Touchpoint

Gruppe C

2. semesters eksamens projekt

Rapport.

Allan Søndergaard, Peter Clausen, Nikolai Christiansen, Kasper Uhre
18-12-2017

Table of Contents

Project Establishment

Introduktion

Vi vil i denne rapport dokumentere og argumentere for de valg der er truffet i forbindelse med vores 2. semesters eksamensprojekt.

Projekt beskrivelse

I dette projekt arbejder vi sammen med virksomheden ”Touchpoint”, vores lærer Mohammed spiller rollen som vores kontaktperson hos virksomheden og det er i samarbejde med ham og ud fra interviews af ham at krav og lignende er blevet udarbejdet.

Virksomheden ledes af tidligere fagfolk og de udbyder kurser i

Målet med dette projekt er at udvikle en software løsning der automatiserer flere aspekter af Touchpoints nuværende system. Dette vil gøre det muligt for virksomhedens kunder at tilmelde sig kurser uden at der skal sidde en person og godkende det i den anden ende. Vores løsning vil endvidere automatisere oprettelsen og opkrævningen af betaling for disse kurser.

Positioning

Problem Statement

The problem of	Touchpoint
affects	Kunder, undervisere
the impact of which is	Da tilmelding til kurser fra en brugers side skal godkendes manuelt af en administrator, skal brugeren sende en mail til administratoren efter de har tilmeldt sig kurset på hjemmesiden. Derudover foregår betalingen eksternt og bliver ikke klaret af systemet. Dette gør at touchpoint skal bruge tid på at godkende disse tilmeldinger og manuelt oprette og udsende opkrævning for betaling af kurserne.
a successful solution would be	Et system, der udover den funktionalitet der i forvejen er, automatisk kan registrere tilmeldinger af brugere til kurser og i forbindelse med dette, oprette betaling af depositum og betaling af selve kurset og sende dette til den given kunde. Derudover skal systemet kunne gemme disse oplysninger i en database.

Product Position Statement

For	Touchpoint
Who	Vil optimere processen omkring tilmelding af brugere og opkrævning af betaling for kurser.
The (product name)	Touchoint.exe
That	Udover den oprindelige funktionalitet, vil automatisere tilmeldingen til kurser og opkrævningen af betalinger herfor.
Unlike	Den nuværende metode hvor en ansat skal bruge tid på at gøre disse ting manuelt, og dermed højner chancen for fejl.
Our product	Gemmer detaljer om brugere, undervisere, undervisningssteder, lokaler og kurser. Og kan oprette og redigere disse, samt benyttes til tilmelding og opkrævning.

Stakeholder Summary

Name	**Description**	**Responsibilities**
Mohammed	Spiller rollen som chef af virksomheden	Fastsætter krav, forklarer og klarlægger den situation vi skal gå ud fra i forhold til firmaets og dets forhold. Har sidste ord på projektet.

User Environment

Systemet bliver brugt af folk hvis faglighed omhandler menneskekroppen, som skal bruge programmet til at tilmelde sig kurser indenfor deres fag.

Programmet skal ikke interagere med andre programmer på computeren, men det skal agere med en database.

Programmet skal benyttes på Windows baserede computere.

Gruppe dannelse.

Allan Søndergaard – Design

Nikolai Christiansen - Kode

Peter Clausen – Design

Kasper Uhre - Kode

Team contract.

Hvis man ikke møder op til gruppearbejde, skal man senest om morgenen den pågældende dag informere et af de andre gruppemedlemmer om dette.

Hvis et medlem overskrider deadline (uden at informere i god tid om at dette vil ske), vil det stykke arbejde blive lavet af resten af gruppen og det medlem der ikke overholdt deadline vil ikke få kredit for dette stykke arbejde.

Project management.

- Vi bruger Github til at holde styr på alle dokumenter og kode, på den måde har vi altid adgang til alt der har med projektet at gøre.

- Vi bruger Discord til at kommunikere i gruppen. Det gør vi altid kan komme i kontakt, både på computere såvel som på telefonen.

- Vi bruger trello til at ud delegere opgaver til det enkelte gruppemedlem, samt at holde styr på hvad der mangler at blive lavet og hvad der er lavet. Ved hjælp at trello kan vi også se hvem der har lavet hvad og hvem der har hjulpet.

Technical tools.

- Microsoft visual studio
- Microsoft visio
- Microsoft word

Vi benytter microsoft visual studio til at skrive vores software løsning i, vi benytter visio til at lave visuelle repræsentationer af vores design artifacts og vi bruger word til at skrive vores rapport.

Risikoanalyse på teamet.

ID	Navn	Beskrivelse	Påvirkning	Sandsynlighed	Ansvarlig	Next step
R01	Sygdom	Medlemmer møder ikke op grundet sygdom	Tid	Mellem	Alle	Få omfordelt arbejdsopgaver
R02	Tidsfrist	Vi når ikke de satte tidsfrister	Tid	Mellem	Alle	unscope
R03	Design	Design er dårligt/svært at implementere i praksis	Tid, funktionalitet	Mellem	Alle	Sørge for at design og kode hænger sammen
R04	Bugs	Fejl i koden	Tid, funktionalitet, stabilitet	Høj	Alle	Sørge for at koden er fejlfri inden man introducerer den til systemet
R05	Frafald	Person i gruppen falder fra	Tid	Lav	Alle	Omfordele arbejdet, evt. finde nyt medlem

Business analysis

Porter five forces.

Threat of new entry:

Der er ingen barriere der forhindrer nye virksomheder at tilbyde samme services som touchpoint, det er samtidigt relativt billigt at oprette og vedligeligholde en hjemmeside. Det kræver ingen viden eller erfaring at tilgå sig markedet, da alt vi leverer er et system hvor kunder kan tilgå undervisere, vi sørger for at der er undervisere, og at der er lokaler til rådighed. På baggrund af det, mener vi at det er let at komme ind på markedet.

Buyer power:

Målgruppen for touchpoint er relativ lille, da de primært tilbyder kurser og foredrag til allerede uddannede specialister inden for meget specialiseret områder. Da der også er andre på markedet der tilbyder det samme som touchpoint og kan konkurrere med priserne, samt kundernes mangel på grund til at være loyale, har kunderne meget magt.

Supplier power:

Vores leverandører er vores undervisere. Da der er et begrænset antal undervisere og andre virksomheder som tilbyder det samme som touchpoint, gør det, det let for underviserne at skifte mellem virksomheder, og blive hos dem de føler sig bedst behandlet af.

Derfor mener vi at leverandørerne har meget magt.

Threat of substitution:

Der findes mange virksomheder som tilbyder kurser og foredrag inden for større og bredere emner, med en større eksisterende kunde base. Skulle disse virksomheder brede sig ud på touchpoints marked, er der en reel trussel for udskiftning.

Competitive rivalry:

Der er et moderate antal virksomheder som tilbyder det samme, som touchpoint og da der er lav kunde loyalitet, og det ikke koster noget at skifte kursus udbyder, mener vi at der er en høj konkurrence, mellem virksomheder på markedet.

SWOT.

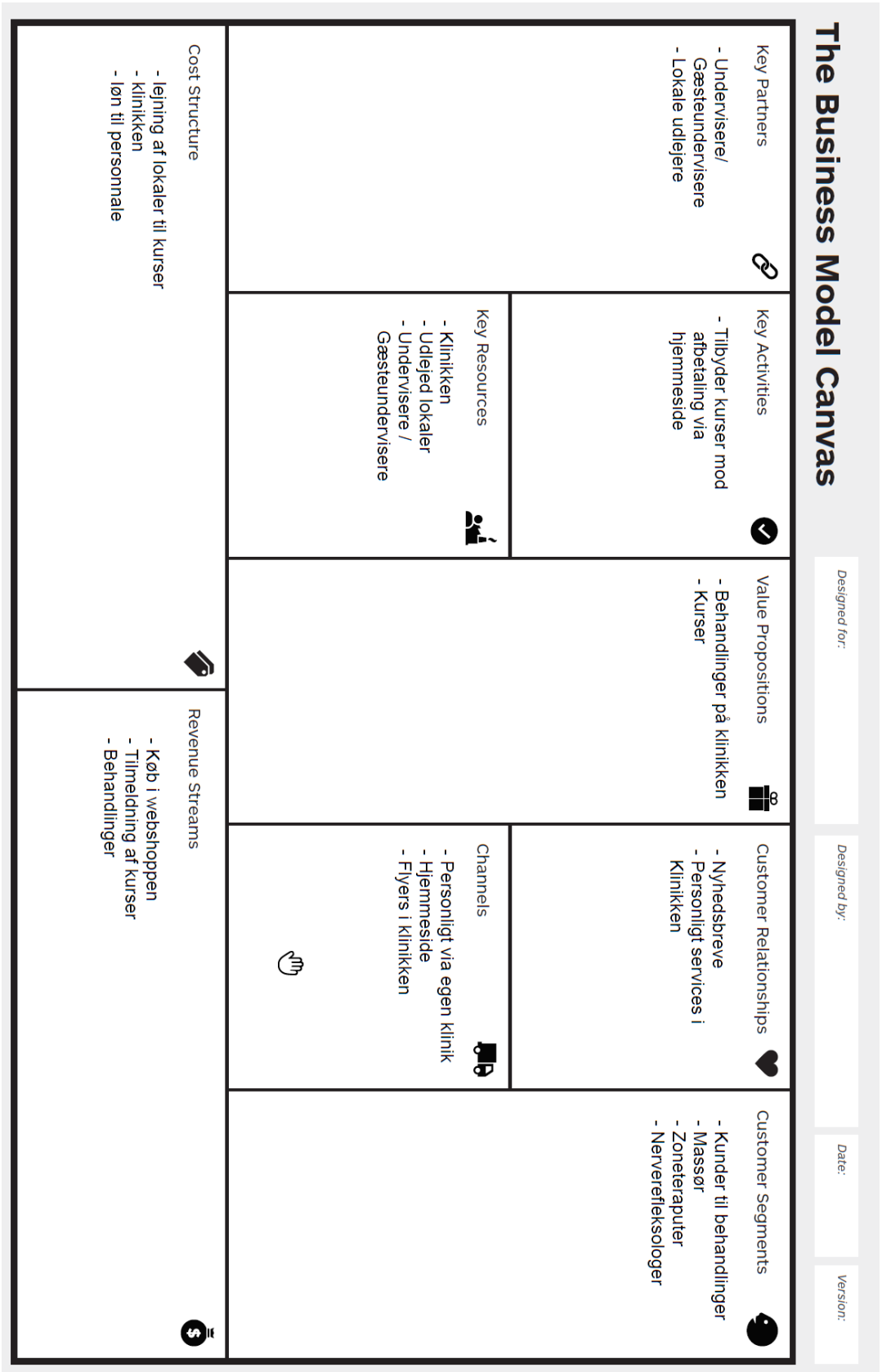
Man benytter SWOT analysen til at se på 4 faktorer, 2 interne og 2 eksterne.

De interne faktorer er defineret som dem vi som firma selv har kontrol over og de eksterne faktorer er dem vi ikke har direkte indflydelse på.

Ideen med analysen er at man efterfølgende skal forsøge at mindske sine svagheder og trusler ved at udnytte sine styrker og satse på de rigtige muligheder.

Figur 1: SWOT

Business Model Canvas



Figur 2: Business Model Canvas

IT-strategi.

Et nyt system der optimerer hele processen omkring det system kunden allerede anvender. Dette indebærer automatisering af de processer som systemet allerede benyttes til, og gennem denne automatisering vil arbejdsgangen og informations flowet i kundens virksomhed blive optimeret.

Programmet skal kunne:

- Registrere brugere til kurser.
- Oprette nye kunder og undervisere.
- Oprette undervisningssteder med tilhørende lokaler.
- Oprette faktura, og pr e-mail sende dem til kunder.

Vi skriver programmet I C# og benytter MVVM arkitektur. (se under punktet ”arkitektur” for at få en mere uddybende gennemgang ar vores arkitektur)

Kategorisering af krav

Vi benytter os af FURPS for at kategorisere hvor vigtige de forskellige krav er.

Functionality

- sikkerhed

Høj prioritet, da vi har med persondata, samt betalinger at gøre. (1)

Udover sikkerhed har vi følgende funktionelle krav til programmet:

- Opret lokale
- Opret kursist
- Opret underviser
- Opret undervisningssted
- Opret kursus
- Opret betaling
- Opret serie
- Opret ekstra materiale
- Tilmeld kursus

Vores resterende funktionelle krav er at finde under punktet Design, hvor vi har defineret alle vores krav og går en enkelt use case fuldt i gennem med tilhørende artifacts.

Usability

- grænseflade, hjælp, dokumentation

Høj prioritet, da de folk der skal bruge systemet i deres hverdag ikke nødvendigvis har noget IT-erfaring. (3)

Reliability

- fejlfrekvens, recoverability

Høj prioritet, da vi skal benytte systemet til at registrere kunder og holde styr på betalinger er det vigtigt at systemet ikke crasher (2)

Performance
- response time, ressourceforbrug

Lavere prioritet, da systemet er lokalt og ikke er så krævende (4)

Supportability
- testability, maintainability, compability, configurability

Lav prioritet, vores system bliver udviklet til kundens OS og ikke andre. Når vores løsning er færdig og leveret til kunden er det ikke meningen at der skal arbejdes mere med det. (5)

Feasibility study

Legal

Vores program, overtræder ingen love og er derfor lovligt at lave, vi skal dog være opmærksomme på persondataloven – da vi har med privatpersoners data at gøre.

Schedule

Projektet skal afleveres den 18. december 2017, hvilket er den deadline vi arbejder med.

Resource

systemet bliver lavet på Windows computere til Windows computere, der er ikke andet udstyr påkrævet til udviklingen af systemet. Det er derfor ikke et ressource tungt projekt.

Technical

Systemet har ikke noget teknisk avanceret aspekt. Løber vi ind i problemer undervejs støtter vi os op af hinanden og vores undervisere.

Economical

Systemet ville kunne sælges til en moderat pris, da der ikke er meget originalitet i det og derfor ikke en ny dyb tallerken der skal opfindes, samt en kort udviklings proces.

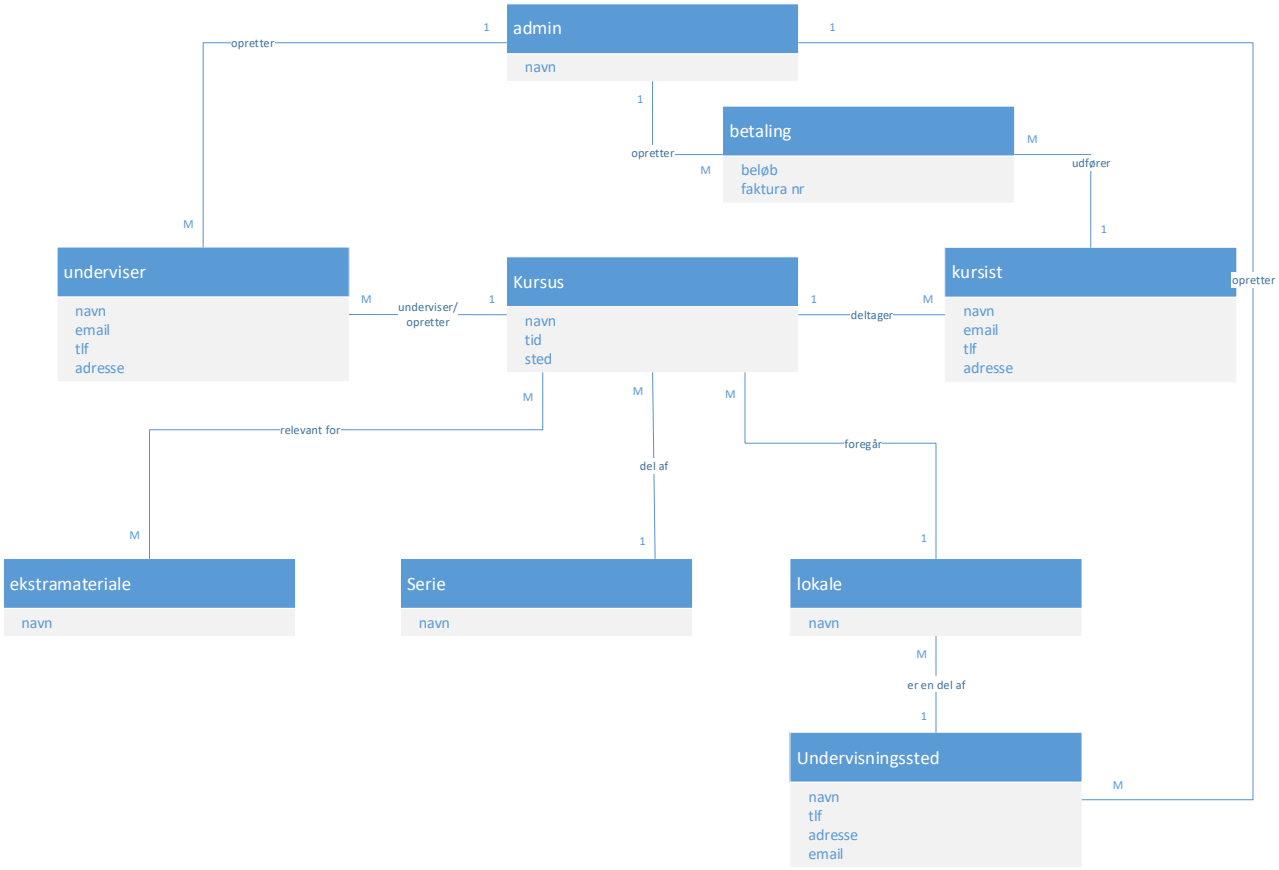
Cultural

Vores system har ingen kulturelle indflydelser på hverken virksomheden eller omverdenen.

På baggrund af disse oplysninger mener vi at det er feasible at fortsætte.

Design

Domæne Model

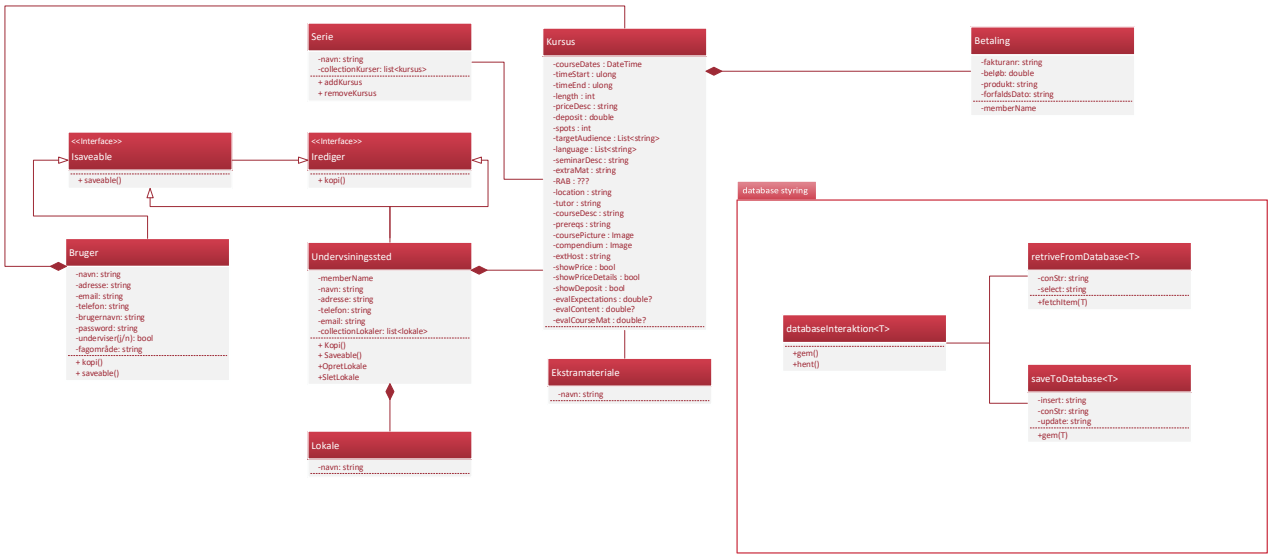


Figur 3: Domæne Model

Domæne-modellen bruges til at skabe et overblik over de begreber og koncepter man har med at gøre, i domæne-modellen taler man ikke software klasser (det kommer i næste afsnit om Design Class Modellen).

Man laver en domæne model for at skabe sig et overblik over de potentielle softwareklasser eller objekter man skal kunne skabe for at projektet bliver succesfuldt.

Design Class Model



Figur 4: Design Class Model

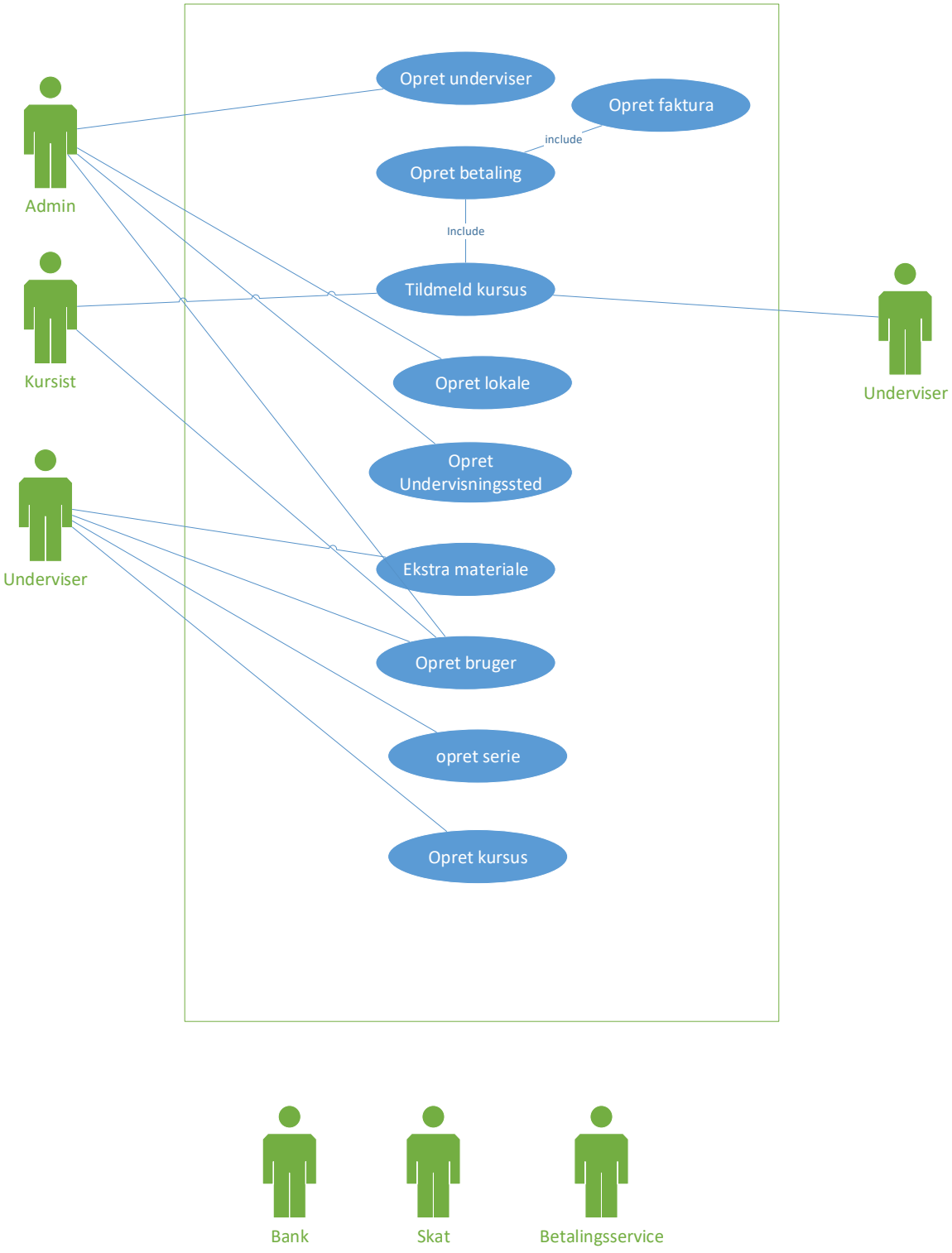
Som man kan se så er der stor lighed mellem domæne-modellen og design class modellen, den store forskel består i at man i design class modellen har de attributter og metoder som klassen indeholder inklusiv deres type.

Ligesom i domæne-modellen har man også associationer mellem klasserne, disse fortæller om hvilke klasser der er afhængige af hinanden.

Et eksempel på dette er at lokale klassen er afhængig af undervisningssted klassen, den er afhængig fordi, som man kan se, så ligger metoden for at oprette et lokale i undervisningssted klassen.

Som vi taler om i GRASP afsnittet så har vi effektivt gjort undervisningssted til en creator af lokaler, dette har vi gjort da de oprettede lokaler skal behandles af undervisningssted klassen og da den allerede har den information der er krævet for at gøre dette, en såkaldt information expert, så har den klasse det ansvar.

Use Case Diagram



Figur 5: Use Case Diagram

Man bruger et use case diagram (*Figur 5*) til at skabe et overblik over hvilke aktører der har med hvilke use cases at gøre.

De aktører der står på den venstre side af firkanten, kaldet vores scope, er primære aktører, en primære aktør er den som udfører den pågældende use case.

På højre side af vores scope finder vi de sekundære aktører, det er en aktør der er nødvendig for at den primære aktør kan udføre en vilkårlig use case.

I bunden har vi de tertiære aktører, det er aktører udenfor systemet som har et ansvar for at en use case kan udføres. Dette kan være en service som NETS, der gør det muligt at udføre betalinger med betalingskort. Hvis man forestiller sig en use case i vores projekt der omhandler betaling af depositum for et kursus eller betaling af selve kurset, så vil vores use case ikke kunne udføres over nettet med betalingskort, hvis ikke vi gør brug af deres service.

Når man har fundet sine use cases kan man gå mere i dybden med den enkelte og detaljeret beskrive hvad der skridt for skridt sker i denne use case.

Dette går vi i dybden med i de følgende afsnit, hvor vi tager udgangspunkt i den use case der hedder ”opret bruger”, vi har gjort det samme for de andre use cases og disse er at finde som bilag, men det virker tosset at skrive og fylde rapporten op med næsten identiske artifacts for alle use cases.

Use Case Text

Da vores “opret burger” use case udelukkende opretter et bruger-objekt og sender brugeren videre til et view hvor dette objekt kan redigeres er denne use case delt op i flere use case texts. Udover at der er forskel på at oprette en bruger og på at gemme information i dette objekt, er der også forskel på hvad man kan gemme af information i objektet.

Alle kan oprette en bruger men anden funktionalitet er altså gemt bag et ”credential check”

Som kursist, altså en bruger uden ekstra credentials er der kun en funktionalitet man kan benytte, og det er ”tilmeld kursus”,

Som underviser kan man oprette serier af kurser, oprette og redigere kurser og tilføje ekstra materiale til kurserne.

Som administrator har man mulighed for at oprette og redigere lokaler og undervisningssteder. Endvidere har administratoren mulighed for at give brugere credentials, altså er det administratoren der efter at en bruger er blevet oprettet, går ind og giver denne bruger underviser eller administrator credentials hvis dette er nødvendigt.

Hvis vi ser på vores use case text for opret bruger (*Figur 6*), kan man se at den indeholder alle de oplysninger man kan være interesseret i omkring hvem der gør hvad i en given use case.

Man giver som udgangspunkt hver use case et unikt ID, så man nemt kan referere til de forskellige use cases uden at skabe forvirring med navne der kan se ens ud.

Vi har i vores use cases valgt at navngive en use case med et ”B”, et tal, en bindestreg og endnu et tal. Det betyder at den use case der hedder ”**B10-1**” er en use case vi har fra **B**illede nummer **10** af de screenshots vi fik af vores kunde. Det sidste **1** angiver at det er use case nummer 1 vi har udledt fra det givne billede.

Det er vigtigt at forstå når man har med use cases at gøre at den ikke skal indeholde tekniske termer, som kode eller fagudtryk. Det skal være en simpel gennemgang af hvilke aktører der indgår i use casen, hvad der igangsætter denne use case og hvilke aktører der gør hvad.

Use Case:	**Opret Bruger**
ID:	**B10-1**
Prioritet:	
Summary:	Opretter en bruger
Primær aktør:	Brugeren
Sekundær aktør:	
Generalisation:	
Include:	
Extend:	B10-2 Rediger bruger
Pre-betingelser:	En ikke oprettet bruger har trykket opret bruger.
Trigger (igangsættende hændelse)	Primær aktør trykker på opret knappen.
Normal flow:	1 PA trykker opret knappen 2. Pa bliver taget til Rediger vindue *Extend B10-2*
Alternate flow/ Exceptions:	
Post betingelser:	Brugeren bliver ført til rediger vindue.
Author:	Nikolai Christiansen – Allan Søndergaard – Kasper Urhe – Peter Clausen
Revision & Date:	v2.0 - 17/11/2017

Figur 6: use case text opret bruger

Som man kan se i vores use case **B10-1**, så har vi ved et punkt der hedder ”Extend” skrevet en anden use case **B10-2**, dette betyder at man skal køre den anden use case i gennem for at møde de betingelser vi har besluttet der skal opnås for at use casen kan anses som værende afsluttet.

Grunden ti at vi har valgt at extende den på denne måde i stedet for at have både en lang use case for både **B10-1** og **B10-2,** er at dette gør det nemmere og mere overskueligt at læse.

Man kunne tro at **B10-2** skule includes i **B10-1** da den skal køres efterfølgende, men hvis man includer en use case så skal den køres for at den anden kan køres. Det betyder helt enkelt at vi ikke ville kunne køre vores **B10-2** use case for at redigere en bruger uden først at oprette en ny bruger, og det ville ikke give mening.

Der er i hver use case text en eller flere ting der skal ske som en del af use casen. Disse kaldes ”post-betingelser”, og hvis ikke disse er opfyldt så er use casen ikke udført.
Hvis vi for eksempel ser på use case **B10-2** har denne en post betingelse der kræver at brugeren ser en bekræftelses besked.
Det betyder at vores use case ikke er udført korrekt, selvom oplysningerne bliver gemt det rigtige sted, hvis ikke brugeren får en bekræftelsesbesked.

Use Case:	**Rediger Bruger**
ID:	**B10-2**
Prioritet:	
Summary:	Redigere en Bruger
Primær aktør:	Bruger
Sekundær aktør:	
Generalisation:	
Include:	
Extend:	
Pre-betingelser:	En bruger redigerer en eksisterende profil En bruger er taget til rediger vindue fra B10-1
Trigger (igangsættende hændelse)	En Bruger trykker redigerer på deres profil. En bruger har trykket opret ny bruger.
Normal flow:	1. indtaster navn 2. indtaster e-mail 3. indtaster adresse 4. indtaster telefon 5. indtaster fagligområde 6. password 7. brugernavn 8. Bruger trykker gem
Alternate flow/ Exceptions:	8a – Ugyldig information indtastet fejlbeskrivelses besked - Brugeren bliver taget til pågældende sted.
Post betingelser:	Oplysningerne er gemt og brugeren får en bekræftelses besked.
Author:	Nikolai Christiansen – Allan Søndergaard – Kasper Urhe – Peter Clausen
Revision & Date:	v2.0 – 17/11/2017

Figur 7: use case text rediger burger

Da vi har forskellige grader af brugere og disse giver adgang til dele af programmet som det ikke er alle der skal kunne tilgå alle disse dele af programmet, har vi to forskellige use cases til at redigere brugerne.

Dette giver mening når man ser på hvilke informationer man kan tilgå i henholdsvis use case **B10-2** og **B10-4**, hvor i den use case man som bruger, uden ekstra credentials, tilgår (**B10-2**) kan man udelukkende ændre ting som sit navn, email, adresse samt ens kodeord.

Hvis man er underviser og skal have ændret sine credentials så man for eksempel kan oprette og redigere et kursus, så skal en administrator først ved at igangsætte den use case (**B10-4**) hvor man ændrer en brugers credentials. Det er kun folk med administrator credentials der kan tilgå denne use case, det vil sige at en underviser ikke kan ændre i dette medmindre denne underviser også er administrator.

Use Case:	**Rediger Bruger admin**
ID:	**B10-4**
Prioritet:	
Summary:	Redigerer en Bruger
Primær aktør:	Admin
Sekundær aktør:	
Generalisation:	
Include:	
Extend:	
Pre-betingelser:	En bruger redigerer en eksisterende profil En bruger er taget til rediger vindue fra B10-1
Trigger (igangsættende hændelse)	En admin trykker rediger på en bruger.
Normal flow:	1. PA tildeler underviser privileger 2. gem.
Alternate flow/ Exceptions:	
Post betingelser:	Oplysningerne er gemt og brugeren får en bekræftelses besked.
Author:	Nikolai Christiansen – Allan Søndergaard – Kasper Urhe – Peter Clausen
Revision & Date:	v2.0 – 17/11/2017

Figur 8: use case text rediger bruger admin

System Sequence Diagram

Når man har udformet sin use case text kan man lave et system sequence diagram (forkortet SSD.) over den. Dette er en mere visuel repræsentation af den valgte use case.

En SSD giver et hurtigt overblik over hvilke informationer der kommer fra brugeren og hvilke der kommer fra systemet uden at definere hvilke klasser i systemet der har ansvaret for det forskellige metoder.

Når man har lavet en SSD kan man udforme en Operation Contract(OC) for denne SSD, en OC fortæller hvilke ændringer der sker internt i systemet når en metode bliver udført.

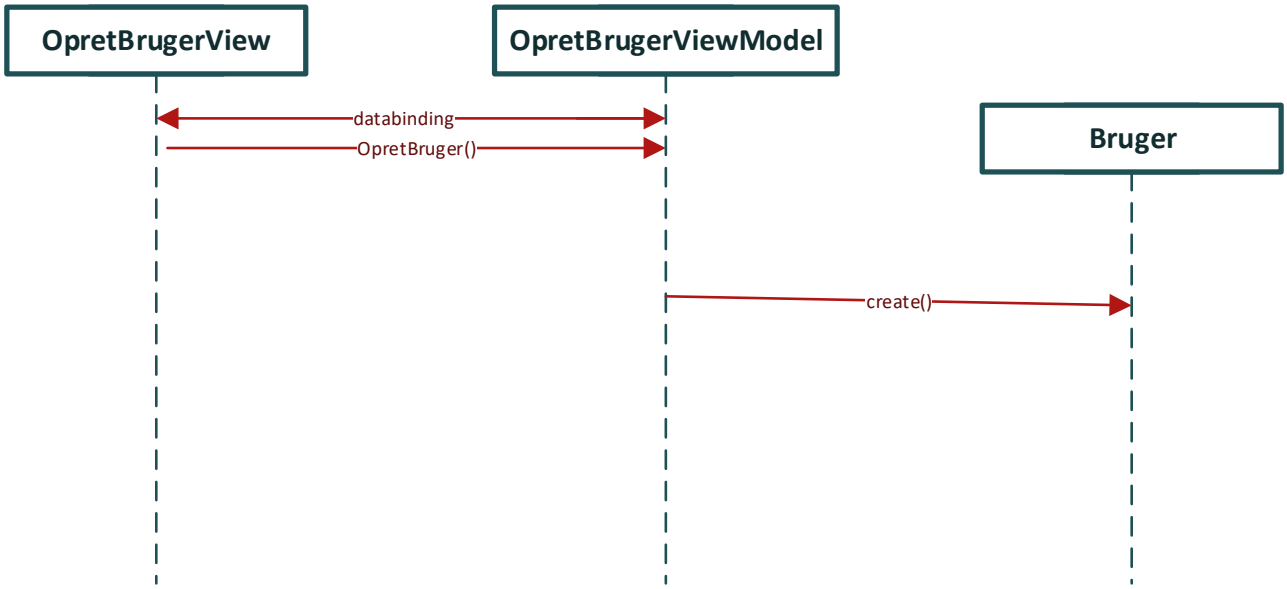
Operation Contract

Name:	Rediger Bruger.
Responsibilities:	Gemme de indtastede informationer.
Cross References:	Use case: Rediger Bruger **B10-2**
Exceptions:	None
Preconditions:	Man har en allerede oprettet burger gemt I system.
Postconditions:	Bruger objektets attributes er ændret til de nye værdier og gemt i databasen.

Design Sequence Diagram

Hvor en SSD giver et let overblik over en use case, giver et design sequence diagram (forkortet DSD.) et indblik i hvilke softwareklasser der tager sig af de forskellige opgaver (hvorfor det er sådan går vi i dybden med i GRASP afsnittet.)

En DSD skal læses fra øverste venstre hjørne og mod højre.



Figur 9: Design Sequence Diagram for opret bruger

Den use case vi har valgt at se på er som tidligere nævnt opdelt i to dele.

Det gør også at vores DSD bliver en del nemmere at overskue, hvis vi havde lavet vores use case som en enkelt og derefter havde lavet vores DSD som en enkelt over både opret og rediger så ville vi have haft betingelser og lignende på diagrammet

Det man kan se ud fra denne DSD er at vores view er databindet til vores viewmodel og at det er viewmodellen der har ansvaret for at få de kommandoer der bliver sendt fra viewet til at ske.

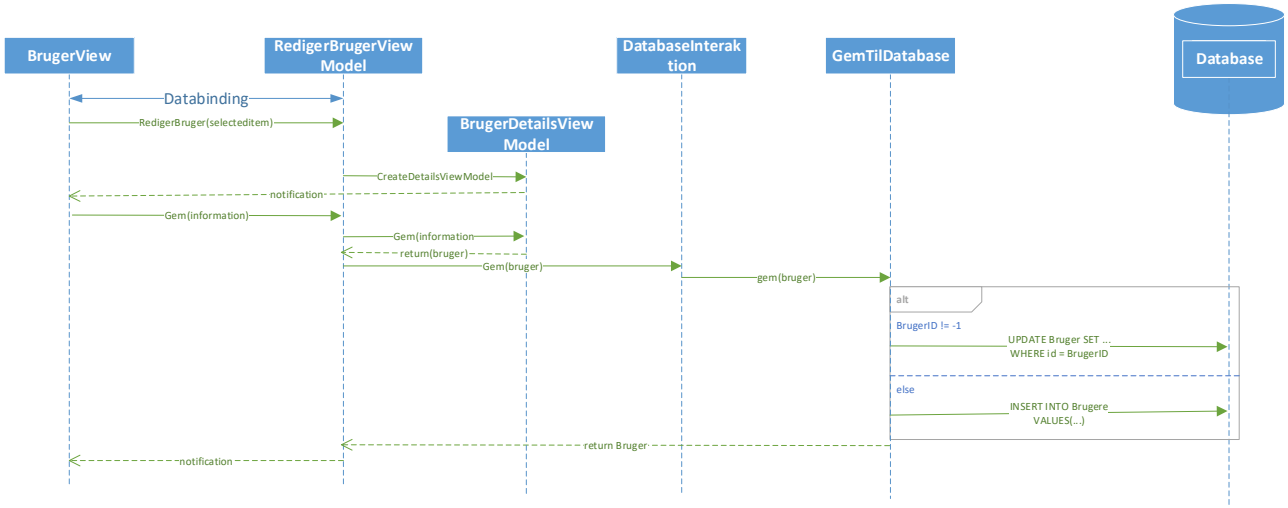
Det metodekald man kan se der går fra viewet og ned til viewmodellen har ikke andet ansvar end at give viewmodellen besked om at der er bedt om dette.

Viewmodellen har så ansvaret for at sende denne kommando ned til den korrekte klasse, i dette tilfælde er det i bruger klassen at informationerne for at skabe et bruger-objekt ligger, og derfor er det denne klasses ansvar.

Pilenes retning er vigtige for at man kan se hvilken klasse der sender noget og hvilken klasse der modtager dette og derfor er ansvarlig.

Den klasse som pilen peger på er den klasse som modtager signalet, og den klasse hvor enden uden pil er den klasse der sender.

Som man kan se er der mellem vores view klasse og viewmodel klasse en pil der peger i begge retninger som hedder ”databinding”, denne pil peger begge veje, da den er ansvarlig for både at sende og modtage data frem og tilbage mellem vores brugergrænseflade og vores program.



Figur 10: Design Sequence Diagram for rediger bruger

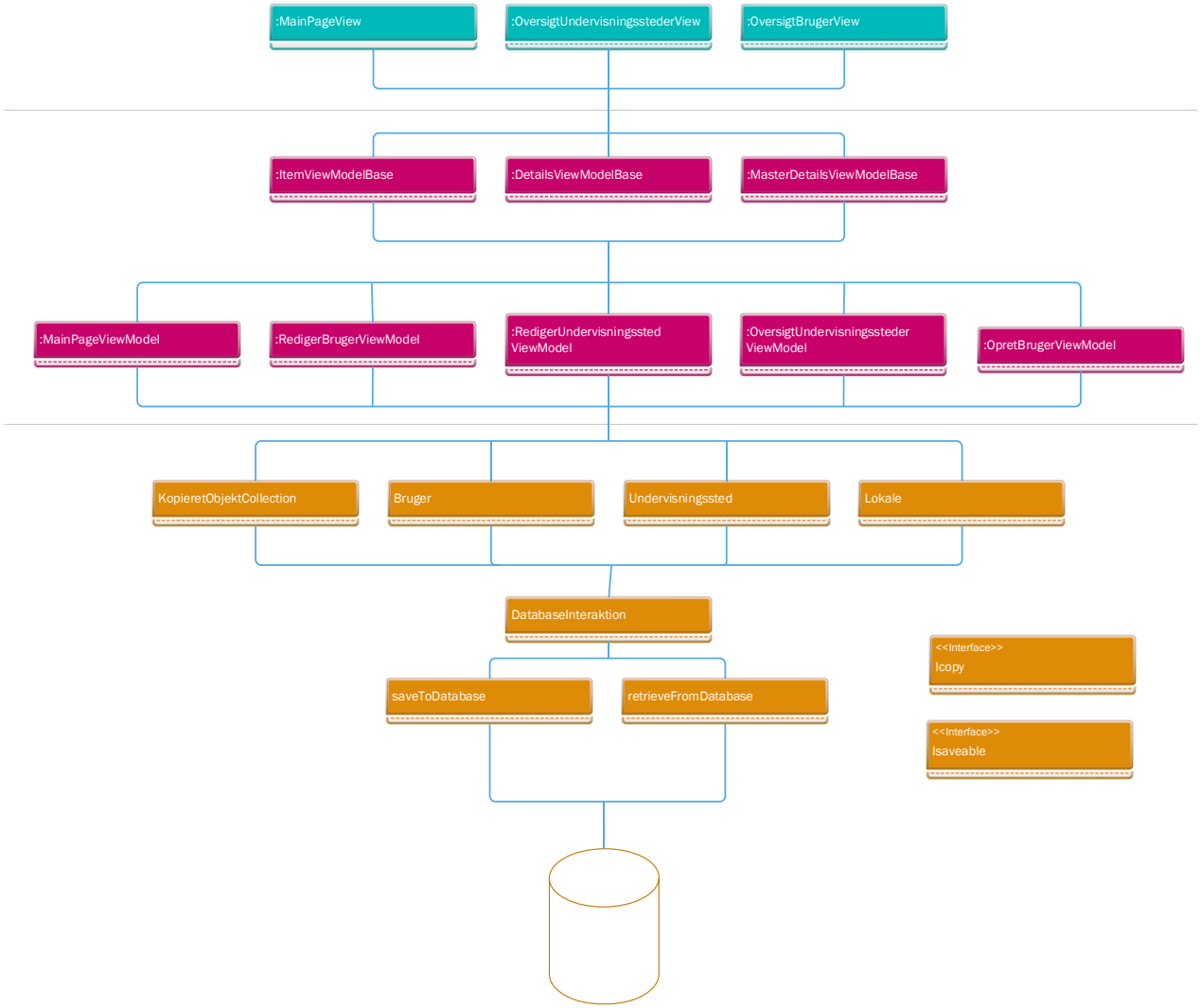
DSD for rediger delen af vores use case ser noget mere indviklet ud, men det eneste der er anderledes fra den første er at vi her har et loop hvor der er to forskellige udfald afhængigt af om brugeren eksisterer i databasen eller ej.

Denne løkke er repræsenteret ved en firkant med et krav i venstre hjørne, her checkes der for om en bruger med det specifikke brugerID allerede eksisterer i databasen. Hvis denne bruger allerede er i databasen vil dennes attributes blive opdateret til de attributes der er givet med fra brugeren, hvis der ikke findes en bruger med dette brugerID, vil databasen oprette og indsætte en bruger med disse attributer.

Software klasser er i denne model repræsenteret ved de rektangulære kasser, grunden til at den kasse der hænger længere nede end de andre er at der i den bliver oprettet et objekt og det repræsenteres sådan i en DSD.

Man vil altid repræsentere en database med en cylinder som man kan se yderst til højre.

Arkitektur



Som vi har skrevet i vores afsnit om IT-strategi, så benytter vi os af en MVVM arkitektur.

Dette er en forkortelse for Model, View og ViewModel. Disse skal ses som tre adskilte lag der har forskellige ansvar afhængigt af hvilket lag de ligger i.

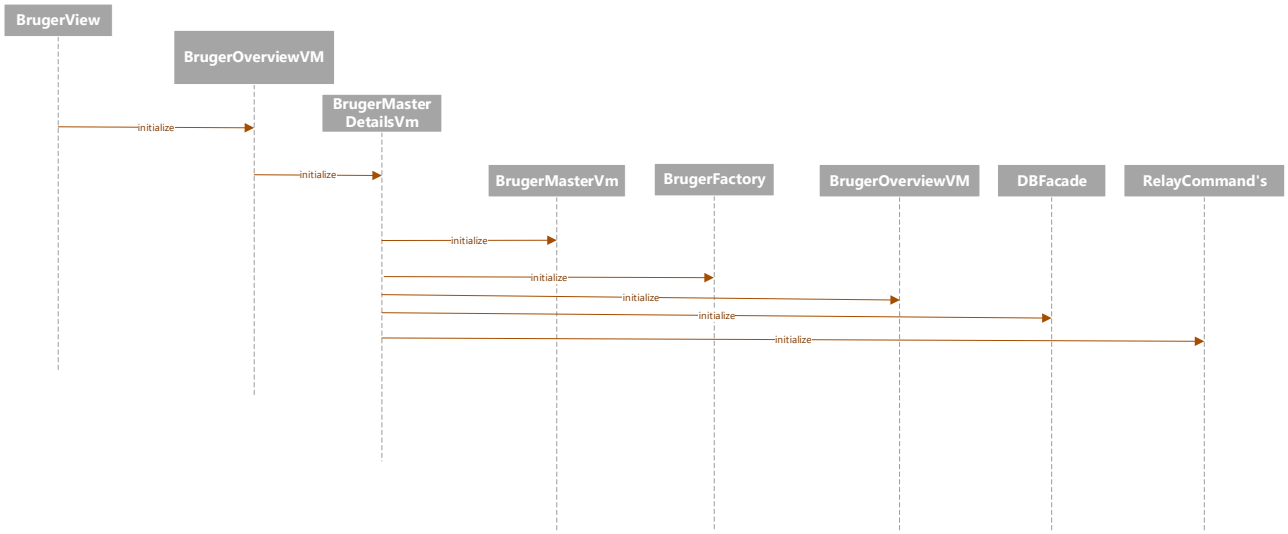
Det er vigtigt at forstå at der ikke er forskel på en model klasse og en viewmodel klasse i forhold til hvordan den er bygget op, forskellen ligger i hvad klassen tager sig af. Vi benytter os af MVVM fordi vi ved at skabe disse lag, kan skabe low coupling og high cohesion (læs mere i GRASP afsnittet)

View laget er ansvarligt for programmets UI, altså brugergrænsefladen. Viewet kan ikke andet end at vise det specifikke view det er lavet til, det betyder ikke at man ikke får funktionalitet gennem ens view.

Der bliver selvfølgelig vist ting fra programmet i viewet og der er knapper og lignende med funktion, men disse går gennem ModelView laget der agerer som bindeled mellem vores view og vores model lag.

I viewmodel laget er der stadig ingen reel logik for programmet klasserne i VM-laget er der udelukkende for at behandle og sende metodekald og notifikationer til den rigtige view klasse eller den rigtige modelklasse afhængigt af hvilken vej kommunikationen går. Et view kan godt have flere forskellige viewmodeller, dette hænger sammen med at vi på denne måde kan opretholde low coupling, ved ikke at skulle hive fat i en klasse som har en masse overflødigt i forhold til den specifikke ting vi ønsker at udføre.

Start Up Design Sequence Diagram



På et start up design sequence diagram kan man se hvilke klasser det bliver sat i gant som en del af at være navigeret til et bestemt view eller når man starter programmet op første gang.

GRASP

GRASP står for General Responsibility Assignment Software Patterns, og vi benytter det til at vurdere hvilken af vores software klasser der vil være bedst til at tage sig af eller udføre et bestemt stykke arbejde indenfor objekt orienteret programmering.

Vi opererer hovedsageligt med 5 principper/mønstre til at bestemme dette:

- Creator
- Information Expert
- Low Coupling
- Controller
- High Cohesion

Creator

En creator er den klasse der har ansvaret for at oprette et givent objekt, vi kan få alle klasser til at oprette det given objekt, det der er opgaven når man designer sit program og beslutter hvor tingene skal udføres, er at få det til at foregå et sted hvor det giver mening.

Hvis vi ser på hvordan vi gjorde i vores første semesters opgave oprettede vi for eksempel et objekt af typen bil, dette gjorde vi vores bilkatalog klasse ansvarlig for da det var den der skulle bruge objektet.

I dette projekt var vi inde på at ville gøre det samme da vi startede, men i dette projekt har vi med databaser at gøre og ikke med et lokalt katalog. Det vil sige at når vi skal oprette og gemme et bruger-objekt så er der ikke en katalog klasse i programmet der kan gemme det lokalt i det vi opretter det.
Vi har i dette projekt givet brugerklassen selv ansvaret for at oprette objektet.

Information Expert

En information expert er en klasse der har alt den information der er krævet for at udføre en given opgave.

Når man vurderer hvilken af klasserne i ens kode der har bedst forudsætninger for at udføre en opgave omtaler man den klasse der har alt den nødvendige information som information expert.
Ved at gøre det på denne måde kan man udnytte en klasses potentiale fuldt ud og man kan samtidigt undgå at en anden klasse skal have mere information end nødvendigt i sig for at kunne klare en opgave som den første klasse er bedre rustet til at udføre.

Low Coupling

Low coupling er et koncept der skal sørge for at ens kode ikke er for afhængig af hvad der sker i andre klasser.

Dette betyder effektivt at hvis man ændrer noget i en software klasse så skal ens design være udformet sådan så det har så lidt indflydelse på de resterende klasser.
Det er derfor vi har valgt at bruge generics i vores program, dette gør at vi har mindre specifik kode og mere general kode. Når vores metoder er lavet generic fremfor indskrænket af parameter typer, gør det at vi kan udnytte den metode for nye objekter der kunne tilføjes til projektet.

Et eksempel på dette i vores projekt er at den klasse der er ansvarlig for at gemme ting til databasen er ansvarlig for alle vores objekter der skal gemmes, altså både brugere og kurser for eksempel.
Så i stedet for at vi har en specifik metode per objekt så har vi en generel metode som vi også kan udnytte

til undervisningssted og en helt sidste ting hvis kunden beslutter sig for at skulle udvide programmets funktionalitet.

Controller

En controller er en klasse eller en samling af klasser der har ansvaret for at vores domæne layer og vores UI kan tale sammen.

Da vi benytter os af MVVM arkitektur er det vores view model layer der har rollen som controller.

Grunden til at man har dette lag mellem ens UI og ens domæne lag er for at ens business logik ikke skal være direkte bundet til et view.
I "the model-view separation principle" forsvares denne opdeling med at de ting som viewet tager sig af kun skal relatere til åbning af andre vinduer og sådanne opgaver og ikke være kodet til at kalde ting direkte fra domæne layeret. Dette gøres fordi at man kan genbruge sin business logik i andre applikationer uden det samme view og uden at man skal skrive alle kaldene (som view modellen står for) om for at genbruge dem i et andet projekt.

High Cohesion

High cohesion går ud på at man skal holde sine softwareklasser fokuserede på det bestemte man vil have den til. Altså ikke at have en masse overflødigt stående i en klasse som den ikke skal bruge, men som en anden klasse kan bruge, dette ville være bedre at skrive i en anden klasse.

Dette hjælper med at holde software-klasserne overskuelige så man ikke står med for eksempel den software klasse der indeholder information om hvad et bruger-objekt indeholder også indeholdt noget information om hvad information et lokale indeholder.

Dette understøtter også vores muligheder for at skabe low coupling.

Øvrige patterns

Der findes i alt 9 GRAS Patterns, ud af disse har vi haft stort fokus på Polymorphism.

Dette har vi brugt i vores bruger klasse, hvor alle de brugere der er af programmet alle kommer fra den samme klasse.
Man kunne have skrevet en klasse til hver af disse, hvilket ville have givet os 3 forskellige klasser der i bund og grund indeholde det samme.

Valg af kode

Model og viewmodel lag

Vores udfordring med koden har ikke været domæne koden, men viewmodel koden. Da vores domæne logik er relativ simpel har vores udfordring været at få det vist i vores view på den måde vi ønsker.

Vi har hovedsageligt gjort brug af en master details model, til visning af vores data, som illusteret på figur 1.

Figur 1

Dette bliver gjort ved hjælp af 5 klasser: Details-Klasse, Master-Klasse, Item-Klasse, Factory-Klasse og en Master-Details klasse.

Vi har forsøgt at lave vores program så generalt som muligt, inden for vores evner. Det gør sig gældende i at vi har lavet de 5 før nævnte klasser generic, dem kalder vi baseklasser, så alle klasser som skal repræsenteres som et master details view kan parametiseres i baseklasserne.

Master details forholdet bliver brugt i næsten hvert af vores views og derfor er der generet en del kode med de 5 base klasser så vi vil gennemgå dem kort herunder:

FactoryVmBase

Factory klassen er har til ansvar at tage et model objekt, så som bruger og laver det til et viewmodel objekt, så som BrugerItemVm.

```
public override ItemVMBase<Bruger> ItemViewModel(Bruger obj)
{
    return new BrugerItemVM(obj);
}
```

Figur 2

ItemViewModelBase

ItemViewModelBase klassen har til ansvar at skulle være logik til de enkelte objekter der bliver vist i master delen af master details, eller sagt på en anden måde er de individuelle objekter du kan se og trykke på i listviewet.

```
public class ItemVMBase<T>
    where T : class
{
    public T DomainObject;
    6 references | Fridai, 23 days ago | 1 author, 1 change | 0 exceptions
    public ItemVMBase(T Obj)
    {
        DomainObject = Obj;
    }
    5 references | Fridai, 16 days ago | 1 author, 1 change | 0 exceptions
    public virtual string Description { get => "we er i base klassen"; }
}
```

Figur 3

Klassen har en enkelt propertie, som er gjort virtuel så i den paramatiseret klasse, kan vi overwrite den med den aktuelle beskrivelse.

MasterVmBase

Masterviewmodelbasen har til ansvar at tage en normal liste af objekter og lave den om til et observeable liste af viewmodel objekter, vi er nød til at bruge observeablelist da en normal liste ikke kan bruges til visning i et listview, i XAML koden.

Det gøres ved at give metoden en factoryVmBase med af typen som er relevant, samt den list, man ønsker lavet om til en observeablelist.

```
3 references | Fridai, 22 days ago | 1 author, 1 change | 0 exceptions
public ObservableCollection<ItemVMBase<T>> CreateItemVMCollection(FactoryVMBase<T> factory,List<T> collection)
{
    _ItemVMCollection.Clear();

    foreach (T Obj in collection)
    {
        _ItemVMCollection.Add(factory.ItemViewModel(Obj));
    }
    return _ItemVMCollection;
}
```

Figur 4

Vi starter med at .Clear, kollektionen da vi godt kan kalde på den samme metode flere gange og ikke ønsker kopier af det samme objekt i vores liste mere en 1 gang.

Derefter laver vi et simpelt foreeach løkke som tilføjer alle objekter fra den medgivet liste til den nye observeablekolletion, med metoden fra den medgivet factory, med det medgivet objekt som parameter.

DetailsVmBase

DetailsViewModelBase, har til ansvar at fungere som en mellemled mellem viewet og en model klasse, så der kan skrives og hentes fra det pågældende objekt.

```
public abstract class DetailsVMBase<T>
    where T : class
{
    private T _domainObject;
    4 references | Fridai, 15 days ago | 1 author, 3 changes | 0 exceptions
    public DetailsVMBase(T obj)
    {
        _domainObject = obj;
    }

    73 references | Fridai1, 10 days ago | 2 authors, 3 changes | 0 exceptions
    public T DomainObject
    {
        get => _domainObject;
        set => _domainObject = value;
    }

}
```

Figur 5

MasterDetailsVmBase

Masterdetailsviewmodelbase klassen er hvor det hele bliver samlet og instanceret, samt generalt er klassen ansvarlig for alt hvad der har med master details forholdet at gøre, så som knapper og relevante metoder/properties.

For at viewet kan fungere er vi nød til at havde nogle Properties som kan binde de før omtalte klasser sammen, vi starter med ItemVMSelected

```
8 references | Fridai1, 2 days ago | 2 authors, 7 changes | 0 exceptions
public virtual ItemVMBase<T> ItemVMSelected
{
    get
    {
        FieldsEnabled = false;
        OnPropertyChanged(nameof(FieldsEnabled));
        return _itemVMSelected;
    }
    set
    {
        _itemVMSelected = value;
        DetailsVM = CreateDetailsVM();
        OnPropertyChanged(nameof(DetailsVM));
        OnPropertyChanged(nameof(TrueIfSelected));
        OnPropertyChanged(nameof(USMasterDetailsVm.LokaleCollection));
        OnPropertyChanged(nameof(USMasterDetailsVm.SelectionChanged));

    }
```

Figur 6

På figur 6 kan det ses at vores propertie har en Get og en Set. Når vi ber om data fra propertien, bliver en bool sat til false, denne bool er bindet til viewet og er ansvarlig for at gører felter enables eller disabled, vi ønsker ikke at brugeren kan redigere i objektet før de trykker på en rediger knap.

OnpropertyChanged bliver kaldt for at gører FieldsEnabled properien opmærksom på at den er blevet ændret.

I SET, sætter vi Instancefieldet `private ItemVMBase<T> _itemVMSelected;` lig med den valgte itemviewmodel i listviewet. Der efter sætter vi DetailsVM propertien lig med metoden CreateDetailsVM().

```
public DetailsVMBase<T> DetailsVM
{
    get => _detailsVM;
    set
    {
        _detailsVM = value;
        OnPropertyChanged();
    }
}
```

Figur 7

```
public DetailsVMBase<T> CreateDetailsVM()
{
    if (_itemVMSelected == null)
    {
        return null;
    }
    else
    {
        return _vMFactory.CreateDetailsViewModel(ItemVMSelected.DomainObject);
    }
}
```

Figur 8

Så det valgte viewmodel objekt i listviewet bliver brugt til at oprette et details view ved hjælp af Factory, som set på figur 8. DetailsVM propertien bliver så bindet til UI, hvor man så er i stand til at gå ned og vælge den attribute på det valgte objekt. `Text="{Binding MasterDetails.DetailsVM.DomainObject.Deposit,`.

Master detail parametiseret

Det har været nødvendigt for os at tilføje yderligere kode til de instanceret masterdetail klasser, som er for specifict til at komme i base klassen, for eksempel har vi i undervisningsstedview, lavet et mini masterdetails, i form af at kunne tilføje lokaler til et undervisningssteds objekt. Dette kræver selfølgelig yderlige kode som er unikt til lige netop dette view.

At få det lavet var lidt af en udfordring, at finde ud af hvor koden skulle skrives henne til at starte med og hvordan det bedst kunne svarer sig. Vores tankegang var om det skulle i overview klassen eller om det hørte til i masterdetails klassen, men efter som at stadig var en del af et større master details forhold, bestluttede vi os for at skrive koden i den instanceret master details klasse.

Koden til at tilføje lokaler til en liste, viste sig at være meget lignende den måde vi havde valgt at gøre det på i baseklassen, med en metode der tilføjer den string som blev indtastet i lokale textboxen.

```
private void CreateRoom()
{
    DetailsVM.DomainObject.CreateRoom(_lokale);
    OnPropertyChanged(nameof(LokaleCollection));
    OnPropertyChanged();
    _lokale = null;
}
```

For at få det vist i et listview i viewet, gjorde vi det på samme måde som i figur 4.

OverView viewmodel

Da der i et UI vindue også foregår mere end bare et master details forhold så som Login, har vi valgt at instancere alle relevante klasser til det enkelte view i en overview klasse, for eksempel BrugerOverview.

GodBase

Efter at havde lavet et bar OverviewVm gik det hurtigt op for os at vi gentog os selv en del med henhold til login releteret kode. For at gøre vores kode mere dry valgte vi at lave en Baseklasse til overviewet, som inholder alt Login logikken.

LoginVM

LoginVm klassen er ansvarlig for at styrer hvem der bliver lukket ind i systemet og gemme den bruger som bliver lukket ind i en static attribut. Vi bruger en static attribut så vi, i hver klasse kan sætte _loggedInBruger `private Bruger _loggedInBruger;` attributen lig den statiske attribute `_isSomeOneLoggedIn = LoginVm.IsSomeOneLoggedIn();`.

Når LoginVm bliver instanceret kører den en metode i constructoren FetchFromDB().

```
public async void FetchFromDB()
{
    List<Bruger> L = new List<Bruger>();
        L = await _dbFacade.LoadMultiple("User");

    foreach (Bruger b in L)
    {
        _users.Add(b.Username,b);
    }

}
```

Som laver en ny liste og gør den lig med en liste af bruger objekter fra databasen, derefter bliver de bruger objekter tilføjet til en dictionary, med brugerens brugernavn som key og bruger objektet som value.

Grunde til at vi benytter en dictionary er primært pågrund af 2 årsager:

- Når der kommer mange brugere og der skal søges efter en bruger er en dictionary meget hurtigere.
- Ved at bruge en dictionary kan vi lettere udfører opgaven at søge efter de 2 atributes vi ønsker og sætte den endelige bruger lig med den statiske attribut givet oplysningerne er korrekte.

Selve login metoden er af typen bool. Den starter med et sanity check

```
if (_brugernavn == null)
{
    // mangler exception
    throw new NotImplementedException();
}
```

Hvorefter hvis brugernavn != null, `bool userfound = _users.ContainsKey(_brugernavn);`

Ser vi om det username som er indtastet også findes i vores dictionary, vi søger efter key.

Hvis brugeren findes gør vi den fundne brugere lig med en attribute af typen bruger

```
Bruger FoundUser = new Bruger();

// hvis der findes en user med det brugernavn
if (userfound)
{
    // findes user gør vi Founduser li med den fundne user
    FoundUser = _users[_brugernavn];
}
```

Til sidst checker vi for om den fundne brugers password stemmer overens med, det password som er blevet indtastet, er det korrekt gør vi den statiske attribute lig med den fundne bruger, og returner true.

```
if (userfound && FoundUser.Password == _password)
{
    _loggedinUser = FoundUser;

    return true;


}
```

Hvis det er ikke er gældende, sætter vi brugernavn og password til null og returner false.

Problemet:

Problemet vi løb ind i med dette var at få det bindet til en knap og få den til at kører metoden før den sendte brugeren videre. I første omgang checkede den koden men om den så var true eller false sendte den alligevel brugeren videre, ved hjælp af Click eventen i XAML `Click="Login"`.

Løsningen:

Løsningen blev at kører commandoen fra codebehind, til click eventen.

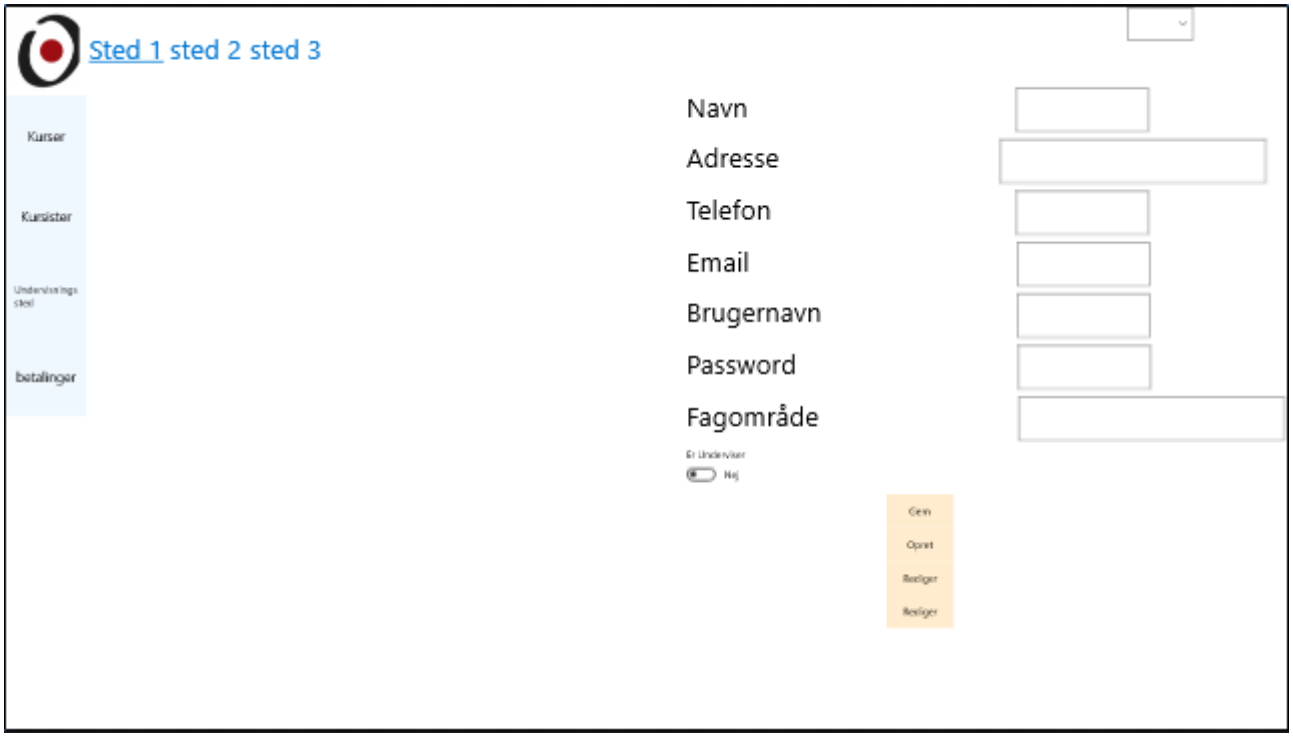
```
private void Login(Object sender, RoutedEventArgs e)
{
    ((LoginVm)this.DataContext).LoginCommand.Execute(null);

    if (((LoginVm)this.DataContext).Brugernavn != null)
    {
        Frame.Navigate(typeof(BrugerOverView));
    }
```

View lag

Vi har i løbet af projektet haft kontakt med vores kunde (mohammed) når vi havde noget nyt at vise ham, så vi kunne se om det levet op til kundens forventninger, og hvad der skulle laves om. Vores første møde med kunde viste vi ham vores UI og login funktion samt oprettelse af en bruger. Dette var kunden ganske tilfreds med og vi fortsatte. Den næste demo gik ud på at vist kunden oprettelse af undervisningssted og tilhørende lokale, her var kunden også glad for funktionalitet men var lidt utilfreds over brugervenligheden og generalt vores GUI.

Vores første udkast af GUI:

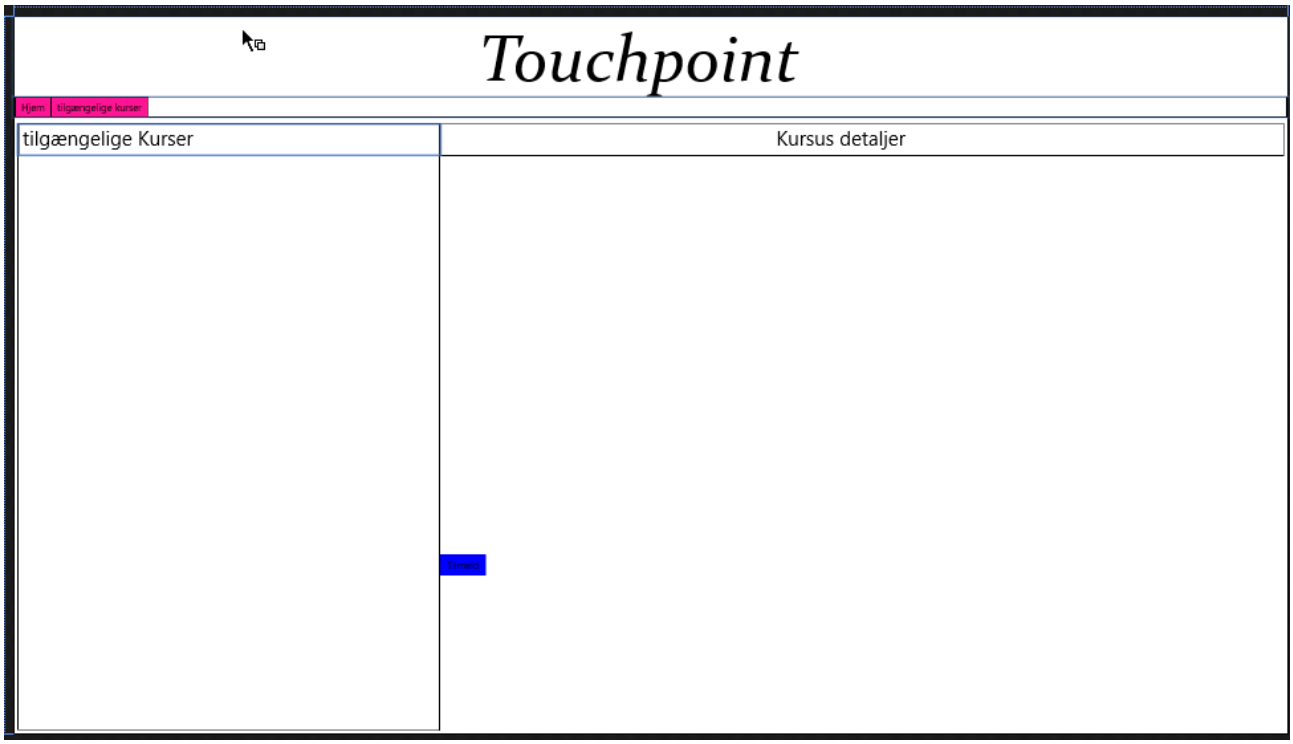


Tanken bag dette design var at havde venstre navigations menu og toppen af programmet altid være det samme, for let at kunne arbejdet ind i muskel hukommelsen hvordan man navigeret rundt som bruger.

Med et listview til venstre og details til højre.

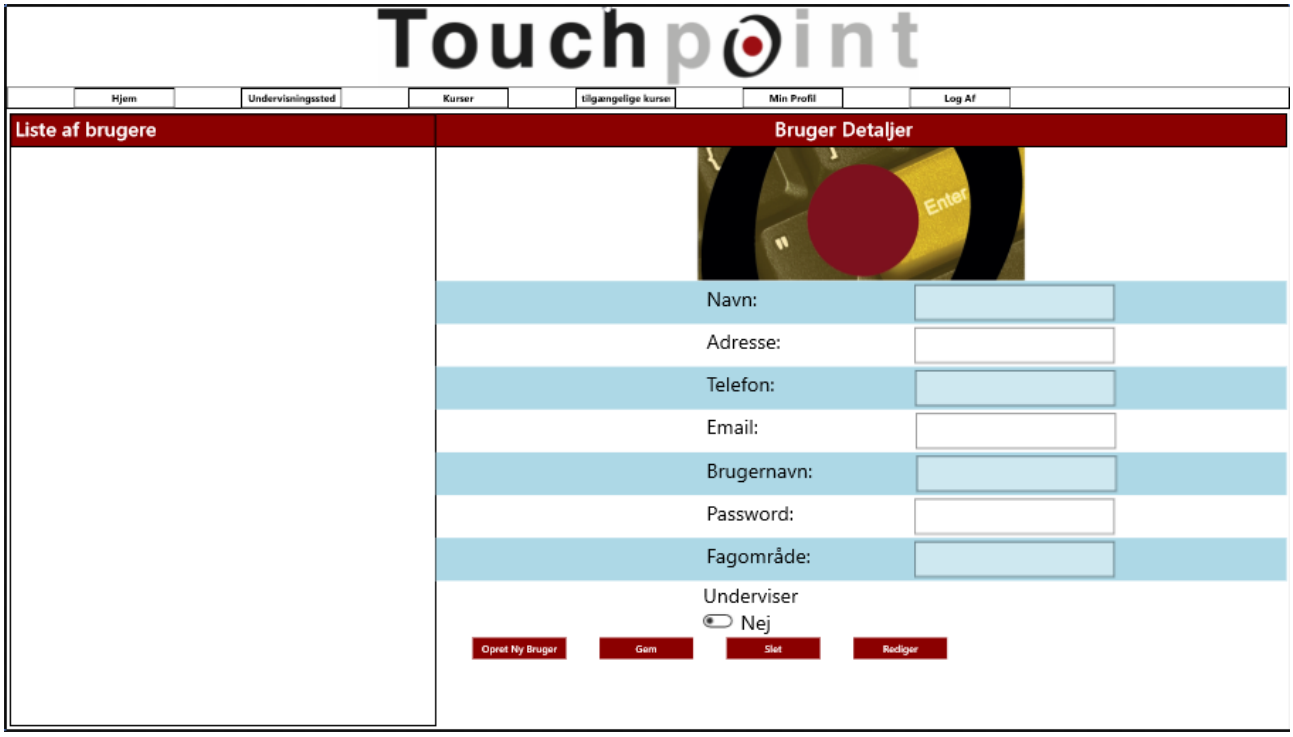
Ude i venstre side havde vi, som sagt en navigations menu, så vi kunne komme rundt til de andre views. I toppen havde vi planlagt en form for stifinder, så jo længere ind i programmet man kom, ville de opdatere sig. Så for eksempel, klikkede man på kurser, så ville sted 1 vise kurser, derefter trykkede man på opret kursus og blev taget til opret kursus view, og sted 2 vil nu skrive opret kursus, så var tanken det skulle kunne fungere som knapper så man kunne gå tilbage til et tidligere skridt nemt, som var det en stifinder i Windows.

Andet udkast:



Her er et første udkast af det nye GUI. Her har vi ændret GUI helt og starter med en stor titel af firmaets navn, herunder en menu bar for navigation rundt i programmet, vi bibeholder vores tidliger setup med et listview til venstre og detaljer til højre. Dette GUI har mere detaljeret beskrivelse af hvad du kigger på, og klar afgrænsning af hvor grænserne af hvert GUI element starter og stopper i form af sorte linjer.

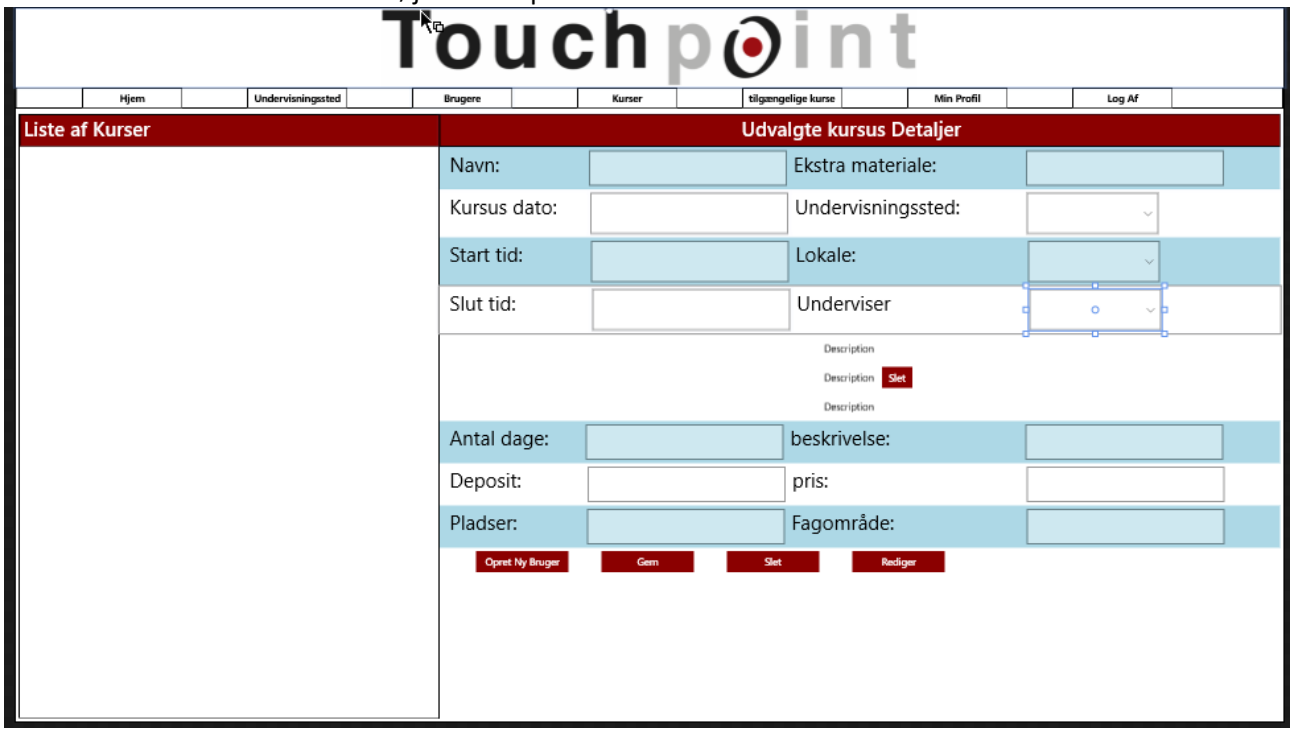
endelige udkast:



Her har vi igen taget udgangspunkt i Touchpoints originale farver, deres mørke røde og brugt det farve tema, til at bringe mere liv i vores GUI. Vi har tilføjet en baggrunds farve til de stackpanels, som holder detaljerne, for lettere at kunne adskille dem fra hinanden. Vi er meget tilfredse med hvordan vore endelig GUI kom til at se ud.

Oprettelse af kursus

For at kunne oprette et kursus, skal man havde oprette et undervisningssted, nogle lokaler og mindst en underviser. Derefter skal man tilføje dem i opret kursus GUI.



Figur 18

Her var tanken, at bruge combo boxe, ved undervisningssted, lokale og underviser. Når man vælger hvilke undervisningssted man ønsker kursus skal forgå på, bliver combo boxen lokale, sat til en liste af det valgtes undervisningssteds lokaler.

Når det kom til underviser skulle det fungere på samme måde, som det gør når man opretter et undervisningssted, når man har valgt et element i combo boxen bliver det automatisk gemt i listen og i kursus objektet.

Problemet der opstår er designet af systemet, som det er lige nu er det designet til at, i det du instancere et masterdetails viewmodel objekt, beder det objekt om data fra databasen.

```
public List<Bruger> TutorList
{
    get
    {
        List<Bruger> underviserList = new List<Bruger>();
        foreach (Bruger b in _Catalog)
        {
            if (b.IsTutor)
            {
                underviserList.Add(b);
            }

        }
        //underviserList.Add(new Bruger("berta", "asd", "asd", "12345678", "asd st. 3", false, "asd@asd.com", "asdery", true));
        //underviserList.Add(new Bruger("prop", "asd", "asd", "12345678", "asd st. 5", false, "asd@asd.dk", "asdery", true));
        return underviserList;
    }
}
```

Figur 19

Som det kan ses i figur 19, har vi en simpel propertie som, returnere en liste af brugere som alle har IsTutor true. Den løber kollektionen _Catalog igennem for at finde disse brugere, den får _Catalog, fra base klasen. _Catalog er det instancefield som er ansvarlig for at holde alle brugere, som den får tildelt af databasen, ved hjælp af metoden set på figur 20.

```
public async void RefreshList()
{

    List<T> waitinglist = await _dbFacade.LoadMultiple(_table);

    if (waitinglist != null)
    {
        _Catalog = waitinglist;
    }
```

Figur 20

Problemet opstår så i at når propertien på figur 18, tutorlist køres. Da der på tidspunktet den kører ikke kommet nogen bruger objekter i _catalog og den retunere derfor en tom liste tilbage til viewet.

Løsningen

Umidlbart kan vi se 2 løsninger.

- Lave _catalog en singleton. Da programmets første side efter login er en brugeroversigt, bliver _catalog fyldt med bruger objekter allerede der og havde det været en singleton, behøver vi ikke hente brugere igen, til opret kursus scenariet, problemet med dette er at vi aldrig er sikker på om de brugere vi henter stadig er up to date, da de er gemt lokalt, og vi derfor risikere at bruge forældet data
- Den anden løsning ville være at lave en helt ny database forbindelse inde i kursusoverview viewmodellen, altså kursus's datacontext. Dette vil betyde at når instanceret viewmodellen ansvarlig for kursus viewet, ville en liste blive fyldt med bruger objekter, som vi senere kunne sortere i.

Begge løsninger har fordele og ulemper ved deres brug og på nuværende tidspunkt kan vi ikke sige hvad der ville være den bedste løsning, men den sikreste løsning ville være nr 2. da vi altid er sikre på at bruge de nyeste data.

Del konklusion og reflektion:

Vi synes at model lags koden var relativ simpel at skrive, det var nødvendigt at lave lidt ændringer i designet, som det skred fremad, men det er jo pointen med Unified process. Nævn ligt at vi havde glemt at tage højde for nogle metoder i kursus og undervisningssteds klasserne. Udover det var der ikke andre vanskeligheder med model laget. Vores problemer kom først rigtigt på spil da vi skulle skrive viewmodellerne, at få det hele til at snakke sammen, som vi havde forstillet os under designet viste sig at tage meget længere tid end forventet, og er desværre årsagen til at vi har måtte unscope lidt i projektet. Dvs. at vi har valgt ikke at lave iteration 3, for at få mere tid til at fikse de problemer vi havde med viewmodellen.

Da vi startede projektet var vi alle enige om at vi ville prøve at skrive det så smart, som vi var i stand til med vores nuværende erfaring, det betød blandt andet at lave database klasserne generic for at gøre programmet mere dry, dette havde vi regnet ville tage 1 uges tid, men ente med at tage 2-3 uger. Hvilke betød, at når viewmodellerne skulle kodes, blev der taget udgangspunkt i lokale kollektionerne og ikke database forbindelsen. Tanken bag de lokale kollektioner var dog at de skulle kundes sættes lig med en liste fra databasen med objekter der fra. Dette program er ikke et stort program, men er trods alt det største program vi har lavet til dato. C# og Xaml, virker som et godt værktøj til at skrive programmer i denne størrelse, med mere erfaring og bedre koordination er vi sikker på at vi ville kunne havde færdig gjort alle projektets iterationer.

Database klasser

Laget til alt databaseinteraktion. Dette lag kører parallelt til model-laget og bliver accessed fra ViewModel-laget.

Alle klasser heri benytter asynkronitet ved at awaite visse handlinger i Task-type metoder.

Alle metoder heri med undtagelse fra metoder i DatabaseFacade.cs og hjælper-metoden PluraliseTableName(string table) i klassen FetchFromDb.cs arbejder via den eksterne klasse HttpClient. Da denne klasse implementerer IDisposable er det anbefalet at bruge klassen ved et "using" statement. Alt skabt inden i et using statement er fjernet – disposed – når statementet slutter. Hvis vi havde høj traffik og hvis applikationen kørte *serverside* og blev accessed fra mange brugere samtidigt ville dette være en dårlig løsning da det ville kræve for mange resurser at konstant skabe nye HttpClient objekter. Det ville muligvis også kræve flere resurser for databasen at konstant håndtere nye connections, men vi er ikke sikre på det. Da vi derimod kun kommer til at have få connections kan vi godt håndtere at dispose HttpClient efter brug.

DatabaseFacade.cs

Facade klasse til databaselaget.

Klassen indeholder de 4 metoder vi bruger til at interagere med databasen: SaveSingle(int id, T item, string table) som kalder Save(id, item, table) i SaveToDb klassen, LoadSingle(string table, int id) som kalder FetchSingle(table, id) i FetchFromDB klassen, LoadMultiple(string table) som kalder FetchMultiple(table) i FetchFromDB klassen, og DeleteSingle(int id, string table) som kalder Delete(id, table) i DeleteFromDB klassen.

SaveToDB.cs, FetchFromDB.cs, DeleteFromDB.cs

Klasser til save-, load-, og delete-funktionalitet i vores database. Klassen tager den generiske type T som har kravet at implementere ISaveable interfacet. Hver klasse indeholder et instance field, en constructor som initialiserer det instance field, og mindst en metode til databaseinteraktion:

Save(int id, T item, string table)

Denne metode ligger i SaveToDB.cs og er ansvarlig for at gemme et objekt i vores database. Den tager et objekt af typen T og omdanner dens værdier til en JSON streng, omdanner denne til en StringContent klasse (som nedarver fra HttpContent, hvilket PostAsync() og PutAsync() kræver, gør tabelnavnet til flertal, og skaber derefter en ny række i en af vores database tabeller hvis vores objekts ID er 0 (PostAsync()) eller opdaterer en eksisterende række hvis objektets ID ikke er 0 (PutAsync()).

Vi skaber en string fra vores objekt gennem JsonConvert fordi vi ikke kan give StringContent et rent objekt og fordi det er den korteste måde at få en kompatibel string af vores objekts properties. StringContent (og

formentlig alle andre klasser af typen HttpContent) er simpelt sagt en streng omdannet til et format som en HTTP server kan forstå. Vores parametre inkluderer selve strengen, måden hvorpå den bliver kodet (her UTF8), og den medie type som bliver brugt til ”header.” Standard medie type er ”text/plain” hvilket vi ikke kan bruge til en JSON string, så vi sætter vores til *application/json*.

FetchSingle(string table, int id)

Denne metode ligger i FetchFromDB.cs og er ansvarlig for at hente *en* række fra databasen. Den tager en string med tabellens navn samt en ID, gør tabellens værdi til flertal, henter en række fra en tabel i vores database (GetAsync() med stien ”api/{table}/{id}”) og awaiter dette, læser svarets resultatet og returnerer det hvis rækken var succesfuldt hentet, og returnerer *null* hvis ikke.
Omdannet til en SQL query ville GetAsync() kaldet være ”GET * FROM table WHERE Id = {id}” hvor {id} er den ID vi får i kaldet til FetchSingle().

FetchMultiple(string table)

Denne metode ligger i FetchFromDB.cs og er ansvarlig for at hente en hel tabel fra databasen. Den fungerer præcis som FetchSingle() bortset fra at resultatet her bliver smidt i en liste ved ToList() og at stien givet til GetAsync() er ”api/{table}” for at få alle værdier i en tabel. Omdannet til en SQL query ville det blive ”GET * FROM table”

Delete(int id, string table)

Denne metode ligger i DeleteFromDB.cs og er ansvarlig for at slette en række i databasen. Det er den korteste af de fire metoder vi har skrevet til databaseinteraktion. Den skaber en ”flertal”-string fra table, kalder DeleteAsync med stien ”api/{pluralised}/{id}” (hvor pluralised bare er værdien af table med et ”s” på), og awaiter dette.

Alle metoderne benytter et using-statement med HttpClient som beskrevet ovenfor i ”Databaselaget” sektionen. Udover dette har de alle tre linjer i starten af using-statementet hvor de først sætter ”base” adressen af klienten (altså hvor vores web service ligger), fjerne alle headers, og tilføjer en ny header til det HTTP kald vi skal lave. Base adressen er den samme for alle metoderne, så vi har lavet en public static string i en metode vi har kaldt _GLOBALS.cs som så er tilgængelig for alle klasser. Vi er klar over at den her form for globale definitioner ikke er anbefalet da en global statisk variabel af en data type går ”imod” idéen af objektorienteret programmering, men nu skal vi kun ændre én variabel hvis vores web service skifter adresse. Med mere tid ville det have været smart at kunne hente den adresse fra en tekstfil i programmet (da den så kunne ændres selv efter programmet bliver kompileret) men det fik vi ikke gjort.
Da FetchSingle() og FetchMultiple() er næsten ens ville det være muligt at samle FetchSingle() og FetchMultiple() i én metode hvis definition måske ville hedde Fetch(string table, int id = 0). Hvis man ikke kalder metoden med en ID ville den så være 0, ellers ville den blive overskredet. I starten af metoden kunne

man skabe en tom string ved navn *path*. I delen af metoden hvor databasen bliver kaldt til ville man så have et if-statement som checkede om id var ikke-lig 0 og satte værdien af *path* til ”api/{table}/{id}”, ellers til ”api/{table}”. Derefter kunne GetAsync kaldes som normalt. I delen af metoden hvor en værdi skal returneres skulle man så igen have et check om id er 0 eller ej for at beslutte om resultatet skal være ”rent” eller om der skal kaldes ToList() på det.

Databasen

Vores database er hosted i ”skyen” hos Microsoft's Azure service. Den består af fire tabeller:

- Users
- Courses
- EducationSites
- Rooms

Vores Users tabel indeholder al relevant information om vores brugere. Vi har valgt at smide administratorer, undervisere, og normale brugere under den samme ”Bruger” type og så bare give dem ekstra adgang via boolean (som er repræsenteret i databasen ved bit typen) IsTutor.
En Bruger beskrives i Users tabellen ved: Id, Username, Password, Name, PhoneNr, Address, IsTutor, Field, Email. Med undtagelse for Id, som er en int, og IsTutor, som er en bit, er alle kolonner her nvarchar med en længde på 64. Primary key i Users-tabellen er Id, som også automatisk bliver assigned og incrementer selv med 1 per nye række. Nvarchar er ligesom char en type som accepterer regulær tekst, men den har en variabel længde (”var”) med *maks* den specificerede længde (64) i stedet for at fylde resten af pladsen ud med whitespace. Præfikset ”N” burde betyde at den tillader mange flere speciale tegn, hvilket er godt i en database som gerne skal kunne indeholde danske tegn, men vi fik ikke konfirmeret dette.
Vores Courses tabel indeholder de kurser som bliver oprettet i systemet. Et Kursus beskrives i Courses tabellen ved: Id, Data, Peroid, PriceText, AvailSpots, SeminarDesc, ExtraMat, EduSiteName, IdRoom, Tutors, CourseDesc, Prereqs, Compendium. Id, Period, AvailSpots, og IdRoom er alle int typer. Date er en date type. Alle andre kolonner er nvarchar typer. Primary key i Courses-tabellen er Id, som bliver automatisk assigned og incrementet ligesom i Users tabellen. EduSiteName er en foreign key som refererer til EducationSites-tabellens primary key Name. IdRoom er en foreign key som refererer til Rooms-tabellens primary key Id.
Vores EducationSites tabel indeholder de undervisningssteder som er tilgængelige til at leje for undervisere. Det beskrives ved: Name, PhoneNr, Address, Email. Name er en primary key. Alle kolonner er af typen nvarchar (og med en maksimal længde på 64 karakterer).
Vores Rooms tabel indeholder de værelser som eksisterer på hvert undervisningssted. Det beskrives ved: Id, EduSiteName, RoomName. Id er en int, er primary key, og bliver automatisk skabt på samme måde som i Users og Courses. EduSiteName og RoomName er nvarchar typer med maksimal længde på 64.

EduSiteName er en foreign key og refererer til EducationSites-tabellens primary key Name. Rooms eksisterer kun til at være en ”undertabel” til EducationSites da dens data kun er relevant til undervisningssteder.

Web service

Vi kører et ASP web service lokalt langs vores primære applikationsprojekt. Der er ikke noget særligt ved det da det automatisk er blevet genereret af Visual Studio ved at skabe et ASP .NET Web Application, skabe et ADO.NET Entity Data Model fil i *base directory* af dette projekt (”TouchPointDBContext”), og sætte et scaffolded Web API Controllers med actions ved Entity Framework ind i dette projekts Controllers folder – alt standard efter vores uddannelse og lektioner om entity frameworks og (RESTful) web services.

Også efter vores læring har vi sat følgende ind i TouchPointDBContext:

```
base.Configuration.LazyLoadingEnabled = false;
base.Configuration.ProxyCreationEnabled = false;
```

Det er ikke helt klart for os hvad dette skal gøre ud over at det burde forhindre servicet fra at være for ”smart” om at loade.

Ud over de to linjer er vores web service absolut standard.

Yderligere dokumenter

Alle vores udarbejdede artifacts, use cases og lignende er sendt med som bilag.

Refleksion og konklusion

Vi har prøvet at følge Unified process ideologien, bedst muligt. Vi må dog erkende at med vores begrænsede erfaring, blev processen lidt en bagtanke til sidst, da vi synes vi var lidt bagud i forhold til de planlagte iterationer. Vi fortsatte med at vise vores demo'er til kunden, og rette til efter kundens ønsker.

Unified process og mvvm arkitektur, virkede ganske udmærket for et projekt i denne størrelse, det hjalp os med at fokusere på de vigtige og kritiske opgaver først, hvilke har tilladt os at reevaluere hvor meget vi kunne nå at lave i tiden givet til os. I dette projekt havde vi planlagt 3 iterationer og kunne halvvejs igennem se at vi kun havde tid til 2.

Da vi i UP laver de vigtige og kritiske opgaver først, og ikke bruger tid og energi på design og kode som ikke er med i den igangværende iteration tillader det os at fokusere der hvor der er brug for det. Havde vi brugt waterfall for eksempel, ville vi havde lavet alt design, for så at gå videre til konstruktion og se at vi kun har tid til halvdelen og derfor har spildt tid på design vi aldrig kommer til at lave.

Udover at kunne spilde tid på design der ikke ville blive implementeret så er der også høj risiko for at ens design ikke er optimalt hvis man laver det hele og låser det fast og så følger det slavisk efterfølgende.

Vi har ikke nået at implementere alle de ting vi havde planlagt, men disse ville blive implementeret i den efterfølgende iteration som vi ville begynde på nu i stedet for at afslutte projektet.

Skulle vi starte på samme projekt igen efter dette, vil vi uden tvivl følge samme fremgangsmåde med unified proces, of mvvm. Vi dog være mere forsigtige med den måde vi designer programmet på, og bruge mere tid på at danne os it komplet overblik over hvad der skal gøre hvad. Når vi designer visse aspekter af programmet, vil vi være mere realistiske i vores tilgang til hvad der er inden for vores evner og ikke prøve at være for smarte.

Alt i alt, har vi lært en masse vigtige lektioner igennem dette projekt, som vi glæder os at gøre brug af i fremtidige projekter.