Web Environment Migration Application

Diplom-IT Eksamensprojekt 2012

Studerende:	Jannick Kaas Johansen, s072640
	Nicolai Emil Børger, s072650
Projektvejleder:	Bjarne Poulsen, DTU-IMM
Projekttitel:	Applikation til migrering mellem webmiljøer.
Eksamensnummer:	IMM-B.Eng-2012-19
Afleveringsdato:	21.09.2012

Danmarks Tekniske Universitet
Institut for Informatik og Matematisk Modellering
Bygning 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk
IMM-B.Eng-2012-19

Abstrakt

Applikation til migrering mellem webmiljøer, udviklet i samarbejde med Stayhard.

Formålet med projektet er at udvikle en prototypeapplikation, som kan sammenligne to udviklingsmiljøer og finde forskelle mellem dem. Den udviklede prototype er afgrænset til at omfatte sammenligning mellem tabeller og rækker i to databaser. Endvidere skal det være muligt for brugeren, gennem prototypens brugergrænseflade, at kunne vælge hvilke forskelle, som prototypen skal migrere. Selve migreringen foregår ved at prototypen dynamisk genererer SQL-kommandoer, baseret på de forskelle som brugeren har valgt og benytter disse til at opdatere den ene database.

English Abstract

Web environment migration application, developed in collaboration with Stayhard.

The purpose of the project is to develop a prototype application, capable of comparing two development environments and find the differences between them. The developed prototype is limited to focus on comparing the tables and rows in two databases. Furthermore, the user must be able to select which differences the prototype should migrate, by using the user interface. The migration itself is done by dynamically generating SQL-queries, based on the user's selections, which is used to transfer the differences from one database to the other.

Forord

Dette eksamensprojekt er det afsluttende projekt på linjen; Diplomingeniør IT på DTU (Danmarks Tekniske Universitet). Projektet udgør de sidste 20 ECTS point af uddannelsen.

Vi vil gerne rette en stor tak til vores projektvejleder, Bjarne Poulsen fra IMM ved DTU, som nar rådet og vejledt os gennem projektet. En stor tak skal også gå til den svenske virksomhed Stayhard, for at give os muligheden for at lave dette projekt i samarbejde med dem. Sidst, men ikke mindst, en stor tak til vores kontakt person hos Stayhard, Stephan Kaas Johansen, som har hjulpet med konstruktiv kritik af rapport samt produktet.

Rapport såvel som implementering er blevet delt ligeligt mellem begge studerende.

Studerende:

Jannick Kaas Johansen, studienummer s072640

_____

Nicolai Emil Børger, studienummer s072650

_____

Dato for aflevering: 21/9/2012
Dato for forsvar: 22/10/2012

Lyngby, september 2012

Indholdsfortegnelse

Kapitel 1 – Introduktion

Formålet med dette kapitel er at give en generel introduktion til projektet. Først gives en kort beskrivelse af virksomheden der samarbejdes med. Herefter vil det blive uddybet, hvad baggrunden for projektet er, og hvad der danner baggrund for motivationen. Når baggrunden og motivationen er beskrevet, vil visionen for projektet klarlægges og der vil blive gennemgået hvilke funktioner det endelige produkt vil understøtte. Der vil blive kigget nærmere på selve problemet, og opstillet en egentlig problemformulering og afgrænsning af projektet. Afsluttende beskrives hvilke udviklingsmetoder der forventes at blive brugt til projektet, samt opstillet en oversigt over hvordan resten af rapporten er opbygget.

1.1 Kort om Stayhard

Stayhard er en svensk webbutik, som fører et bredt sortiment af forskellige tøjmærker til mænd. De sælger primært deres tøj online gennem deres webbutik, men har også to fysiske butikker, en i Göteborg og en direkte tilknyttet centrallageret i Herrljunga. Virksomheden blev grundlagt i 2005 af Daniel Möller og Joakim Naumburg. Virksomheden startede ud i Joakims forældres pulterkammer på små 15 kvadratmeter, hvor den i dag har omkring 30 ansatte og har en lagerhal på ca. 4000 kvadratmeter. I løbet af de sidste 6-7 år har de arbejdet sig op til at blive én af de største svenske tøjbutikker på nettet, og har kunder i op mod 28 lande. Deres fokus ligger på nuværende tidspunkt primært på det svenske marked, men de har også en stor aktivitet i Danmark, Norge og Finland.

For mere information, se www.stayhard. dk

1.2 Stayhards Systemer

Over 90 % af Stayhards salg sker gennem deres webbutik. Det er derfor meget vigtigt at deres hjemmeside hele tiden fungerer optimalt, således at kunderne nemt og gnidningsløst kan foretage et køb når det passer dem. Stayhard opdaterer løbende deres webbutik med ny funktionalitet og rettelser, for at give kunderne den bedste oplevelse.

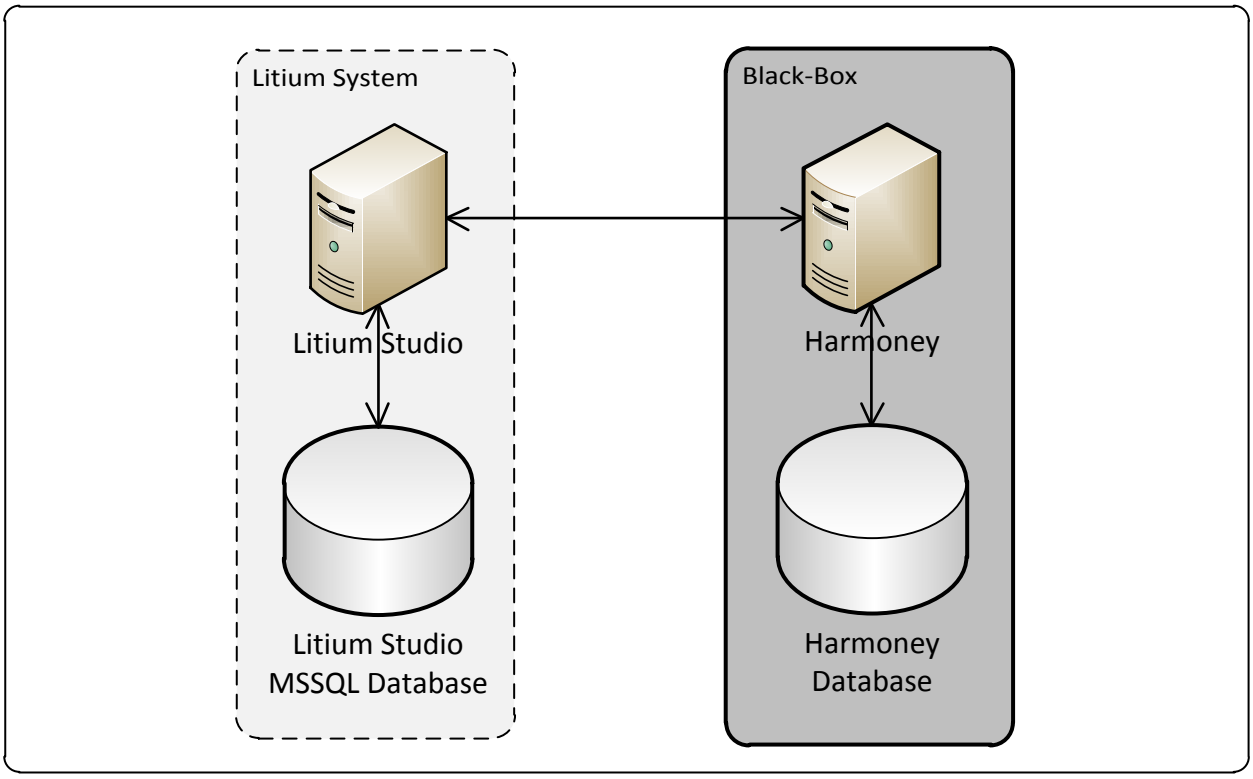
Stayhards hjemmeside bygger på systemet Litium Studio. Litium Studio er en e-handels platform udviklet af det svenske firma Litium, som indeholder de basale komponenter man har brug for i en webbutik:

- Content Management System(CMS)-del til at styre hjemmesidens indhold og udseende,
- ordre- og betalingshåndterings-del
- basalt produkt- og lagerstyringssystem
- kundekartotek
- mediabank (til at håndtere video og billeder)

Litium Studio kan også holde styr på produkter i flere forskellige sprog, samt priser i flere forskellige valuta, hvilket er én af årsagerne til at Stayhards valg faldt på Litium Studio.

Litium Studio bruges i dag af rigtig mange webbutikker (primært svenske). Blandt Litiums største kunder finder man Stayhard, Gudrun Sjördén og CDON.com. Stayhards nuværende hjemmeside bygger på Litium Studio (4.6.1) der blev lanceret i juni 2012.

Selv om Litium Studio har indbyggede funktioner til at styre lager og priser, bruger Stayhard ikke disse elementer. Derimod benytter de et Warehouse Management System(WMS), som hedder Harmoney (udviklet af det svenske firma Pulsen[1]), til at holde styr på lager, sortiment og priser. Alle disse oplysninger bliver siden eksporteret fra Harmoney, og importeret ind i Litium Studio, hvilket betyder at alle ændringer relaterede til produkt (priser, lagerbeholdning osv.) altid bliver lavet i Harmoney. For at muliggøre dette, er der lavet en tæt integration mellem de to systemer.



Figur 1 - Stayhards System-opbygning

Da dette projekt vil rette fokus mod databasen til Litium Studio, betragtes Harmoney-systemet som et Black-Box system, se Figur 1.

Kombinationen af disse to systemer bliver brugt af flere store webshops i Sverige. Se for eksempel www.GudrunSjoden.com[(2)(3)]

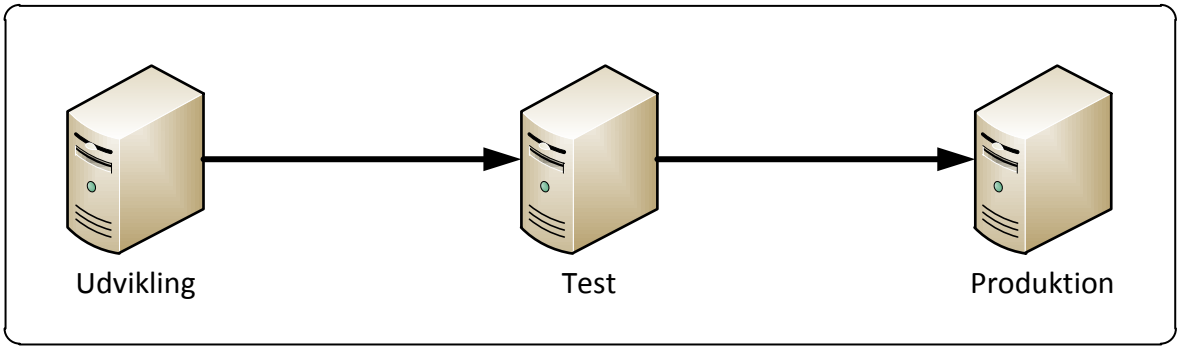
[1] http://www.pulsen.se/varatjanster/distanshandel/tjanster/affarssystemharmoney.html
[2] http://www.litium.se/nyheter/pressmeddelanden/gudrun-sjoden
[3] http://www.mynewsdesk.com/se/pressroom/litium/pressrelease/view/stayhard-se-lanserar-ny-e-handel-med-litium-756298

1.3 Baggrund og Motivation

Med en stabil vækst i omsætning og kundetilstrømning samt et ønske om højere kvalitet og nye funktioner, har Stayhard fundet det nødvendigt at ændre deres IT-strategi. De har siden starten i 2005 benyttet sig af eksterne konsulenter til at vedligeholde og opdatere deres webbutik. I løbet af 2011, i forbindelse med overvejelser om en ny webbutik, følte Stayhard ikke længere at konsulenter var en holdbar løsning, primært ud fra en økonomisk vinkel, samt et ønske om højere fleksibilitet. De har derfor valgt at oprette deres egen web-udviklingsafdeling, og vil i stigende grad skifte over til interne udviklere til at vedligeholde og udvikle webbutikken i tæt samarbejde med Litium.

Hos Stayhard har man valgt et setup som består af tre miljøer(se Figur 2), et udviklingsmiljø som ligger lokalt på udviklerens computer, et testmiljø(der i så høj grad som muligt ligner produktionsmiljøet) som alle hos Stayhard har adgang til og et produktionsmiljø som er tilgængeligt på internettet, det miljø som kunderne bruger til at handle på. Dette setup er valgt for at sikre så høj en kvalitet og så lille en fejlmargin, som muligt.

Da Stayhards udviklere begyndte at arbejde på den nye webbutik i Litium Studio, indså de en begrænsning i Litium Studio, som betyder at indholdsændringer (altså ændringer som gemmes i Litium Studios database og som ikke er en del af webprojektet) lavet i udviklings- eller testmiljøet er meget besværlige at flytte videre til henholdsvis test- og produktionsmiljøet. Dette er ikke særligt hensigtsmæssigt, og medfører en høj risiko for fejl i webbutikken eller i værste fald nedetid, hvis der skulle gå noget galt. Formålet med udviklingsmiljøet er at udviklerne kunne lave ændringer til systemet, uden at der var risiko for at det påvirkede produktionsmiljøet. Når ændringer var lavet færdig i udviklingsmiljøet, bliver de flyttet til testmiljøet. Formålet med testmiljøet var at sikre, at alle de nye ændringer fra udviklingsmiljøet virker som planlagt. Når alle test er gennemført succesfuldt, blev ændringerne migreret til produktionsmiljøet.



Figur 2 - Stayhards miljøer

De adskilte miljøer giver flere fordele, så som at reducere nede tiden for produktionsmiljøet samt sikre at nye ændringer/funktioner virker som det er intentionen, når de kommer på produktionsmiljøet. Der er dog også ulemper ved at splitte udviklingen ud på flere miljøer. En af disse ulemper er, at Litium Studio ikke er lavet med udvikling til flere miljøer i tankerne. Dette bliver tydeligt, når man skal migrere ændringer mellem miljøerne.

Ændringer som er foretaget i kode er lette at flytte fra det ene miljø over til det andet. Litium Studio er dog designet således, at mange ændringer foretaget i Litium Studio sker direkte i systemets database. Ønsker man at flytte disse ændringer fra et miljø til et andet, så er der to muligheder. Den ene er at flytte hele

databasen, og den anden er at oprette alle ændringer manuelt igen (med stor risiko for at glemme noget når flere udviklere sidder og arbejder på hver sin ende af systemet).

Når man kører med flere miljøer, er det sjældent hensigtsmæssigt blot at overskrive produktionsmiljøets database, da den ofte indeholder data som er nyere end dem man erstatter med. Det kan for eksempel være ændringer i priser, varer, ordre el. lign.

Når man skal migrere mellem miljøer, er man i dag derfor nødt til at flytte ændringerne manuelt. Det betyder, at jo mere der er ændret mellem miljøerne, jo større er risikoen for at der opstår fejl i forbindelse med migreringsprocessen. Dette er grundet at man kan komme til at glemme at migrere nogle vigtige ændringer. En risiko der ville være betydeligt mindre, hvis processen var automatiseret.

1.4 Vision

Med fokus på problemet med at migrere data (oftest gemt i en database eller i filer) mellem to miljøer, som vi beskrev i forrige afsnit, vil vi udvikle en prototype af en applikation, der kan løse problemet.

Applikationen skal kunne sammenligne databaser fra to Litium Studio miljøer, og lave en visuel fremstilling af alle deviationer mellem de to. Det skal herefter være muligt at kunne vælge præcis hvilke ændringer man ønsker at migrere fra det ene miljø til det andet.

1.5 Problemformulering

For at løse den udfordring med datamigrering, som Stayhards udviklere sidder med til dagligt, er det tanken at udvikle en prototypeapplikation, som kan bruges af udviklerne til hurtigt at skabe sig et overblik over de ændringer der er lavet i systemet. Det skal være muligt at kunne se de enkelte forskelle mellem miljøerne, og vælge hvilke ændringer der skal migreres fra det ene til andet. Hensigten med applikationen er, at eliminere risikoen for menneskelige fejl under migrering, samt at gøre hele processen simplere ved at automatisere dele af den. Det er hensigten at applikationen skal gøre det mere overskueligt at migrere mellem to miljøer.

Da formålet med applikationen er at lette udviklernes arbejde, vil der blive lagt vægt på, at brugergrænsefladen skal være intuitivt opbygget, således at den er hurtigt og let at anvende. Dette kan gøres ved at dele applikationen op i tre fokusområder, hvor det første er funktionen som sammenligner de to databaser. Andet område er funktionen som kopierer/migrerer ændringer mellem databaser mens det sidste fokusområde er Graphical User Interface (GUI)-delen.

Funktionen til sammenligning skal, som navnet antyder, bruges til at sammenligne to databaser, og finde eventuelle forskelle mellem disse. Sammenligningen skal være envejs, hvilket betyder at der skal være en kildedatabase (SourceDB) og en måldatabase (TargetDB). Dette kan både være nye eller slettede data, samt rækker, hvor data er ændret. Det skal være muligt at vælge, hvilke tabeller i databasen, som skal sammenlignes. Følgende parametre har Stayhards udviklere givet udtryk for, at de gerne så sammenlignet:

- **Tabeller** - findes de samme tabeller, og indeholder de samme kolonner og indstillinger(for eksempel primærnøgler, fremmednøgler, datatyper)
- **Stored procedures og funktioner** - er der de samme stored procedures og funktioner til rådighed, og er de lavet ens.
- **Indhold af tabeller** - er der ændringer, slettede/tilføjede data, osv.

Funktionen til kopiering og migrering opretter, sletter og ændrer i mål-databasen på baggrund af de kriterier, som er defineret af brugeren i GUI-delen af applikationen. Dette er en kritisk del af applikationen, da der potentielt laves ændringer i produktionsdatabasen, hvor fejl kan føre til at webbutikken ikke længere fungerer, og dette kan blive meget dyrt i tabt indtjening.

Applikationen skal designes sådan, at det er lettere på sigt, at udvide fra grundfunktionerne, sammenlignings-funktionen og kopierings-/migrerings-funktionen, således at der også kan udbygges til at understøtte andre datakilder, for eksempel konfigurationsfiler og billedfiler.

GUI-delen skal kunne præsentere brugeren for en stor mængde data på en hensigtsmæssig måde. Den skal derfor tænkes grundigt igennem, således at den overholder de forskellige usability-principper og standarder.

Grundet projektets begrænsede tidshorisont vil der blive fokuseret på at sammenlige miljøernes databaser, samt at udvikle en GUI der præsenterer forskellene mellem databaserne på en hensigtsmæssig måde.

For at opsummere, vil dette projekt fokusere på følgende problemstillinger:

1. Hvilke data skal prototypen præsentere for brugeren.
2. Hvordan kan der opbygges en simpel og intuitiv brugergrænseflade, som overholder standarden ISO 9241(se side 20).
3. Hvilke metoder kan man bruge til at sammenligne og kopiere/migrere databaser, og hvilken er mest hensigtsmæssig for dette projekt.
4. Der skal udarbejdes en kravspecifikation for prototypen, som tager udgangspunkt i ovenstående punkter.
5. Design af brugergrænsefladen.
6. Bestem hvordan prototypens opbygning skal designes, ved at bruge designarkitektur.
7. Funktionerne i prototypen skal udvikler således at de opfylder de opstillede krav.
8. Der skal laves unit-tests til prototypens kerne-funktioner.

1.6 Udviklingsmetoder

I dette projekt vil udviklingsprocessen Agile Unified Process (AUP) blive brugt. AUP er en af flere agile systemudviklingsmetoder, som alle er kendetegnet ved at være iterative og evolutionære. I modsætning til vandfalds-modellen og sekventiel systemudvikling, kendetegnes Agil systemudvikling ved at man bryder programmet der skal udvikles op i mindre dele. De mindre dele gør, at man meget tidligere i forløbet kan påbegynde udvikling og test af dele af systemet. Udviklingen og testen gentager man i cyklusser, indtil det opfylder de forudbestemte krav til programmet.

Normalvis benyttes udviklingsmetoden Test-Driven Development (TDD), når man følger AUP. Der er dog nogle opgaver, hvor TDD ikke er egnet. Det er for eksempel opgaver, hvor data skal hentes ind fra en ekstern kilde, så som en database. Grunden hertil er, at det er en enorm opgave at skulle lave en grundig test, som bearbejder de, som oftest, store mængder data med dynamisk indhold.

I stedet for TDD, er det i dette projekt valgt at adaptere dele af programmerings-filosofien fra en anden agil udviklingsmetode, Extreme Programming. De dele vi har adapteret, er:

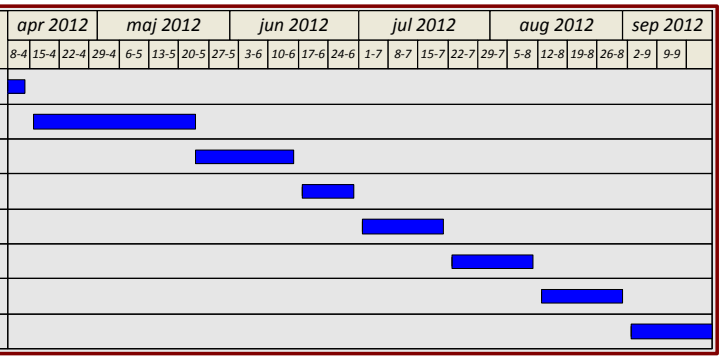
- **Parprogrammering** – Når man sidder flere sammen og koder, opdager man hurtigere eventuelle fejl, og man kan supplere hinanden, hvis man sidder fast.
- **Gem optimeringer til sidst** – Hovedmålet er at få noget kode som fungerer. Eventuelle optimeringer kommer i anden række.
- **'Kunden' er altid tilgængelig** – Når man har nogle kritiske spørgsmål, hvor man er nødt til at rådføre sig hos kunden, før man kan fortsætte, er det vigtigt at man kan få fat på 'kunden'.

Udviklingsforløbet for prototypen kommer til at foregå således, at der ud fra de nødvendige analyse og design-overvejelser, vil blive fastslået hvilke funktioner der skal implementeres i prototypen. Prototypen vil herefter blive splittet op i mindre dele, baseret på de nødvendige funktioner. Hver del vil efterfølgende blive designet og implementeret. Afslutningsvis vil den samlede prototype, samt dens hovedfunktioner blive testet.

1.7 Tidsplan

Til at sikre, at både rapport og programmering får tilstrækkeligt med opmærksomhed, vil projektet følge en tidsplan, se Figur 3:

ID	Task Name	Start	Finish	Duration
1	Opstart på Projektet	10-04-2012	13-04-2012	4d
2	Dokumentation	16-04-2012	23-05-2012	28d
3	Iteration 1	24-05-2012	15-06-2012	17d
4	Ferie	18-06-2012	29-06-2012	10d
5	Iteration 2	02-07-2012	20-07-2012	15d
6	Iteration 3	23-07-2012	10-08-2012	15d
7	Iteration 4	13-08-2012	31-08-2012	15d
8	Færdiggør Rapport	03-09-2012	21-09-2012	15d



Figur 3 - Tidsplan

Tidsplanen er opdelt i flere forskellige grupper, som dækker over følgende:

- ”Opstart på Projektet” dækker over et besøg hos virksomheden i Sverige, hvor detaljerne om projektet fastlægges.
- Dokumentation (rapporten) påbegyndes. Første kapitel, indledningen, skrives færdig i denne periode.
- En Iteration er, i dette projekt, et todelt forløb, hvor der først bliver brugt tid på at udfærdige kode, hvorefter der bliver arbejdet med rapporten på baggrund af dette.
- Den sidste del af projektet kommer udelukkende til at handle om at færdiggøre rapporten.

1.8 Overblik over rapportens kapitler

Resten af rapporten er organiseret i følgende kapitler og omhandler de beskrevne emner:

Kapitel 2 – Analyse

I dette kapitel vil projektets problemstilling blive analyseret, således at der dannes et solidt grundlag for design-fasen af projektet. Dette gøres ved brug af use-case-modeller.

Der vil blive analyseret på hvordan applikationen kan løse problemet med at sammenligne data fra to miljøer, samt opstille hvilke forskellige måder dette kan gøres på. Endvidere vil nogle af prototypens mere komplekse funktioner blive analyseret grundigt.

For at sørge for at prototypens brugergrænseflade bliver opbygget på en hensigtsmæssig måde, vil der i dette kapitel blive lavet nogle mock-ups til at guide udviklingen af brugergrænsefladen. Der vil blive undersøgt hvordan GUI-delen kommer til at opfylde de væsentligste usability-principper og standarder.

I dette kapitel vil punkterne 1, 2, 3 og 4 fra problemformuleringen blive belyst.

Kapitel 3 – Design

Design-kapitlet vil indeholde klassediagrammer og sekvensdiagrammer, som er lavet på baggrund på usecases lavet i analysen, og som vil give et overblik over hvordan prototypens opbygning og funktioner skal implementeres. De mest grundlæggende funktioner beskrives i dybden, så som indlæsning af data, hvordan der sammenlignes og migreres data mellem databaser.

Baseret på analysen, vil den mest optimale og hensigtsmæssige løsning udvælges, og ud fra denne løsning vil en egnet arkitektur udvælges.

Baseret på mock-ups og usability-analyse fastlægges det endelige design af brugergrænsefladen.

I dette kapitel vil punkterne 5 og 6 fra problemformuleringen blive belyst.

Kapitel 4 – Implementering og test

Dette kapitel vil overordnet beskrive hvorledes koden til prototypen er blevet udfærdiget, og vil gå i dybden med udvalgte dele af koden.

Kapitlet vil desuden beskrive, hvordan prototypens implementering er blevet testet.

I dette kapitel vil punkterne 7 og 8 fra problemformuleringen blive belyst.

Kapitel 5 – Konklusion

Konklusionen vil opsummere del-konklusionerne fra de andre kapitler, og lave en samlet konklusion for hele projektet.

Der vil som afslutning blive kigget på fremtidsperspektiverne for prototypen.

Kapitel 2 – Analyse

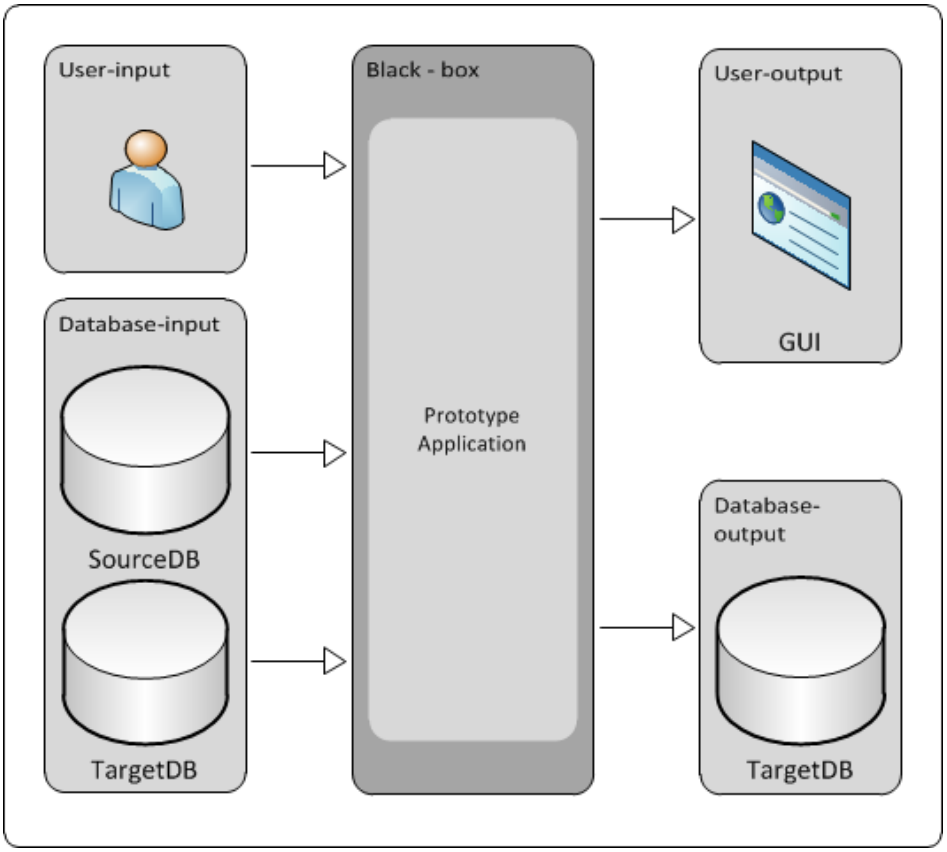
Formålet med dette kapitel er at analysere projektet og opstille, samt udvælge projektets problemstillinger. Indledningsvis vil der blive analyseret, hvilke input og output som prototypen har. Derudover vil der blive opstillet en kravspecifikation, hvor samtlige krav til prototypen vil blive defineret og beskrevet. Desuden vil projektet blive afgrænset til en passende delmængde af de opstillede krav. Denne afgrænsning vil blive lavet i slutningen af kravspecifikationen, og vil være baseret på hvad der er realistisk at kunne få med i dette projekt. Brugen af prototypen vil blive beskrevet gennem use-cases, samt tilhørende sekvensdiagrammer.

Der vil også blive undersøgt, hvorledes den valgte usability-standard kan benyttes i prototypens GUI, samt hvilke problematikker, man skal være opmærksom på, når man skal lave et program med GUI.

Efterfølgende vil de vigtigste problematikker vedrørende prototypens kernefunktioner blive belyst. De mest relevante teknologier vil blive beskrevet, og der vil blive forklaret, hvorfor netop disse teknologier er blevet valgt.

2.1 Analyse af komponentdata

Nedenstående figur giver et overblik over hvilke input prototypen tager, samt hvad prototypen har som output.



Figur 4 - Overblik over Input og Output

Som man kan se på Figur 4, er inputtet til prototypen delt op i to forskellige undergrupper; database-input og user-input. Dette er gjort for at gøre det nemmere at kunne skelne mellem, hvor de enkelte input kommer fra.

2.1.1 Database-input

Database-input dækker over alle de informationer, som prototypen henter ud fra de to valgte databaser.

Når prototypen starter, vælges der to databaser, Source Database(SourceDB) og Target Database(TargetDB), som skal sammenlignes. For at sammenligningen mellem de to databaser skal give mening, skal det være to versioner af den samme database. Det kunne for eksempel være hvis man sammenlignede en database fra et testmiljø, med en database i produktionsmiljøet. Man kan sagtens sammenligne to vilkårlige databaser, men resultatet vil blot blive, at source-databasen vil blive kopieret over i target-databasen.

Prototypen skal sammenlige så stor en del af databaserne, som muligt. Dette betyder at når prototypen skal sammenligne de to databaser, skal følgende data hentes ud fra begge databaser:

- **Tabeller** – Tabel-navne, kolonne-navne og datatyper skal hentes ud for hver tabel der skal sammenlignes. Endvidere skal der hentes oplysninger omkring tabellernes primær- og fremmed-nøgler.
- **Rækker** – Al indhold i de enkelte rækker skal hentes ud.
- **Stored Procedures** – Både navne og funktioner på samtlige af databasernes stored procedures skal hentes ud.
- **Views** - Navne, indstillinger og indhold skal hentes ud for samtlige views databaserne indeholder.
- **Generelle Database-indstillinger** – Databasernes indstillinger skal hentes ud fra begge databaser.

Hvis samtlige dele af en database skal sammenlignes med en anden database, vil det hurtigt blive en meget omfattende opgave. Der bliver derfor sandsynligvis valgt nogle dele, som der bliver fokuseret på, mens resten vil blive en del af en fremtidig udvidelse. Derfor bliver de forskellige dele prioriteret med ”must have” og ”nice to have”.

- ”Must have” er de dele, som anses for at være systemkritiske, og som er nødvendige for at prototypen kan opfylde den ønskede funktion.
- ”Nice to have” er de dele, som ikke er essentielle for prototypens kernefunktioner.

I samarbejde med Stayhard, er de forskellige dele blevet prioriteret således, at tabeller og rækker prioriteres som ”must have”, mens de resterende dele prioriteres som ”nice to have”. Grunden til denne prioritering er, at tabeller og rækker dækker over databasens grundlæggende data-indhold, mens de resterende dele bliver anskuet som ekstra funktionalitet i en database.

2.1.2 User-input

User-input dækker over hvordan brugeren interagerer med prototypen.

Brugeren starter processen, ved at vælge hvilke to databaser der skal sammenlignes. Dette gøres ved at fortælle destinationen(URL) på databaserne, samt brugernavn og kodeord for at tilgå dem. Brugeren skal herefter bestemme om det skal være en 'Full' eller 'Partial' sammenligning der skal laves på de to databaser. De to sammenligningstyper dækker over følgende:

- **Full** – Hvis denne vælges påbegynder prototypen en fuldstændig sammenligning af databaserne, uden at brugeren skal gøre andet. Når sammenligningen er færdig præsenteres brugeren for resultatet.
- **Partial** – Hvis denne vælges, foretages der først en sammenligning på databasernes tabeller, hvorefter brugen skal vælge hvilke tabeller der skal sammenlignes. Som med 'Full' præsenteres brugeren for resultatet når sammenligningen er færdig.

Når sammenligningen er færdig og brugeren er blevet præsenteret for resultatet, kan der vælges hvilke data der skal migreres. Dette gøres ved at markerer ud for hver ændring om denne skal migreres eller ej. Efter ønskede ændringer er blevet valgt trykkes der på en knap, og prototypen fortsætter til næste del af processen. I sidste ende vælger man om man selv vil køre SQL-Kommandoerne, eller om programmet automatisk skal lave migreringen.

2.1.3 Database-output

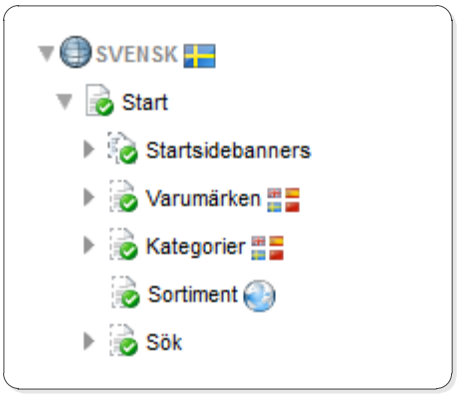
Database-output dækker over de SQL-Kommandoer, som bliver genereret af prototypen.

Når brugeren har valgt hvilke ændringer der skal migreres, genererer prototypen en række SQL-kommandoer, som bruges til at opdatere TargetDB med de valgte ændringer. Dette kunne for eksempel være nye rækker i en tabel, eller ændringer til tabelindstillinger, så som ændrede kolonnenavne eller datatyper.

2.1.4 User-output

User-output dækker over hvordan prototypens GUI kommunikerer med brugeren.

Efter de to databaser er blevet sammenlignet, vises resultatet for brugeren. Resultatet bliver opstillet i en træstruktur, som minder om den træstruktur, der er i Litium Studio(se Figur 5).



Figur 5 – Træstruktur brugt i Litium Studio

Prototypen viser hvilke elementer, der er forskellige i de to databaser, og skelner mellem følgende forskelle:

- **Nyt element** – Et element, som er i SourceDB, men ikke i TargetDB.
- **Manglende element** – Et element, som er i TargetDB, men ikke i SourceDB.
- **Indholdsændring** – Et element, som findes i både SourceDB og TargetDB, men hvor indholdet er forskelligt i de to databaser.

Efter brugeren har valgt de ændringer som han ønsker at overfører, genererer prototypen en række SQL-kommandoer, som kan bruges til at opdatere TargetDB med. Disse SQL- kommandoer bliver vist i en tekst-boks, som ikke kan redigeres, hvor brugeren kan vælge mellem at:

- **Redigere i kommandoer** – Tekstboksen skifter status, således at der kan redigeres i kommandoerne, og brugeren kan manuelt lave ændringer til SQL- kommandoerne.
- **Kopiere SQL-kommandoer til udklipsholderen** – SQL- kommandoerne bliver sendt til brugerens udklipsholder, og brugeren kan herefter selv bestemme hvad der skal ske med kommandoerne.
- **Lade prototypen køre kommandoerne på TargetDB** – Prototypen kører kommandoerne på TargetDB, og migreringen gennemføres.

Vælger brugeren en, eller begge, af de to første muligheder, kan brugeren selv bestemme om hvorvidt sidste valgmulighed skal benyttes. Ønskes det ikke, at prototypen skal stå for migreringen, kan prototypen lukkes på dette stadie, og brugeren skal selv stå for at køre kommandoerne på databasen. Vælger brugeren derimod, at prototypen skal køre kommandoerne, vil prototypen køre kommandoerne på TargetDB, og et nyt vindue med en tekstboks vil rapportere hvordan migreringen er forløbet. Herefter kan prototypen afsluttes.

2.2 Kravspecifikation

Her vil der blive kigget på de krav der er til projektet, dette omfatter funktionelle såvel som ikke-funktionelle krav.

2.2.1 Funktionelle krav

De funktionelle krav dækker over de krav, som bliver stillet til prototypens funktioner.

Prototypen skal opfylde følgende funktionelle krav:

FK1 • **Sammenligning af to databaser** – Det skal være muligt at vælge to databaser, som skal sammenlignes. I dette projekt er kravet, at der skal sammenlignes på de dele af databasen, som i afsnit 2.1.1 - Database-input(side10) blev vurderet som værende ”must have”. Dette betyder, at der skal sammenlignes på tabeller og rækker.

FK2 • **Visuel præsentation af forskelle mellem to databaser** – Prototypen skal kunne vise resultatet af sammenligningen af to databaser.

FK3 • **Migrering** – En vigtig funktion er at man gennem interfacet skal kunne udvælge hvilke ændringer man ønsker at migrere fra den ene database til den anden.

FK4 • **Sammenligning af miljø-variabler** – Dette omfatter eventuelle Config-filer samt Billede-filer.

FK5 • **Versionsstyring** - Således at der er en historik over hvilke ændringer der er blevet migreret.

FK6 • **Udvidet versionsstyring** - Som giver mulighed for at fjerne en migrering fra et miljø.

FK7 • **Historik over udeladte ændringer** - Mulighed for at se ændret data, som ikke er blevet migreret ved en tidligere sammenligning.

2.2.2 Ikke-funktionelle krav

De ikke-funktionelle krav dækker over de begrænsninger og betingelser, som prototypen skal overholde. For at skabe bedre overblik er det valgt at inddele afsnittet i to, navnlig Performance- og projektbegrænsninger.

Eftersom at der udvikles en prototype vil mange af nedenstående krav ikke kunne testes. Grunden til dette er at der i en prototype hovedsageligt fokuseres på de funktionelle krav, mens der er i det færdig produkt vil blive lagt ligeså meget vægt på ikke-funktionelle krav.

2.2.2.1 Performancebegrænsninger

Performancebegrænsninger er de krav der bliver stillet til et programs kørsel efter endt udvikling. Det kan for eksempel være krav til stabilitet eller programmets interface.

PFB1 • **Usability**:

Prototypens interface skal overholde ISO 9241, som er en ISO-standard som har en række krav angående usability og design. Endvidere er der opstillet et krav fra Stayhard om at prototypens interface skal være opbygget, så det minder Litium Studio. Såfremt der skulle opstå designmæssige komplikationer, hvor der skulle være konflikt mellem ISO-standarden og Stayhard's krav, vil der i dette projekt blive fokuseret på at efterkomme Stayhard's krav.

PFB2 • **Skalerbarhed**:

Stayhard har udtrykt interesse for at de gerne vil videreudvikle på prototypen, således at de får et færdigt sammenligningsværktøj. Det er derfor et krav fra Stayhard's side, at prototypen skal være nem at bygge videre på. Dette opnås ved at benytte design-patterns, samt sørge for at koden har høj binding og lav kobling.

PFB3 • **Stabilitet**:

Da prototypen potentielt skal bruges i et produktionsmiljø, er det meget vigtigt, at programmet er meget stabilt. Dette betyder at der skal være omfattende fejlhåndtering, og at fejl skal fanges og håndteres så tidligt i forløbet, som muligt.

PFB4 • **Respons- og kørsels-tid**:

Stayhard har givet udtryk for, at de anser det som en acceptabel løsning, at prototypen kan køre i løbet af natten, og har derfor ikke nogen specielle krav til prototypens respons- og kørsels-tider. Der vil derfor ikke blive lagt vægt på optimering af algoritmer i dette projekt.

2.2.2.2 Projektbegrænsninger

Projektbegrænsninger er de krav der bliver stillet til projektet, samt det miljø prototypen skal køre i. Det kan for eksempel være hvilket sprog koden skal laves i (C#, Java osv.) eller hardware-krav/begrænsninger.

PJB1 • **Database**:

Da Stayhard udelukkende bruger MSSQL-databaser, skal prototypen kun fokusere på denne type databaser. Det er derfor også nødvendigt, at prototypen kan kommunikere med en MSSQL-database.

PJB2 • **Udviklingsteknologier**:

Stayhard har ønsket, at prototypen i dette projekt, vil blive udformet i en kombination af C# og WPF. Dette ønskes, da deres udviklere primært arbejder med disse teknologier. Desuden vil der blive brugt SQL, da Stayhards databaser kører MSSQL.

PJB3 • **Software**:

Stayhard har et krav til at systemet, som skal køre prototypen, skal være et windows-baseret miljø. Grunden til dette er at de fleste af deres systemer kører på Windows-platformen.

PJB4 • **Hardware**:

Stayhard har givet udtryk for, at de ikke har nogen hardwaremæssige krav eller begrænsninger til prototypen. Den eneste hardware-begrænsning som projektet opstiller for systemet, som skal køre prototypen, er at begge databaser skal kunne tilgås lokalt, eller via Stayhards lokale netværk(LAN).

2.2.3 Projektafgrænsning

Projektafgrænsningen specificerer hvilke dele af de funktionelle krav, projektet vil fokusere på. Dette kan være grundet flere forskellige ting, så som for eksempel en tidsbegrænset projektperiode.

Projektet er blevet afgrænset således, at der bliver fokuseret på de funktionelle krav FK1, FK2 og FK3. Samtlige af de ikke-funktionelle krav vil blive inkluderet i projektet.

De funktionelle krav, der ikke vil blive fokuseret på i dette projekt, vil blive anskuet som en række mulige fremtidige udvidelser til programmet, men vil ikke blive inkluderet i dette projekt.

Det er valgt at afgrænse projektet på denne måde, da det umiddelbart vurderes at dette er passende i forhold til projektets tidsramme, som er 20 uger.

2.3 Use-Case

For at skabe bedre overblik over hvordan en bruger interagerer med prototypen, er der lavet følgende Use-Case-beskrivelser og Use-Case-diagrammer. Til trods for at der kunne laves en enkelt Use-Case til at beskrive hovedforløbet, er det valgt at opdele dette i tre dele, således at hver del beskriver et forløb som løser et at de funktionelle krav. Dette er gjort for at give et bedre overblik over, hvad de enkelte use-cases beskriver. Tilknyttet hver Use-Case-beskrivelse vil der være et sekvensdiagram som uddyber forløbet i systemet.

2.3.1 Use-Case-beskrivelser

I dette projekt, er det valgt at benytte Alistair Cockburn's skabelon til at lave fully-dressed Use-Cases. Dette er gjort for at få så mange detaljer med i de respektive Use-Cases, som muligt.

2.3.1.1 Use-Case-beskrivelse 1

Use-Case afsnit:	**Kommentar:**
Use-Case Navn	Valg af Databaser og Tabeller.
Scope	Prototype til WEMA-program
Level	Brugermål
Primær aktør	Udvikler
Interesse partner	Udvikler: Ønsker at få sammenlignet to databaser.
Prækondition	- Computeren skal kunne forbinde til de ønskede databaser.
Postkonditioner	- SourceDB og TargetDB er succesfuldt blevet sammenlignet ud fra udviklerens ønsker.
Hovedsucces scenarie	1. Udvikleren vælger to databaser der skal sammenlignes, ved at indtaste URL, brugernavn og kodeord. 2. Udvikleren vælger at der skal laves en 'Full'-sammenligning. 3. Prototypen sammenligner samtlige dele af de to databaser.
Alternativt scenarie	2-3a. Udvikleren vælger at der skal laves en 'Partial'-sammenligning 1. Prototypen præsenterer Udvikleren for en liste af tabeller hvorefter han/hun kan vælge hvilke tabeller der ønskes sammenlignet. 2. Prototypen laver en sammenligning på de valgte tabeller.
Specielle Krav	Denne Use-Case gør brug af følgende ikke-funktionelle krav: PJB1, PJB2, PJB3 og PJB4.
Hyppighed	Minimum en gang per kørsel af prototypen.
Diverse	-

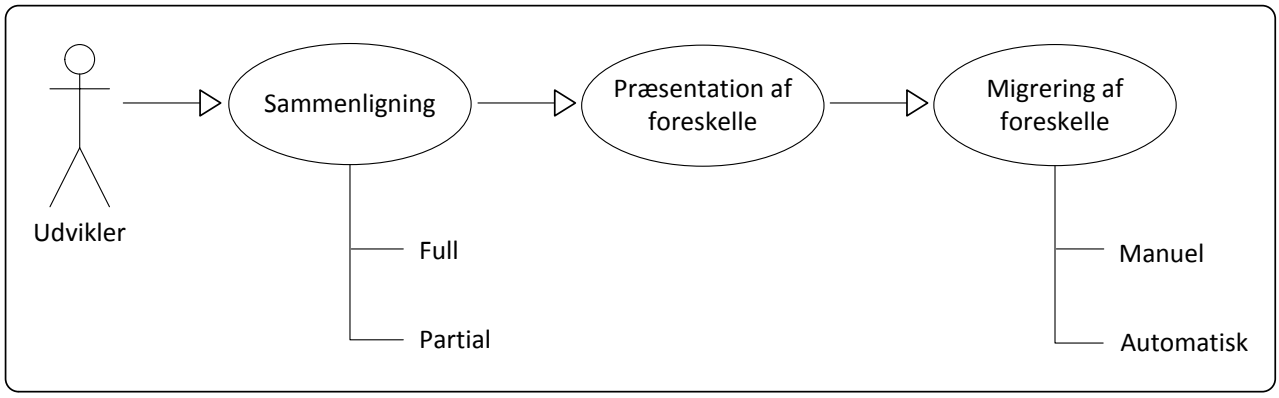
2.3.1.2 Use-Case-beskrivelse 2

Use-Case afsnit:	**Kommentar:**
Use-Case Navn	Præsentation af forskelle mellem databaser.
Scope	Prototype til WEMA-program
Level	Brugermål
Primær aktør	Udvikler
Interesse partner	Udvikler: Vil gerne præsenteres for en oversigt over hvilke forskelle der er mellem de to databaser
Prækondition	- Databaser og eventuelle tabeller skal være sammenlignet.
Postkonditioner	- Forskelle mellem de to databaser, som skal migreres, er blevet valgt.
Hovedsucces scenarie	1. Prototypen præsenterer Udvikleren for forskellene mellem de to databaser. 2. Udvikleren udvælger de ændringer han/hun ønsker at flytte fra SourceDB til TargetDB.
Alternativt scenarie	- Ingen
Specielle krav	Denne Use-Case gør brug af følgende ikke-funktionelle krav:PJB2, PJB3 og PJB4.

2.3.1.3 Use-Case-beskrivelse 3

Use-Case afsnit:	Kommentar:
Use-Case Navn	Genereret SQL samt migrering af ændringer
Scope	Prototype til WEMA-program
Level	Brugermål
Primær aktør	Udvikler
Interesse partner	Udvikler: TargetDB opdateres med de ønskede forskelle.
Prækondition	- De ønskede forskelle mellem de to databaser, som skal migreres, er blevet valgt.
Postkonditioner	- Udvikleren har fået et SQL-kommandoer som kan opdatere TargetDB med de ønskede forskelle. - Prototypen har opdateret TargetDB med de ønskede forskelle.
Hovedsucces scenarie	1. Prototypen genererer en række SQL- kommandoer, som kan bruges til at migrere de ønskede forskelle til TargetDB. 2. Prototypen præsenterer Udvikleren for SQL- kommandoerne. 3. Udvikleren vælger at prototypen skal opdatere TargetDB med de ønskede ændringer. 4. Prototypen opdaterer TargetDB med de ønskede ændringer og rapporterer om udfaldet af opdateringen til Udvikleren.
Alternativt scenarie	2a. Udvikleren kopierer SQL- kommandoer fra tekstboksen over i udklipsholden i Windows. 1. Udvikleren lukker prototypen og kan herefter arbejde videre med SQL-kommandoerne i et andet tekst-redigeringsprogram.
Specielle krav	- Simultan kørsel af prototypen af to forskellige Udviklere. Kan føre til uønskede resultater. - Denne Use-Case gør brug af følgende ikke-funktionelle krav: PJB1, PJB2, PJB3 og PJB4.

2.3.2 Use-Case-diagram



Figur 6 - Use-Case-Diagram

Det ovenstående Use-Case-Diagram giver et overblik over flowet i programmet, samt hvilke alternative scenarier flowet kan have. Da programmet har en meget specifik opgave, det skal opfylde, er flowet meget lineært.

2.3.3 Use-Case-/Funktions-matrix

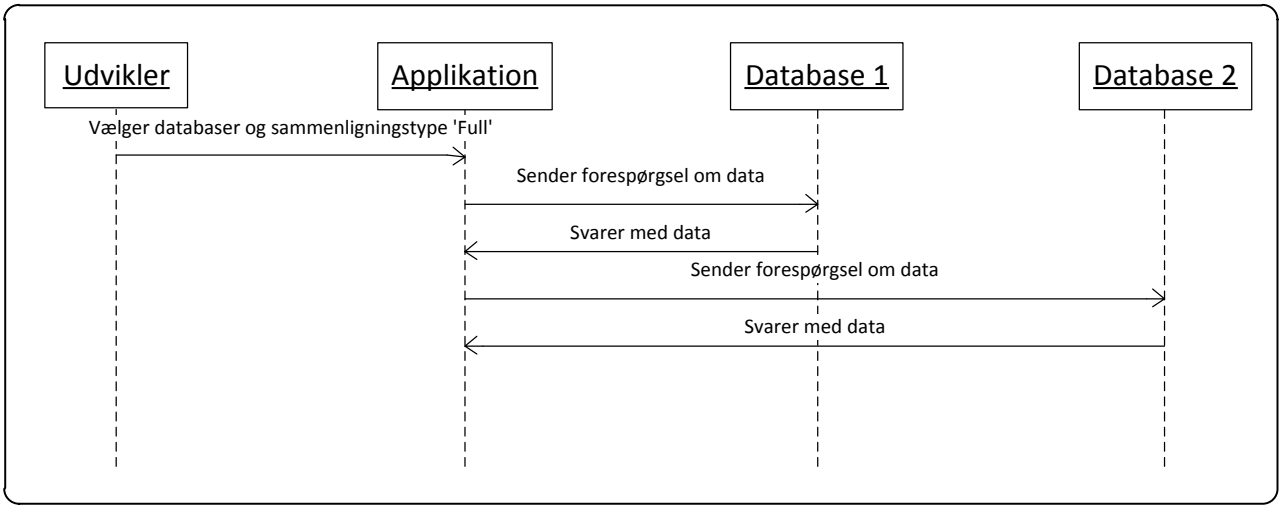
Den nedenstående tabel viser sammenhængen mellem de funktionelle krav, og punkterne i hovedsucces scenariet i Use-Case-beskrivelserne. For eksempel: UC2 S2 referer til step 2 i Use-Case 2's hovedscenarie.

Tabel 1- Use-Case / Funktions-matrix

	FK1	FK2	FK3
UC1 S1	X		
UC1 S2	X		
UC1 S3	X		
UC2 S1		X	
UC2 S2		X	
UC3 S1			X
UC3 S2		X	
UC3 S3			X
UC3 S4			X

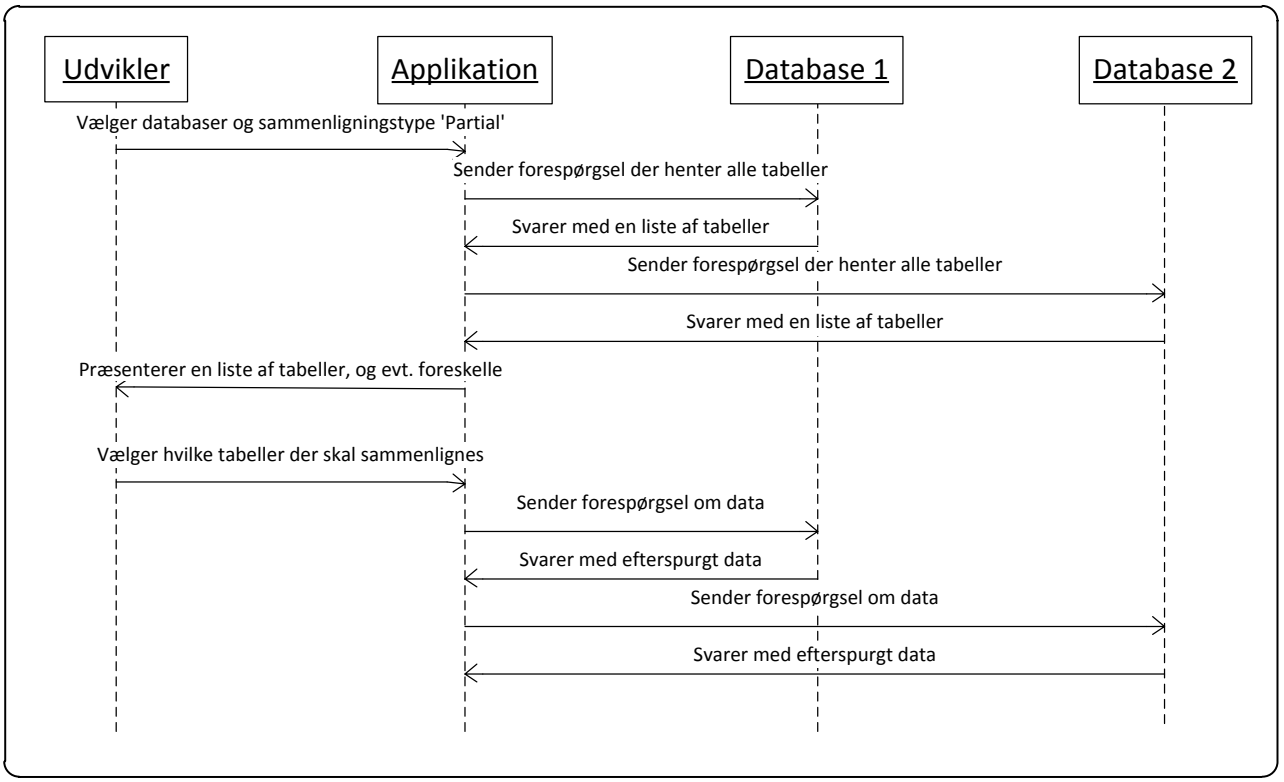
2.4 Sekvens-Diagram for Use-Case

Disse sekvensdiagrammer beskriver, hvordan de forskellige komponenter i prototypen kommunikerer. Hvert sekvensdiagram er tilknyttet en bestemt Use-Case-beskrivelse. Dette er gjort for at skabe et overblik over hvordan bruger og applikation interagerer med hinanden.



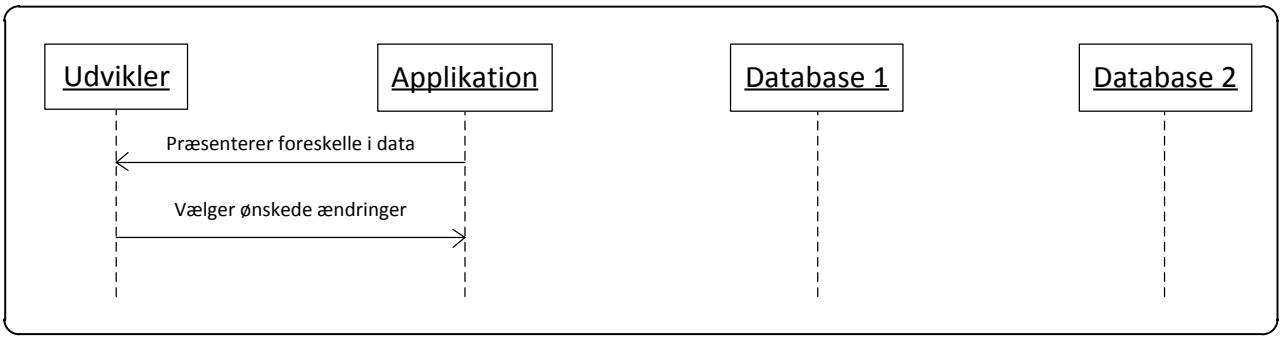
Figur 7 - Sekvensdiagram for Use-Case 1.1

Sekvensdiagrammet ovenfor (Figur 7), viser processen som brugeren "Udvikler" skal igennem for at lave en sammenligning af typen 'Full' på de to databaser. Forløbet afsluttes i applikationen, hvor der foretages en komplet sammenligning af de valgte databaser.



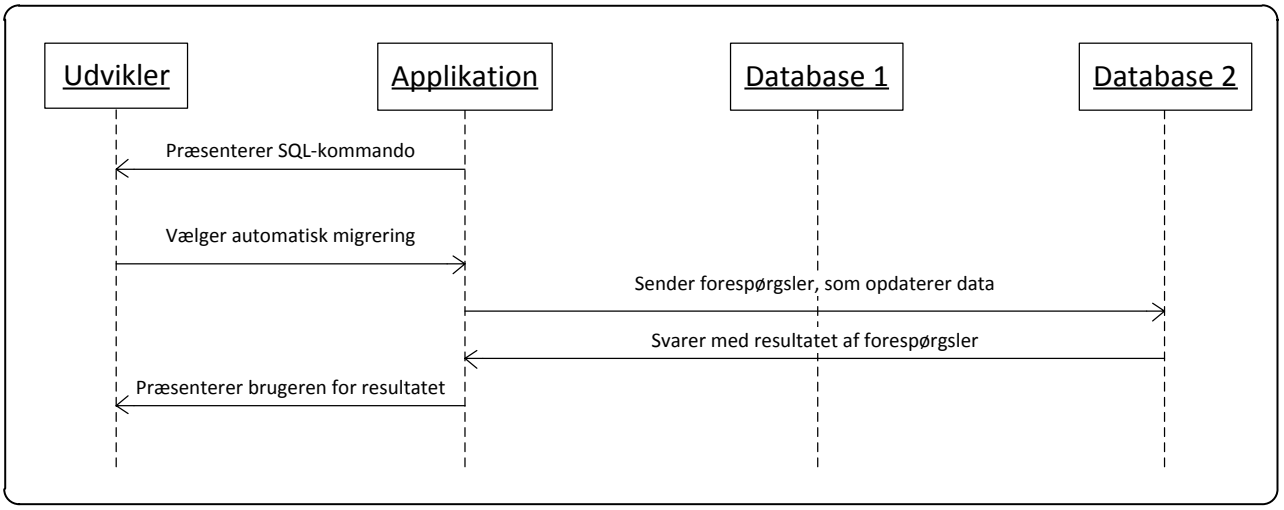
Figur 8- Sekvensdiagram for Use-Case 1.2

Sekvensdiagrammet ovenfor (Figur 8), viser det alternative forløb som brugeren ”Udvikler” skal igennem, hvis der vælges en ’Partial’ sammenligning frem for ’Full’. Denne sekvens afsluttes ved at der foretages en sammenligning af de valgte tabeller.



Figur 9- Sekvensdiagram for Use-Case 2

Sekvensdiagrammet ovenfor (Figur 9), illustrerer hvordan brugeren ”Udvikler” bliver præsenteret for eventuelle forskelle mellem de to valgte databaser. I denne sekvens er der ingen kommunikation med databaserne, da alle nødvendige data allerede er hentet ind i prototypen.

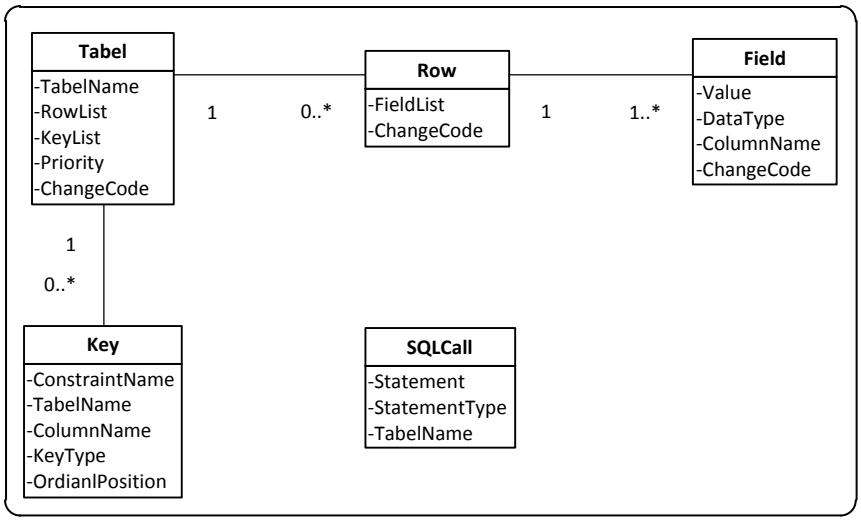


Figur 10- Sekvensdiagram for Use-Case 3

Sekvensdiagrammet ovenfor (Figur 10), illustrerer forløbet, som brugeren "Udvikler" skal igennem, i forbindelse med at prototypen skal foretage en migrering af de udvalgte ændringer. Som det fremgår af Use-Case-beskrivelse 3, er der et alternativt forløb, hvor brugeren kopierer SQL fra prototypen for at bruge den uden for prototypen. Da den eneste handlig fra systemet i denne sekvens er at præsentere brugeren for SQL-kommandoerne, er det valgt blot at beskrive dette forløb, frem for at lave et sekvensdiagram til at beskrive det.

2.5 Domænemodel

Domænemodellen viser, hvilke objekt-entiteter som prototypen skal indeholde, samt relationer mellem dem. Som det fremgår af modellen, kommer Table til at være hovedobjektet i prototypen, og vil indeholde samtlige data fra en database-tabel.



Figur 11 – Domæne model

Da prototypens funktion er at hente og bearbejde indhold fra en database, for siden at sætte det ind i en anden database, er det nødvendig at gemme database-indholdet i en struktur der er lig den der er i en database.

2.6 Usability

Usability er det, som på dansk kendes som brugervenlighed, og dækker over hvor intuitivt et objekt, som et menneske kan interagere med, er. Det handler om, hvordan en bruger opfatter og føler, i forbindelse med at han/hun interagere med et givent objekt. Det kan være alt lige fra en bil eller en skovl, til et stykke software. Det betyder i princippet, at der skal tænkes usability ind i designet for alt, som bliver skabt til menneskebrug. Usability er relevant for mange produkttyper, hvilket er grund til at der er mange faggrupper inden for videnskab repræsenteret, når der diskuteres og forskes i hvordan man opnår god usability.

Usability kan overordnet defineres ud fra følgende komponenter:

- **Indlæring** – Hvor let er det for en førstegangs-bruger at gennemføre en simpel opgave med produktet?
- **Effektivitet** – Når brugeren er blevet bekendt med produktet, hvor hurtig kan en given opgave gennemføres?
- **Uforglemmelighed** – Når brugeren vender tilbage til produktet, efter ikke at have brugt det i en periode, hvor lang tid tager det så at blive bekendte med produktet igen?
- **Fejl** – Hvor mange fejl laver en bruger, hvor alvorlige er disse fejl, og hvor lang tid tager det at komme ovenpå efterfølgende?
- **Tilfredsstillelse** – Hvor behageligt er det at arbejde med produktet?

Det er svært at tale om usability uden at komme ind på design, da begge dele er noget man tænker ind i udviklingen af et produkt. Sammenspillet mellem usability og design er som oftest et trade-off, hvor det ene kommer på bekostning af det andet. Set i henhold til hjemmesider, så får for meget usability ofte en hjemmeside til at fremstå utroværdig, mistænkelig og gammeldags. Et eksempel på dette, er usability-konsulenten Jakob Nielsens hjemmeside[4]. Hvis der derimod bliver lagt for meget vægt på design, resulterer det i "støj" for brugeren, som fjerner fokus fra budskabet/hovedformålet med hjemmesiden. Det handler om at finde en gylden middelvej, mellem design og usability. Der er heller ikke noget definitivt svar på hvad der er god usability, da der er så mange faktorer der spiller ind, at det betragtes som noget subjektivt.

I forbindelse med programmer eller hjemmesider, handler det i høj grad om hvor ligetil brugergrænsefladen/GUI'et er at gå til, samt den måde data bliver fremstillet på. Til at sikre at usability bliver tænkt ind i udvikling af GUI til programmer, er der udviklet standarder til dette. Standarder kan være med til at sikre at følgende:

- **Sikre ensartethed** – Standarder hjælper med at skabe ensartethed, så man opnår et homogent interface.

[4] http://www.useit.com/

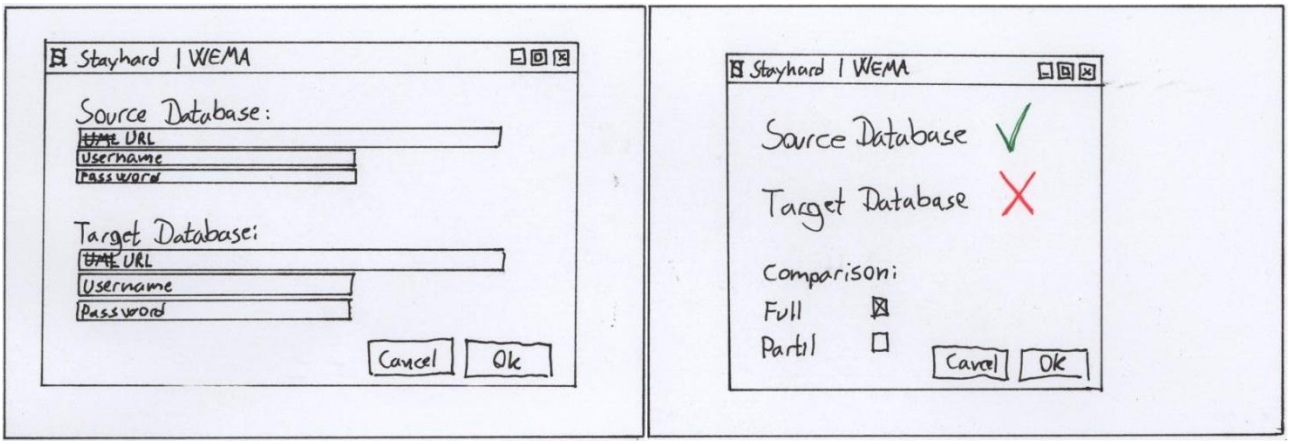
- **Definerer good practice** – Der er mange syn på hvad der er good practice, med en standard er dette fastlagt fra starten.
- **Hjælper med at opfylde juridiske forpligtigelser** – for eksempel hjælp til handikappede, ikke at det er alle der er underlagt krav om at understøtte dette, men ved at følge en standard viser det en vis føjelighed overfor berørte grupper.

Den mest populære standart til at sikre funktionalitet og brugervenlighed i software produkter er ISO 9241. ISO 9241 har også den mere sigende titel "Ergonomics of human-system interaction", som den fik i 2006, i forbindelse med at standarden blev revideret til at dække mere bredt.

Som nævnt i performancebegrænsningerne(PFB1) i kravspecifikationen, vil der i dette projekt blive lagt vægt på at prototypen kommer til opnå god usability. Dette opnås ved at følge de i ISO9241 forskrevne retningslinjer så vidt muligt, samt ved at følge Jakob Jensen's retningslinjer, alt i mens Stayhards ønsker også bliver taget med i betragtning.

2.7 Mock-Ups

På baggrund af use-case-beskrivelserne, samt afsnittet om usability, er nedenstående mock-ups blevet udfærdiget. Større billeder af de enkelte mock-ups er at finde i bilag 0.

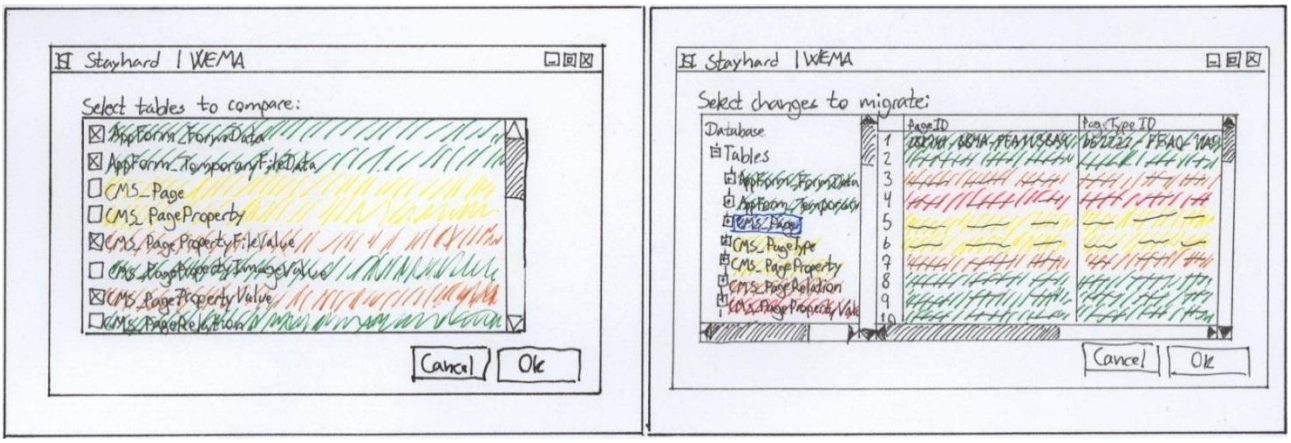


Figur 12 - Mock-up 1 & 2

Figur 12 ovenfor tager udgangspunk i use-case-beskrivelse 1, hovedsucces scenarie 1 og 2.

Mock-up 1 til venstre viser, hvordan hovedsucces scenarie 1 i use-case-beskrivelse 1, forventes at tage sig ud i prototypen. Det vil være et vindue, hvor man henholdsvis for SourceDB og TargetDB kan indtaste databasens 'URL', 'username' og 'password'.

Mock-up 2 til højre viser, hvordan hovedsucces scenarie 2 i use-case-beskrivelse 1, forventes at tage sig ud i prototypen. Dette vindue vises efter man har angivet den nødvendige information om databaserne, og trykket 'Ok' i mock-up 1. I mock-up 2 vises der om der er forbindelse til de angivne databaser, grønt 'V' betyder der er forbindelse, rødt 'X' betyder at der ikke er forbindelse. Herudover vælger brugeren hvilken type sammenligning der skal laves. Vælges 'Full' bliver brugeren efterfølgende præsenteret for mock-up 4. Vælges 'Partial' bliver brugeren efterfølgende præsenteret for mock-up 3.



Figur 13 - Mock-ups 3 & 4

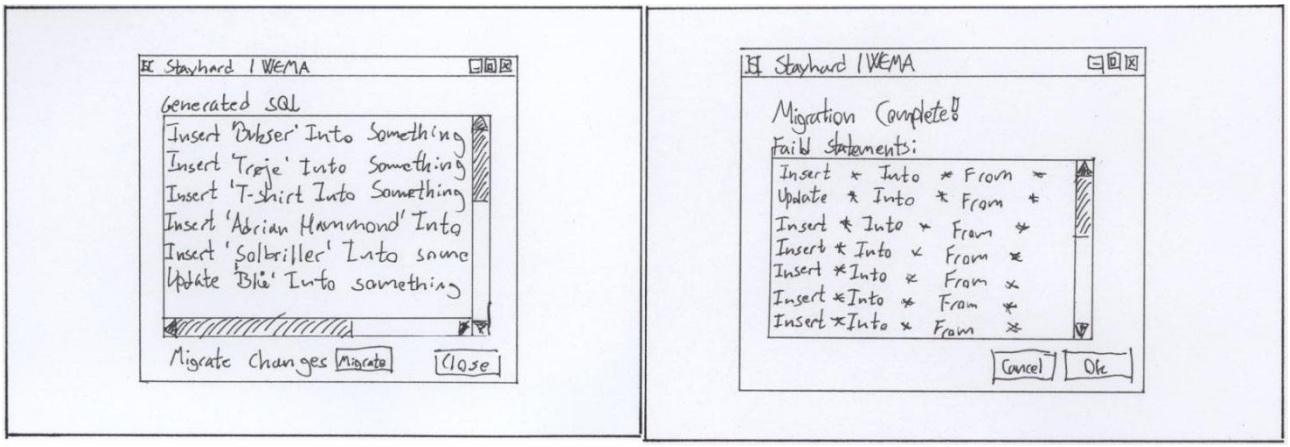
Figur 13 ovenfor tager udgangspunk i use-case-beskrivelse 1, alternativt scenarie 1, og use-case-beskrivelse 2, hovedsucces scenarie 1.

Mock-up 3 til venstre viser, hvordan alternativt scenarie 1, forventes at tage sig ud i prototypen. Det vil være et vindue, hvor brugeren har mulighed for at krydse de tabeller af, som han/hun ønsker skal sammenlignes. Tabellerne har farvekoder der indikerer hvilke af de to databaser de findes i.

- **Grøn** – Tabellen findes både i SourceDB og TargetDB.
- **Gul** – Tabellen findes udelukkende i SourceDB.
- **Orange** – Tabellen findes udelukkende i TargetDB.

Mock-up 4 til højre viser, hvordan hovedsucces scenarie 1 i use-case-beskrivelse 2, forventes at tage sig ud i prototypen. Dette vindue vises efter der er lavet en 'Full'-sammenligning eller efter brugeren har valgt hvilke tabeller der ønskes sammenligninget. Træstrukturen i den venstre side viser samtlige tabeller, klikker man på en af tabellerne, vil de rækker den indeholder blive vist i den højre side. På samme måde som i mock-up 3 er der farvekoder der indikerer forskellige ting, for den enkelte række eller tabel:

- **Grøn** – Rækken eller tabellen er ikke ændret
- **Gul** – Rækken eller tabellen er blevet ændret
- **Orange** – Rækken eller tabellen findes kun i Source
- **Rød** – Rækken eller tabellen kun i TargetDB



Figur 14 - Mock-ups 5 & 6

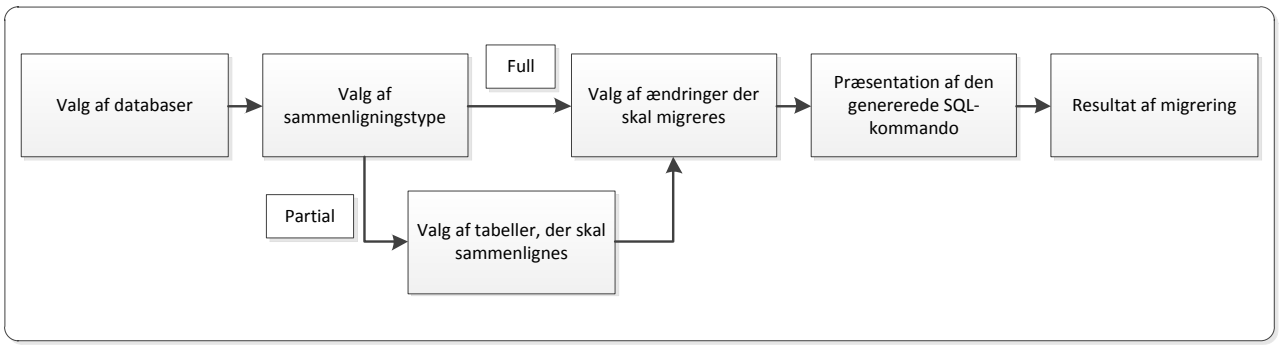
Figur 14 ovenfor tager udgangspunk i use-case-beskrivelse 3, hovedsucces scenarie 2 og hovedsucces scenarie 4.

Mock-up 5 til venstre viser, hvordan hovedsucces scenarie 2 i use-case-beskrivelse 3, forventes at tage sig ud i prototypen. Brugeren vil blive præsenteret for dette vindue, efter det er blevet valgt hvilke forskelle der skal migreres.

Mock-up 6 til højre viser, hvordan hovedsucces scenarie 4 i use-case-beskrivelse 3, forventes at tage sig ud i prototypen. Brugeren vil blive præsenteret for dette vindue efter, at brugeren har trykket på 'Migrate' i Mock-up 5. Brugeren bliver informeret om at migreringen er gennemført, og hvis er der SQL-kommandoer som er fejlet, vil de blive vist.

2.7.1 Navigationsdiagram

Et navigationsdiagram viser den hierarkiske-struktur for et systems brugerinterface. Det er en simpel måde at illustrere, hvordan en bruger kan navigerer rundt i et system.



Figur 15 - Navigations-diagram

Som det fremgår af navigationsdiagrammet ovenfor, kommer prototypens brugerinterface til at have et meget lineært forløb. Det skyldes prototypens specifikke formål, som ikke giver anledning til flere alternative forløb end det ene, som frem går af Figur 15.

2.8 Problematikker ved sammenligning af databaser

Når man skal sammenligne to databaser, er der en række problematikker, man skal være opmærksom på. En af hovedproblematikkerne er, hvordan man finder frem til den samme unikke række i begge databaser. Man kan ikke bare tage én række fra Database 1(DB1) og lede efter en række i Database 2(DB2), som har de samme værdier, da man kun vil finde den, hvis rækken er uændret. Hvis bare ét felt i rækken er ændret, vil man ikke kunne benytte denne metode, til at finde den samme unikke række i DB2. Man er derfor nødt til at kigge på den tilhørende tabels nøgler, for at kunne identificere hvilke kolonner der skal bruges til at finde en unik tabel.

Dette fører videre til en anden problematik, som handler om hvordan man håndterer de forskellige typer af nøgler, som en tabel kan have. En nøgle er tilknyttet en eller flere specifikke kolonner i én enkelt tabel, således at en tabel kan have mange nøgler, men en nøgle kun kan være tilknyttet én tabel. Der er tre forskellige slags nøgler, og de har alle tre hver deres karakteristika:

- **Primær-nøgler** – Er kendetegnet ved at værdierne i kolonnen, markeret som primær-nøgle, alle er unikke, samt ikke er null. Generelt set vil der kun være én kolonne pr. tabel, som vil være markeret som primær-nøgle. Primær-nøglen vil derfor oftest være den første nøgle som benyttes, når man ønsker at finde en unik række.

- **Unikke-nøgler** – Er kendetegnet ved at kombinere to eller flere kolonner til at skabe en unik nøgle. Til forskel fra en kolonne, markeret som primær-nøgle, kan kolonner markeret som unikke, godt være null, så længe den unikke nøglekombination forbliver unik.

- **Fremmed-nøgler** – Er kendetegnet ved at markere en kolonne, som bruges til at referere til en kolonne i en anden tabel. Kolonnen der refereres til skal være enten primær- eller unik-nøgle i den anden tabel. En fremmed-nøgle har ikke noget at sige, i henhold til om hvorvidt værdien af den markerede kolonne er unik.

Når man skal finde den samme unikke række i to databaser, vil man som oftest kigge på en eventuel primær-nøgle, som det første. Hvis ikke en tabel har en primær-nøgle, vil det næste logiske skridt være at kigge på unikke nøgler, såfremt de findes. Skulle man rende ind i en tabel, som hverken har primær- eller unikke-nøgler, vil det ikke umiddelbart være muligt at finde den samme unikke række i begge databaser, da man ikke kan vide, hvilke kolonner man skal kigge på, for at få en unik værdi.

På baggrund af ovenstående er det besluttet, at der udelukkende vil blive kigget på tabeller der indeholder primær- og unikke-nøgler. Endvidere er primær-nøgler prioriteret over unikke-nøgler, således at der først vil blive kigget på primær-nøgler, såfremt tabellen har sådanne en. Kun hvis en tabel ikke har en primær-nøgle, vil der blive kigget på tabellens unikke-nøgler.

2.9 Teknologianalyse

I teknologianalysen vil relevante teknologier blive fremhævet og beskrevet. Det vil blive belyst, hvorfor de forskellige teknologier er blevet valgt, samt hvilke fordele og ulemper de har.

2.9.1 C#./NET

Som nævnt i performancebegrænsningen PJB2 i kravspecifikationen, har Stayhard bedt om, at prototypen bliver udviklet i C#/.NET, da de har mest erfaring med netop dette programmeringssprog.

C Sharp (C#) er et multiparadigme programmeringssprog, som understøtter flere forskellige programmeringsdiscipliner, så som objektorienteret, generisk og imperativ programmering. Det er udviklet af Microsoft på baggrund af deres .NET-framework, og har til formål at være simpelt, men samtidig være brugbart til at løse mange forskellige programmerings-opgaver. Det benytter mange af de samme syntakser som C, C++ og Java også bruger, hvilket betyder at det er relativt nemt at lære, hvis man har erfaring med nogle af de førnævnte sprog.

At det er baseret på .NET-frameworket betyder, at men nemt kan forbinde et C# program med applikationer, som også benytter .NET-framworket. Det er for eksempel relativt simpelt at forbinde en C#-applikation med en webapplikation, såfremt denne er lavet i ASP.NET. Her tilbyder .NET-frameworket en række klassebiblioteker, som sørger for at de to applikationer kan kommunikere, samt at de kan kommunikere effektivt.

Eftersom C# er baseret på .NET, kan det næppe overraske nogen, at C# er stærkt platforms-afhængig, forstået på den måde, at C#-applikationer kun kan bruges på systemer, hvor der findes en .NET-version til. Officielt er der kun udviklet et .NET-framework til windows-miljøer, men der findes også open-source-versioner af frameworket som for eksempel Mono[5], som implementerer frameworket i GNU/Linux.

Da C# er udviklet af Microsoft, er der også sørget for, at man relativt hurtigt kan forbinde sin applikation med andre Microsoft-programmer. Det kunne for eksempel være office-programmer eller en Microsoft SQL-database. Ligesom ved kommunikation mellem .NET-applikationer, er de nødvendige værktøjer en grundlæggende del af .NET-platformen, i form af en lang række klassebiblioteker. Dette er en stor fordel, eftersom at Stayhard bruger Microsofts SQL-databaser, hvilket betyder at C# også ville være det oplagte valg, hvis ikke Stayhard selv havde bedt om det.

2.9.2 WPF - Windows Presentation Foundation

Da WPF er det User-Interface (UI)-framework, som Stayhards udviklere har mest erfaring med, har de bedt om, at prototypen udvikles med netop dette framework. Dette har de gjort, således at det bliver en mere overskuelig proces, hvis de ønsker at lave eventuelle udvidelser til prototypen i fremtiden.

Dette er også en del af performancebegrænsningen PJB2 fra kravspecifikationen.

Windows Presentation Foundation(WPF) er et UI-framework, som Microsoft frigav som en del af .NET Framework 3.0.

[5] http://www.mono-project.com/Main_Page

WPF benytter sig af DirectX i stedet for at bruge det noget ældre Graphics Device Interface(GDI), som mange andre windows-applikationer ellers benytter sig af. Dette medfører, at en del af programmets grafiske arbejdsbyrde kan blive givet til systemets GPU, og dermed aflaster CPU'en en smule. Derved er et WPF-programs grafiske opdateringsfrekvens bestemt ud fra systemets GPU-hastighed. Dette kan både være en fordel, og en ulempe. På systemer med en kraftig GPU, vil det betyde at opdateringsfrekvensen vil være meget høj, og bevægelser i programmet vil se meget flydende ud. Sidder man derimod med en netbook, som ikke har en særlig kraftig GPU, vil éns program virke meget hakkende, da opdateringsfrekvensen vil være noget lavere.

En anden grund til at WPF benytter sig af DirectX er, at det gør det muligt for WPF at fokusere mere på vektorgrafik. Det betyder, at de fleste kontroller og elementer i WPF kan forstørres, uden at der opstår kvalitetstab eller pixelering.

Sammen med WPF, udgav Microsoft et nyt programmeringssprog kaldet eXtensible Application Markup Language (XAML), som er baseret på XML. XAML er designet til at være en mere effektiv måde at lave grafisk brugerinterface til programmer.

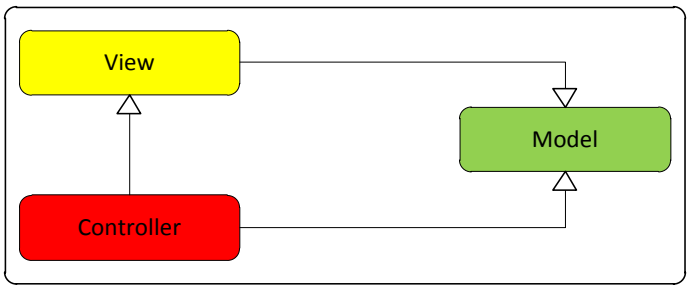
En fordel i WPF er, at det giver udvikleren mulighed for at kombinere XAML, og et andet .NET-kodesprog til at opbygge sit program. Man kan også vælge at lave programmet kun ved brug af XAML, eller ved helt at undgå XAML. Microsoft understreger dog, at der er en række fordele ved at bruge en kombination af XAML og et andet .NET-kodesprog[6]. Dette gøres ved, at man bruger XAML til at opbygge den grafiske del af UI'et, mens man bruger det andet .NET-kodesprog til at styre logikken(code-behind). Fordelene dækker blandt andet over at opdelingen mellem UI og code-behind medfører en mere testbar kode, samt at det hjælper udviklerne, som kan arbejde på hver sin del, på samme tid.

2.9.3 MVVM – Model-View-ViewModel

Det er vigtigt for at få en gennemført kode, at have valget af arkitektur-pattern fastlagt fra starten, da det ikke er noget man ubesværet tilpasser koden til efterfølgende. Arkitektur-patternet er med til at gøre koden lettere at udarbejde, da den generelle opbygning er fastlagt på forhånd. Desuden letter det opgaver, så som at lave rettelser og tilføjelser til koden.

Model-View-ViewModel (MVVM) er Microsofts udgave af design-patternet Presentation Model(PM), som blev udviklet af Martin Fowler. I forhold til PM, er MVVM tilpasset til udvikling af applikationer, som benytter Microsofts WPF. MVVM og PM er en videreudvikling af design-patternet Model-View-Controller (MVC). MVC er et arkitektur-pattern som blev udviklet i 70'erne, og som stadig er relevant den dag i dag.

[6] http://msdn.microsoft.com/en-us/library/aa970268(v=vs.100)#Markup_And_Codebehind

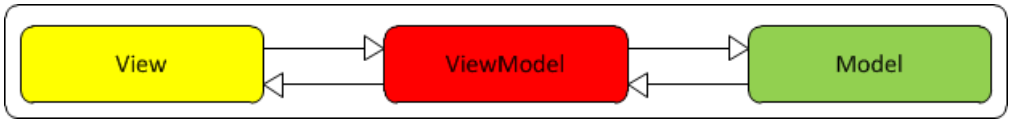


Figur 16 - MVC's opbygning

Kommunikationen mellem elementer i MVC sker på følgende måde:

- **Model** – Har til opgave at styre data internt i applikationen, videregive data som bliver forespurgt (som regel fra View), samt reagere på ændrede status og instruktioner (som regel fra Controller).
- **View** – Har til opgave at videregive information til brugeren.
- **Controller** – Har til opgave at videregive informationer om user-input til henholdsvis View og Model.

Når man ser figurer, der viser relationerne mellem objekterne i MVC og MVVM, er der elementer der går igen. View og Model er at finde i begge, mens Controller er blevet skiftet ud med ModelView. Selve relationerne mellem objekterne, og deres funktioner, i de to patterns er dog forskellige.



Figur 17 - MVVM's opbygning

Kommunikationen mellem elementer i MVVM sker på følgende måde:

- **Model** - Har til opgave at videregive og behandle applikations-data. Størstedelen af funktionaliteten og logikken placeret i denne del.
- **View** – Indeholder brugerinterfacet samt funktioner til at push og pull data til brugeren. View vil i WPF øjemed indeholde XAML og Code-Behind.
- **ViewModel** – Har til opgave at gøre data og funktioner/objekter fra Model tilgængelige i View. ViewModel skal være fuldt uafhængigt af View, så man i princippet kan skifte View'et ud, uden at skulle ændre i ViewModel.

I og med der er den stærke adskillelse af View og ViewModel i MVVM, gør det opgaven at teste applikationer lettere. Eftersom at al logik ligger i ViewModel og Model, gør det ikke nogen forskel om det er et View eller en unit test som pusher eller puller data fra ViewModel.

Som nævnt i afsnit 2.9.2 – WPF - Windows Presentation Foundation(side 25) har Stayhard ønsket, at der benyttes WPF til udvikling af prototypen. Derfor er det et meget nærtliggende valg at benytte MVVM som arkitektur-pattern til udvikling af prototypen.

Ud over den fordel, at MVVM gør det lettere at teste ens kode, er der også de generelle fordele, man får af at benytte et pattern til at bygge sin kode op omkring, så som at den fastlagte struktur gør det lettere at udvide og vedligeholde kode, både for en selv og for folk ude fra.

Når man skal bruge MVVM til et projekt, kan man godt vælge at implementere det hele fra bunden af. Dette kan dog meget hurtigt ende ud med at være en meget kompleks og omfattende opgave. Derfor er der blevet udformet en række frameworks, som indeholder den grundlæggende MVVM-struktur. Ved at bruge et sådanne framework, undgår man unødvendig kompleksitet, samt fordelen ved at bruge et framework, som allerede er dokumenteret.

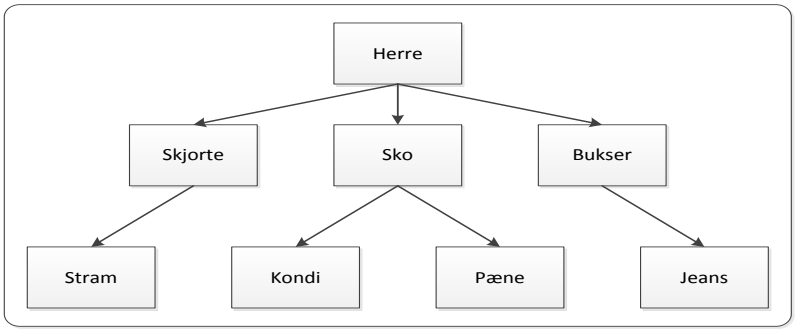
Det er i samarbejde med Stayhard, blevet besluttet at der i dette projekt skal bruges MVVM light-frameworket, som er udviklet af Galasoft[7].

2.9.4 Microsoft SQL Server

Stayhard har bedt om, at der i dette projekt bliver fokuseret på at sammenligne to databaser af typen Microsoft SQL Server (MSSQL). Grunden til dette er, at Stayhard hovedsageligt benytter databaser af denne type.

MSSQL er et database-system, som Microsoft lancerede tilbage i 1989 i samarbejde med virksomheden Sybase. Samarbejdet fortsatte indtil 1994, hvor Microsoft og Sybase opløste partnerskabet, og Microsoft fortsatte på egen hånd. I dag er MSSQL nået til version 11, og udviklingen fortsætter stadig i høj fart.

MSSQL er et database-system, som er baseret på den relationelle datamodel. Den relationelle datamodel er bare én af de mange modeller, en database kan bruge. Et alternativ kunne for eksempel være den hierarkiske model, som er baseret på parent-/child-forholdet.

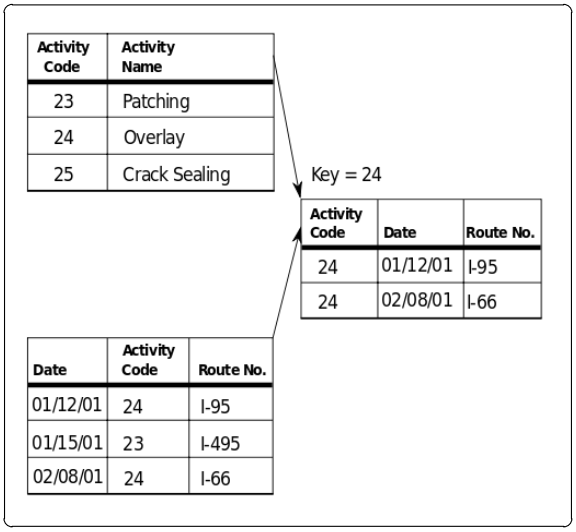


Figur 18 - Hierarkisk database-struktur

Som illustreret i den ovenstående figur, vil hvert element i en hierarkisk database have mellem nul og mange child-elementer, mens det kun kan have ét parent-element. Denne opbygning minder meget om

[7] http://www.galasoft.ch/mvvm/

opbygningen af XML, hvilket også kan forklare at denne database-model blev populær igen, kort efter XML blev lanceret[8].



Figur 19 - Relationel Databasemodel (billede fra Wikipedia[9])

Lige siden den relationelle database-model blev designet i 1969, er den langsomt men sikkert blevet den mest populære database-model[10]. Denne model er opbygget omkring tabeller, således at hver tabel indeholder data og nøgler. Disse nøgler kan bruges til at finde unikt data i den valgte tabel, eller bruges til at referere til en anden tabel. Datamodellens høje popularitet har medført, at den nu bruges i mange større systemer, herunder Oracle Database, MySQL og MSSQL[11].

En fordel ved MSSQL er, at der findes mange forskellige udgaver. Der er for eksempel Express-udgaven, som er en gratis-version, hvor der blandt andet er indført en begrænsning, således at den kun kan udnytte én enkelt CPU. En anden begrænsning er på størrelsen af databasen som maksimalt kan være på 10gb. En udgave i den anden ende af skalaen er enterprise-udgaven, hvor der ikke er nogen begrænsninger, og som understøtter kæmpe databaser, samt enorme mængder RAM og CPU'er.

De mange forskellige udgaver muliggør, at man kan anskaffe sig lige præcis den udgave, som dækker éns behov optimalt.

Som nævnt i afsnit 2.9.1 - C#./NET(side 25), er der meget høj kobling mellem Microsofts produkter, og der er derfor mange muligheder for at arbejde med en MSSQL-database, når man laver et program i C#/.NET.

[8] http://en.wikipedia.org/wiki/Hierarchical_database_model#History
[9] http://en.wikipedia.org/wiki/File:Relational_Model.svg
[10] http://en.wikipedia.org/wiki/Database_model
[11] http://en.wikipedia.org/wiki/Relational_model

2.10 Delkonklusion

I dette kapitel blev projektets problemstillinger og omfang specificeret. Der er dannet et overblik over hvilke input og output som prototypen kommer til at gøre brug af.

Baseret på problemstillingerne er der blevet opstillet en række funktionelle og ikke-funktionelle krav til prototypen. Efterfølgende er projektets omfang blevet afgrænset i henhold til de opstillede krav. Projektet er blevet afgrænset til at omfatte følgende:

- **FK1** – Sammenligning af to databaser
- **FK2** – Visuel præsentation af forskelle mellem databaserne
- **FK3** – Migrering af forskelle mellem databaserne

På baggrund af de opstillede krav er der formuleret use-case-beskrivelser, samt tilhørende sekvens-diagrammer, som beskriver brugerens interaktion med prototypen. Derudover er der udarbejdet en domænemodel, som viser hvilke relationer der skal være mellem prototypens objekt-entiteter.

Der er blevet beskrevet hvilke problemstillinger, man skal tage med i overvejelserne i forbindelse med udvikling af et GUI. På baggrund af dette er der udarbejdet mock-ups, som viser hvordan prototypens GUI forventes at se ud, samt et navigationsdiagram som viser sammenhængen mellem mock-upsne.

Da det er én af prototypens kernefunktioner, er det blevet undersøgt, hvilke problematikker man skal være opmærksom på, når man skal sammenligne to databaser. Som beskrevet i afsnittet blev det besluttet at prototypen kun vil fokusere på tabeller som indeholder primær- og/eller unikke-nøgler, samt prioritere primær-nøgler over unikke-nøgler.

Til slut i kapitlet er relevante teknologier og arkitekturer blevet gennemgået, og der beskrives hvorfor de enkelte teknologier er blevet valgt. Ud fra teknologianalysen er det blevet fastslået at der benyttes C#/.NET som kodesprog, MSSQL til databaser, MVVM som kodearkitektur med MVVM light-frameworket og WPF til at opbygge GUI.

Kapitel 3 – Design

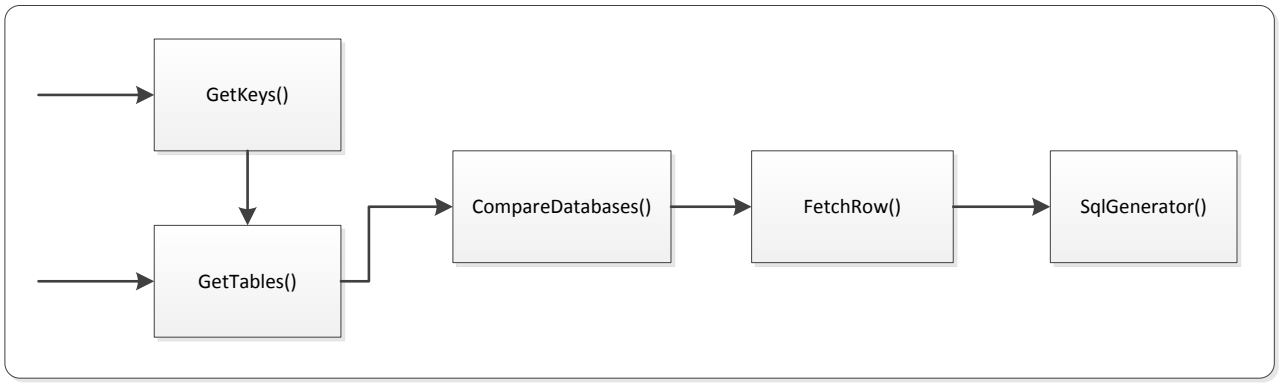
I dette kapitel vil den generelle opbygning for prototypen blive beskrevet. Endvidere vil der, baseret på analysen af sammenligningsfunktionen, blive opstillet forskellige løsningmodeller til hvordan funktionen kan designes. På baggrund af løsningsmodellerne vil den for projektet, mest optimale løsning blive udvalgt.

Herefter beskrives det, hvordan prototypen er struktureret, samt hvilke tanker der ligger til grund for dette. Prototypens opbygning vil blive klarlagt ved at uddybe, hvordan MVVM vil blive benyttet til at strukturere koden. Herudover vil der blive udformet nogle klassediagrammer, samt uddybende sekvensdiagrammer på baggrund af use-cases opstillet i analysen.

Der vil blive kigget på de designmæssige overvejelser, som vil komme til at ligge til grund for det endelige design af prototypens GUI.

3.1 Beskrivelse af prototypens opbygning

For at give et overblik over hvilke dele prototypen vil komme til at bestå af, vil dette afsnit indeholde en overordnet beskrivelse af flowet i systemet.



Figur 20 - Flowet i systemet

Flowet begynder med at prototypen benytter funktionen GetKeys() til at hente nøglerne ud for de tabeller, som brugeren har valgt at sammenligne. Når nøglerne er blevet hentet ud, hentes tabellerne ud ved at bruge GetTables(), hvorefter de respektive tabeller bliver sat sammen med deres tilhørende nøgler. Efter samtlige af de valgte tabeller er oprettet for begge databaser, og der er lavet to lister med tabeller, som repræsenterer de to databaser, sendes listerne til CompareDatabases(), hvor der skal udføres en sammenligning af de to databaser. Sammenligningen foregår ved at den samme tabel hentes ud fra begge databaser, for at blive sammenlignet. Inden man kan foretage sammenligningen, er man dog nødt til at finde den samme unikke række i de to tabeller, som man er i gang med at sammenligne. Til dette formål vil FetchRow() benyttes, som kigger på en række fra den ene tabel, og søger efter den samme række i den anden tabel. Når man har fundet frem til den samme unikke række fra begge tabeller, udføres selve sammenligningen af de to. Efter sammenligningen er blevet fuldført, bliver resultatet returneret i form af en enkelt række, som indeholder den sammenlignede data, samt information om hvad der er blevet ændret, såfremt der er nogen forskelle mellem de to rækker. Når samtlige tabeller er blevet sammenlignet, sendes resultatet til SqlGenerator(), som vil bearbejde resultatet af sammenligningen. Der vil blive oprettet

en række SQL-kommandoer baseret på resultatet fra CompareDatabases(), og en liste med samtlige genererede SQL-kommandoer vil blive returneret. Herfra bliver kommandoerne kørt på TargetDB, hvorefter prototypens forløb er fuldført.

3.2 Design af funktioner

Dette afsnit vil beskrive designet af nogle af de funktioner, som prototypen vil indeholde. Endvidere vil et par af de løsningsmodeller, der har været for funktionerne, blive beskrevet, samt hvad der har ligget til grund for det endelige design.

3.2.1 GetKeys()

Formålet med denne funktion er, at hente de relevante nøgler ud fra en database.

Som beskrevet i afsnit 2.8 - Problematikker ved sammenligning af databaser(side 24), vil det kun være muligt at udføre en sammenligning, hvis begge tabeller har de nødvendige nøgler. Der vil derfor, som det også er nævnt i analysen, kun blive fokuseret på tabeller, som har enten primær-nøgle, eller unikke-nøgler. Af denne grund er den funktion, som henter nøgler ud af databasen, en af de vigtigere af prototypens funktioner. Måden hvorpå nøgler hentes ud af en database, afhænger meget af hvilken databasetype, der er tale om.

Som det blev fastlagt i teknologi-analysen, fokuseres der i dette projekt udelukkende på MSSQL. Denne database har en række system-views kaldet ”INFORMATION_SCHEMA”, hvor man blandt andet kan finde viewet ”KEY_COLUMN_USAGE”. Ud fra dette view kan man hente alle de data, som vil være nødvendige for at oprette en nøgle i prototypen, angivet i rækkerne ’constraint-name’, ’table-name’, ’column-name’ og ’ordinal position’. ’Constraint-name’ dækker over et sammensat tekst-felt, som både indeholder navnet på den tabel, nøglen er tilknyttet, samt hvilken nøgletype, der er tale om. Hvis feltet er en fremmed-nøgle, indeholder feltet også navnet på reference-tabellen. Ved brug af regular-expressions, vil nøglens ’constraint-name’ blive benyttet til at bestemme hvilken type nøgle der er tale om, når nøglen oprettes i prototypen.

Mens ’table-name’, ’column-name’ og, delvist, ’ordinal position’ er tilnærmelsesvis selvforklarende, vil disse blive uddybet senere i kapitlet, hvor prototypens objekter, metoder og attributter gennemgås.

```
SELECT
    CONSTRAINT_NAME  = KCU.CONSTRAINT_NAME,
    TABLE_NAME       = KCU.TABLE_NAME,
    COLUMN_NAME      = KCU.COLUMN_NAME ,
    ORDINAL_POSITION = KCU.ORDINAL_POSITION
FROM
    INFORMATION_SCHEMA.KEY_COLUMN_USAGE KCU
```

Figur 21 - SQL-sætning til at hente nøgle-info ud af en database

Figur 21 viser den SQL-kommando, som vil blive brugt til at hente nøglerne ud for den pågældende database. Når man skal hente data ud fra databasen, kan man bruge et SqlDataReader-objekt til at gemme det resulterende datasæt i. Et SqlDataReader-objekt kan benyttes til iterativt at gennemgå samtlige rækker i et resultatsæt fra en database. Dette fungerer ved at objektet beholder en åben forbindelse til databasen, og henter en enkelt række ud ad gangen, indtil alle rækker er blevet læst. Det er med dette objekt kun

muligt at gå videre til næste række, og man kan derfor ikke gå tilbage i listen, eller springe frem til et bestemt punkt.

3.2.2 GetTables()

Formålet med denne funktion er at forbinde til en database, og hente de relevante tabeller ud, og oprette prototypens egen repræsentation af databasen.

Denne funktion vil være opbygget således, at den henter én tabel ud af gangen, baseret på et tabelnavn. Resultatet fra databasen, som vil blive opbevaret i en SqlDataReader, vil blive gennemløbet, og baseret på den hentede data, vil prototypen oprette dens egen repræsentation af den hentede tabel. Når tabellen er oprettet, vil de relevante nøgler blive tilføjet til tabellen.

3.2.3 FetchRow()

Denne funktion skal bruges til at finde frem til den samme unikke række i to forskellige databaser.

Den kan laves på flere forskellige måder, hvor én måde er, at man under oprettelse af et Row-objekt, bygger en ”nøgletekst”, som indeholder rækkens nøgle-værdier sat sammen i en bestemt rækkefølge, baseret på tabellens nøgler. Hvis man forestiller sig et scenarie, hvor man har en tabel med to unikke-nøgler, hvor første nøgle er fjerde kolonne, og anden nøgle er første kolonne, vil en række med værdierne ”T-shirt”, ”herre”, ”medium” og ”Diesel” få en nøgletekst som hedder ”DieselT-shirt”.

Idéen med nøgleteksten er, at den vil være rækkens unikke værdi, hvilket vil sige den værdi, som skal bruges for at hente lige netop denne række ud. Det betyder, at man ved at benytte et Dictionary med nøgleteksten som nøgle i Dictionary’et, vil kunne fremsøge en unik række, uden selv at skulle løbe en hel liste af rækker igennem. Man vil altså for hver enkelt række tilgå den samme række i sammenlignings-tabellen direkte, uden først at skulle fremsøge den. Ulempen ved denne løsning er dog, at det tager noget længere at oprette de enkelte Row-objekter. Grunden til dette er, at hver enkelt Row-objekt skal loopes igennem ’m’ gange, hvor ’m’ er antallet af nøgler, for at bygge nøgleteksten. En ting, man skal være opmærksom på med denne løsning er, at kørselstiden forøges på det punkt, hvor man kører funktionen, som sætter nøgleteksten. Endvidere skal man også være opmærksom på, at den først bør køres efter brugeren har valgt hvilke forskelle der skal migreres, da man ellers potentielt kører funktionen på unødvendigt mange rækker. Funktionen skal dog køres på samtlige af de tabeller, som skal sammenlignes.

En anden løsning er at bruge en rekursiv metode(en metode der kalder sig selv). Denne metode får så følgende parametre:

- En række, som man ønsker at sammenligne med den samme unikke række i sammenlignings-tabellen.
- En liste af relevante nøgler fra sammenligningstabellen.
- Listen af rækker fra sammenligningstabellen.
- Nøgleindeks, som fortæller indekset på den nøgle, som skal benyttes. Vil være ’0’ ved første gennemkørsel.

Ved første gennemkørsel bruger metoden den nøgle med højest prioritet, til at hente den første unikke værdi ud fra den givne række. Listen af rækker fra sammenlignings-tabellen gennemsøges for alle de rækker, som indeholder den første unikke værdi, hvorefter de fundne rækker bliver tilføjet til en liste.

Metoden kaldes nu rekursivt med samme række og liste af nøgler, men med den nyoprettede liste, og et nøgleindeks, der er forøget med én. Denne fremgangsmåde gentages for hver nøgle af samme nøgletype, som var på plads '0' i listen af nøgler. Når samtlige nøgler er blevet gennemgået, bør der kun være nul eller én række tilbage, alt efter om den findes i sammenlignings-tabellen. Hvis der er flere end én, betyder det, at det ikke har været muligt at finde frem til den unikke række ud fra de givne nøgler. I dette tilfælde vil det ikke være muligt at fuldføre sammenligningen på denne række. Denne løsning kan så optimeres, således at eventuelle fundne rækker fjernes fra listen af rækker fra sammenlignings-tabellen, således at størrelsen af denne liste bliver én mindre for hver gennemkørsel.

I dette projekt er det valgt at løsning to skal benyttes, da den umiddelbart vil medføre en mere struktureret kode, hvilket er en del af performancebegrænsningerne, PFB2 – Skalerbarhed.

3.2.4 CompareDatabases()

Denne funktion skal bruges, når de to databaser skal sammenlignes.

Når man skal lave en sammenligningsfunktion af den type, som der er behov for i dette projekt, kan man designe funktionen på flere forskellige måder. Én løsning kunne være, at have en funktion med nestede loops, hvor det yderste loop gennemgår en liste af tabeller fra første database. For hvert tabel, skal den tilsvarende tabel findes i listen af tabeller fra den anden database. Hvis denne ikke kan findes, betyder det at tabellen findes i den første database, men ikke anden, og der derfor ikke er nogen grund til at behandle tabellen yderligere, udover at markere den som "ny". I de fleste tilfælde vil man dog finde den pågældende tabel i listen.

Når man har fundet den korrekte tabel, er det en god idé at sammenligne de to tabellers nøgler, da det kan skabe problemer, hvis de ikke har ens nøgle-sæt. Når det er blevet valideret, at de to nøgle-sæt er ens, er man nået til at skulle sammenligne tabellernes rækker. Dette gøres ved først at finde den samme unikke række i begge tabeller, som kan gøres ved at bruge en af metoderne beskrevet i forrige afsnit, 3.2.3 - FetchRow()(side 33). Herefter skal felterne i de to rækker sammenlignes, hvilket kan gøres ved at gennemløbe den ene rækkes liste af felter, og sammenligne værdien og datatypen med det samme felt i den anden række. Hvis der er en ændring i bare ét af felterne, skal feltet, rækken og tabellen markeres som "ændret".

Når samtlige felter fra den ene række er bearbejdet, skal man kontrollere om den anden række har nogle felter, som den første række ikke har. Disse felter vil nemlig ikke blive registreret, hvis man udelukkende fokuserer på den første rækkes felter. Dette gælder ligeledes, når man har sammenlignet alle tabeller og rækker fra den første database. Finder man et objekt af denne type, uanset om det er en tabel, en række eller et felt, skal det markeres som "slettet". Det skal markeres som "slettet", da det betyder at objektet ikke er at finde i den første database, som indeholder de "nyeste" data. Når man har sammenlignet samtlige rækker og felter i en tabel, fortsætter loopet til næste tabel, indtil samtlige tabeller er sammenlignet, hvorefter sammenlignings-funktionen returnerer en liste med resultatet af sammenligningen.

En anden metode er, at flytter sammenligningsfunktionaliteten ud i de respektive objekter, ved at benytte interfacet IComparable. Funktionaliteten vil på mange punkter være den samme, som i den anden løsningsmodel, men forskellen er her, at sammenligningsfunktionen for hvert objekt vil være meget tæt

knyttet til det respektive objekt. Mange af de loops, som er at finde i den første løsning, er også at finde i denne løsning, men her er de blot rykket ned i ”CompareTo”-metoden for de enkelte objekter. En ulempe ved dette er, at det gør det en smule sværere at beholde overblikket over den samlede sammenligningsfunktion. Til gengæld viser denne løsning meget tydligt, præcis hvad der sker med det enkelte objekt, når sammenligningsfunktionen køres.

I dette projekt er det valgt at benytte en blanding af disse to løsninger, for at udnytte fordele fra begge løsninger. Den første løsning er benyttet, indtil det punkt, hvor man har fundet den samme unikke række fra begge tabeller. Herfra benyttes den anden metode, blot modificeret en smule. Konceptet med at flytte funktionaliteten ud i objektet er stadig beholdt, men IComparable benyttes dog ikke, da det kræver at ”CompareTo”-funktionen returnerer en Integer. Funktionen er i stedet ændret således at objektet opdaterer sig selv, baseret på resultatet af sammenligningen, hvorefter det returnerer sig selv. Idéen med denne løsning er, at beholde overblikket, så vidt muligt, men samtidig vise præcis hvad der sker med en række og et felt, når disse bliver sammenlignet.

3.2.5 SqlGenerator()

Idéen med denne funktion er, at man giver den en liste af ændringer, som skal migreres, hvorefter der genereres en række SQL-kommandoer. Disse kommandoer vil herefter blive returneret i en liste.

Ligesom med de andre funktioner, er der også flere forskellige løsningsmodeller for denne funktion. Der er flere ting, man skal have med i sine overvejelser, når man designer denne funktion. Én overvejelse er, hvordan man ønsker at gemme den genererede SQL-kommando i systemet. Dette kunne gøres ved at bruge en simpel string, som indeholdte kommandoen. Fordelen ved denne løsning er, at den ville være meget simpel, og ville ikke fylde meget i systemet, hvilket er at foretrække, hvis man laver et letvægts-program. Ulempen med den løsning er dog, at man ikke ville vide noget omkring den enkelte kommando, og ikke på nogen let måde ville kunne spore kommandoen tilbage til en tabel/række/felt, såfremt man skulle have brug for dette.

Man kunne også vælge at lave et helt objekt, som ville indeholde kommandoen, samt yderligere informationer, så som hvilken tabel/række/felt, kommandoen var relateret. Fordelen ved denne løsning er, at man vil kunne beholde tilhørsforholdet mellem en SQL-kommando og det relaterede objekt. Ud over denne relation, ville man også kunne gemme en række andre informationer, så som kommando-typen, og resultatet, efter kommandoen er blevet kørt. Det ville være en fordel, når man skal vise resultatet for hver SQL-kommando, således at man kan se præcis hvilken tabel/række/felt, som kommandoen var tilknyttet, hvis den fejlede. Ulempen ved denne løsning er, at den fylder noget mere end den anden løsning, da man her vil bruge et noget større objekt, end en string.

En anden overvejelse, som man bør tænke på, er opbyggelsen af SQL-kommandoerne. Som det blev beskrevet i designet af sammenligningsfunktionen, vil det blive markeret på hvert objekt i prototypen(tabeller, rækker og felter), hvordan det pågældende objekt er blevet ændret. Det er derfor muligt, at lave SQL-kommandoer helt nede på felt-niveau. Fordelen ved denne løsning er, at man vil kunne se præcis hvilket felt der skabte problemer, hvis der var problemer med en migrering. Ulempen er, at man vil ende op med en enorm mængde SQL-kommandoer, som prototypen skal holde styr på. En anden løsning er, at holde kommandoerne på række-niveau. På denne måde formindsker man antallet af kommandoer,

men man vil stadig kunne identificere, hvilken række der skabte problemer, hvis der opstod problemer med en migrering.

I dette projekt er det valgt at benytte en string, til at holde styr på SQL-kommandoerne, for at de ikke kommer til at fylde for meget i systemet. Endvidere er det valgt at holde kommandoerne på række-niveau, for at holde antallet af kommandoer på et overskueligt niveau.

3.3 Systemarkitektur

For at afklare hvordan prototypen opnår en systemarkitektur, som overholder kravene PFB2 og PFB3 fra kravspecifikationen, vil dette afsnit beskrive grundtanken for den arkitekturmodel, som prototypen skal benytte. På den måde sikres det at prototypens opbygning vil være ensartet, og samtidig sørge for at fremtidige udvidelse af funktionalitet vil være en mere overskuelig opgave.

3.3.1 MVVM

For at sørge for, at prototypen opfylder kravet PFB2 om skalerbarhed, vil der benyttes en systemarkitektur, som sørger for en struktureret opbygning af prototypen. En god struktur gør koden mere læsbar, og giver generelt et bedre overblik. Endvidere vil en god struktur sikre et godt grundlag for en nem og smertefri overdragelse til fremtidige udviklere. Dette kunne for eksempel være i forbindelse med fremtidige udvidelser, eller vedligeholdelse af prototypen.

Som det konkluderes i analysen vil der i dette projekt benyttes arkitektur-patternet MVVM. En fordel ved MVVM er måden hvor ved systemet er opdelt i de tre dele, Model, View og ViewModel. Denne opdeling medfører at eventuelle rettelser eller udvidelser ikke nødvendigvis skal foretages i alle tre dele. Den stærke adskillelse mellem View og ViewModel medfører, at man kan foretage ændringer til systemets GUI uden at det påvirker logikken bagved, og den anden vej rundt. En anden fordel ved opdeling er at man kan teste direkte mod GUI og logik uafhængigt af hinanden. Dette muliggør at man kan skrive unit-tests til et systems kernefunktioner uden at tage hensyn til GUI. Adskillelsen gør det også muligt at opnå en højere abstraktion af ens model-objekter. På denne måde kan man beholde model-laget så simpelt som muligt, mens behandling af rå-data finder sted i ViewModel og bliver sendt videre til View.

I MVVM står et View udelukkende for en brugers interaktion med systemet. Et View bør kun indeholde kode som styrer den logiske interaktion mellem de grafiske elementer, samt den nødvendige kode til at binde op mod ViewModel, men har ingen kendskab om resten af systemet. ViewModel fungerer som formidler mellem View-delen og Model-delen, og bør kun indeholde simpel logik. Formålet med ViewModel er at hente relevante data ud fra model-laget og bearbejder det således at det er tilpasset det enkelte View. Det er derfor at man oftest har en eller flere ViewModels for hvert View, alt efter hvilke data View'et har behov for. I MVVM indeholder Model-delen programmets kerne, hvilket vil sige systemets objekter, services, dataforbindelser, med mere.

Når man starter et program, som benytter MVVM-arkitekturen, indledes programmet i ViewModel-delen. Herfra henter ViewModel'en de relevante data ud fra Model-delen, og binde de behandlede data til View'et. Når brugeren foretager en handling, som systemet skal reagere på, er det ViewModel'en, som skal håndtere dette. Hvis handlingen medfører en ændring i de relevante data, skal View'et have dette at vide, således at det kan hente de opdaterede data fra ViewModel'en.

I dette projekt vil MVVM blive implementeret ved at benytte frameworket MVVM Light. Grundtanken er at der vil blive implementeret et View for hvert vindue i GUI'en. Til hvert View vil der være tilknyttet en ViewModel, som binder det enkelte View til Model. Model-delen kommer til at indeholde objekterne fra domænemodellen, sammenligningsfunktion, migreringsfunktionen samt dataforbindelse til databaserne.

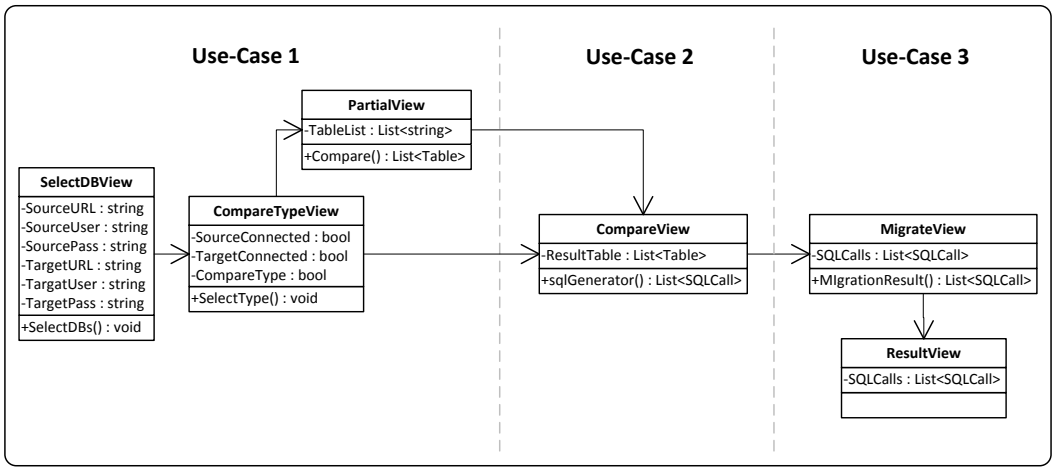
3.3.2 Klassediagrammer

I forbindelse med udvikling af software er der udformet en række modeller, man kan benytte til at skabe overblik over hvordan systemet skal opbygges. Dette letter også implantationsprocessen, da man allerede i designfasen har gennemtænkt, hvordan koden er opbygget og hvilke dele der kommunikerer med hinanden. Herudover sikrer det også, at man overholder eventuelle arkitekturmodeller, som man måtte ønske at benytte. Herudover bliver det lettere, hvis man skal overlevere et projekt, da man så med standart termer kan beskrive, hvordan applikationen er opbygget.

For overskuelighedens skyld er det valgt at der først laves et klassediagram der viser sammenhængen mellem prototypens Views, hvor man samtidig kan se hvilke use-cases der dækker hvilke dele. Herefter vil der blive opstillet et klassediagram for prototypens objekt-klasser, og et for prototypens funktionelle-klasser. Det er besluttet at holde et vist abstraktions niveau således at det ikke er alle funktioner og attributter der bliver vist.

3.3.2.1 View-klasser

Figur 22 viser et klassediagram over sammenhængen mellem systemets View-klasser, samt hvilke attributter og metoder de indeholder.



Figur 22 - Klassediagram over Views

Som nævnt viser dette klassediagram hvilke Views der er inkluderet i hvilke use-cases. Dette er gjort for at man bedre kan se hvilke funktioner den enkelte use-case-del skal opfylde. Hvert View i Figur 22 kommer til at være forbundet til én ViewModel som forbinder til de bagved liggende funktioner i Model. For at bevare overblikket vil Model-delen blive beskrevet med separate klassediagrammer.

Samtlige metoder som sender brugeren videre fra et View til et andet, vil blive kaldt, når brugeren trykker på en knap.

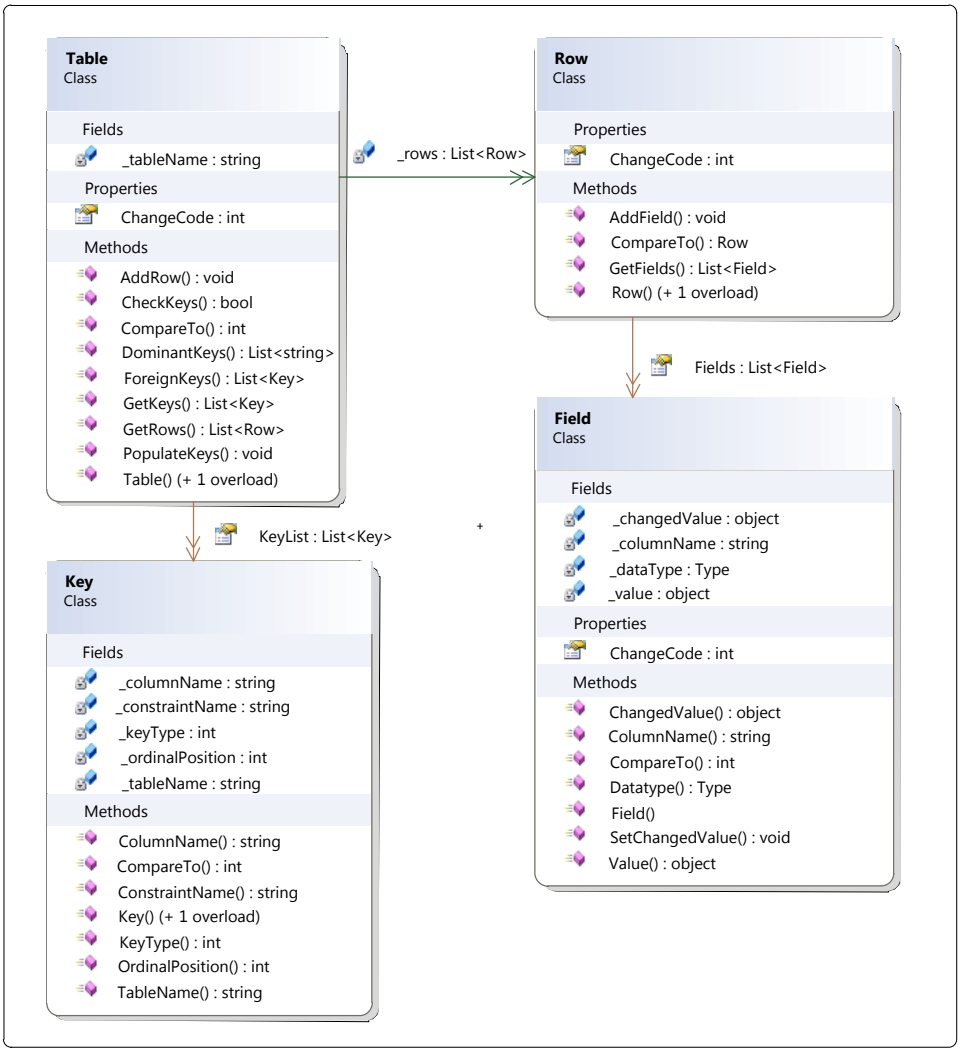
Det første View, som brugeren vil blive præsenteret for, er 'SelectDBView', hvor brugeren skal vælge hvilke databaser der skal forbindes til, ved at indtaste URL, brugernavn og kodeord til hver database. Herfra vil brugeren fortsætte til 'CompareTypeView', når metoden SelectDBs() bliver kaldt. I dette View vil SourceConnected og TargetConnected benyttes til at indikere om hvorvidt der er forbindelse til de respektive databaser. Brugeren skal vælge hvilken type sammenligning der skal laves, ved at sætte værdien for CompareType. Afhængigt af CompareType's værdi bestemmes det om brugeren skal sendes direkte til CompareView(sammenligning af typen Full), eller om brugeren først skal forbi PartialView(sammenligning af typen 'Partial') og herefter til CompareView. PartialView vil vise en liste af tabelnavne, hvor brugeren udvælger hvilke tabeller der skal foretages sammenligning på. Herfra vil brugeren blive sendt videre til CompareView ved at Compare() bliver kaldt.

I CompareView vil brugeren blive præsenteret for resultatet af sammenligningen, hvorefter brugeren kan udvælge hvilke forskelle der skal migreres. Når de ønskede forskelle er blevet valgt fortsætter brugeren til MigrateView ved at metoden sqlGenerator bliver kaldt.

MigrateView vil præsentere brugeren for de SQL-kommandoer, som blev genereret baseret på de valgte forskelle. Brugeren vil herefter kunne vælge mellem at kopiere de genererede SQL-kommandoer og manuelt køre kommandoerne uden for prototypen, eller lade prototypen køre kommandoerne. Prototypen fortsætter kun videre til ResultView, såfremt brugeren vælger at lade prototypen migrere forskellene. I ResultView vil resultatet af de kørte SQL-kommandoer vises, og brugeren vil kunne se om der har været problemer i forbindelse med migreringen.

3.3.2.2 Objekt-klasser

Nedenstående klassediagram viser hvordan prototypens objekt-klasser vil være forbundet med hinanden.



Figur 23 - Klassediagram over objekt-klasser

Som det også er nævnt i beskrivelsen af domænemodellen i analysen, er formålet med objekt-klasserne at kunne gengive indholdet af en relationeldatabase. Objekt-klasserne vil derfor komme til at minde meget om elementer, som er at finde i en database af denne type.

I prototypen vil en database blive gengivet ved at have en liste af Table-objekter. Hvert Table-objekt vil indeholde en Liste af Row-objekter, som igen vil indeholde en liste af Field-objekter. Herudover vil Table-objektet også indeholde en liste af Key-objekter.

De enkelte objekt-klassers vigtigste attributter og metoder vil kort blive beskrevet i følgende tabeller.

Tabel 2 – Table-Objekt: beskrivelse af attributter og metoder

Attributter:	Type:	Beskrivelse:
ChangeCode	Int	ChangeCode skal benyttes til at holde styr på om der er lavet en ændring på tabellen. Der vil skelnes mellem følgende typer af ændringer: Ingen ændring(0), Indholdsændring(1), Ny tabel(2), Slettet tabel(3).
Metoder:	**Retur type:**	**Beskrivelse:**
CheckKeys()	Bool	Vil have til formål at kontrollere at to tabeller, der skal sammenlignes, har de samme nøgler.
DominantKeys()	List<String>	Skal returnere en liste af de højest-prioriterede nøgler. Vil returnere tabellens primær-nøgle, såfremt den har sådanne en. Ellers returneres en liste af dens unikke-nøgler.
PopulateKeys()	void	Vil benyttes når et Table-objekt oprettes, til at hente og sortere dets nøgler.

Tabel 3 – Row-Objekt: beskrivelse af attributter og metoder

Attributter:	Type:	Beskrivelse:
ChangeCode	Int	Vil have samme funktion som i Table-objektet.
Metoder:	**Retur type:**	**Beskrivelse:**
AddField()	void	Vil benyttes til at tilføje et nyt felt til en række.

Tabel 4 - Field-Objekt: beskrivelse af attributter og metoder

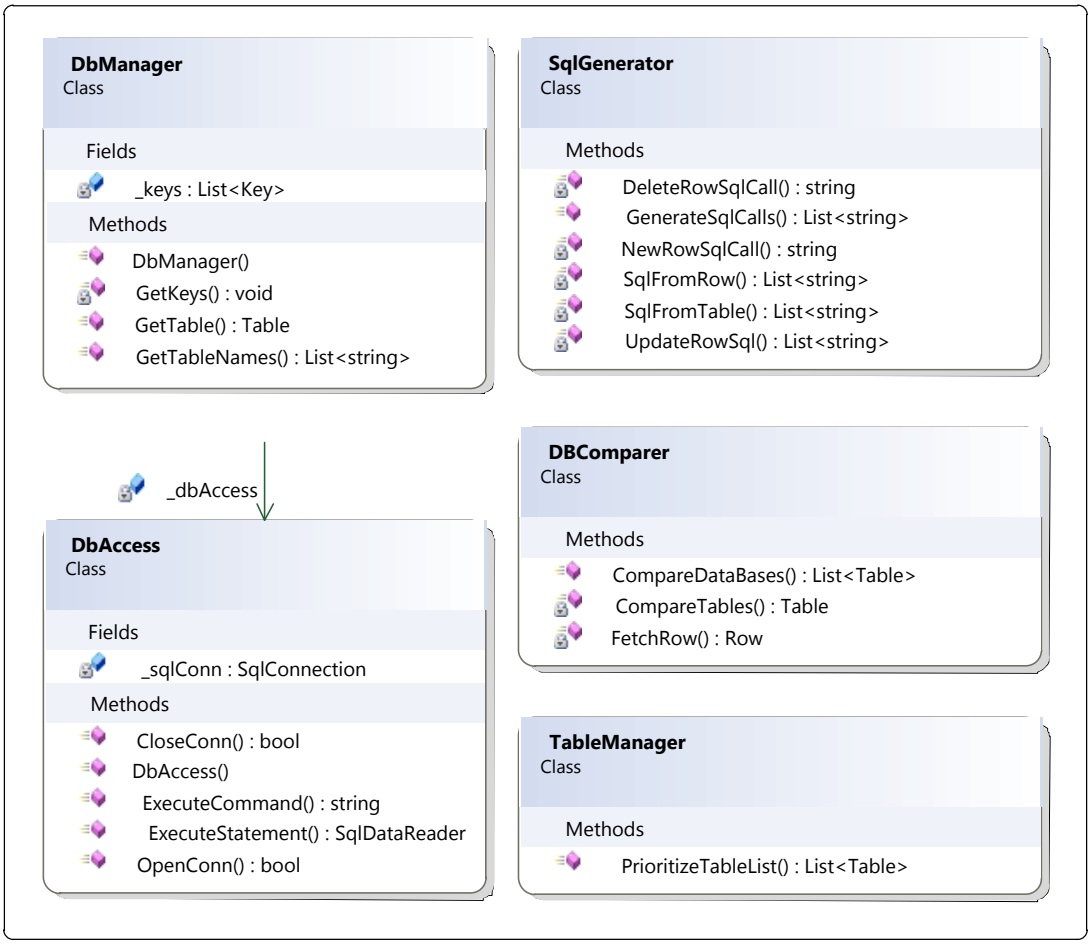
Attributter:	Type:	Beskrivelse:
_value	objekt	Feltets værdi vil blive gemt som typen objekt da feltets egentlige type ikke vil være kendt på forhånd.
_changedValue	objekt	Hvis det opdages at der forskel i et felts værdi, i forbindelse med sammenligningen, gemmes den ændrede værdi i dette felt. Typen objekt benyttes af samme årsag, som nævnt i _value.
ChangeCode	Int	Vil have samme funktion som i Table-objektet.
Metoder:	**Retur type:**	**Beskrivelse:**
SetChangedValue()	void	Vil benyttes til at sætte _changedValue.

Tabel 5 - Key-Objekt: beskrivelse af attributter og metoder

Attributter:	Type:	Beskrivelse:
_columnName()	String	Vil indeholde navnet på den kolonne som nøglen et tilknyttet.
_tableName()	String	Vil indeholde navnet på den tabel som nøglen er tilknyttet.
Metoder:	**Retur type:**	**Beskrivelse:**
keyType()	Int	En Integer-repræsentation af nøgle typen: Primær-nøgle(0), Unik-nøgle(1) og Fremmed-nøgle(2).
ordinalPosition()	Int	Vil fortælle hvor i rækkefølgen den pågældende nøgle er, såfremt der er flere af samme slags nøgletype.

3.3.2.3 Funktionelle-klasser

Figur 24 - Klassediagram over funktionelle-klasser et klasse-diagram som viser hvilke funktionelle klasser prototypen kommer til at indeholde. Ved funktionelle-klasser menes der her, at det er en klasse i model-laget, som ikke er betegnet som en objekt-klasse.



Figur 24 - Klassediagram over funktionelle-klasser

Klassen DbManager skal håndtere enhver handling, hvor data skal hentes ud, eller gemmes i en database. Når DbManager-klassens konstruktør bliver kaldt, skal der oprettes en instans af DbAccess, som leverer forbindelsen til databasen. Hver gang der skal kommunikeres med databasen, skal der bruges funktioner fra DbAccess, som sørger for at åbne og lukke forbindelsen, samt køre forespørgsler på databasen. Der vil blive oprettet en DbManager for hver af de to databaser, som skal sammenlignes.

Nedenfor beskrives de vigtigste funktioner der vil være i DbManager-klassen.

Tabel 6 - DbManager: beskrivelse af metoder

Metoder:	Retur type:	Beskrivelse:
GetKeys()	void	Denne metode vil bruges til at hente de nøglerne ud af databasen. Den vil gemme nøglerne i attributten _Keys, som vil være en liste af Key-objekter.
GetTableName()	List<string>	Vil hente en liste af tabelnavne ud af databasen, og returnere dem som en liste af strings.
GetTable()	Table	Vil hente en enkelt tabel ud af databasen, oprette et Table-objekt med de hentede data, og returnere det oprettede objekt.

DbAccess-Klassen vil blive benyttet til at oprette forbindelse til en database, som vil blive specificeret, når klassens konstruktør bliver kørt. Al forbindelse til databasen vil foregå gennem DbAccess, som vil stille relevante metoder til rådighed for klasser, som skal bruge en forbindelse til databasen. Følgende tabel indeholder en beskrivelse af de væsentlige metoder, DbAccess kommer til at indeholde.

Tabel 7 - DbAccess: beskrivelse af metoder

Metoder:	Retur type:	Beskrivelse:
OpenConn() CloseConn()	bool	Vil håndtere forbindelsen til databasen, og sørge henholdsvis for at åbne og lukke for forbindelsen til databasen.
ExecuteStatement()	SqlDataReader	Vil benyttes til at lave en forespørgsel på databasen, hvor man forventer at få de resulterende rækker.
ExecuteCommand()	string	Vil blive brugt til at lave en forespørgsel på databasen, hvor man ikke forventer at få resulterende rækker. Det kunne for eksempel være hvis man indsætter en række.

DbComparer kommer til at stå for at håndtere sammenligningsfunktionen i prototypen. Størstedelen af de metoder, som har med sammenligningen at gøre, kommer til at ligge i denne klasse.

Tabel 8 - DbComparer: beskrivelse af metoder

Metoder:	Retur type:	Beskrivelse:
CompareDataBases()	List<Table>	Vil blive brugt til at sammenligne to lister af tabel-objekter som repræsenterer de to databaser. Der returneres en nye liste som indeholder resultatet af sammenligninngen.
CompareTables()	Table	Skal bruges til at sammenligne to tabel-objekter.
FetchRow	Row	Ud fra en række i én tabel, fra den ene database, vil metoden finde, den samme unikke række i den tilsvarende tabel, i en anden database.

Klassen SqlGenerator vil være den klasse, som skal, som navnet også antyder, stå for at generere de SQL-kommandoer, som er nødvendige for at kunne fuldføre migreringen. Klassen vil derfor indeholde samtlige metoder, som har noget med denne generering af SQL- kommandoer.

Tabel 9 - SqlGenerator: beskrivelse af metoder

Metoder:	Retur type:	Beskrivelse:
GenerateSqlCall()	List<string>	Vil være den metode, som bliver kaldt for listen med Table-objekter, hvor der skal genereres SQL-kommandoer for.
SqlFromTable()	List<string>	Vil blive kaldt fra GenerateSqlCall(), for samtlige tabeller i den givne liste.
SqlFromRow()	List<string>	Vil blive kaldt fra SqlFromTable(), for samtlige Row-objekter i det givne Table-objekt. Metoden vil sørge for at kalde den korrekte metode, alt efter hvilken ChangeCode den givne row har.
NewRowSqlCall()	string	Skal generere en string, som indeholder en Insert-kommando baseret på den givne række.
UpdateRowSqlCall()	string	Skal generere en string, som indeholder et Update-kommando baseret på den givne række. SQL-kommandoen vil kun opdatere de felter som markeret som ændret.
DeleteRowSqlCall()	string	Skal generere en string, som indeholder et Delete-kommando sletter den givne række.

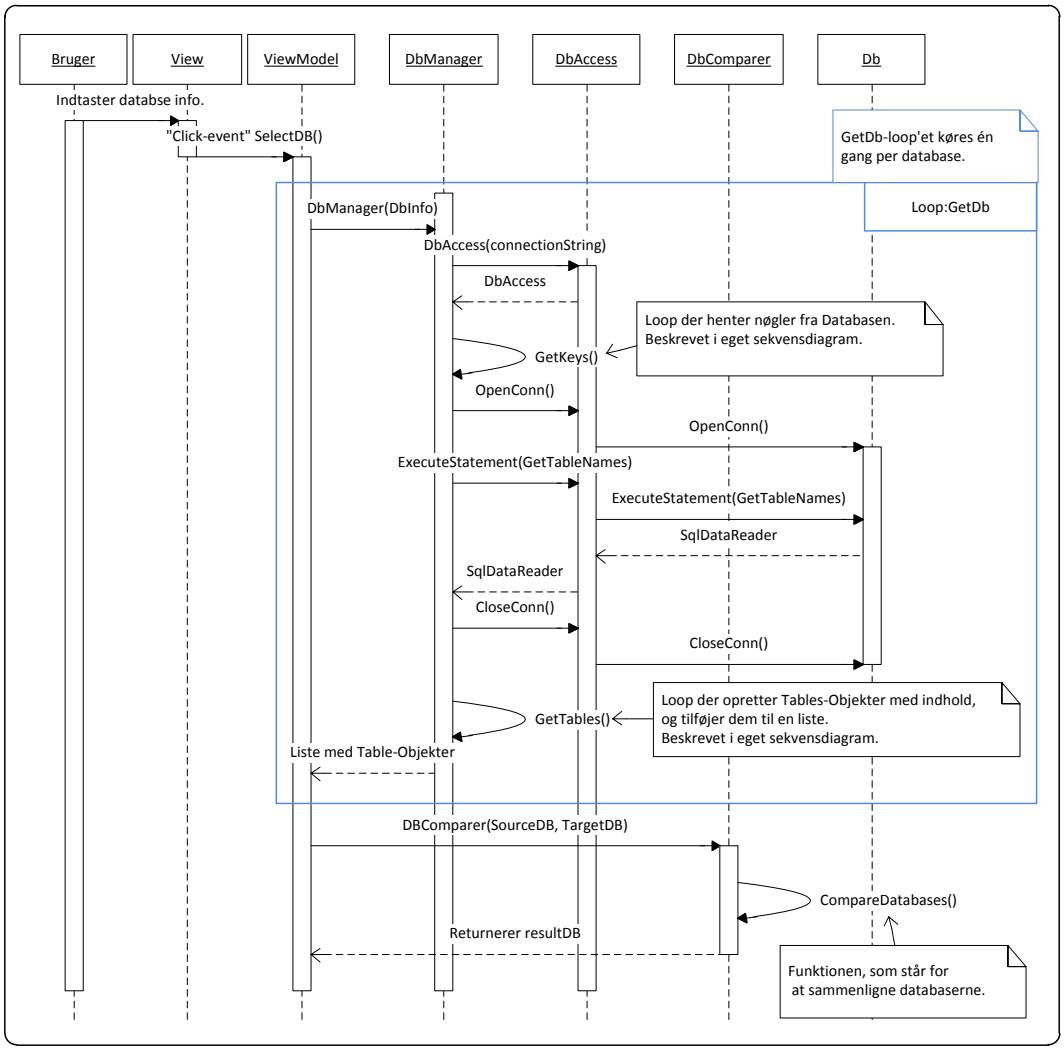
3.3.3 Uddybende sekvensdiagrammer

Når man vil beskrive, hvordan et system reagerer, når en bruger interagerer med det, kan det være meget nyttigt, at benytte et sekvensdiagram. Det, som for brugerens synsvinkel blot er et simpelt klik på en knap, kan i det bagvedliggende system ofte resultere i en lang række af metode-kald gennem systemets dele. Ved at benytte et sekvensdiagram, er det muligt at kortlægge hvilke metoder, der bliver kaldt på hvilke tidspunkter igennem et givent forløb.

Dette afsnit vil uddybe de sekvensdiagrammer fra afsnit 2.4 - Sekvens-Diagram for Use-Case(side 17), som beskriver et hovedscenarie i en Use-Case.

3.3.3.1 Sekvensdiagram for Use-Case 1

Dette sekvensdiagram viser hvilke metoder der vil blive kaldt igennem prototypen, når brugeren følger scenariet beskrevet i Use-Case 1. Scenariet beskriver hvordan brugeren vælger, hvilke databaser der skal sammenlignes.



Figur 25 - Uddybende sekvensdiagram for Use-Case 1

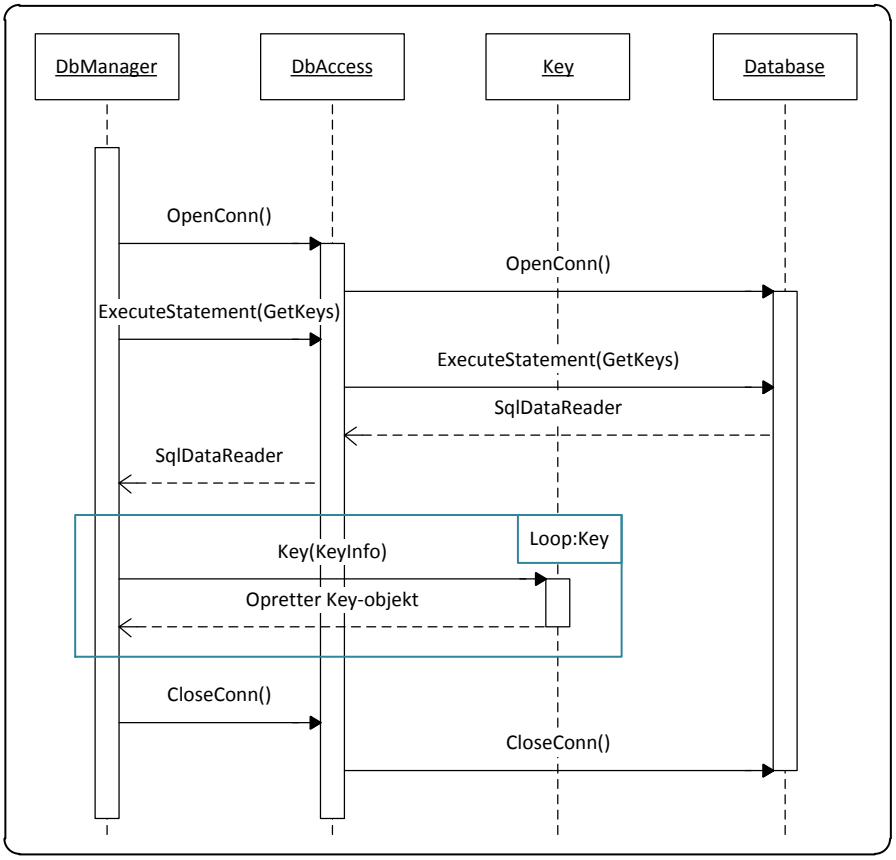
Som det kan ses på Figur 25, er de to loops, GetKeys() og GetTables(), ikke beskrevet på denne figur. De bliver derimod beskrevet i seperate sekvensdiagrammer, for at beholde overskueligheden i denne figur.

Forløbet vil starte ved at brugeren vil blive præsenteret for prototypens første vindue, hvor brugeren skal indtastes de relevante informationer(brugernavn, password og URL til den enkelte database), som skal bruges, når der skal forbindes til de to databaser. Når brugeren klikker på ”ok”-knappen, vil der opfanges et event som vil kalde SelectDB-metoden. Herfra oprettes der en instans af DbManager-klassen for hver af de to databaser. Når DbManager-klassen bliver oprettet skal dens konstruktør kaldes med de indtastede data som argument. I DbManager’ens konstruktør vil der blive oprettet en instans af DbAccess-klassen, som skal bruge de indtastede data til at forbinde den enkelte database. Herefter bliver nøglerne hentet ud af databasen, som beskrevet i afsnit 3.3.3.2 - Sekvensdiagram for Use-Case 1 – GetKeys(side 45). Efter

nøglerne er blevet hentet ud, vil GetTableNames(), som åbner forbindelsen til databasen gennem DbAccess, blive kørt. Når forbindelsen er blevet åbnet, skal ExecuteStatement i DbAccess kaldes med en SQL-kommando som henter samtlige tabelnavne ud fra en database. Efter tabelnavnene er blevet hentet ud, vil forbindelsen til databasen blive lukket, hvilket vil blive gjort gennem CloseConn()-metoden i DbAccess. Den hentede liste vil herefter blive sendt til GetTables(), hvor den skal loopes igennem, og der vil blive returneret en liste med Table-objekter, som igen vil blive returneret til ViewModel-delen. Metoden GetTables() er beskrevet i afsnit 3.3.3.3 - Sekvensdiagram for Use-Case 1 – GetTables(side 46). Når der er blevet lavet en liste med tabeller for begge databaser, vil de blive sendt til metoden CompareDatabases() i DBComparer. Som sidste skridt i forløbet beskrevet i Use-case 1, vil resultatet blive returneret til ViewModel-delen.

3.3.3.2 Sekvensdiagram for Use-Case 1 – GetKeys

Dette sekvensdiagram giver et overblik over hvilke metoder der skal køres, når loopet GetKeys bruges til at hente nøgler ud af databasen.



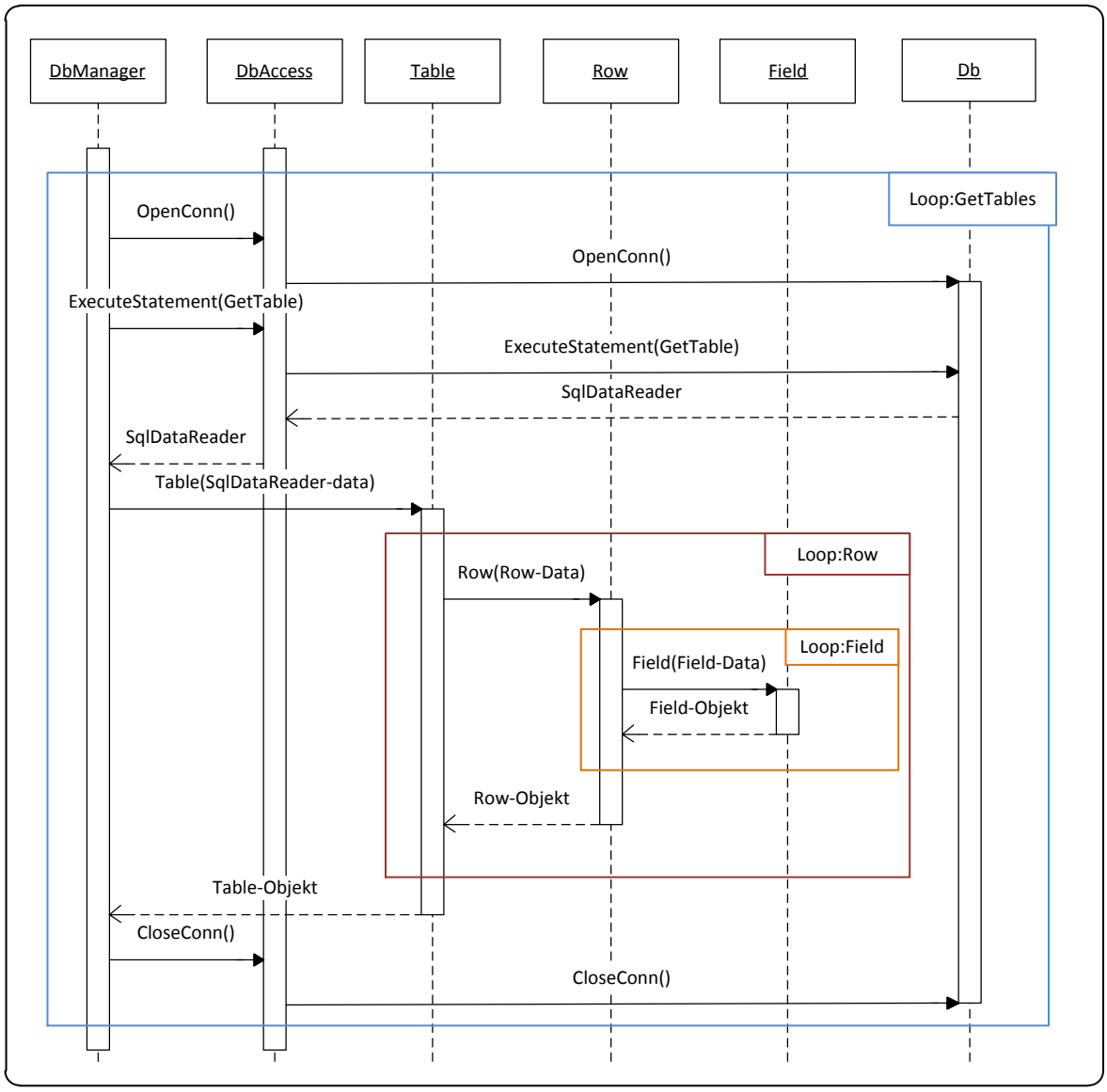
Figur 26- GetKeys-loop fra det uddybende sekvensdiagram for Use-Case 1

Loopet begynder ved, at DbManager-klassen skal kalde metoden OpenConn() i DbAccess til at oprette forbindelsen til den pågældende database. Når forbindelsen er blevet åbnet, vil metoden ExecuteStatement() i DbAccess kaldes med en SQL-kommando, som kan bruges til at hente de relevante nøgler ud. Svaret fra databasen modtages og gemmes i et SqlDataReader-objekt, som skal returneres

tilbage til DbManager-klassen. SqlDataReader-objektet vil blive loopet igennem, og der vil blive oprettet et Key-objekt for hver række, som SqlDataReader'en returnerer. Når der er blevet oprettet et Key-objekt for samtlige fundne rækker, skal CloseConn() i DbAccess kaldes, og forbindelsen til databasen lukkes.

3.3.3.3 Sekvensdiagram for Use-Case 1 – GetTables

Dette sekvensdiagram er lavet med henblik på at give et overblik over hvilke metoder der vil blive benyttet til at oprette prototypens repræsentationer af de to databaser.



Figur 27 - GetTables-loop fra det uddybende sekvensdiagram for Use-Case 1

Den første handling i loopet, vil være at metoden benytter OpenConn() i DbAccess, til at åbne forbindelsen til databasen. Herefter vil ExecuteStatement() blive kaldt med en SQL-kommando, som vil hente samtlige rækker ud for den pågældende tabel. Ligesom i GetKeys-loopet, vil resultatet fra databasen gemmes i en SqlDataReader, som vil blive returneret tilbage til DbManager-klassen.

Inden SqlDataReader bliver behandlet, vil der oprettes et Table-objekt, som skal indeholde prototypens repræsentation af tabellen. DbManager foretager en iterativ gennemgang af den givne SqlDataReader og opretter et Row-objekt for hver af de fundne rækker. Derudover vil der også gennemføres en iterativ gennemgang for hver funden række, hvor der vil blive oprettet et Field-objekt for hver funden felt. Hver gang der oprettes et Field-objekt, vil det blive tilføjet til en liste af Fields i det oprettede Row-objekt. Når alle felter på en række er blevet behandlet, tilføjes det resulterende Row-objekt til det oprettede Table-objekt. Når der er blevet oprettet et Row-objekt for samtlige fundne rækker, og disse er tilføjet til Table-objektet, lukkes forbindelsen til databasen, og Table-objektet tilføjes til en liste. Dette loop bliver kørt for hvert tabelnavn, som den givne liste indeholder.

3.4 Design af GUI

I dette afsnit vil der blive kigget på de designovervejelser samt valg der er blevet truffet, i forbindelse med udarbejdelse af prototypens GUI. Der vil blive taget udgangspunkt i de mock-ups der er blevet lavet, samt de standarder og retningslinjer der findes for usabiliy, som er beskrevet i afsnit 2.6 - Usability(side 20).

Prototypen kommer til være en Windows applikation, som vil blive kørt på brugerens arbejdscomputer. Brugeren er en softwareudvikler med et generelt højt kendskab til brugen af en computer, og som derfor højest sandsynligt besidder evnen til hurtig at sætte sig ind i nye programmer, på baggrund af erfaring med utallige programmer. I og med det ikke er en førstegangs-bruger der er tale om, vil der blive taget højde for dette i udarbejdelsen af prototypens GUI.

Eftersom at programmet, som prototypen kunne ende ud med at blive, er tiltænkt til intern brug hos Stayhard, samt at den definerede bruger er en erfaren computerbruger, vil der ikke være meget idé i at indarbejde standarder for hjælp til handikappede i prototypen.

På baggrund af brugernes erfaringsniveau, og det faktum at det er en prototype der udvikles, vil der i projektet ikke blive lagt vægt på sikkerhed, i forbindelse med interaktion med prototypen. Dette omfatter sikkerhedscheck i forbindelse med at man foretager en handling, der laver ændringer i databasen. Eller for eksempel implementeringen af Undo/Redo-funktioner. Dette vil selvfølgelig være væsentlige funktioner at få implementeret i et færdigt program, uafhængigt af brugerens erfarings niveau, men er altså udeladt i prototypen.

Som det fremgår af prototypens navigationsdiagram(Figur 15, side 23), kommer der kun til at være et enkelt alternativt forløb i prototypen. Så den interaktionen, der kommer til at forgå, er et meget lineært forløb, hvor brugeren mere eller mindre vil blive ført igennem prototypen. Dette gør at indlæring af prototypen bliver relativ hurtig, da det kommer til at være et næsten identisk handlingsforløb man skal igennem, hver gang man benytter prototypen.

Dette øger på samme tid effektiviteten af prototypen, da de få, men rutinepræget handlinger, hurtigt bliver noget, man klikker igennem. Der hvor bruger kommer til at brug sin tid, er når han/hun skal udvælge, hvilke tabeller der skal sammenliges, eller der hvor det skal besluttes hvilke forskelle der skal migreres. Dette kunne muligvis gøres mere automatisk, men det ville hurtigt blive et helt projekt i sig selv. Det kunne for

eksempel ske ved, at applikation indikerede hvilke forskelle man højest sandsynligt ville migrere på baggrund af tidligere kørsler af applikationen.

Det enkle handlingsforløb der er i prototypen er med til at gøre at uforglemmeligheden er god, da brugeren ikke vil være i tvivl om, hvilke handlinger der skal foretages igennem forløbet.

Tilfredsstillelsen for brugeren kan godt komme til at volde problemer, da prototypen godt kan have en meget lang kørselstid, hvis der laves en sammenligning af typen 'Full' mellem to af Stayhards databaser. Derfor er der lavet mulighed for at lave en sammenligning af typen 'Partial' mellem to databaser, så man kan reducere køretid ved at fokusere på kun at sammenligne, hvad der er nødvendigt. Dette er dog noget som Stayhard på forhånd har været opmærksomme på, og derfor forventer, at en fuld sammenligning er noget, der bliver kørt som batch i løbet af natten.

Som nævnt er det når brugeren skal vælge hvilke tabeller, der skal sammenlignes, samt når der skal udvælges hvilke rækker der skal migreres, at brugeren skal bruge tid på at foretage sine valg. Her er det fra Stayhards side blevet ønsket at der bliver benyttet en træstruktur(Figur 5, side 11) til at repræsentere en oversigt over ændringer. Årsagen til dette er at Litium Studio, som de benytter hos Stayhard, gør brug af træstukturen til at organisere menuer. Og da prototypen som et færdigt produkt, er tiltænkt fra Stayhards side som en tilføjelse til Litium Studio, giver det god mening at gøre dette.

Træstrukturen vil dog ikke blive benyttet i fuldt omfang da det ikke den bedste løsning til at gengive indholdet af en database. Generelt vil en relationel database som bliver overført til en træstruktur, ikke have en særlig dyb struktur, og man vil let miste overblikket. Den del af Stayhards database der bliver udgangspunkt i dette projekt, omfatter hovedsageligt tabeller med tilhørende rækker, og vil i en træstruktur have over 230 tabeller yderst i træet, og under den enkelte tabel vil der være op mod en million rækker. Der er ikke umiddelbart noget god måde at viser over en million rækker på så det bliver overskueligt, men at putte dem i en træstuktur hjælper ikke. Derudover vil det ikke være muligt med en træstuktur at gengive data for den enkelte række på en overskuelig måde. Det er valgt at benytte en træstruktur til at vise data ned på tabel niveau. Information om hvad en tabel indeholder, vil blive vist i et separat control-objekt(se Figur 13 - Mock-ups 3 & 4, billedet til højre).

Når brugeren vælger at lave en sammenligning af typen 'Partial', kan der være tabeller, som ikke er i begge tabeller. For at hjælpe brugeren med at se hvilke tabeller der er i SourceDB, men ikke i TargetDB, og omvendt, vil der blive benyttet farvekoder der indikerer dette. For at tage højde for farveblindhed, er der blevet undersøgt, hvilke farver der er let adskillelige, hvad enten man er normaltseende, lider af rødblinde eller grønblinde. Der er fundet frem til, at følgende tre farver er let adskillige: gul, blå og orange[12].

Farvekoderne vil være følgende:

- **Hvid** – tabellen findes i begge databaser.
- **Gul** – Tabellen findes i SourceDB, men ikke i TargetDB.
- **Blå** – tabellen findes i TargetDB, men ikke i SourceDB.

[12] http://www.vos.dk/nye_publikationer/vos_farvesyn_72dpi.pdf

På samme måde er der tilføjet farvekoder på den del, hvor man udvælger hvad der skal migreres. Farvekoderne vil være følgende:

- **hvid** – Der er ingen ændringer i tabellen eller rækken.
- **Orange** – Der er ændringer tabellen eller rækken.
- **Gul** – Det er en tabel eller række som kun findes i SourceDB. Rækker eller tabeller som er nye, da de ikke findes i TargetDB.
- **Blå** – Det er en tabel eller række som kun findes i TargetDB. Rækker eller tabeller som er blevet slettet, da de ikke findes i SourceDB.

Farvekoderne er med til at gøre det hurtigere og lettere, for brugeren at finde frem til de tabeller der skal sammenlignes, samt hvilke forskelle der skal migreres. Selv om det ikke fremgår af mock-ups'ne, vil der være farveskemaer der viser hvilke farver der betyder hvad, på de relevante View.

Som nævnt ovenfor kommer prototypen til at overholde de overordnede krav der er for at opnå god usability, som er nævnt i afsnit 2.6 - Usability(side 20). Derudover kan det på baggrund af design-overvejelserne besluttes at mock-ups'ne generelt set leverer et fornuftigt design. Dog skal der tilføjes farvekoder, for at gøre prototypen mere brugervenlig, samt bedre egnet til brug af farveblinde. Der er også lavet enkelte ændringer i forhold til den planlagte brug af træstrukturen, som er beholdt i det omfang, den er egnet.

3.5 Delkonklusion

I dette kapitel er det endelige design af prototypens GUI, samt funktioner blevet fastlagt. Designet af prototypens funktioner er blevet fundet ved først at klarlægge, hvilke funktioner der er kernefunktioner i prototypen, og herefter opstille nogle løsningmodeller for hver funktion. Herefter er hver løsningsmodel blevet gennemgået, og det løsningforslag som bedst opfylder projektets krav, er blevet valgt som den endelige løsningsmodel.

Da designet var fundet til samtlige kernefunktioner, blev der opstillet en række klassediagrammer til at afspejle hvordan den endelig struktur vil komme til at se ud. Den generelle interaktion med prototypen er blevet uddybet ved brug af sekvensdiagrammer, baseret på klassediagrammer og sekvensdiagrammer for use-cases fra afsnit 2.4 - Sekvens-Diagram for Use-Case(side 17).

På baggrund af analysen i afsnittene 2.6 - Usability(side 20) og 2.7 - Mock-Ups(side 21), er det blevet vurderet hvorvidt de indledende design idéer er gangbare i det endelige design. Enkelte ændringer er blevet lavet til det endelige design for at forbedre prototypens usability.

Kapitel 4 – Implementering & Test

Når man går fra design-fasen til implementeringsfasen, kan man risikere at løbe ind i komplikationer, som der ikke er blevet taget højde for i designet. Det vil så medføre, at man kan være nødt til afvige en smule fra det planlagte design, og helt ændre designet til noget andet.

Disse uforudsete komplikationer har der været nogle stykker af i dette projekt, og disse vil blive beskrevet, samt hvordan de er blevet løst.

Til at starte med vil dette afsnit beskrive, hvordan kernefunktionernes implementering ser ud, samt hvad der har ligget til grund for eventuelle ændringer. Herefter vil andre relevante dele af prototypen blive gennemgået, herunder objekt-klasserne, og eventuelle afvigelser fra designet vil blive beskrevet.

Da der hurtigt kan være mange små ændringer mellem design og implementering, er det valgt kun at fokusere på at beskrive de væsentligste afvigelser.

Når prototypen er blevet gennemgået, vil der blive beskrevet hvilke tests der er blevet lavet på prototypen, samt resultatet af de respektive test.

4.1 Implementering af kernefunktioner

Det endelige flow i systemet endte med at ligge sig meget tæt på mod det flow, som blev beskrevet i afsnit 3.1 – Beskrivelse af prototypens opbygning(side 31), og der er derfor ingen grund til at lave en ny figur til dette afsnit, som viser det samme som den tidligere figur.

Der har dog været nogle ændringer til de enkelte kernefunktioner, og der vil i dette afsnit blive beskrevet, hvordan implementeringen af disse funktioner har afveget fra det oprindelige design, samt hvorfor.

4.1.1 Getkeys()

Implementeringen af denne funktion, er endt med at blive en smule anderledes end beskrevet i afsnit 3.2.1 – GetKeys()(side 32). Hoved-ændringen ligger i, at GetKeys()-metoden er blevet splittet op i tre metoder, én for hver nøgletype. Hovedgrunden til dette er, at der opstod problemer i forbindelse med fremmed-nøgler, som vil blive beskrevet i afsnit 4.2.3 - GetRequiredTablesAndPrioritize()(side 56). Det medførte, at der opstod et behov for at hente ekstra informationer ud, når fremmed-nøglerne skulle oprettes i systemet. SQL-kommandoen ville være nødt til at hente en række informationer ud, som var nødvendige i henhold til fremmed-nøgler, men som primær- og unikke-nøgler ikke havde brug for. Ved at opdele metoden i tre, blev det muligt at specificere hvordan hver nøgletype skulle hentes og oprettes i systemet.

Mens primær- og unikke-nøgler ikke behøvede nogen ekstra information i forhold til det oprindelige design, krævede den nye opdeling en ændring til SQL-kommandoerne, således at det blev muligt at hente nøglerne ud for den specifikke nøgletype. Dette blev gjort ved at lave et join mellem to views, således at det blev muligt at binde den enkelte nøgle sammen med en nøgletype. Dette fjernede også behovet for at bruge regular expressions til at bestemme nøgletyper, hvilket også viste sig at indebære en vis usikkerhed. Funktionen var nemlig afhængig af, at udviklerne benyttede nogle bestemte opbygninger af nøglens 'constraint name', hvilket desværre ikke altid blev overholdt. Det medførte, at man var nødt til at indføre

ekstra undtagelser, når man fandt en nøgle, som ikke var lavet korrekt. Dette er helt undgået, da prototypen nu bruger databasens egen definition af hvilken nøgletype de forskellige nøgler har.

```
SELECT
    CONSTRAINT_NAME = KCU.CONSTRAINT_NAME,
    TABLE_NAME = KCU.TABLE_NAME,
    COLUMN_NAME = KCU.COLUMN_NAME ,
    ORDINAL_POSITION = KCU.ORDINAL_POSITION
FROM
    INFORMATION_SCHEMA.KEY_COLUMN_USAGE KCU
INNER JOIN INFORMATION_SCHEMA.TABLE_CONSTRAINTS TC
    ON KCU.CONSTRAINT_NAME = TC.CONSTRAINT_NAME
where TC.CONSTRAINT_TYPE = 'PRIMARY KEY'
```

Figur 28 - SQL-kommando til at hente primær-nøgler fra en database

Figur 28 ovenfor viser den SQL-kommando, som blev udfærdiget til dette formål. Kommandoen er næsten identisk med den funktion, som henter unikke-nøgler, hvor 'PRIMARY KEY' er ændret til 'UNIQUE'. View'et "INFORMATION_SCHEMA.KEY_COLUMN_USAGE" indeholder samtlige af de værdier, som er nødvendige for at oprette en nøgle, mens "INFORMATION_SCHEMA.TABLE_CONSTRAINTS" forbinder en nøgles 'constraint name' med en nøgletype. Et join af disse to tabeller på nøglernes 'constraint name' vil give et resultatsæt, som indeholder samtlige af de nødvendige værdier for en nøgle, samt en nøgletype for hver nøgle.

Kommandoen for fremmed-nøgler er dog en smule mere kompliceret, da man er nødt til at hente data fra tre forskellige views(og to gange fra de ene af de tre).

```
SELECT
    CONSTRAINT_NAME = C.CONSTRAINT_NAME,
    TABLE_NAME = FK.TABLE_NAME,
    COLUMN_NAME = CU.COLUMN_NAME,
    ORDINAL_POSITION = CU.ORDINAL_POSITION,
    PRIMARY_KEY_SCHEMA_TABLE = PK.TABLE_SCHEMA,
    PRIMARY_KEY_TABLE_NAME = PK.TABLE_NAME
FROM
    INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS C
INNER JOIN INFORMATION_SCHEMA.TABLE_CONSTRAINTS FK
    ON C.CONSTRAINT_NAME = FK.CONSTRAINT_NAME
INNER JOIN INFORMATION_SCHEMA.TABLE_CONSTRAINTS PK
    ON C.UNIQUE_CONSTRAINT_NAME = PK.CONSTRAINT_NAME
INNER JOIN INFORMATION_SCHEMA.KEY_COLUMN_USAGE CU
    ON C.CONSTRAINT_NAME = CU.CONSTRAINT_NAME
```

Figur 29 - SQL-kommando som benyttes til at hente fremmed-nøgler ud

De ekstra data, som fremmed-nøgler havde brug for, var information omkring hvilken tabel den refererede til. For at finde frem til dette, benyttes "INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS", som holder styr på fremmed-nøglerne i databasen. View'et indeholder for hver nøgle to 'constraint name'-felter. Ét for fremmed-nøglen selv, og ét for den nøgle, som fremmed-nøglen refererer til. For at binde de to 'constraint name'-felter sammen med de relevante tabel-navne, udføres der to joins på "INFORMATION_SCHEMA.TABLE_CONSTRAINTS", hvor der først joines på fremmed-nøglens egen 'constraint name' og bagefter den refererede nøgles 'constraint name'. På den måde får man et resultat-sæt, som indeholder størstedelen af de data, man har brug for. For at få de sidste, udføres der et join på "INFORMATION_SCHEMA.KEY_COLUMN_USAGE", hvor der joines på fremmed-nøglens 'constraint name'.

På trods af, at man er nødt til at joine flere forskellige views, kan man med én SQL-kommando hente samtlige data ud, som man skal brug for at oprette en fremmed-nøgle.

4.1.2 FetchRow()

Implementeringen af denne funktion er endt med at ligge meget tæt op af det design, som blev beskrevet i afsnit 3.2.3 – FetchRow()(side 33). Den største forskel ligger i, at der benyttes en LINQ-sætning til at finde frem til de rækker, hvor den pågældende nøgle-værdi svarer overens med den række, man sammenligner med.

```
List<Row> foundRows = rows.Where(row1 =>
                        row1.Fields[keys[keyIndex].ColumnName()].Value().Equals(value)).ToList();
```

Figur 30 - LINQ-sætning, som benyttes til at sortere i liste af rækker

De argumenter, som den rekursive metode tager, er følgende:

- **row** – Række fra SourceDB. Ud fra denne række, skal den samme unikke række findes i listen af rækker fra tabellen fra TargetDB.
- **rows** – En liste af rækker fra TargetDB.
- **keys** – En liste af nøgler fra de pågældende tabeller. Inden metoden køres, kontrolleres det, at de to tabeller har ens nøgler. Det er derfor vilkårligt, hvilken database nøglerne tages fra.
- **keyIndex** – Starter på '0'. Forøges med én, for hver gennemkørsel af den rekursive funktion.
- **keyType** – sættes til at være typen af den første nøgle i 'keys'.

LINQ-sætningen skal læses således, at der fra 'rows' hentes alle de elementer, hvor den givne sætning returnerer 'true'. For at gøre sætningen lidt mere overskuelig, vil den blive brudt ned i mindre dele.

Hver række gemmer dens tilknyttede felter i et 'Dictionary <string,Field>', hvor kolonne-navnet for de enkelte felter benyttes som nøgle. "keys[keyIndex].ColumnName()"-delen henter kolonne-navnet på den nøgle, som metoden er nået til. Kolonne-navnet benyttes så i rækkens 'Dictionary', således at det ønskede felt findes. Når feltet er fundet, sammenlignes værdien for dette felt, med værdien for det samme felt i 'row'. Hvis de to værdier er ens, returnerer sætningen 'true', og rækken tilføjes til listen 'foundRows', som så bruges som argument, til at kalde metoden igen, såfremt den skal kaldes flere gange.

Til at bedømme, hvorvidt den rekursive funktion skal kaldes igen, bliver der kigget på om typen af næste nøgle i 'keys' er den samme som 'keyType'. Hvis typerne er ens, kaldes funktionen igen, og ellers returneres den fundne række, såfremt der er fundet et match.

For at optimere køretiden for funktionen, fjernes de fundne rækker fra listen af rækker fra TargetDB, således at der for hver gennemkørsel vil være én række mindre i TargetDB's liste af rækker. Når samtlige rækker fra SourceDB er bearbejdet, vil TargetDB's liste kun indeholde de rækker, som er slettet i SourceDB. Disse markeres derfor som 'deleted', ved at ændre deres 'ChangeCode' til '3'. Kørsels-tiden for FetchRow() vil derfor forkortes, for hver gennemkørsel af det rekursive forløb.

CompareDatabases()

Da denne funktion blev implementeret, endte opbygningen med at ligge meget tæt op ad det fundne design i afsnit 3.2.4 – CompareDatabases()(side 34). Som nævnt i afsnittet, bestod grunddesignet af denne funktion af en blanding mellem de to løsningsmodeller opstillet i design-afsnittet. Den ene løsning bestod i at al sammenlignings-logikken var samlet i én funktion, mens den anden løsning var hvor dele, som var meget specifikke for det enkelte objekt, blev lagt ind i objekts klasse, så det selv stod for sammenligningen.

Den endelige implementering er opbygget således, at den yderste del består af en metode, som står for håndteringen af tabeller. Udover at gennemløbe listerne af tabeller, står denne metode også for at sikre, at de to tabeller, som skal sammenlignes, har ens nøgler. Har de ikke dette, bliver tabellerne sprunget over, da det ikke vil give mening at sammenligne dem.

De tabeller, som når igennem nøglekontrollen, sendes videre til en anden metode, som står for at håndtere sammenligningen af to tabeller. I denne metode gennemløbes listen af rækker fra SourceDB, og FetchRow()-metoden køres for hver række. Såfremt FetchRow() finder den givne række i TargetDB, sammenlignes rækkerne fra de to databaser. Sammenligningen udføres, ved at kalde ”CompareTo” på rækken fra SourceDB, med rækken fra TargetDB, som argument. ”CompareTo”-funktionen bliver kørt som en del af det ”Row”-objekt, som det bliver kaldt på, hvilket i dette tilfælde vil give funktionen adgang til attributterne i rækken fra SourceDB.

Funktionen gennemgår alle rækkens felter, og finder frem til det tilsvarende felt i rækken, som blev givet som argument. Når det tilsvarende felt er blevet fundet, kaldes feltets ”CompareTo”-funktion med det fundne felt. I denne funktion bliver de to felters attributter sammenlignet, og hvis der bare er én attribut, hvor der er en afvigelse, sættes feltets ’ChangeCode’ til ’1’, som markerer feltet som ændret. Dette medfører så, at den tilhørende rækkes ’ChangeCode’ ændres til ’1’, som igen opdaterer tabellens ’ChangeCode’ til ’1’.

Måden, hvorpå ”CompareTo” bliver kaldt, medfører det, at samtlige ændringer der sker i ”CompareTo”-funktionen, vil foregå i det objekt, som ”CompareTo” er blevet kaldt på. Det betyder, at når ’ChangeCode’ ændres i en ”CompareTo”-funktion, vil ændringen ske i det objekt, hvorpå ”CompareTo” er kaldt. Dette medfører, at det ikke er nødvendigt at oprette nye instanser af objekterne, for at gemme ændringerne, da de laves direkte i de givne objekter.

Når alle felterne i rækken fra SourceDB er blevet gennemgået, undersøges det, om der er nogle felter i rækken fra TargetDB, som ikke er sammenlignet endnu. Hvis der opdages nogle felter af denne type, markeres de med ’ChangeCode’ ’3’, som svarer til ’deleted’. Denne fremgangsmåde gentages, hvor der kigges på rækker, i stedet for felter, og til sidst også for tabeller, således at det sikres at samtlige tabeller, rækker og felter fra begge databaser er blevet gennemgået.

Når samtlige felter i en række er gennemgået, tilføjes rækken til en resultat-tabel, som indeholder alle de ændrede rækker. I sidste ende vil tabellen repræsentere resultatet af sammenligningen mellem de to tabeller fra SourceDB og TargetDB, og vil være en sammenlægning mellem de to. For at samle alle ændringerne, tilføjes alle resultat-tabellerne til et ’Dictionary’, som repræsenterer en sammenlægning af de to databaser, som indeholder samtlige tabeller, både ændrede og uændrede, fra begge databaser.

4.1.3 SqlGenerator()

Denne funktion er blevet implementeret, ved at benytte en af løsningsmodeller, som blev valgt fra i afsnit 3.2.5 – SqlGenerator()(side 35). Funktionen er ændret således, at de genererede SQL-kommandoer ikke gemmes i en 'string', men i et objekt. Se uddybende forklaring i afsnit 4.2.2 - SqlCall-objekt(side55).

Selve funktionen er opbygget ved at objekternes 'ChangeCode' bliver brugt til at bestemme, hvilken type SQL-kommando der skal laves. Hvis en tabel er markeret som 'Changed', sendes den videre til en funktion, som gennemløber tabellens liste af rækker. For hver række undersøges det, hvilken 'ChangeCode' den har, og baseret på resultatet bestemmes det, hvilken type SQL-kommando der skal oprettes.

Rækken sendes så videre til den funktion, som står for at oprette den givne kommando-type, hvor der, baseret på 'ChangeCode' for hvert felt i rækken, dynamisk bliver oprettet en SQL-kommando, som gemmes i et 'sqlCommand'-objekt. 'SqlCommand'-objektet vil så indeholde en SQL-kommando, hvor værdien for hvert felt, som skal indsættes, opdateres eller slettes, bliver gemt som en parameter, mens den egentlige værdi vil blive tilføjet til en parameter-liste.

Fordelen ved at gøre det på denne måde er, at det bliver nemmere at behandle det givne data dynamisk, samt sikkerhed imod SQL-injections, når det bliver implementeret, at brugere kan ændre i SQL-kommandoen[13].

Når samtlige SQL-kommandoer er blevet genereret, returneres de i form af en liste af SqlCalls. Denne liste vil herefter blive sendt til en funktion i DbManager-objektet for TargetDB, hvor listen gennemgås, og hvert SqlCall bliver sendt til en funktion i DbAccess, som kører SQL-kommandoen på databasen, som svarer med antallet af rækker, der er blevet berørt af den kørte kommando.

4.2 Implementering af anden relevant kode

Prototypens klasser som er beskrevet i afsnit 3.3.2 - Klassediagrammer(side 37), er stort set blevet implementeret i overensstemmelse med før nævnte klassediagrammer. Enkelte ændringer og tilføjelser er dog fundet nødvendige i forbindelse med implementering af prototypen. Beskrivelser af de væsentligste ændringer er at finde i detaljer nedenfor.

4.2.1 Row-objekt

Oprindeligt var det planen, at Row-objektet skulle indeholde en liste af Field-objekter, et for hver kolonne i rækken. Men i forbindelse implementering af FetchRow(se afsnit 4.1.2 - FetchRow()(side 52), blev det klart at det var upraktisk og uoverskueligt at have Field-objekterne i en liste. Det blev derfor besluttet at benytte 'Dictionary' i stedet for 'List', da det gør koden mere overskuelig, at der ikke skal laves et loop til at søge listen igennem. I og med at der i størstedelen af tabellerne er mere end 5 kolonner, er det i langt de fleste tilfælde hurtigere at søge i et 'Dictionary'[14], frem for en liste.

[13] http://www.codeproject.com/Articles/9378/SQL-Injection-Attacks-and-Some-Tips-on-How-to-Prev
[14] http://www.codebullets.com/dictionary-vs-list-lookup-658

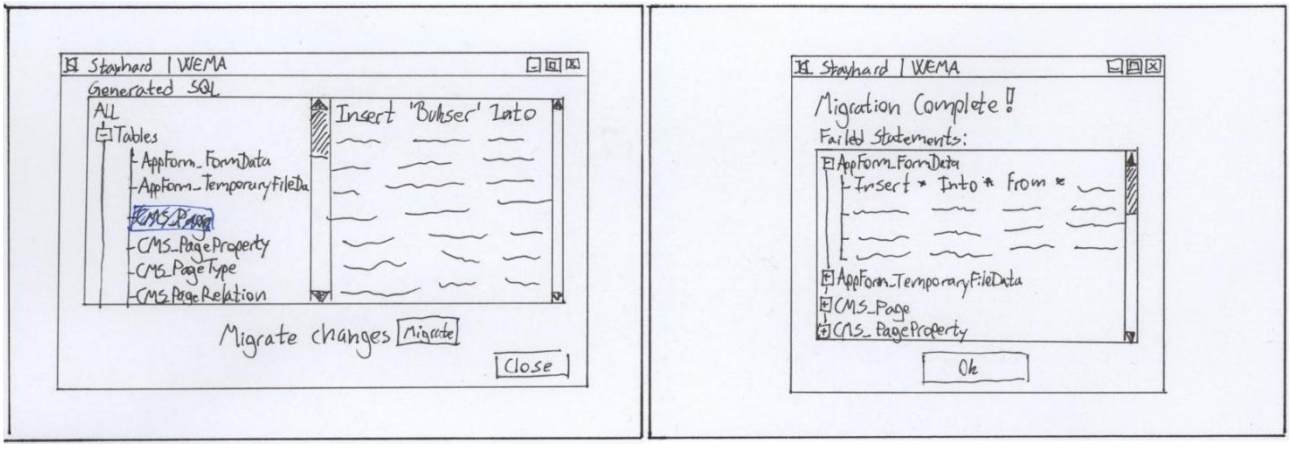
4.2.2 SqlCall-objekt

I forbindelse med at SqlGenerator-klassen blev implementeret, viste sig at de ikke var nogen god idé blot at have de genererede SQL-kommandoer i en liste af strings. Der var ikke blevet tænkt på, hvor mange SQL-kommandoer der ville blive genereret, og at det efterfølgende ville være nærmest umuligt finde rundt i dem uden at kunne differentiere dem. Der er derfor blevet oprettet et SqlCall-objekt for at skabe tilhørsforhold mellem en SQl-kommando, og den tabel som den relaterer til. Et SqlCall-objekt indeholder følgende attributter:

Tabel 10 - SqlCall-Objekt: beskrivelse af attributter

Attributter:	Type:	Beskrivelse:
Statement	SqlCommand	Den genererede SQL-kommando.
statementType	int	Fortæller om SQL-kommandoen påvirker på tabel-, række- eller felt-niveau, samt om det er en UPDATE-, INSERT- eller DROP-kommando. En tabel er 0x, række er 1x, felt er 2x på samme måde er UPDATE x0, INSERT x1, DROP x2. Så en ny tabel er 10 og et opdateret felt vil være 21.
tableName	string	Navnet på den tabel som kommandoen er tilknyttet.
RowsAffected	int	Holder styr på antallet af rækker som er blevet påvirket af at SQL-kommandoen er blevet kørt.

Med SqlCall, er det muligt at opdele SQL-kommandoerne når de skal vises i GUI'et, så man fået et bedre overblik over kommandoerne. Det er muligt at differentiere ud fra hvilken tabel kommandoerne påvirker, om det er på felt-, række- eller tabel-niveau, hvilken type SQL-kommando der er tale om, eller en kombination af disse. Et mock-up af ændringerne til GUI'et er afbilledet i Figur 31 - Alternativer til mock-ups 5 & 6. Det vil også være muligt at præcisere, hvis der opstår en fejl når SQL-kommandoerne bliver kørt.



Figur 31 - Alternativer til mock-ups 5 & 6

Tilføjelsen af SqlCall-objektet til prototypens er en af de ændringer i løbet af implementeringen, som har bidraget til at forbedre prototypens usability. Behovet for at differentiere resultatet af SQL-kommandoer burde nok være opdaget tidligere, men separation af prototypens moduler gjorde at det ikke var noget problem, at tilføje den i løbet af implementeringen.

4.2.3 GetRequiredTablesAndPrioritize()

I forbindelse med at der skulle migreres forskelle mellem databaserne opstod der problemer, når man prøvede at indsætte en række i en tabel, hvis fremmed-nøgle endnu ikke var blevet indsat. Fejlen opstod ligeledes når man ville slette en række, som en eller flere andre rækker refererede til som fremmed-nøgle. For at undgå denne fejl, var det nødvendigt at prioritere tabellerne, således at SQL-kommandoerne bliver kørt først på de tabeller, hvor der er flest fremmed-nøgler som refererer til. Metoden GetRequiredTablesAndPrioritize() som er at finde i DbManager løser dette problem.

GetRequiredTablesAndPrioritize() fra DbManager-klassen,er en overloaded metode, som består af to dele, en hovedmetode og en overloaded metode. Hovedmetoden tager en reference til et 'Dictionary<string, Table>' som argument, og har ingen return type. I metoden bliver der oprettet en liste af strings som indeholder navnene på de tabeller, der er i det 'dictionary', som argumentet refererer til. Herefter bliver der lavet en foreach på listen af nøgler, og for hver nøgle bliver den overloadede metode kørt.

Den overloadede metode tager følgende argumenter:

- **ref Dictionary<string, Table>tableDict**– Reference til Dictionary af tabelnavne, samt tabel-objekter.
- **stringtableName** – Navnet på en tabel.
- **List<string>visitedTables = null** – Liste som skal indeholde de tabeller som er blevet behandlet.

Den overloadede metode har returntype void, hvilket betyder at den ikke returnerer noget, efter endt gennemkørsel.

Indledende i den overloadede metode bliver der oprettet en 'List<Key>keyList', som indeholder fremmed-nøgler fra den tabel i 'tableDict', som stemmer overens med 'tableName'. I tilfælde af at en tabel ikke indeholder nogen fremmed-nøgler bliver 'return' kaldt, metoden bliver stoppet og returnerer til metoden, hvori metoden var blevet kaldt.

En ny 'List<string>tablesVisited' bliver oprettet, og hvis 'visitedTables' er forskellig fra nul, bliver 'tablesVisited' sat lig 'visitedTables'. Dette bliver gjort da det var nødvendigt at have listen lokalt i metoden.

Herefter vil indholdet af 'keyList' blive gennemgået, for hvert Key-objekt 'key', med en 'foreach'. I de tilfælde, hvor 'tablesVisited' indeholder navnet på den tabel som fremmed-nøglen 'key' peger på, vil 'continue' blive kaldt. Formålet med dette er at undgå at ende i et uendeligt loop, som ville forekomme hvis en tabel har en fremmed-nøgle som refererer til en primær-nøgle i samme tabel. Navnet på den tabel, som fremmed-nøglen 'key' peger på bliver fundet med metoden PrimaryKeyTableName(), som findes i Key-objektet.

Der oprettes et Table-objekt 'primaryTable', som skal bruges til at holde værdien af den tabel, som fremmed-nøglerne peger på. Herpå startes den if-sætning, som hovedfunktionaliteten i GetRequiredTablesAndPrioritize() ligger i, se koden nedenfor i Figur32.

```
Table primaryTable;
if (tableDict.TryGetValue(key.PrimaryKeyTableName(), out primaryTable))
{
    tableDict[key.PrimaryKeyTableName()].Priority++;
    GetRequiredTablesAndPrioritize(ref tableDict, key.PrimaryKeyTableName(), tablesVisited);
}
else
{
    primaryTable = GetTable(key.PrimaryKeyTableName());
    primaryTable.Priority++;
    tableDict.Add(primaryTable.FullName,primaryTable);
    GetRequiredTablesAndPrioritize(ref tableDict, key.PrimaryKeyTableName(), tablesVisited);
}
```

Figur32 - If-sætningiGetRequiredTablesAndPrioritize()

I if-sætningen i figuren ovenfor, bliver der først kigget på om 'tableDict' indeholder den tabel, som man får når man kører 'key.PrimaryKeyTableName()'. Hvis tabellen findes 'tableDict', vil Table-objektet 'primaryTable' blive sat til at være en instans af den pågældende tabel. Der kan opstå tilfælde hvor tabellen ikke findes i 'tableDict' og den skal derfor hentes, hvilket er formålet med else-delen af denne if-sætning.

Herefter vil 'priority' for tabellen, som i 'tableDict' hedder det sammen som 'key.PrimaryKeyTableName()', blive talt op med én. 'priority' bestemmer hvilken prioritet den pågældende tabel har, når elementer skal indsættes eller slettes i databasen. Jo højere værdi jo højere prioritet. I sidste ende vil de tabeller, der har flest fremmed-nøgler som peger på sig, både direkte og indirekte, være dem der har den højeste prioritet.

Efterfølgende vil GetRequiredTablesAndPrioritize() blive kaldt med en reference til 'tableDict', 'key.PrimaryKeyTableName()' samt 'tablesVisited'. Så man har en rekursiv funktion, der tager udgangspunkt i en fremmed-nøgle og den tabel som den peger på. Derfra gennemgås de tabeller, som den givne tabels fremmed-nøgle, samt de efterfølgende tabellers fremmed-nøgler, peger på. Løbende vil prioriteten på de tabeller som man støder på blive talt op.

Hvis forudsætning i if-sætningen ikke er overholdt, betyder det at tabellen, som fremmed-nøglen refererer til, ikke findes i 'tableDict'. Tabellen bliver hentet ved hjælp af metoden GetTable() fra DbManager-klassen, og gemt i Table-objektet 'primaryTable'. Herefter bliver 'priorty' for tabellen talt op med én, og tabellen bliver tilføjet til 'tableDict'. Herefter bliver GetRequiredTablesAndPrioritize() kaldt fuldstændig på samme måde, og af de samme grunde, som er nævnt i if-sætningen.

4.3 Test

Når man udvikler et program, er det vigtigt, at man sørger for at gennemteste det, således at man kan formindske antallet af fejl, så meget som muligt, inden programmet når slutbrugeren. For at sikre, at man får testet programmet på den ønskede måde, kan man opstille en teststrategi, som beskriver hvilke typer test man vil køre, samt hvordan de skal køres.

4.3.1 Testmetoder

Når man skal teste et system, er der forskellige måder at gøre det på. Overordnet kan man bl.a. benytte en box-opdeling, til at beskrive de forskellige typer af tests, herunder white-box og black-box[15].

4.3.1.1 White-box test

White-box dækker over en række metoder som programmører kan benytte til at sikre at deres kode fungerer efter hensigten. Generelt for white-box tests er, at de benyttes til at teste den interne struktur i en applikation. White-box test kan udføres på tre niveauer i applikation:

- **Unit test** – En test af den enkelte enhed i et system. Kunne for eksempel være metoder eller klasser.
- **Integrations test** – Benyttes til at teste hvorledes et systems enheder kommunikerer indbyrdes.
- **System test** – En overordnet test af det samlede system.

Ud af disse tre benyttes unit test-niveauet oftest, når der skal udføres en white box test[16]. Endvidere anslås det at man ved brug af unit test kan opdage op mod 65 % af alle fejl[17].

Som nævnt i afsnit 3.3.1 - MVVM(side 36), er en af fordelene ved at benytte MVVM, at det gør det muligt at brug unit test på et systems funktioner uafhængigt af systemets GUI.

4.3.1.2 Black-box test

Black-box dækker over de test-metoder, hvor testeren ikke nødvendigvis kender det bagvedliggende system. Der bliver derfor typisk fokuseret på de input og output, som det givne system har.

Ligesom med white-box test, kan black-box test udføres på flere forskellige niveauer i en applikation. Fordelen med black-box test er, at kompleksiteten af testen ikke nødvendigvis stiger, når man tester på forskellige niveauer. Dette medfører, at det bliver mere normalt at bruge black-box test, jo højere op i applikationens niveau, man når[18].

4.3.1.3 Teststrategi for projektet

Til at teste prototypen, vil der i dette projekt blive benyttet en kombination af white-box og black-box tests, og der vil blive lagt vægt på at teste de scenarier, som er blevet beskrevet i de opstillede Use-Cases i afsnit 2.3 - Use-Case(side 14).

Prototypens metoder vil blive testet ved at bruge unit-tests, da dette også vil hjælpe fremtidige udviklere, således at de kan kontrollere, at de forskellige dele fortsat virker efter hensigten, efter en eventuel udvidelse.

Prototypens overordnede funktionalitet vil blive testet ved bruge af black-box test, som bør udføres af personer uden kendskab til applikationen, og med IT-kvalifikationer, som svarer til en typisk bruger af prototypen.

[15] http://www.ece.cmu.edu/~koopman/des_s99/sw_testing/#concepts
[16] http://softwaretestingfundamentals.com/white-box-testing/
[17] http://www.chaudhary.org/WhiteBox.pdf
[18] http://softwaretestingfundamentals.com/black-box-testing/

Det kan være utroligt svært at fange samtlige fejl i et program, inden det frigives til slutbrugeren, men ved at benytte disse testmetoder, kan man minimere antallet af fejl i programmet, og de mest kritiske fejl bør blive fanget og rettet.

4.3.2 Resultat af black-box test

Der er blevet udført en black-box test for hvert af de tre hoved-scenarier, beskrevet i de opstillede Use-Cases i afsnit 2.3 - Use-Case(side 14).

4.3.2.1 Test af use-case 1

Hvad skal der testes:	Prækondition:	Forventet Resultat:	Faktisk resultat:	Status:
Brugeren indtaster forbindelses-info korrekt.	Brugeren skal kunne forbinde til begge databaser.	Prototypen forbinder til databasen, og spørger som hvilken sammenlignings-type der ønskes.	Prototypen forbinder korrekt til databasen, fortsætter direkte til at foretage en 'Full' sammenligning.	OK
Brugeren indtaster forbindelses-info forkert.	Brugeren skal kunne forbinde til begge databaser.	Prototypen fortæller brugeren, at der ikke er forbindelse til databasen, og stopper programmet.	Prototypen skriver, at det ikke har været muligt at oprette forbindelse, og afslutter operationen.	OK
Brugeren vælger 'Partial' sammenligning	Brugeren skal have indtastet korrekt data for begge databaser.	Prototypen præsenterer brugeren for et vindue, hvor brugeren kan vælge hvilke tabeller der skal sammenlignes.	Prototypen fortsætter direkte til sammenligningen, og brugeren bliver ikke spurgt hvilke tabeller der skal sammenlignes.	FEJL

Fejlbeskrivelse:

- **Brugeren vælger 'Partial' sammenligning** – Fejlen opstår, grundet manglende GUI.

4.3.2.2 Test af use-case 2

Samtlige dele af dette test-case ville fejle, da de alle ville være afhængige af prototypens GUI. Eftersom dette mangler, kan det ikke lade sig gøre at gennemføre denne test.

4.3.2.3 Test af use-case 3

Hvad skal der testes:	Prækondition:	Forventet Resultat:	Faktisk resultat:	Status:
Brugeren ønsker at se resultatet af sammenligningen.	Brugeren skal have valgt én eller flere forskelle, som skal flyttes mellem miljøerne.	Brugeren bliver præsenteret af et vindue, hvor samtlige SQL-kommandoer er præsenteret.	Prototypen spørger blot, som den skal påbegynde migrering, uden brugeren vælger nogen forskelle.	FEJL
De valgte forskelle skal migreres.	Der skal have været mindst én forskel i den del af databasen, som brugeren har valgt.	TargetDB skal opdateres således, at den afspejler de ændringer, som brugeren har valgt.	De valgte ændringer bliver flyttet over i TargetDB, og samtlige valgte ændringer er repræsenteret i TargetDB.	OK

Fejlbeskrivelse:

- **Brugeren ønsker at se resultatet af sammenligningen** – Denne fejl opstår, også, grundet manglende GUI.

4.4 Delkonklusion

Der er i dette kapitel blevet beskrevet, hvordan den overordnede funktionalitet er blevet implementeret, baseret på analyse- og design-kapitlerne. Kapitlet beskriver hvilke ændringer der har været mellem design og implementering, samt hvad der har ligget til grund for disse ændringer.

Der er blevet udviklet en prototype, som kan tage to databaser, finde eventuelle forskelligheder mellem de to, og flytte fundne forskelligheder over i den ene database.

Endvidere er der blevet lagt en test-strategi, som beskriver hvordan det var hensigten, prototypen skulle testes. Grundet uheldig planlægning, er der kun blevet udført en black-box test.

Den uheldige planlægning har også medført, at der ikke er lavet en fuld implementering af MVVM, og MVVM Light-frameworket, da der kun er implementeret Model-delen i prototypen. Dette har resulteret i manglende GUI.

Kapitel 5 – Konklusion

Dette kapitel vil indeholde den samlede konklusion for det samlede projekt, baseret på en opsummering af delkonklusionerne fra de enkelte kapitler i rapporten.

Til at runde kapitlet og rapporten af med, vil der være et afsnit, som beskriver fremtidige udvidelser af prototypen. Afsnittet vil også indeholde en beskrivelse af hvilke dele af prototypen, som ikke nåede at blive implementeret.

5.1 Opsummering af rapport

For at give et hurtig overblik rapportens kapitler, vil delkonklusionerne blive opsummeret.

5.1.1 Analyse

Som det første i kapitlet bliver der overordet defineret hvilke input og output systemet har, i forhold til brugeren og eksterne datakilder. Herefter bliver der udformet en kravspecifikation hvor der bliver kigget på de funktionelle og de ikke funktionelle krav til prototypen. På baggrund af kravspecifikationen er der blevet udformet en projektafgrænsning, som reducerer projektets omfang til en realistisk størrelse, set i forhold til projektets tidshorisont.

- **FK1** – Sammenligning af to databaser
- **FK2** – Visuel præsentation af forskelle mellem databaserne
- **FK3** – Migrering af forskelle mellem databaserne

Der er herefter udformet use-case-beskrivelser, som beskriver brugerens interaktion med prototypen, i henhold til de opstillede krav. Derudover er der brugt use-case-beskrivelser, sekvensdiagrammer, en domænemodel samt mock-ups til at uddybe prototypens opbygning. Der bliver også kigget på hvilke overvejelser man skal gøre sig, når man udvikler applikationer med brugergrænseflader.

Afsluttende i kapitlet bliver relevante teknologier for projektet beskrevet.

5.1.2 Design

I dette kapitel bliver det endelige design af prototypens funktioner og brugergrænseflade fastlagt. Kapitlet indledes med at det overordnede flow i prototypen fastlægges. Det bliver defineret hvilke kernefunktioner prototypen skal bestå af, hvorefter de vil blive beskrevet, og deres design bliver fastlagt. Til at uddybe prototypens design bliver der udformet en række klassediagrammer, samt sekvensdiagrammer.

Der vil blive beskrevet hvilke problemstillinger man skal være opmærksom på i forbindelse med udvikling af en funktion der kan sammenligne databaser, samt migrerer de observerede forskelle mellem databaserne.

Herudover bliver det beskrevet hvilke designovervejelser der ligger til grund for prototypens brugergrænseflade. Der bliver også forklaret hvordan prototypen overholder de retningslinjer, der er nødvendige for at opnå god usability. For at sikre den bedst mulige kommunikation og interaktion mellem bruger og prototype.

5.1.3 Implementering og test

Der bliver i dette kapitel forklaret hvordan de forskellige funktioner i prototypen, herunder sammenlignings- og migreringsfunktionen, er blevet implementeret. Endvidere bliver der beskrevet hvordan implementeringen afviger fra designet, samt hvorfor. Der er lavet uddybende beskrivelser af prototypens hovedfunktioner og andre interessante funktioner. Herudover bliver de funktioner, som ikke var en del af det oprindelige design, beskrevet.

Grundet tidsmangel er der ikke blevet implementeret en brugergrænseflade til prototypen.

Kapitlet indeholder også en beskrivelse af den teststrategi der benyttes, samt hvilke test-typer der bruges til at teste prototypen. Efterfølgende beskrives hvordan de forskellige tests er blevet udført, samt hvilket resultat testene har givet.

5.2 Samlet konklusion

Der er i dette projekt blevet udviklet en prototype af et databasemigreringsværktøj, som kan overskuelig gøre processen til at migrere forskelle fra en database til en anden, for virksomheden Stayhard. Projektet er afgrænset således at prototypen fokuserer på sammenligning mellem tabeller og rækker i en database. Dette er gjort for at begrænse projektets omfang til en størrelse der er passende til et projekt af denne type.

Lige siden projektets begyndelse har flere af de teknologiske aspekter af været fastlagt, grundet ønske fra Stayhards side. Dette omfatter kodesprog, som er fastlagt til C#/.NET, og databasetype som er Microsoft SQL Server, begge er teknogier som Stayhard benytter. Til udvikling af brugergrænsefladen er Microsoft WPF-framework blevet valgt, efter ønske fra Stayhard. Herudover er det besluttet at benytte designarkitekturen Model-View-ViewModel for at strukturere koden, så der opnås lav kobling og høj binding. Dette er med til at gøre kode mere læsbar, hvilket gør det nemmere for en ny udvikler at sætte sig ind i den. Dette er vigtigt, da Stayhard har givet udtryk for at prototypen potentielt skal videreudvikles til en fuldt funktionel applikation.

Prototypen er udviklet således at det er muligt at udføre en sammenligning mellem to databaser, og finde samtlige forskelle mellem dem. Endvidere kan prototypen migrere de observerede forskelle over i den ene af de to databaser.

Der er taget stilling til, hvordan brugergrænsefladen skal opbygges, så prototypen følger de retningslinjer der er opstillet i afsnittet om usability i kapitel 2. Herudover er der udarbejdet et farvekodesystem, som prototypens brugergrænseflade benytter til at skabe overblik over resultatet af en sammenligning.

Målet med projektet var at udvikle en applikation, som kunne sammenligne databaser fra to Litium Studio miljøer. Det skulle være muligt for brugeren, at udvælge hvilke tabeller der skal sammenliges, gennem en brugergrænseflade. Herefter skulle der udføres en sammenligning på de valgte tabeller, og samtlige fundne forskelle skulle blive vist til brugeren, hvorpå brugeren skulle kunne udvælge hvilke forskelle der skulle migreres. På baggrund af de valgte forskelle skulle applikationen generere SQL-kommandoer, som kunne bruges til at opdatere databasen i det ene miljø.

Da der overordnet er lagt vægt på funktionaliteten, der er tilknyttet model-laget i systemets arkitektur, er størstedelen af denne funktionalitet blevet implementeret. Det omfatter oprettelse af database-objekter, sammenligningen af to database-objekter, generering af SQL-kommandoer baseret på fundne forskelle samt muligheden for at benytte SQL-kommandoerne til at opdaterer en database.

Grundet skred i tidsplanen samt uhensigtsmæssig disponering af resterende tid, har det været nødvendigt at fokuser på de mest essentielle dele af prototypen. Det har medført at prototypens brugergrænseflade, samt de dele af designarkitekturen som relaterer til den, ikke er blevt implementeret. Herudover har det påvirket testforløbet således, at det har været nødvendigt at fokusere på black-box test.

Det har været en spændende og lærerig proces, at lave et større projekt i samarbejde med en virksomhed. Hele processen med at skulle efterkomme virksomhedens ønsker, samt at tilpasse projektet til virksomhedens krav, har været en meget interessant oplevelse.

5.3 Fremtidige udvidelser

Der er i dette projekt lavet en prototype, som viser hvordan vores vision kan muliggøres. Der er derfor en række funktioner, som er blevet fravalgt, for at tilpasse projektets omfang til en overkommelig størrelse. Det er nogle af de funktioner, som vi gerne ville arbejde videre med, hvis vi skulle fortsætte arbejdet med applikationen, efter vi var færdige med projektet. Disse funktioner kan inddeles under følgende dele:

- Brugerinterfacet
- MVVM
- Fravalgte funktionelle krav.
- Udvidet sammenligning af databaser

Af disse, er de mest nærliggende af de fremtidige udvidelser, GUI og MVVM-delene. Som det blev nævnt i konklusionen, er det kun en begrænset implementering af disse to dele i den nuværende prototype. Det vil derfor være en stor forbedring af applikationen, at få implementeret disse dele fuldt ud i applikationen.

De fravalgte funktionelle krav dækker over de fire udeladte krav, som blev beskrevet i kravspecifikationen. Det første udeladte krav indebærer, at udvide sammenlignings-funktionen, således at der også kan sammenlignes på billed- og config-filer, når man sammenligner to miljøer. De næste to krav omhandler versionsstyring, samt en avanceret funktion hertil. Selve versionsstyringen omhandler et system, således at der skabes en historik over hvilke ændringer, der er blevet migreret. Den udvidede version går ud på, at det skal være muligt, at bruge versionsstyringen til at fjerne en tidligere migrering fra et miljø. Det sidste udeladte krav dækker over en historik, således at man kan se de ændringer, man bevidst har udeladt fra en tidligere migrering.

Delen der omhandler udvidet sammenligning af databaser, dækker over en udvidelse af applikationen, således at der også bliver sammenlignet på de dele af databaserne, som bliver vurderet til at være ”nice to have” i afsnit 2.1.1 - Database-input(side 10). Det betyder altså, at der udvides til også at sammenligne på stored procedures, views og generelle database-indstillinger. Denne udvidelse vil mindske forskelligheds-graden yderligere, ved sammenligning af to databaser.

5.4 Udtalelse fra virksomheden

STAYHARD.COM™
FASHIONLINE

Herrljunga, Sverige
Tirsdag den 2 oktober 2012

Konklusion på projektforløb

Da vi på Stayhard i midten af 2011 begyndte processen med at skifte platformen på vores daværende webshop til Litium Studio (en svensk udviklet e-handels platform). Samtidig påbegyndtes arbejdet med at opbygge en egen webudviklingsafdeling, og hvad der dertil hører af udviklingsværktøjer og –miljøer. Vi indså ganske hurtigt, at vi havde en udfordring når det kom til fuld-/semiautomatisk migrering af ændringer, lavet i backend systemet (disse ligger gemt som ændringer i en database). Dette er en vigtig detalje for os, da vi stræber mod at have en continuous deployment med så lidt manuelt arbejde som muligt.

Vi kiggede på forskellige løsningsmodeller og kom frem til, at projektet kunne egne sig til at lave i samarbejde med nogle studerende, der eventuelt brugte det som eksamensprojekt.
Valget på samarbejdspartnere faldt på Nicolai Børger og Jannick Kaas Johansen fra Danmarks Tekniske Universitet.

Samarbejdet blev indledt ved, at Nicolai og Jannick var på besøg i fire dage hos Stayhard i Sverige. De fik en introduktion til virksomheden, en forståelse af projektets natur, og vi diskuterede mulige løsningsmodeller. Herefter har samarbejdet fungeret med jævnlig kontakt via email og telefon.

Samarbejdet med Jannick og Nicolai har fungeret uden problemer. De tog vores krav til systemet, både med hensyn til funktionalitet og teknologier (.NET, SQL etc.), til sig og kom på denne baggrund med forskellige løsningsmodeller. De har under forløbet arbejdet på at skabe en prototype, som belyser, hvordan vi løser vores migreringsproblemer.
Det har for os været meget vigtigt med en solid foranalyse for at få belyst mulige løsningsmodeller, samt et grundigt system design ved hjælp af usecases, mockups og andre designbeslutninger. Vores sekundære mål har været udvikling af en protype, som viser, at det er muligt at lave den ønskede datamigrering.

Nicolai og Jannick har under projektet udvist en stor interesse for vores forretning og vores interne arbejdsgange i udviklingsafdeligen, hvilket har været nødvendigt for at kunne forstå vores udfordring og lave en grundig analyse.

Vores mål med dette samarbejde var at få lavet en analyse af problemet samt skabt en prototype for at se, om det ville være muligt at bygge en sådan applikation. Vi føler, at vores forventninger til samarbejdet er blevet indfriet både med hensyn til analysen og prototypen.
Vores plan er at forsætte arbejdet med prototypen, højst sandsynlig som en del af fremtidige stundenterprojekter. Først og fremmest ønsker vi at flytte prototypen ud af prototype stadiet og benytte den mellem udviklings- og stagingmiljøerne. Herefter ønsker vi at videreudvikle den med nye funktioner som nævnt i Nicolai og Jannicks rapport.

Stephan Kaas Johansen

Weddeveloper, M.Sc.Eng. IT
stephan.johansen@stayhard.se
Telefon: +46 513 - 22512

Stayhard AB
Box 33
524 21 Herrljunga
Sweden

Kapitel 6 – Bilag

6.1 Litteratur

Larman, C. (2004) – Applying UML and patterns: an introduction to object-oriented analysis and design and iterative development, 3rd edition, Pearson Education

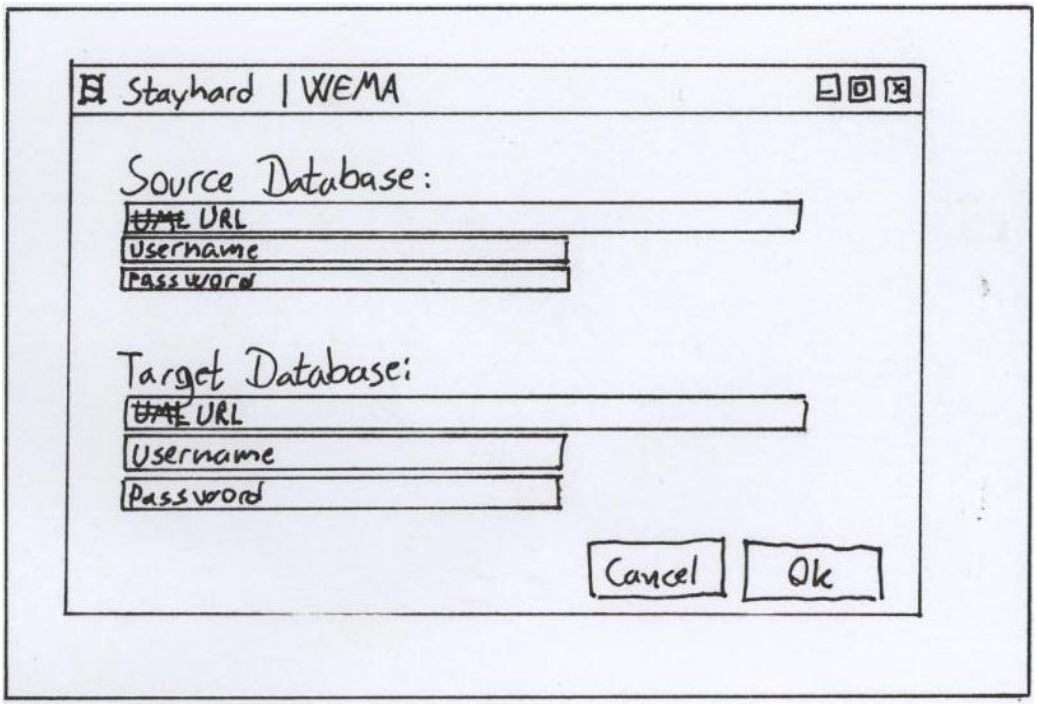
Bruegge, B., Dutoit, A. H. (2004) – Objekt-Oriented Software Engineering: Using UML, Patterns, and Java™, 2nd edition, Pearson Education

Brown, P. (2010) – Silverlight 4: In Action, Revised edition of Silverlight 2. In action, Manning Publications

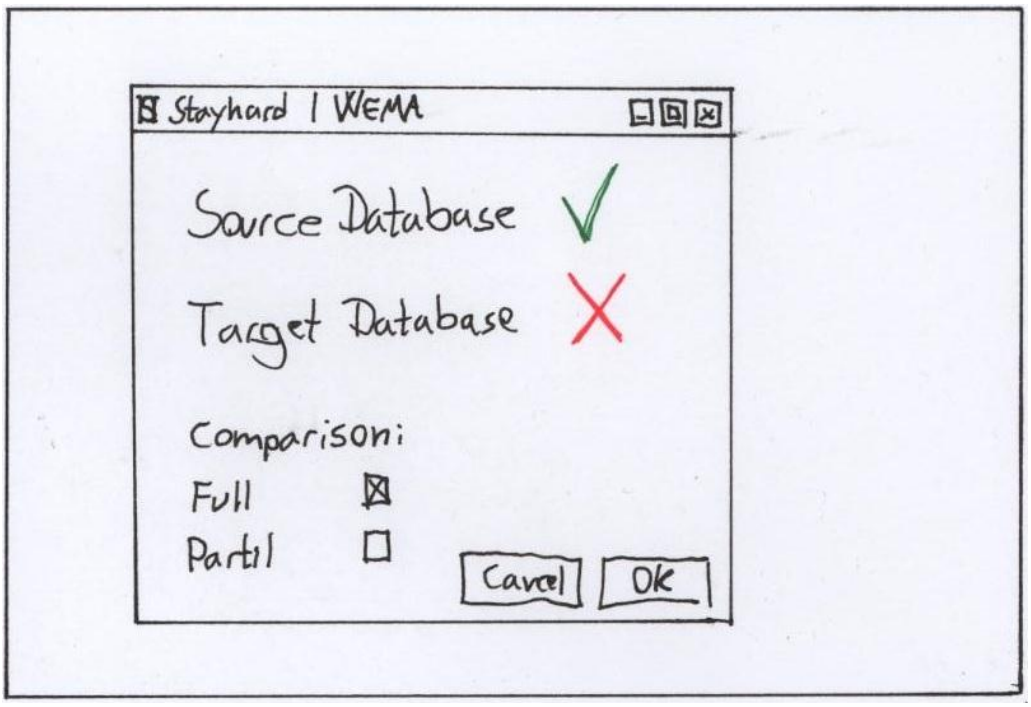
Beck, B. (2003) – Test-Driven Development: By Example, Pearson Education

Northrup, T. (2009) – MCTS Self-Paced Traning Kit (Exam 70-536): Microsoft© .NET Framework-Application Development Foundation, 2nd edition, Microsoft© Press

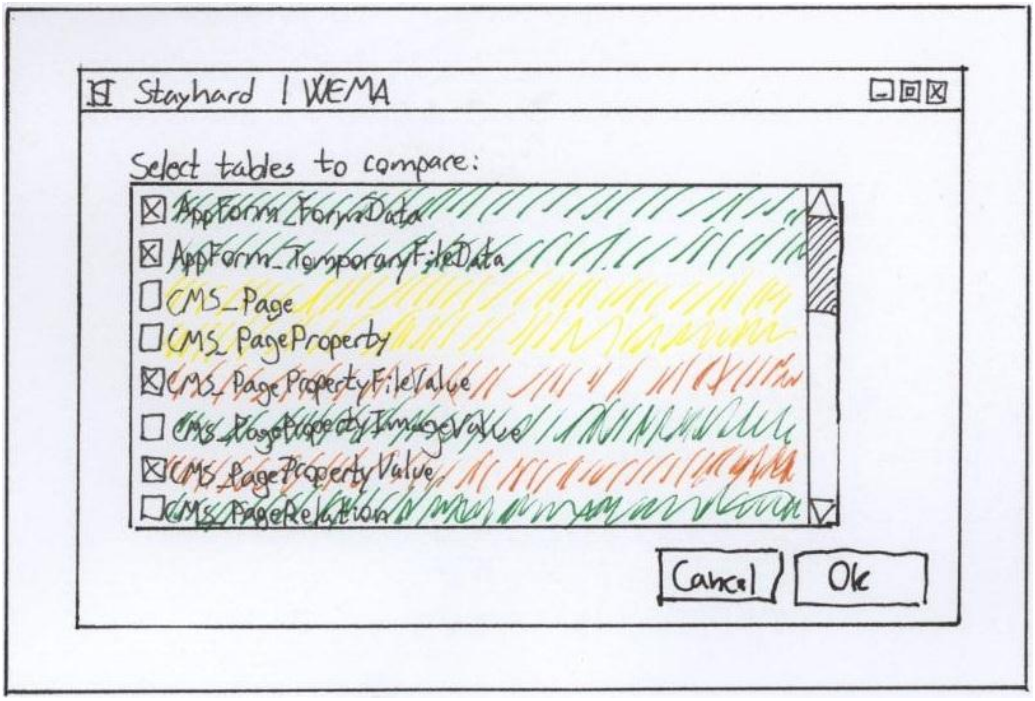
6.2 Mock-ups



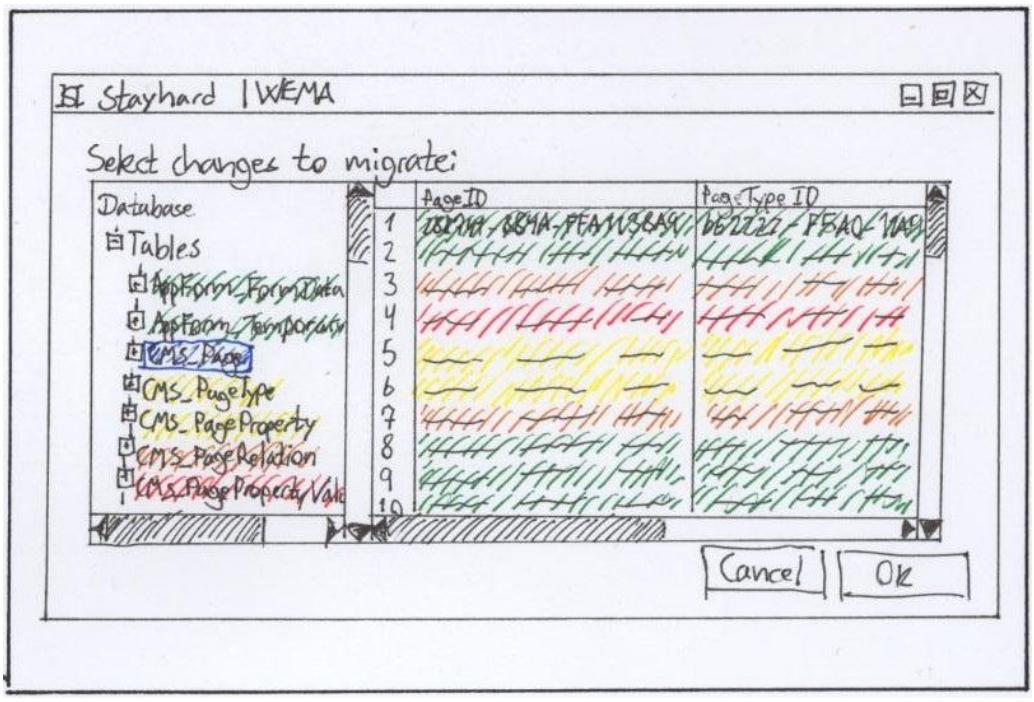
Figur 33 - Mock-up 1



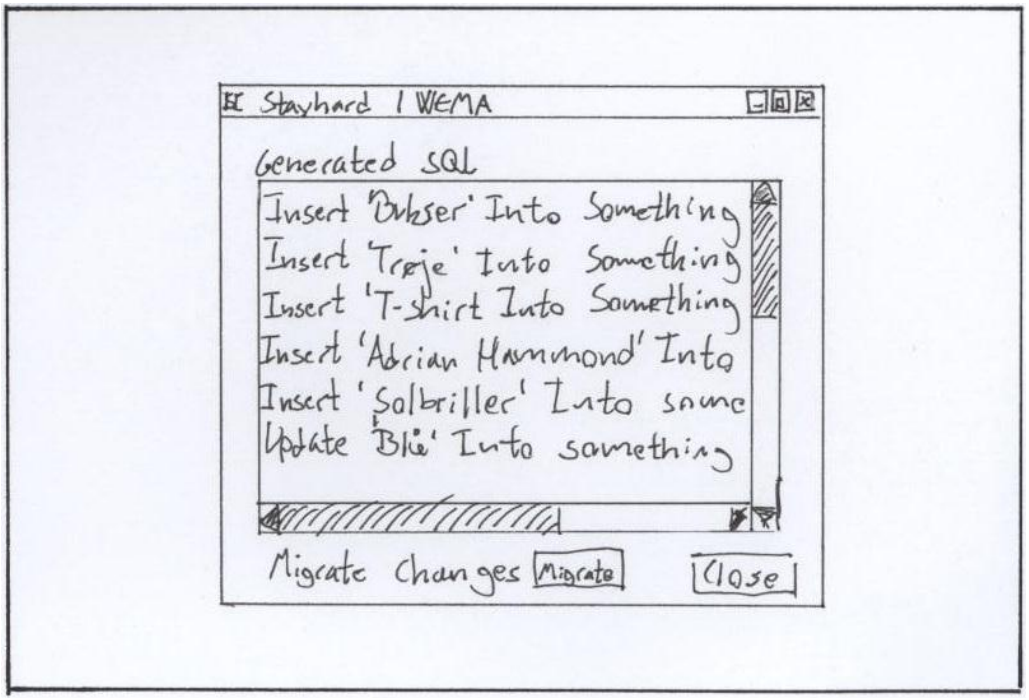
Figur 34 - Mock-up 2



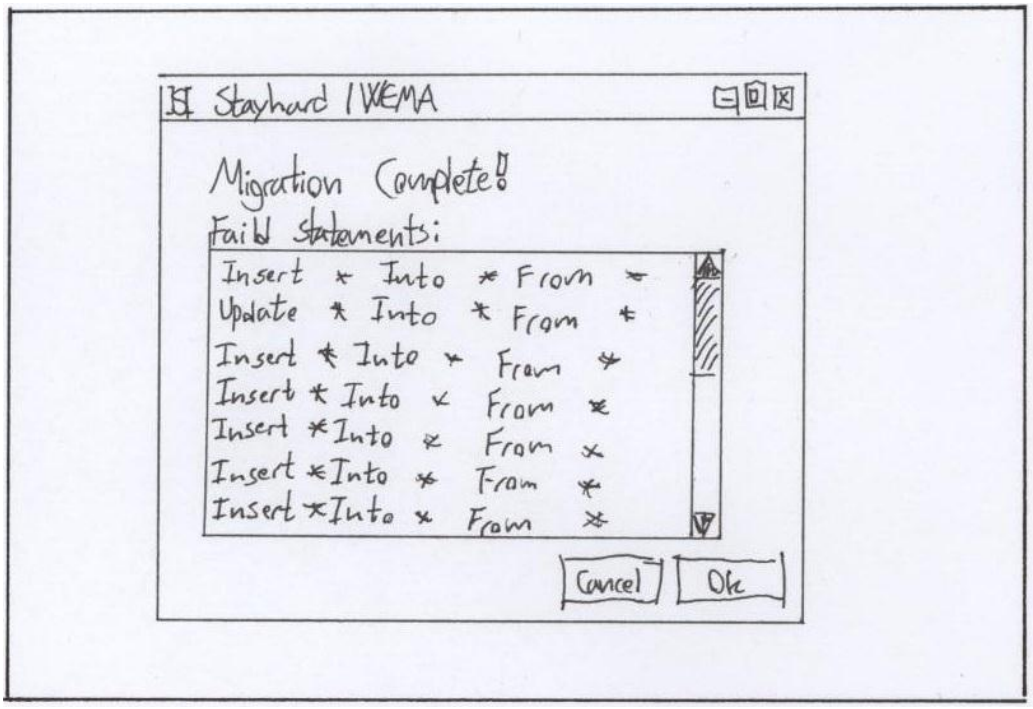
Figur 35 - Mock-up 3



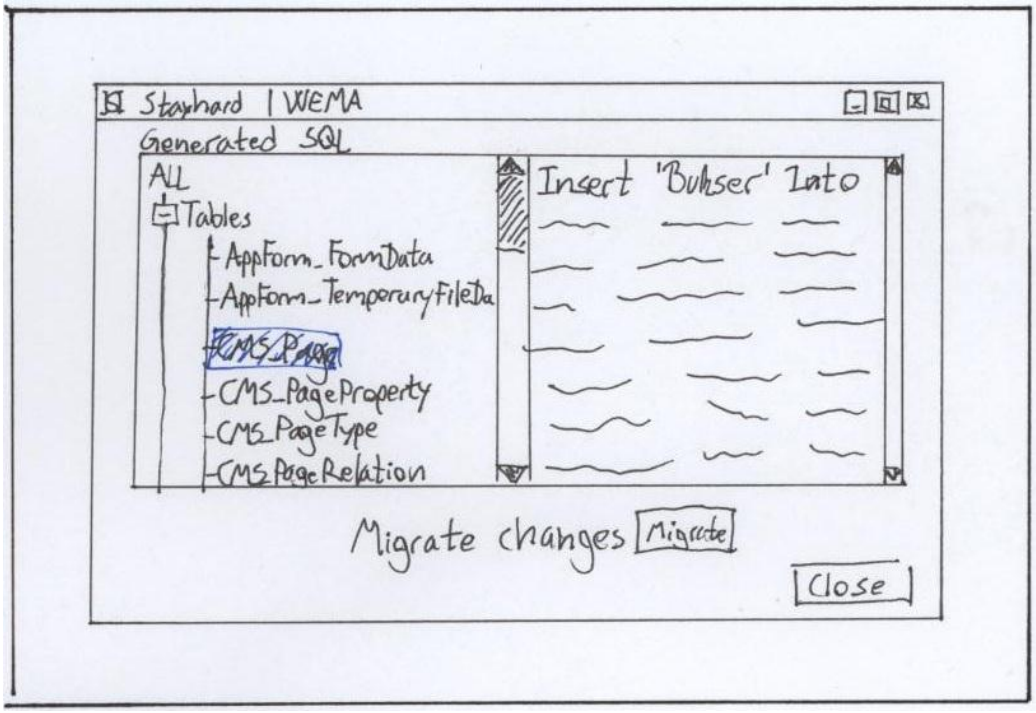
Figur 36 - Mock-up 4



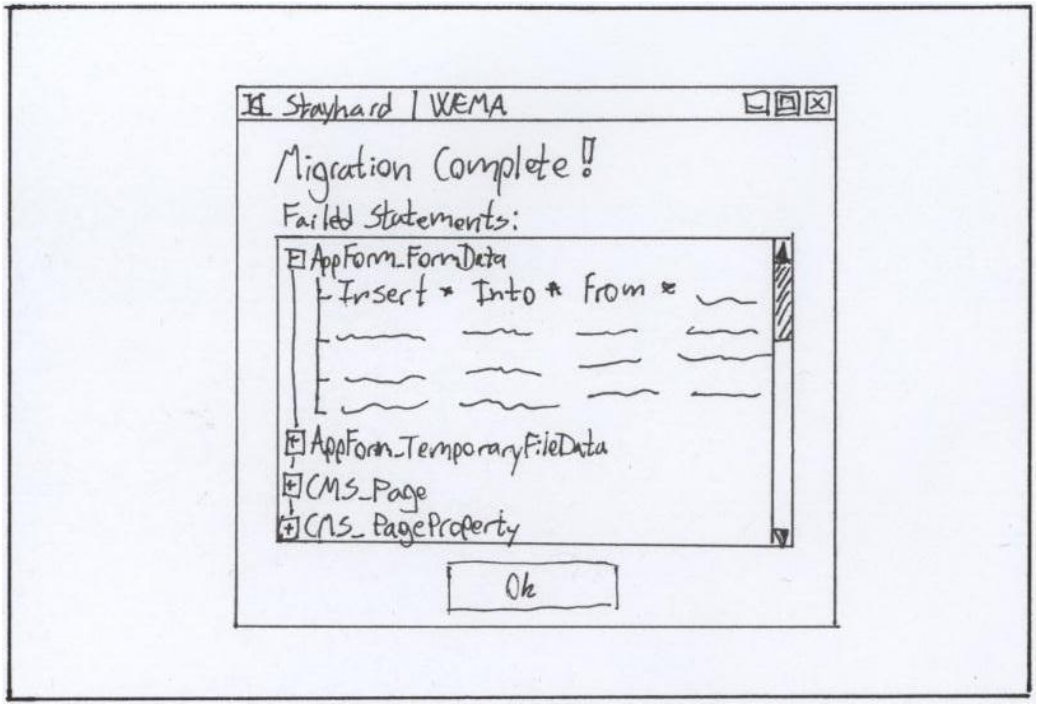
Figur 37 - Mock-up 5



Figur 38 - Mock-up 6



Figur 39 - Alternativ til mock-up 5



Figur 40 - Alternativ til mock-up 6