListener (en listener kan implementere dette interface, og realSubjekt(den som har implementeret subjekt interfacet) har 0…\* instanser af listeners, altså en liste af listeners)
      * propertyChange(evt:PropertyChangeEvent)
    - PropertyChangeSupport (erstatning for vores liste af listeners i PropertyChangeListener, denne klasse er vores liste af listeners)
      * Har samme metoder som Subjekt
    - PropertyChangeEvent(når firePropertyChange(name:String, oldValue:String, newValue:String) bliver kaldt, bliver hele listen i PropertyChangeSupport kørt igennem)
    - PropertyChangeSubject (laver vi selv (Subject klassen)) (et subjekt kan implementere dette interface)
      * addPropertyChangeListener(name:String,listener:PropertyChangeListener)
      * removePropertyChangeListener(name:String,listener:PropertyChangeListener)
    - Når listener objektet implementerer PropertyChangeListener, implementerer den metoden propertyChange(evt:PropertyChangeEvent), som bestemmer hvad der skal ske, når der sker et event

3 måder at implementerer observer design pattern med java:

* Fælles for alle: realSubjekt klassen skal have en instans private PropertyChangeSupport support som er listen af listeners, og klassen selv implementerer Subjekt klassen

1. Listener klassen implementerer PropertyChangeListener interfacet, og får metoden propertyChange(evt:PropertyChangeEvent), som her opdaterer hvad der sker ved event
2. Lambda notation: listener klassen implementere IKKE PropertyChangeListener, i stedet tager man et PropertyChaneSubject subject i listener klassens constructor, og inde i constructoren skriver man ”subject.addPropertyChangeListener(evt->metodeNavn(evt))” – her definerer vi en metode, som tager et parameter ”evt” (anonym metode), og laver herefter metoden med ”PropertyChangeEvent evt” som argument
   1. Hvis man kun vil have at metoden metodeNavn(evt) bliver brugt ved observer pattern, kan man gøre sådan, og undgå at den bliver kaldt udefra
3. næsten samme som 2. : PropertyChangeSubject i constructoren, og så ”subject.addPropertyChangeListener(”stateName”, this::metodeNavn);” – Vi refererer direkte til metoden metodeNavn.
   1. Fordel ved denne, så sender subjektet stateName afsted, og listener skal så bare have en metode tilpasset til denne, f.eks. (”greenChange”, this::onGreen), så skal subjektet bare fortælle ”greenChange”, og så skal listener nok tilpasse sig

Present your own example from an assignment, SEP2, or exercise

* Use UML and describe how your solution follows the pattern
* Show Java code – how the pattern is implemented.
* Show Java code for how the pattern can be used.

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