

I4 PRJ4

Forår 2015

SmartFridge

Rapport

Gruppe 5

Deltagere:

#1 Stud.nr.: 201370952 Navn: Kristoffer Lerbæk Pedersen
#2 Stud.nr.: 201270810 Navn: Mathias Siig Nørregaard
#3 Stud.nr.: 201371027 Navn: Mathias Schmidt Østergaard
#4 Stud.nr.: 201371009 Navn: Mathis Malte Møller
#5 Stud.nr.: 201370747 Navn: Mikkel Koch Jensen
#6 Stud.nr.: 201370786 Navn: Rasmus Witt Jensen

Vejleder: Lars Mortensen

28. maj 2015

Resumé

Denne rapport beskriver et semesterprojekt for 4. semester på Aarhus School of Engineering. Gruppens seks medlemmer er alle ingeniørstuderende inden for Informations- og Kommunikationsteknologi.

Problemstilling

Produktet har til formål at give overblik over varerne i brugerens fysiske køleskab, ved at skabe et digitalt køleskab, hvor varerne kan ses i hjemmet og på farten.

Formål

Formålet med projektet er, ifølge introduktionsoplægget for 4. semesterprojekt, blandt andet at:

- Udvikle applikationer med grafiske brugergrænseflader, databaser og netværkskommunikation.
- Anvende teknikker, metoder og værktøjer til softwaretest.
- Anvende en iterativ udviklingsproces.
- Anvende projekt- og versionsstyringsværktøjer.
- Anvende objektorienteret analyse og design i systemudvikling.

Disse læringsmål tager udgangspunkt i flere af fagene på 4. semester.

Opstillede krav og valgte løsninger

Krav fra introduktionsoplægget:

- *Projektet skal inddrage faglige aspekter fra samtlige fag på 4. semester IKT. Dette skal dokumenteres i projektrapporten og bør inddrages af de studerende til eksamen.*
- *Projektet skal have et passende omfang, så alle i projektgruppen kan arbejde med projektet.*
- *Projektet skal være af en karakter der tillader at læringsmålene i faget opfyldes.*
- *Der skal afleveres et projektforslag (ca. 1 A4-side) med en problembeskrivelse (hvilket problem søger projektet at løse) og projektbeskrivelse (hvordan søges problemet løst) gerne fredag i semesterets 1. uge.*

Løsningen er en webapplikation, to databaser samt en WPF-applikation.

Anvendte metoder

Under udarbejdelse af projektet er der blevet anvendt elementer fra Scrum til overskueliggørelse af opgaver og møder. Der er blevet holdt stand-up møder tre gange om ugen. Sprint-planlægningsmøder og retrospektmøder er blevet holdt én gang hver 1.-3. uge.

Resultater

Systemet med alle dets kernefunktionaliteter er implementeret, herunder opsætning af lokal database, en ekstern database, en WPF-applikation, samt en webapplikation. Herudover er der blevet implementeret to udvidelses-*use cases*.

Abstract

This report describes a 4th semester project made by six Aarhus School of Engineering ICT students.

Problem summary

The idea behind the product is to design a digital refrigerator, which allows registration of the items of a physical refrigerator. This allows customers to manage or view their groceries, even when they are not near their refrigerator.

Goal

Our goal, according to the introductory presentation, is to:

- Develop applications with graphical user interfaces, databases and network communication.
- Practice techniques, methods and tools used in software testing.
- Practice an iterative development process.
- Practice project- and distributed revision control.
- Practice object oriented analysis and design in system development.

Based on this semester's curriculum, these are the learning goals.

Requirements and selected solutions

Requirements from the introductory presentation:

- *The project must include elements from all 4th semester ICT courses. This is to be documented in the report, and should be mentioned by the students during the exam.*
- *The project must be of an appropriate scope, so that everyone in the group is able to contribute.*
- *The project must be substantial enough to allow the learning objectives in the course to be met.*
- *A proposed project (1 A4-page) with a problem summary (which problem does the project aim to solve) and a project description (how do you intend to solve the problem) must be handed in prior Friday of the first week of the semester.*

The selected solution was a web application, two databases as well as a WPF application.

Applied methods

During project execution, we used elements from Scrum to manage tasks and meetings. Three times a week, stand up meetings have been organized. Once every one to three weeks, sprint meetings and retrospect meetings have been held.

Results

The system, with all of its core functionalities, has been implemented; a local database, an online database, a WPF application and a web application. In addition, two extension use cases have also been implemented.

Indholdsfortegnelse

Resumé	I
Problemstilling	I
Formål	I
Opstillede krav og valgte løsninger.....	I
Anvendte metoder.....	I
Resultater.....	I
Abstract.....	II
Problem summary.....	II
Goal	II
Requirements and selected solutions.....	II
Applied methods.....	II
Results.....	II
Arbejdsfordeling	1
Indledning	2
Opgaveformulering.....	2
Projektafgrænsning.....	3
Termliste	3
Fridge app	3
Web app.....	3
Kernefunktionalitet.....	3
Standard-varer	3
Systembeskrivelse.....	4
Krav	5
Aktørbeskrivelse	5
Use case-beskrivelse	6
UC1: Se varer.....	6
UC2: Tilføj vare.....	6
UC3: Fjern vare.....	6

I4 PRJ4, Gruppe 5

IKT

UC4: Rediger vare	6
UC5: Synkroniser til ekstern database	6
UC6: Notifikation om holdbarhedsdato	6
Projektgennemførelse og metoder	6
Projektstyring	6
UML	8
Tidsplan	8
Mødestruktur	8
Continuous Integration	8
Specifikation og analyse	10
Systemarkitektur	11
Design, implementering og test	12
Overordnet design	12
Database	12
Design, implementering og test - Database	Fejl! Bogmærke er ikke defineret.
Fridge app	12
Web app	18
Fridge app	20
Design	20
Implementering	21
Test	23
Web app	23
Design	23
Implementering	24
Test	27
Resultater og diskussion	28
Udviklingsværktøjer	29
Git	29
Microsoft Visual Studio 2013	29
Microsoft Visio 2013	29

I4 PRJ4, Gruppe 5

IKT

LucidChart	29
DDS-Lite.....	29
Entity Framework.....	29
Microsoft Sync Framework	29
NUnit	29
NSubstitute	29
Doxygen	29
Opnåede erfaringer.....	30
Kristoffer Lerbæk Pedersen	30
Mathias Siig Nørregaard	30
Mathias Schmidt Østergaard	31
Mathis Malte Møller	32
Mikkel Koch Jensen.....	32
Rasmus Witt Jensen	33
Fremtidigt arbejde	34
Konklusion.....	34
Referencer.....	36
Bilag.....	37

Arbejdsfordeling

Navn	Arbejdsområder
Kristoffer Lerbæk Pedersen	Softwaredesign af Fridge app, implementering af Fridge app, Business Logic Layer, tovholder for rapport og dokumentation, korrektur, Jenkins.
Mathias Siig Nørregaard	Tests af Business Logic Layer, UX design af Fridge app, grafikansvarlig for Fridge- og Web app, Business Logic Layer, implementering af Fridge app.
Mathias Schmidt Østergaard	Softwaredesign af Fridge app, softwaredesign af Web app, implementering af Fridge app, implementering af Web app
Mathis Malte Møller	Databasemodel, DAL for Fridge app og Web app, Synkronisering, Tests af DAL i Web app og Fridge app
Mikkel Koch Jensen	Implementering af Fridge app, implementering af Web app
Rasmus Witt Jensen	Implementering af Fridge app, implementering af Web app, Jenkins, UX design af Fridge app

Indledning

Denne rapport er skrevet på baggrund af et projektoplæg, som stiller visse krav til hvad projektet skal indeholde, mens selve emnet er frit. Projektet omhandler hvorledes databaser, en desktop-applikation og en webapplikation kan benyttes til sammen at opbygge et digitalt køleskab. Systemet er designet til at få et overblik over varerne i brugerens fysiske køleskab, selv når brugeren ikke er i nærheden af det – deraf er systemet døbt SmartFridge. Flere idéer blev overvejet, men SmartFridge blev valgt på baggrund af gruppens ønske om en simpel platform med rig mulighed for udvidelser.

Opgaven udføres ved hjælp af de forskellige fag som 1. til 4. semester på Aarhus School of Engineering har budt på, med særligt udgangspunkt i fagene på 4. semester. Først er der blevet udarbejdet en kravspecifikation, hvorefter gruppen har arbejdet med elementer fra Scrum, hvor gruppemedlemmerne arbejder i iterationer, som projektstyring.

Opgaveformulering

Formålet med dette projekt er at udvikle et system, som tillader registrering af varer i et køleskab via en grafisk brugergrænseflade på en lokal skærm. Lagring af disse oplysninger sker i en lokal database, der synkroniseres med en ekstern database, som kan tilgås via et web-interface. Systemet vil også tilbyde vedligeholdelse af en indkøbsseddel, samt en liste over standardvarer, der altid ønskes i køleskabet.

Systemet, som er relativt simpelt, byder desuden på rig mulighed for udvidelse, i form af inkorporering af ekstra funktioner; eksempelvis oplysning om ernæringsværdier, opskrifter baseret på eksisterende ingredienser og eksisterende/fremtidige tilbud på manglende varer.

Systemet skal ses som en ekstern tilføjelse til eksisterende køleskabe, og ikke i første omgang som en indbygget feature i nye modeller. Dette sikrer at systemet kan tilbydes til en bredere målgruppe end køberne af "high end"-køleskabe.

Visionen er at tilbyde brugeren et hurtigt og effektivt overblik over køleskabets indhold, til lettelse i en hverdag, hvor man ikke altid har en opdateret indkøbsliste inden for rækkevidde.

Brugerne kan være privatpersoner, såvel som industrielle køkkener og catering-virksomheder, som ønsker overblik over indholdet i deres køleskab(e).

Projektafgrænsning

Udførelsen af projektet er med fokus på softwareudvikling, så problemer som ikke er direkte software-relaterede, vil ikke blive forsøgt løst. Dette inkluderer:

- Strømforsyning til enheden, der kører Fridge app'en.
 - Kabling
 - Batteri
- Udvikling af enhed til kørsel af Fridge app'en.
 - Lenovo Yoga 2 Pro (**bilag 03**) benyttes som platform for Fridge app.

Ydermere vil *Web app* ikke indeholde de udvidelser, der er blevet tilføjet til *Fridge app*, og altså kun indeholde kernefunktionaliteterne. Dog har det også i kernefunktionaliteterne været nødvendigt med en afgrænsning; *Web app* vil ikke automatisk tilføje varer fra *Standard-beholdning* til *Indkøbsliste* ved mangel i *Køleskab*, ligesom det heller ikke vil være muligt at initiere en synkronisering mellem den lokale og den eksterne database. Den æstetiske fremtoning af *Web app* vil heller ikke blive prioriteret, da fokus ligger på at udstille den implementerede funktionalitet.

Termliste

Fridge app

Fridge app er den lokale del af systemet, og dækker over den lokale brugergrænseflade, samt den lokale database.

Web app

Web app er den eksterne del af systemet, og dækker over websitet.

Kernefunktionalitet

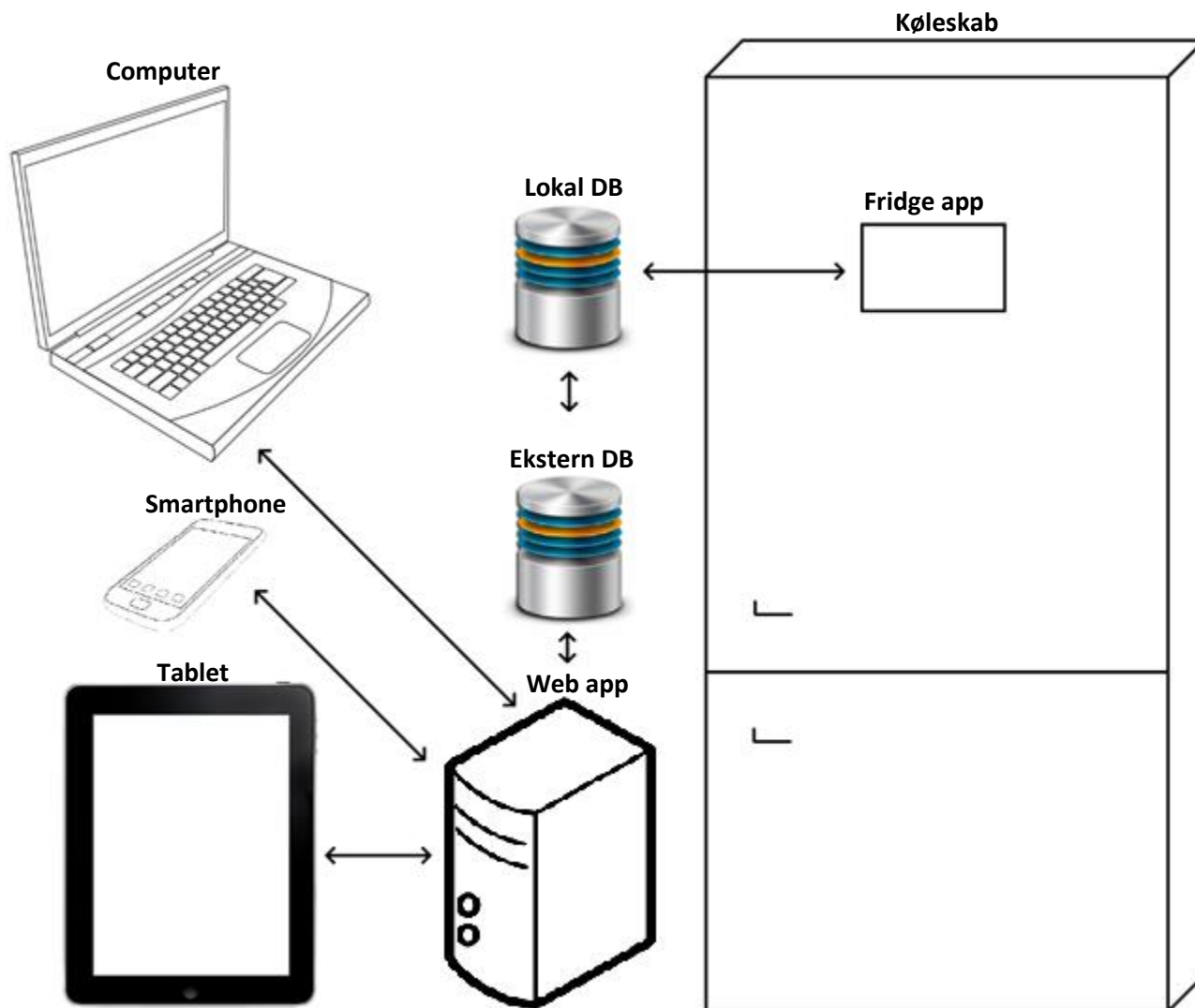
Begrebet dækker over de mest basale funktionaliteter, som gør at systemet er sammenhængende og brugbart. Disse funktionaliteter er repræsenteret ved use cases 1-6.

Standard-varer

Begrebet dækker over en række varer, som Bruger altid ønsker at have i sin varebeholdning. Varerne tilføjes til en liste på lige fod med andre lister i systemet. Kaldes også "standard-beholdning".

Systembeskrivelse

Systemet, skitseret i **Figur 1**, skal kunne assistere brugeren, ved at tilbyde opretholdelse af en liste over hvilke varer, der er i køleskabet, og en indkøbsseddel til manglende varer. Der kan tilføjes og fjernes varer i takt med indkøb og forbrug, samt opretholdes en liste over ønskede varer i køleskabet, der ved mangel automatisk tilføjes til indkøbssedlen. Systemet vil desuden bestå af en lokal og en ekstern database, hvori de oprettede lister gemmes, så listerne også kan tilgås fra en web-applikation.



Figur 1 Skitsering af systemet SmartFridge

Når brugeren starter systemet, vil den lokale database synkronisere med den eksterne database, for at sikre at begge databaser er opdateret og stemmer overens. Herefter vil den lokale og den eksterne database synkronisere jævnligt, så længe *Fridge app* er i brug.

Efter systemet er startet, vil det være muligt for brugeren at tilføje varer til listen over varer i køleskabet, på indkøbslisten eller i standardbeholdningen, hvorpå brugeren får eksisterende varetyper som forslag, eller kan tilføje nye varetyper selv.

Det er muligt at se varerne på de forskellige lister. I denne forbindelse vil det være muligt at redigere mængden af varerne i takt med forbrug.

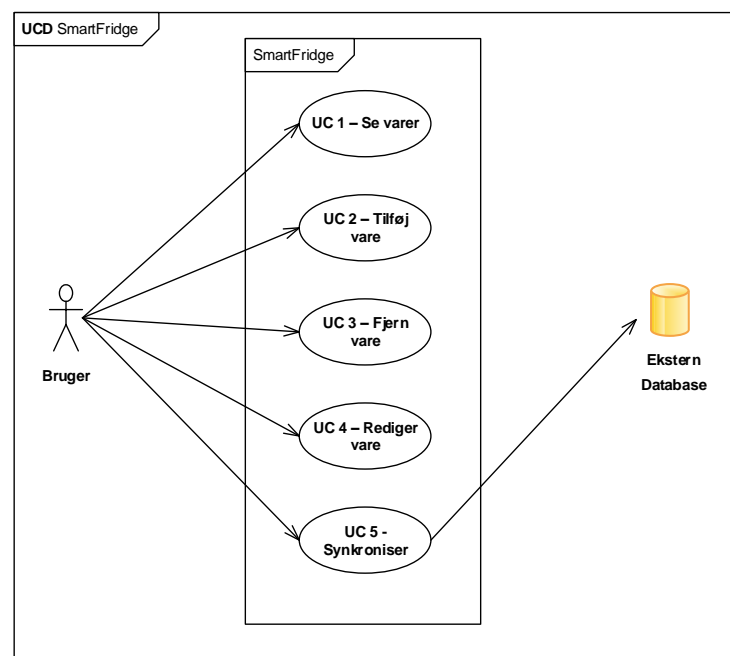
Systemet vil selv tilføje vare til indkøbssedlen når køleskabet indeholder en mindre kvantitet af en vare, end der er angivet på listen over standard-varer.

Krav

Ud fra opgaveformuleringen, er der udarbejdet en række *use cases*, som beskriver aktørernes interaktion med systemet. Disse use cases fungerer som kravspecifikation, og bruges i den tidlige del af udviklingsfasen til at bestemme systemets funktionalitet. For *fully dressed use cases*, henvises til projektdokumentationen fra side 4.

Aktørbeskrivelse

På *use case*-diagrammet på **Figur 2** ses en række aktører. Disse er beskrevet i følgende aktørbeskrivelse. For mere detaljerede beskrivelser, henvises til projektdokumentationens side 2.



Figur 2 Use case-diagram over SmartFridge

Bruger er en primær aktør, som ønsker at benytte systemet ved at tilføje, fjerne, redigere og se varer.

Ekstern Database er en sekundær aktør, som løbende synkroniseres med systemets lokale database. Herudover fungerer den som direkte database for *Web app*.

I4 PRJ4, Gruppe 5

IKT

Use case-beskrivelse

Use cases fra *use case*-diagrammet er beskrevet i følgende afsnit. Hver *use case* beskriver et scenarie, hvor Bruger interagerer med systemet.

UC1: Se varer

Bruger får frembragt alle de nuværende varer på en ønsket liste.

UC2: Tilføj vare

Bruger tilføjer en vare til en ønsket liste.

UC3: Fjern vare

Bruger fjerner en vare fra en ønsket liste.

UC4: Rediger vare

Bruger redigerer vareinformationerne på en vare i en ønsket liste.

UC5: Synkroniser til ekstern database

Bruger initierer en øjeblikkelig synkronisering mellem Lokal Database og Ekstern Database.

UC6: Notifikation om holdbarhedsdato

Systemet viser en notifikation for varer der har overskredet holdbarhedsdatoer

Projektgennemførelse og metoder

Projektstyring

Gruppen har valgt at tage elementer fra blandt andet den anerkendte projektstyringsform Scrum, således at der bliver udviklet iterativt.

Gruppen har kørt sprints på 1-3 uger. Herunder listes de elementer, gruppen har anvendt:

- *Task board*
 - Gruppen har brugt et digitalt Scrum board vha. en in-house web-applikation (*Redmine*), som skolen har stillet til rådighed. Her er det muligt at se hvilke opgaver, der skal laves under nuværende sprint, samt hvem de tilhører, og hvor meget tid der er sat af til opgaven.
- *Stå-op-møde*
 - Mandag, onsdag og fredag har gruppen holdt et møde på 5-10 minutter. Her kommer hvert gruppemedlem ind på 3 ting; Hvad har du lavet? Hvad skal du lave? Har du nogen forhindringer? Det har gavnet gruppen, da man hurtigt får feedback fra hvert medlem og hører om de er gået i stå, eller om der er nogen, der mangler tasks.
- *Retrospekt-møde*
 - Under retrospekt-mødet har gruppen afsluttet nuværende sprint og planlagt næste. Der ses på hvor meget gruppen nåede, og hvert gruppemedlem kommer med feedback til

sidste sprint, samt eventuelle forbedringer til næste sprint. Herefter planlægges et nyt sprint med omhu, således at opgaverne og den normerede tid lyder realistisk ift. forhindringer.

Kort beskrivelse af hvert sprint:

Iterationsnummer	Længde	Formål
1	4 dage	Kravspecifikation.
2	2 uger	De første kernefunktionaliteter implementeres/påbegyndes.
3	2 uger	De sidste kernefunktionaliteter (det første <i>use case</i> -udkast, gruppen lavede) laves færdigt. Nogle ting refaktoreres.
4	3 uger	Udvidelser (nye <i>use cases</i>) påbegyndes. Unit tests sættes op. Databasen integreres med applikationen.
5	2 dage	Et minisprint for at fastlægge hvad gruppen skal lave i næste sprint. Teknologiundersøgelse af MVVM og MVC pattern, samt vurdering af hvorvidt koden med fordel kan refaktoreres til et af disse.
6	1 uge	Påbegyndelse af synkroniseringsprocessen og webapplikationen.
7	1 uge	Sync laves færdig, og der arbejdes videre på webapplikationen.
8	1 uge	Notifikations- <i>use casen</i> laves færdig, og Sync kobles på WPF-applikationen. <i>Controller</i> og <i>Model</i> arbejdes på i web-applikationen
9	1 uge	Rapportskrivning. De enkelte <i>use cases</i> uddelegeres i web-applikationen.
10	1½ uge	Rapportafslutning, Web app afsluttet, Fridge app fejlrettet/testet færdig, accepttest

UML

Til formidling af kravspecifikation og systemarkitektur, har projektgruppen valgt at anvende *Unified Modeling Language (UML)*. Dette er valgt for at formidle systemet bedst muligt, da *UML* er industristandard, og desuden simpel og intuitiv at gå til for omverdenen.

Tidsplan

Gruppen har fulgt en tidsplan (**bilag 20**), hvor der er forsøgt at sætte mere tid af til rapportskrivning end hvad vi gruppens medlemmer har afsat under tidligere semesterprojekter.

Mødestruktur

Gruppe- og vejledermøder er blevet styret ved hjælp af en mødeindkaldelse (**bilag 16**), efterfulgt af et møde med dagsorden, dirigent og referent. De administrative roller er blevet fastlagt vha. en turnusordning (**bilag 22**), hvor de forskellige roller som referent og dirigent er skiftet fra møde til møde. Mødeindkaldelsen var fastlagt. Dette er gjort for at sikre at alle gruppemedlemmer, som måtte ønske det, har fået indblik i det administrative arbejde. For at sikre konsensus i dokumenter, er der udarbejdet skabeloner til mødeindkaldelser og referater. Møderne er blevet afholdt efter behov, med udgangspunkt i et møde ved et sprints begyndelse og afslutning. Referatet (**bilag 21**) fra forrige møde er blevet gennemgået og godkendt ved hvert møde.

Continuous Integration

Jenkins er i det følgende beskrevet, på trods af at det ikke kom til at virke under dette projekt, da der er brugt mange timer på det, og der er kommet en masse erfaringer ud af det.

Jenkins har haft til formål at sikre en objektiv og løbende håndtering af de tests, der er skrevet i projektet. Det kobles sammen med et *git-repository*, og har derigennem adgang til den nyeste publicerede version af projektet. *Jenkins* kompilerer selv projektet og kører de *test-suites*, der er lavet ved hjælp af *NUnit*- og *NSubstitute*-frameworket. Det kan også sættes op til at udføre statisk analyse og *coverage*, hvis det ønskes.

I dette projekt er der blevet brugt lang tid på at få *Jenkins* til at fungere. Dette er dog ikke lykkedes efter hensigten. Det *git-repository*, der er brugt i projektet, kører på *GitHub*, hvilket viste sig at være et problem. Under opsætningen af *Jenkins*-projektet skal der specificeres, hvornår *Jenkins* skal bygge den *Visual Studio*-solution, der henvises til. Til omfanget af dette projekt, vil det være optimalt at bygge projektet hver gang der bliver lavet et nyt push. For at dette kan lade sig gøre med et *git-repository*, der kører på *GitHub*, skal der installeres et plugin på *Jenkins*-serveren. Det har ikke været muligt at finde en måde at gøre dette på, uden at prøve sig frem med opsætningen af et af de plugins, der findes til netop dette. Da projektgruppen ikke har direkte adgang til *Jenkins*-serveren, kunne det derfor ikke lade sig gøre.

Der blev senere udleveret et *git-repository* af skolen, som kører på *Gitswat*, og derfor ikke havde ovenstående problem. I forbindelse med udviklingen af projektets databasetilgang, blev der brugt et *modeling*-projekt i *Visual Studio*. Denne type projekt kræver at der er nogle bestemte filer i

I4 PRJ4, Gruppe 5

IKT

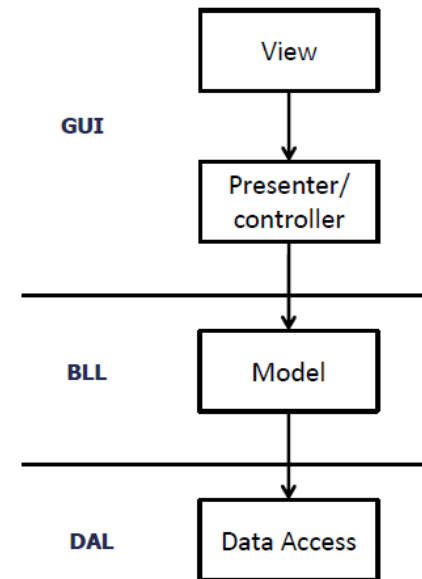
installationen af *Visual Studio*. Disse filer var ikke tilgængelige på *Jenkins*-serveren, og *Jenkins* kunne derfor ikke bygge systemets *solution*. Der blev forsøgt at omgå dette ved at lægge de manglende filer i en mappe i projektet, og manuelt ændre i en af projekternes *.csproject*-fil til at der skulle ledes på den nye lokation. Resultatet heraf blev at *Jenkins* kunne bygge projektet, men at det ikke kunne bygges lokalt i *Visual Studio*.

Da underviseren, der står for *Jenkins* lagde de manglende filer der, hvor *Jenkins* søgte efter den, efterspurgte *Jenkins* ikke længere de pågældende filer, men nu nogle andre. På dette tidspunkt var størstedelen af de ønskede test-suites skrevet og kørt, og grundlaget for at bruge *Jenkins* var ikke længere tilstrækkeligt til at det var tiden og indsatsen værd.

Det ovenstående beskriver processen med *Jenkins* i forhold til *WPF*-applikationen. *Jenkins* skulle efter planen også have været brugt til at teste web-applikationen. Her manglede *Jenkins*-serveren, ligesom ved *modeling*-projektet, en fil for at kunne bygge web-applikationer. Tidligere erfaringer viste, at det ikke kunne betale sig at forsøge at løse dette problem, uden selv at have adgang til serveren, og *Jenkins* blev derfor helt droppet.

Specifikation og analyse

Da udviklingen af WPF-applikationen gik i gang, var det meget nyt. Gruppen lærte undervejs hvordan delene skulle programmeres, og der var ikke de store overvejelser omkring hvilken arkitektur programmet skulle bygges op efter. Dette har givet nogle problemer med at bevare overskueligheden, og med at holde koden let at vedligeholde. I slutningen af april blev der foretaget en analyse af MVC, MVP og MVVM, og om det kunne betale sig at refakturere koden, for at overholde disse arkitekturer. Gruppen fandt, at det ville have været en god idé at opbygge WPF-applikationen efter MVVM-arkitekturen, helt fra starten af. Hvis det var sket, ville mængden af kode i *code behind* have været reduceret væsentligt. Problemet med kode i *code behind* er, at det gør det vanskeligt at teste koden i form af *unit tests*. MVC og MVP blev også undersøgt. Grundlæggende har det vist sig at WPF-applikationen til dels følger MVC/MVP-arkitekturen. Der er et View i form af XAML, *presenter/controller* i form af *code behind*, *Business Logic Layer (BLL)* og *Data Access Layer (DAL)*. Det ville have været en fordel at fjerne koden i *code behind*, og rykke det ud i et lag for sig selv. Fordelene ved at gøre dette kunne dog ikke gøre op for den tid det ville tage at skrive koden om. Gruppen har derfor besluttet at arbejde videre med koden som den var, men at lære af det og bruge den nye viden i forbindelse med udviklingen af *Web app*.



Figur 3 MVC/MVP til 3-lags-arkitektur

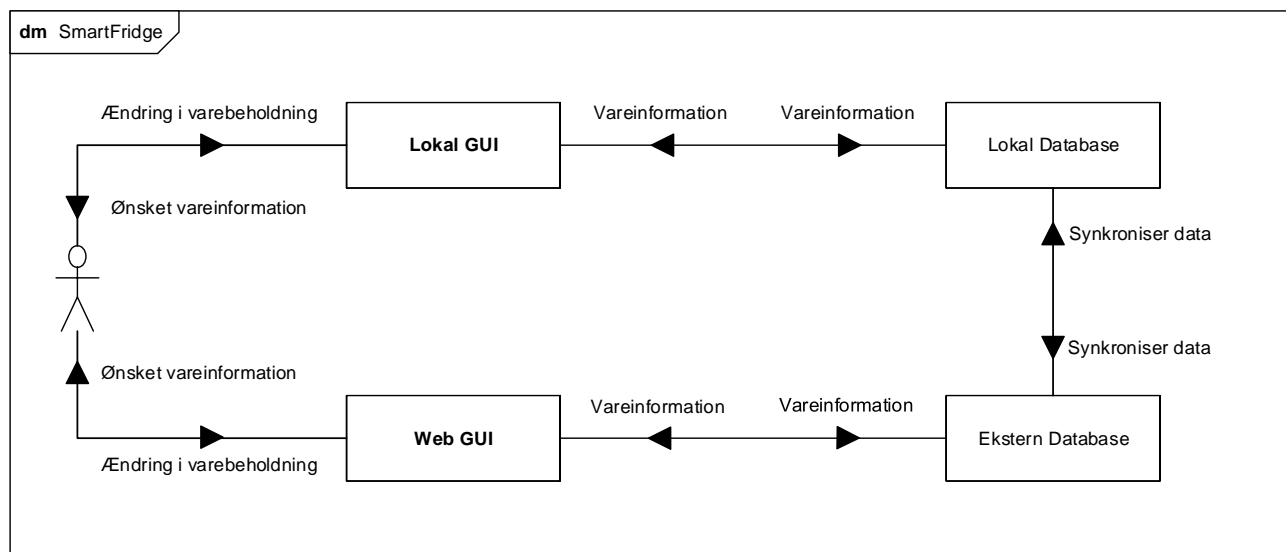
En anden udfordring har været *mapping*'en mellem objekt-udgaven af en *entity* i koden, og dens relationelle database-udgave. Der er brugt lang tid på at få denne *mapping* til at være korrekt, og sørge for at udgaven i koden er den samme som den eksisterende i databasen. Løsningen på dette blev fundet gennem undervisningen i I4DAB. Grundet den fremskredne udvikling i WPF-applikationen blev det ligeledes besluttet at det ikke kunne betale sig at refaktorere koden, men at implementere det som en del af *Web app*.

I *Web app* er der også blevet indført et *GOF-pattern* ved navn *Facade*. *Facade*-mønsteret har i dette tilfælde til opgave at give et enkelt adgangspunkt til DAL. I forbindelse med DAL er der et *repository* for hver tabel i databasen, samt en databasekontekst. For at overskueliggøre dette er der lavet en facade, så man i koden, der skal bruge disse elementer, blot opretter en *Facade*, og ikke alle de forskellige elementer. Dette gør koden mere vedligeholdelsesvenlig, og overskueliggør koden, set fra et højere abstraktionsniveau.

For yderligere info om de truffe beslutninger, henvises til mødereferaterne (Bilag 21).

Systemarkitektur

Domænemodellen, som ses i **Figur 4**, beskriver den overordnede kommunikation i systemet. Brugeren interagerer enten med den lokale GUI (*Fridge app*) eller web-GUI'en (*Web app*). Den lokale GUI får sin information om varebeholdningen fra den lokale database, og har mulighed for at tilføje, fjerne og ændre i dataene. Web-GUI'en får sin information fra den eksterne database, og har samme muligheder for ændring af data som den lokale GUI. Den lokale database og den eksterne database synkroniseres af applikationen som styrer den lokale GUI.



Figur 4 Domænemodel af SmartFridge

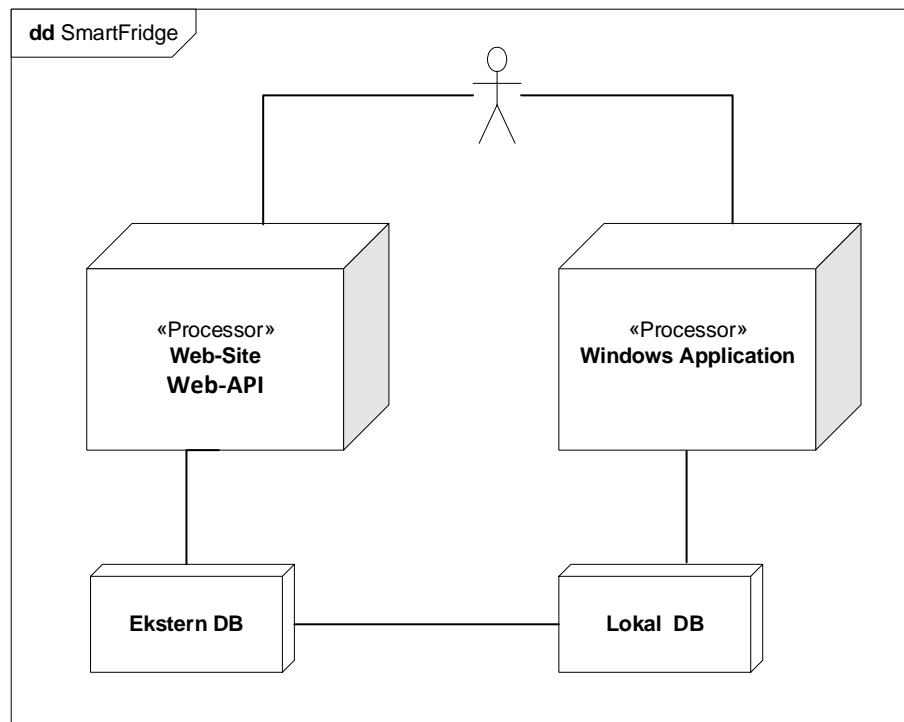
Til den lokale applikation er det valgt at der anvendes 3-lags-modellen; *View*, *Business Logic Layer* og *Data Access Layer*. Derved frakobles meget kode fra *code behind* til *BLL*, og systemet er dermed mere testbart.

For yderligere systemarkitektur, og sekvensdiagrammer for hver *use case*, henvises til projektdokumentationen fra side 12.

Design, implementering og test

Overordnet design

Deployment-diagrammet i **Figur 5** viser, hvordan de interne dele af SmartFridge interagerer. Som det fremgår, interagerer Bruger kun med enten en *Fridge app* eller *Web app*, og har altså ingen direkte interaktion med den bagvedliggende logik.



Figur 5 Deployment Diagram for hele SmartFridge-systemet

For et mere detaljeret *Deployment*-diagram, henvises til **bilag 04**.

Database

I dette afsnit vil designprocessen, implementeringen og test af database-delen af systemet blive beskrevet. Her vil *Data Access Layer (DAL)* for både *Fridge App* og *Web App* blive beskrevet, med de overvejelser der er blevet gjort i designprocessen og implementeringen af *DAL* for begge applikationer.

For mere en detaljeret gennemgang, henvises til projektdokumentationen fra side 19.

Fridge app

Design

Teknologi

Før der har kunnet udarbejdes et design, har det været nødvendigt først at bestemme hvilken teknologi der har skullet anvendes til at implementere *DAL*. Her har de oplagte valg stået mellem *ADO.NET* og

I4 PRJ4, Gruppe 5

IKT

Entity Framework, da det har været de teknologier, der er blevet nævnt i Database-kurset. Her er det blevet valgt at anvende ADO.NET for læringens skyld, foruden at der ville blive overvejet at anvende *Entity Framework* til *Web app*.

Ved brug af *ADO.NET*, har der været en række problemer. Hovedsagligt har brugen af *SQL statements* gjort det vanskeligt at arbejde med og teste. Herudover er det nødvendigt at køre *SSDT*-projektet for at oprette en database, hvilket *Entity Framework* selv administrerer. På flere punkter havde det været lettere at arbejde med *Entity Framework*, dog har det været fordelagtigt i forhold til læring.

Objektmodel

Da der arbejdes med relationelle databaser, har der skullet udarbejdes en objektmodel, hvor data har kunnet gemmes korrekt. Her er det blevet diskuteret hvilke objekter der har været nødvendige at kunne gemme i databasen, og følgende entiteter identificeret:

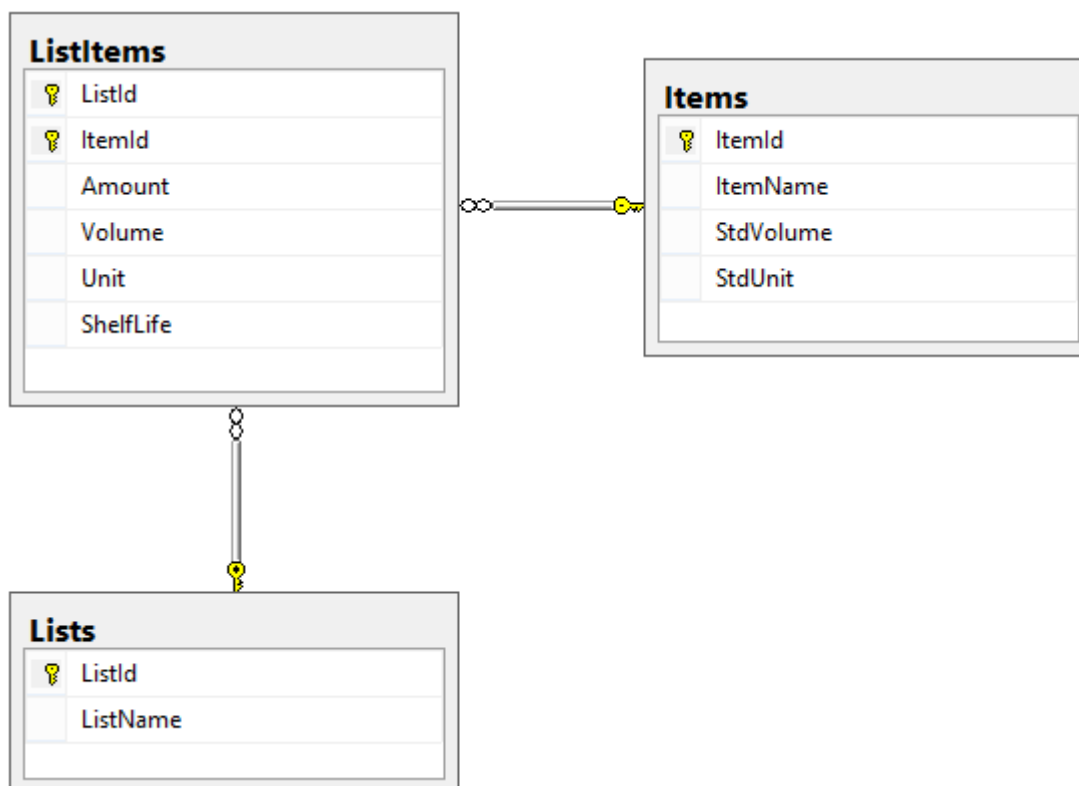
- **Liste (List)**
Eksempelvis køleskab og indkøbsliste. Disse lister skal kunne indeholde de varer som brugeren sætter i køleskabet, og hvad brugeren har på sin indkøbsliste. Denne indeholder listens navn.
- **Vare (ListItem)**
Den konkrete vare, som er på en given liste. Det er f.eks. 2 x 1 liter mælk i køleskabet. Denne entitet indeholder antal, mængde, enhed og holdbarhedsdato.
- **Varetype (Item)**
For at brugeren ikke skal indtaste varetypen hver gang, er varetyper gemt, som f.eks. mælk, æg, hakket oksekød osv. Varetypen skal også have en standardmængde og standardenhed.

Efter disse entities er blevet identificeret, er et ER-diagram blevet udarbejdet, som ses på **Figur 6**, vha. *DDS-Lite*.



Figur 6 ER diagram

Der er udnyttet et *Mange-Til-Mange*-forhold mellem *List* og *Item*, hvor den svage entitet *ListItem* forbinder de to. På denne måde har det været muligt at have den konkrete vare og varetypen adskilt på en fordelagtig måde. Objektmodellen med attributter kan ses på **Figur 7**.



Figur 7 Objekt model med attributter

Som beskrevet, er *ListItem* en *weak entity*, hvilket har introduceret flere udfordringer. *ListItem*'s *update*-funktion er ikke blevet implementeret, ligesom det har været en nødvendighed at implementere en *mapper* for at binde objektmodellen sammen. Dette ville *Entity Framework* have kunnet håndtere, så brug af *Entity Framework* havde været fordelagtigt.

Repository pattern

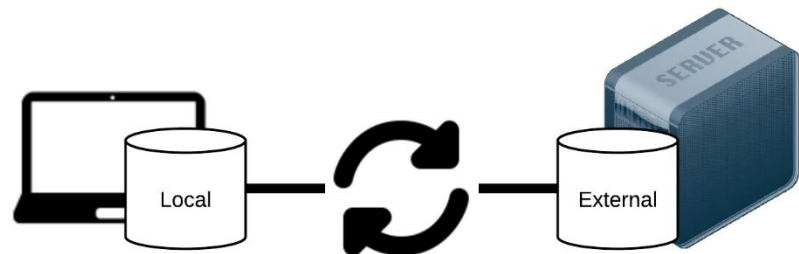
Som en del af designet, er *Repository*-mønsteret blevet anvendt. *Repository*-mønsteret giver mulighed for at have ét sted, hvor al databasetilgang foregår. Formålet med dette er at separere databasetilgangen fra forretningslogikken, hvilket er med til at gøre systemet lettere at vedligeholde, samt lettere læseligt. Herudover giver *Repository*-mønsteret en højere grad af testbarhed, da det vil være muligt at *mocke* repositoryet ud, hvilket gør forretningslogikken testbar. For en mere detaljeret beskrivelse af *Repository*-mønsteret, henvises til projektdokumentationen fra side 20.

Unit of Work

Til designet, er *Unit of Work* blevet anvendt sammen med *Repository*-mønsteret. Formålet med dette, er at alle databasetransaktioner foregår i et *Unit of Work*, hvorpå det er muligt at lave *commits* og *rollbacks*, hvorved transaktionerne er kontrolleret. Dette er fordelagtigt for at sørge for at databasetransaktioner foregår med færre fejl, samtidig med at alle transaktioner bliver *committed* på samme tid, så databasen tilgås i ét hug. For en mere detaljeret beskrivelse af *Unit of Work*, henvises til *Unit of Work*-afsnittet på projektdokumentationens side 21.

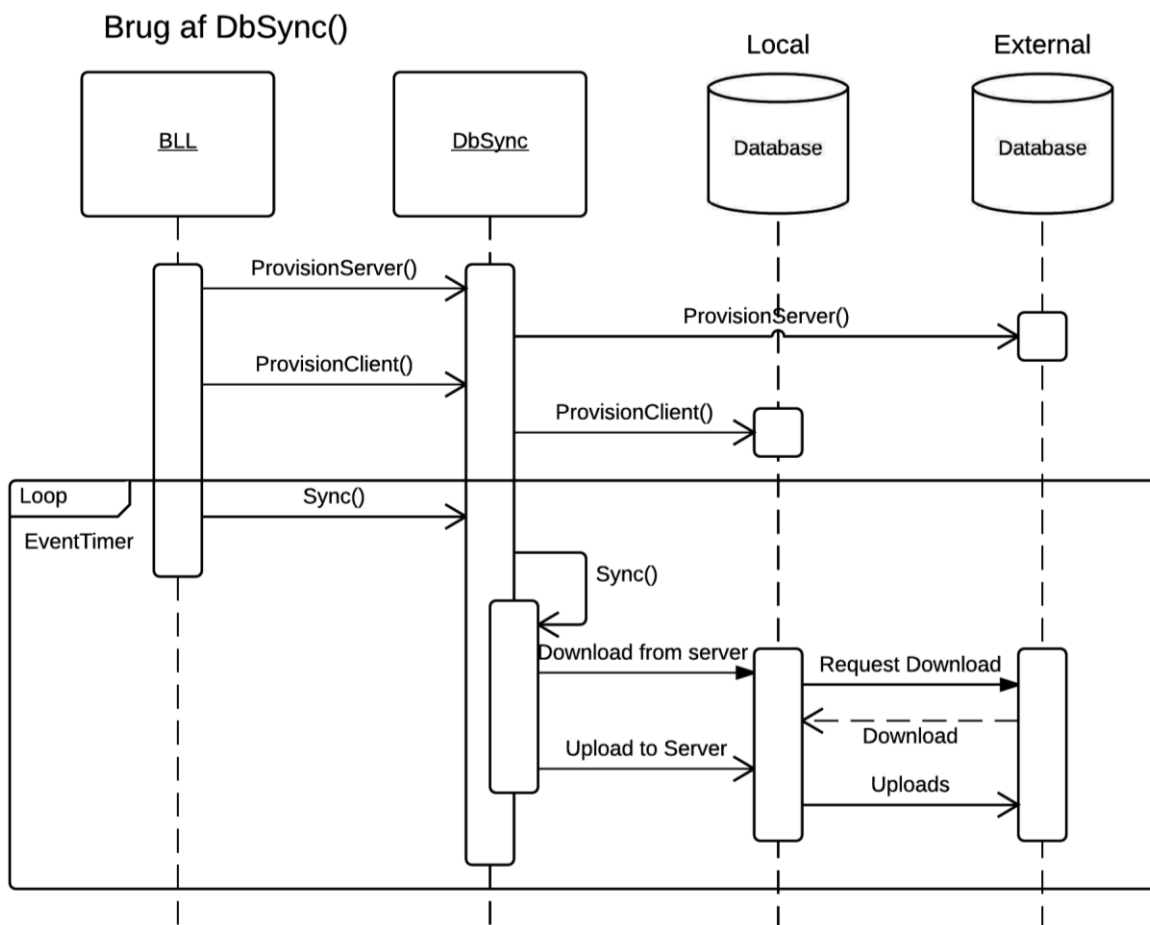
Synkronisering

Da det er et krav at systemet skal kunne fungere lokalt uden internet, har det været nødvendigt at implementere synkronisering mellem en lokal database og en ekstern database. En illustration af dette ses på **Figur 8**. Til formålet er der anvendt *Microsoft Sync Framework*, da dette er en løsning som man selv kan implementere, frem for synkroniseringsværktøjer.



Figur 8 Illustration af synkronisering

På denne måde sørges der for at indholdet af den lokale og den eksterne database altid er ens, hvorved brugeren kan anvende systemet på både *Fridge app* og *Web app*. Et sekvensdiagram for synkronisering kan ses på **Figur 9**, hvor det illustreres hvordan der laves en provision af server-databasen og den lokale database, hvorefter databaserne synkroniseres.



Figur 9 Sekvensdiagram for synkronisering

Endeligt design

Efter de designovervejelser, der er blevet gjort i de forrige afsnit, er der blevet udarbejdet et klassediagram. Klassediagrammet kan ses i **bilag 05**, og uddybes i projektdokumentationen fra side 22. Klassediagrammet er udarbejdet over en iterativ proces, og har dermed ikke været det endelige klassediagram fra start.

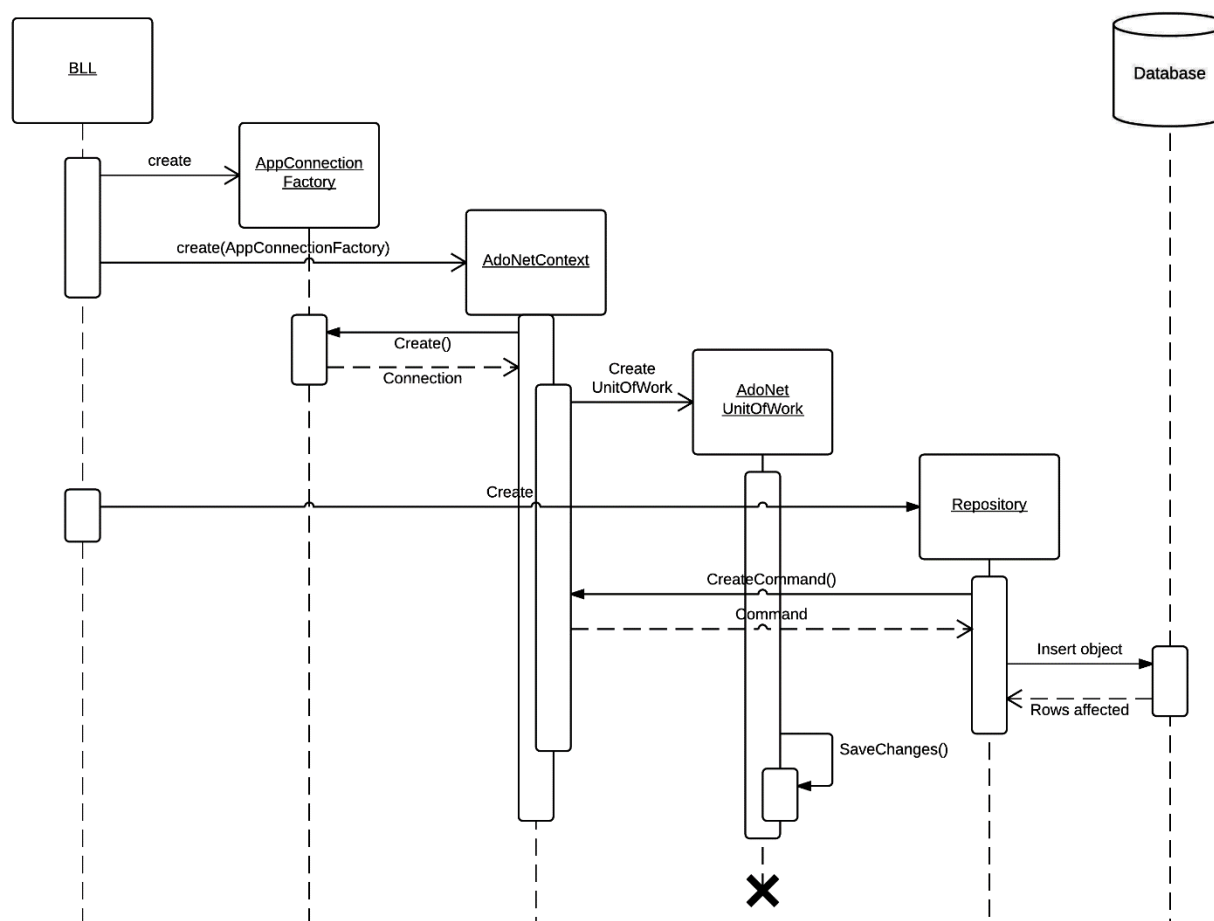
Opbygningen af *DAL*, som klassediagrammet viser, er med en *AppConnectionFactory*-klasse, hvis ansvar er at oprette forbindelse til databasen gennem en *connection string* fra *app.config*. Denne forbindelse *injectes* til *AdoNetContext*, som svarer til *Entity Framework's DbContext*, hvorpå at man kan oprette et *Unit of Work* og eksekvere kommandoer på databasen. I et *Unit of Work*, kan man anvende repositoryet til databasetilgang, som er transaktionstyret. Når man er færdig med sine databaseoperationer, kan man *commite*, og *Unit of Work* kan nedlægges. For en mere detaljeret gennemgang, henvises til projektdokumentationen.

Anvendelse af DAL

Figur 10 viser et sekvensdiagram for en *insert operation* ved brug af *DAL*. Dette beskriver hvordan *Business Logic Layer (BLL)* anvender *DAL* til at indsætte en entitet i databasen.

Først oprettes en *ConnectionFactory*, som beskrevet, der opretter en forbindelse fra en *connection string* i *app.config*. Herefter oprettes *AdoNetContext*, hvor databaseforbindelsen bliver *injected*. Derefter oprettes et *AdoNetUnitOfWork* af *AdoNetContext*, hvori der kan eksekveres databasetransaktioner. Her kan oprettes et *Repository*-objekt til den ønskede entitet, hvorefter de ønskede databasetransaktioner foretages, og herefter *committes* disse transaktioner til databasen. Til sidst nedlægges *AdoNetUnitOfWork*.

Brug af DAL (insert)



Figur 10 Sekvensdiagram for brug af DAL

Implementering

I dette afsnit vil implementeringen af *DAL* blive beskrevet. Alt kode er dokumenteret vha. *XML comments* og *Doxygen*, som kan ses i **bilag 14**. For detaljeret dokumentation, samt dokumentation af væsentlig funktionalitet, henvises til projektdokumentation fra side 26.

Test

I DAL er det meste af funktionaliteten blevet testet. Her er det primært `AppConnectionFactory` med fake `connectionsstring` og `AdoNetContext` og `AdoNetUnitOfWork` med en fake `AppConnectionFactory` for at sørge for at testene ikke tilgår en database. *Repository.cs*, samt dens nedarvede klasser, er ikke testet, hvilket skyldes at disse klassers ansvar er at skrive til databaser med *SQL commands*, hvilket ikke er særlig testbart. Desuden vil disse tests heller ikke give så meget, da det kun er databasetilgang. Herudover er `Sync` heller ikke testet, da der er meget høj kobling til tabellerne, og man ikke kan *mock*'e funktionaliteten ud, er dette heller ikke testet.

Af de testede klasser, er der opnået 100% coverage, der betyder at al koden er blevet berørt, hvilket er et mål i sig selv. Code metrics viser et godt resultat for maintainability, samt complexity. For en mere dybdegående forklaring, henvises til produktdokumentationen fra side 27.

Web app

I dette afsnit vil designprocessen, implementeringen og test af database-delen for *Web app* blive beskrevet, samt de overvejelser der er blevet gjort for database-tilgang fra applikationen.

Design

I dette afsnit vil designprocessen af DAL for *Web app* blive beskrevet. Da objektmodellen, og anvendelsen af *Repository* og *Unit of Work* går igen fra *DAL for Fridge app*, vil de ikke blive beskrevet i dette afsnit. For information om disse, henvises til design-afsnittet for *Fridge app*.

Teknologi

I modsætning til *Fridge app*, hvor der er anvendt *ADO.NET*, anvender *DAL for Web app Entity Framework* til databasetilgang. Dette er valgt for at komme uden om de tidligere beskrevet problemer med brug af *ADO.NET*, såsom abstraktion og testbarhed. Herudover er det også gjort for læringens skyld.

Façade pattern

Foruden anvendelse af *Repository*-mønsteret og *Unit of Work*, er der også anvendt *Façade* til *DAL for Web app*. Formålet med dette mønster er at controllerne har ét sted, hvorigennem de tilgår databasen, hvilket giver højere abstraktion, samt høj testbarhed. For mere detaljeret gennemgang af *Façade*-mønsteret, henvises til projektdokumentations side 28.

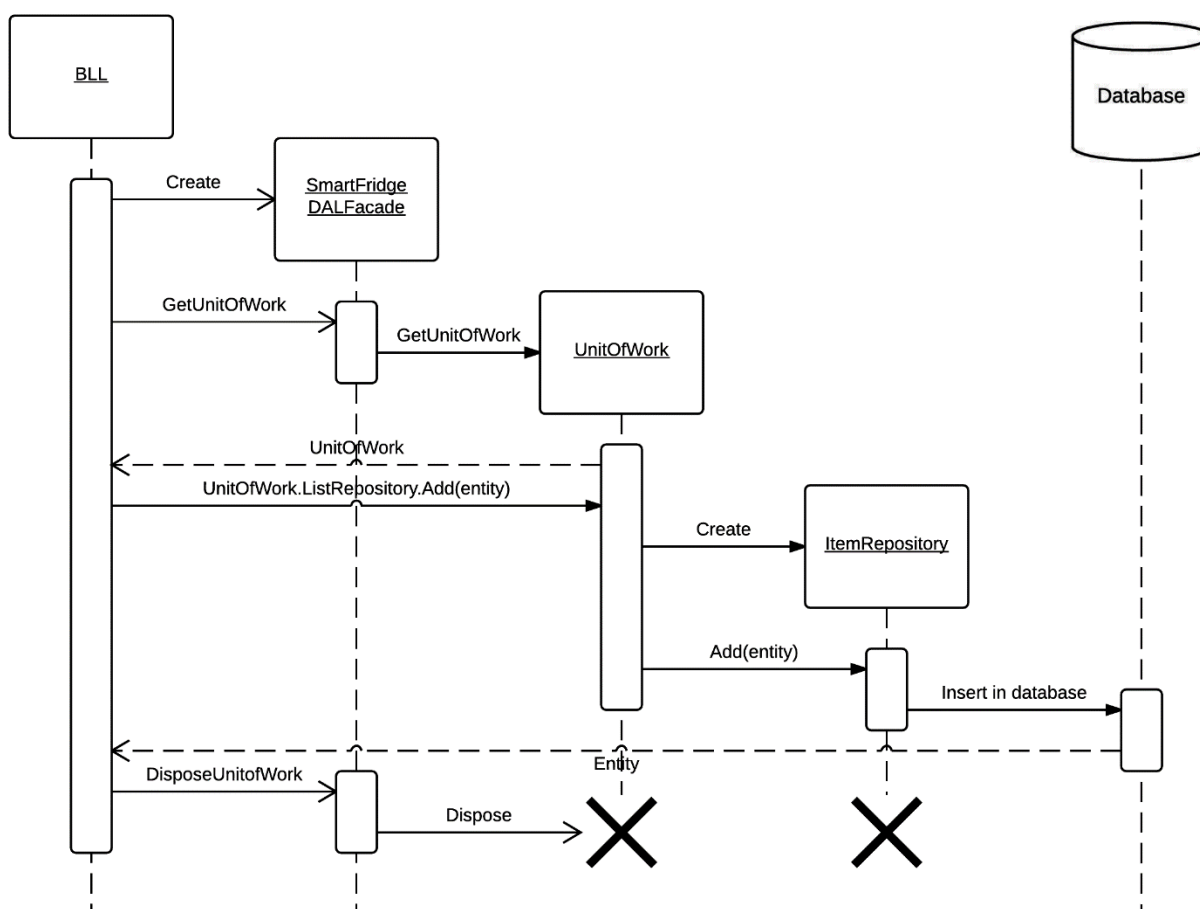
Endeligt design

Efter de designovervejelser, der er blevet gjort i de forrige afsnit, er der blevet udarbejdet et klassediagram. Klassediagrammet kan ses i **bilag 07**, og uddybes i projektdokumentationen fra side 29. Klassediagrammet er udarbejdet over en iterativ proces, og har dermed ikke været det endelige klassediagram fra start.

Her anvendes *Façade*-, *Repository*- og *Unit of Work*-mønstrene, alle sammen med et interface, for at skabe så høj abstraktion og testbarhed som muligt. Det er også væsentligt at *SmartFridgeDALFacade* har ansvaret for at oprette *SFContext*, som er *DbContext* fra *Entity Framework*. Grunden til dette er at der kun ønskes at oprette én *context*, og ikke flere, da dette kan skabe databaseproblemer. For en mere detaljeret gennemgang, henvises til projektdokumentationen fra side 29.

Anvendelse af DAL

Et sekvensdiagram for anvendelse af DAL for Web app, ses på **Figur 11**. Her ses det hvordan BLL (eller en Controller) kan oprette et *SmartFridgeDALFacade*-objekt, hvor der kan oprettes et *UnitOfWork* med *GetUnitOfWork*. Heri kan de ønskede transaktioner laves, hvorefter disse *commit*'es med funktionen *SaveChanges*. Efter brug kan *UnitOfWork dispose*'s, hvorpå det nedlægges og resourcer frigives.



Figur 11 Sekvensdiagram for brug af DAL på Web app

Implementering

I dette afsnit vil implementeringen af *DAL* for *Web app* blive beskrevet. Alt kode er dokumenteret vha. *XML comments* og *Doxygen*, som kan ses i **bilag 15**. For detaljeret dokumentation, samt dokumentation af væsentlig funktionalitet, henvises til projektdokumentationen fra side 31.

Test

I *DAL* for *Web app*, er det meste af funktionaliteten blevet testet. I forskel til *DAL* for *FridgeApp*, har der været interfaces for hver klasse, hvilket har gjort unit testing meget simplere, da der har været lav kobling mellem klasserne. Som i *DAL* for *Fridge app*, er repositoryet ikke blevet modultestet, da det igen er databasetransaktioner, og det ikke er egentligt funktionalitet at teste. Herudover er *SFContext* heller ikke testet, da det kommer fra *Entity Framework*, som må anses som gennemtestet.

Her er blevet opnået 100% coverage af den testede funktionalitet, som i *DAL* for *FridgeApp*. Dog er Code Metrics bedre end *DAL* for *FridgeApp*, hvilket interfaces samt *Entity Framework* har medvirket til. For en mere dybdegående forklaring, henvises til produktdokumentationen fra side 32.

Fridge app

I dette afsnit vil der blive set på processen omkring det visuelle design af *Fridge app*, såvel som den konkrete implementering af systemet, og de mere bemærkelsesværdige funktioner. Ligeledes vil der blive redegjort for overvejelserne omkring systemets testbarhed.

Design

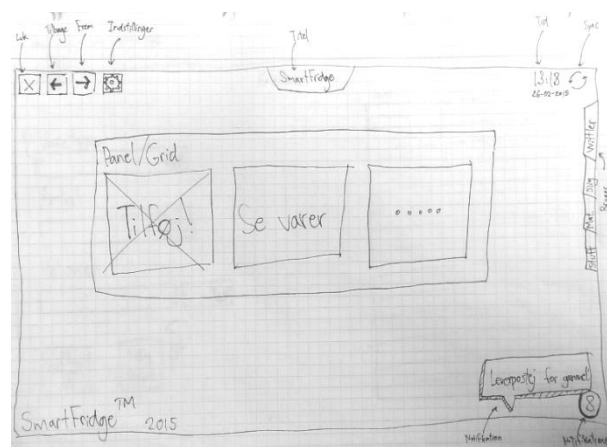
Designet på *Fridge app* er, på baggrund af en forudgående brainstorm, i første omgang skitseret i hånden, hvorefter modelleringen i WPF har taget udgangspunkt i disse skitser.

Dette afsnit vil beskrive de ønsker til designet, som kom frem i brainstormen. For flere detaljer omkring designet, henvises til projektdokumentationen fra side 34.

Helt overordnet består designet, som skitseret på **Figur 12**, af hovedvinduet, som danner en ramme om en *User Control*; et vindue som dynamisk kan skifte indhold.

Størrelse

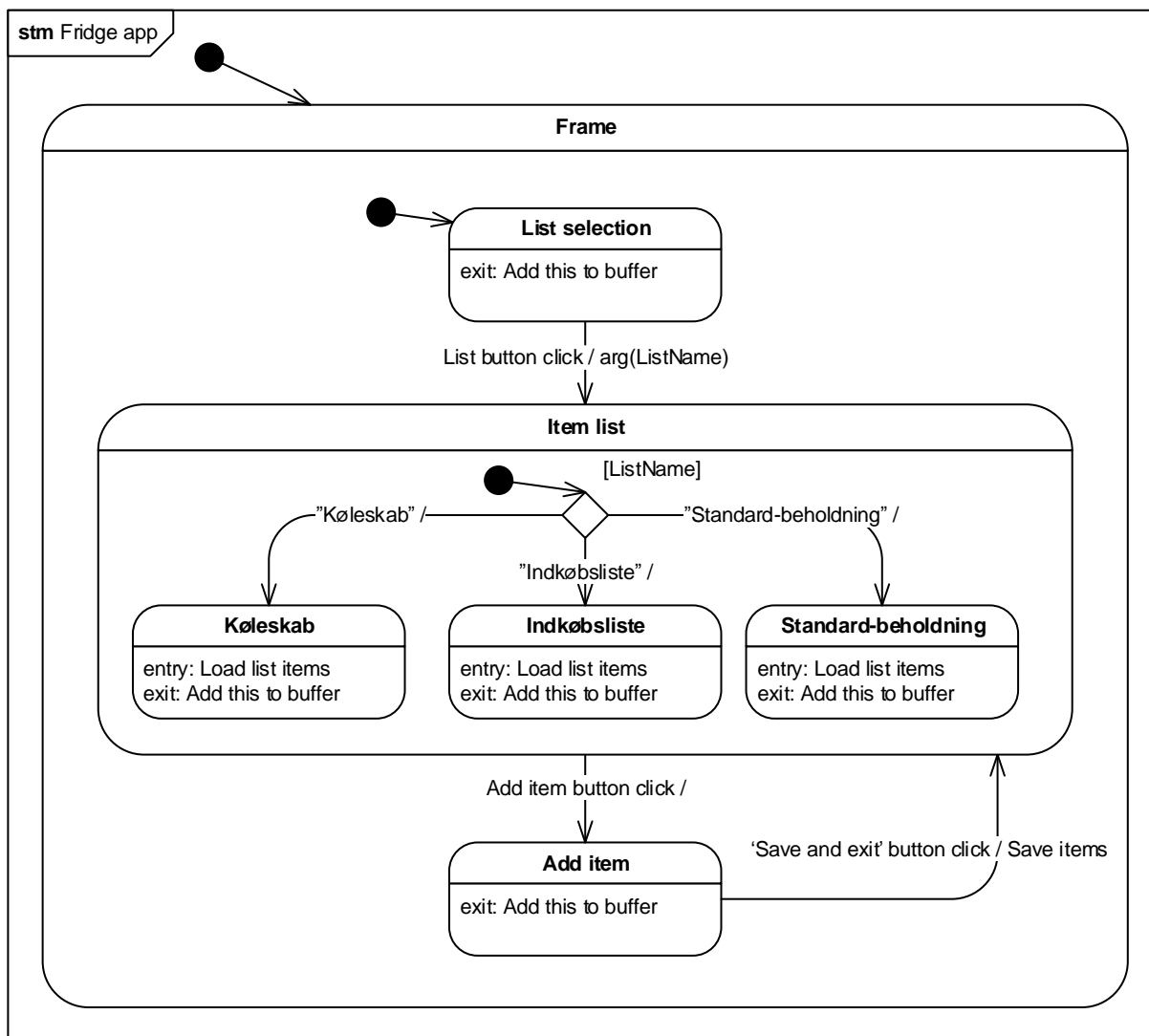
På baggrund af det apparat, systemet i første omgang udvikles til, designs *Fridge app* med en størrelse på 1920x1080 pixels, mens de *User Controls*, som vises i midten af skærmen, har en størrelse på 1280x720 pixels.



Figur 12 Skitse af hovedmenu og ramme

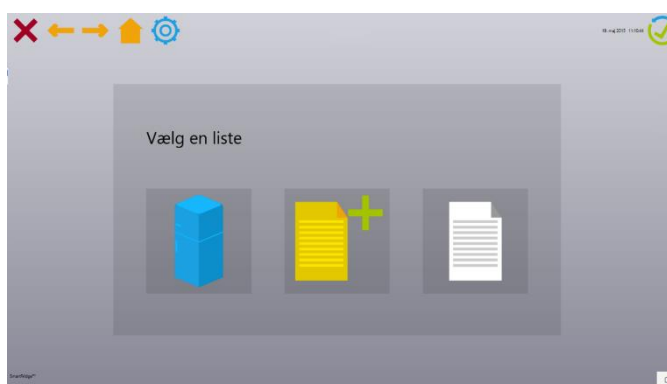
Implementering

Den overordnede navigation i *Fridge app* illustreres som et *state machine*-diagram i **Figur 13**.



Figur 13 State machine diagram over Fridge app

Bemærk at rammen, som til enhver tid omkranser de forskellige *User Controls*, har knapperne *Hjem*, *Frem* og *Tilbage*, som øger lethededen af navigationen, ved til enhver tid at tilbyde muligheden for at gå til listeoversigten, samt en side frem eller tilbage. Disse knapper er ikke illustreret på **Figur 13**.



Figur 14 Hovedmenu

Ved åbning af programmet, indlæses rammen, som indlæser listeoversigten. Ved valg af en liste, udskiftes listeoversigten med den valgte liste; *Køleskab*, *Indkøbsliste* eller *Standard-beholdning*. Ved indlæsning af disse lister, hentes informationerne om deres tilknyttede varer fra databasen, og indlæses i en liste.

Fra listerne er det muligt at redigere varerne, og det er muligt at tilgå den sidste *User Control*, *Tilføj vare*.

I vinduet *Tilføj vare* er det muligt at tilføje varer til den pågældende liste, som har fremkaldt det. Efter de ønskede varer er tilføjet, er det muligt at gemme, hvorved varerne persisteres, og der returneres til *Vis varer*.

View

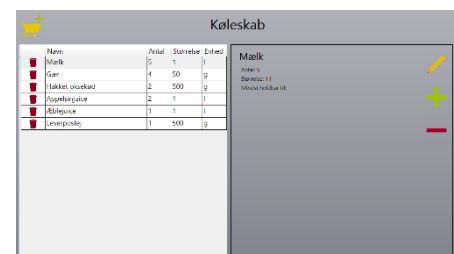
Viewet er blevet implementeret i nogenlunde overensstemmelse med det oprindelige design.

Fra hovedvinduet indlæses listeoversigten, som vist i **Figur 14**. Herfra er det muligt at tilgå de eksisterende lister, som i første omgang består af *Køleskab*, *Indkøbsliste* og *Standard-beholdning*.

Fra den valgte liste er det muligt at se, tilføje, redigere og fjerne varer, som vist i **Figur 15**.

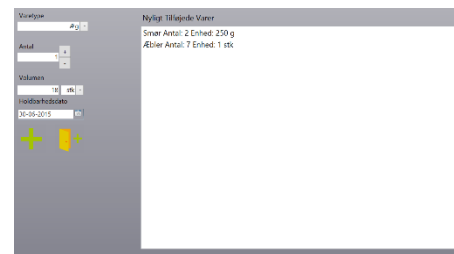
Vælges der at tilføje varer, er det muligt med vinduet vist i **Figur 16**.

For mere detaljerede beskrivelser af *View*-delen af *Fridge app*, henvises til projektdokumentations fra side 36.



Navn	Antal	Enheds	Enheds
Mælk	5	l	l
Gær	4	50	g
Tøkket (vasket)	2	500	g
Suppen (vasket)	2	1	l
Blåbær	1	1	l
Lønspode	1	500	g

Figur 15 Listen "Køleskab"



Figur 16 Tilføj vare(r)

Business Logic Layer

Business Logic Layer er blevet implementeret på en sådan måde, at det står for at facilitere al kommunikation mellem *View* og *Data Access Layer*. Det konverterer *ListItem* og deres tilhørende *items* fra databasen til et samlet objekt i form af et *GUIItem*, som bruges af *viewet*, og omvendt når der skal tilføjes til databasen. Det er også i dette lag, der laves tjek på hvorvidt en vare har overskredet dens holdbarhedsdato, og hvis den mangler fra standard-beholdningen, få den tilføjet til indkøbslisten.

Test

I begyndelsen og under udviklingen af projektet, var målet at skrive unit tests til hver enkelt implementeret funktion. Dette blev af flere årsager ikke realiseret.

Mange af funktionerne i projektet er lavet ud fra "*button click*"-funktioner, som genereres efter behov ud fra knapperne i viewet. I disse funktioner blev der, i starten af projektet, implementeret al logik, som senere overført til projektets *Business Logic Layer (BLL)*. Konsekvensen af dette blev at "*button click*"-funktionerne blev svære at teste vha. *NUnit*-frameworket, da det ikke har været muligt at simulere et museklik, når både eventet og funktionen er private. Ydermere blev det besluttet af disse tests ikke var relevante i samme grad som tests af *BLL*-laget, hvor størstedelen af den reelle databehandling fandt sted. Her blev der, så vidt muligt, lavet unit test, dog mest fokuseret på integrationstests.

Integrationstestene blev, pga. manglende unit tests og den høje kobling, et krav til projektet. Mange af disse tests er baseret på at hente, slette og manipulere data fra databasen.

Skulle problematikken med unit testene være undgået, skulle projektet have været bygget mere op om et mønster/*pattern*, der havde gjort det mere testbart. Dette mønster kunne eksempelvis have været MVVM-mønstret, som adskiller modellerne, viewet og controlleren bedre end måden, hvorpå denne applikation er opbygget. Med dette mønster ville koblingen blive lavere, og ingen tests af viewet være nødvendige, da der ikke ville være noget code-behind. For mere information om overvejelserne omkring forskellige mønstre, henvises til projektdokumentations side 53.

Web app

Design

Valget af *ASP.NET* er baseret på den funktionalitet som frameworket giver, frem for hvis *Web app* udelukkende var blevet lavet i HTML. Vha. *ASP.Net* er der mulighed for at manipulere med det data som brugeren indtaster og det data der ligger i databasen, i C#.

Når der laves en web-applikation i *ASP.NET*, skal der træffes et valg om hvilke frameworks og mønstre, man ønsker at bruge til at udvikle applikationen. Da der skulle besluttet hvordan applikationen skulle udvikles, var to metoder under overvejelse: *ASP.NET Web Forms* og *ASP.NET MVC*. *Web Forms* og *MVC* benytter sig begge af HTML, men ifølge forskellige websites (bl.a. (Sukesh, 2014) og (Hambrick, 2013)), har udvikleren mere kontrol over renderingen af den HTML-kode som brugeren ender med i browseren, hvis der benyttes *MVC* i forhold til, hvis der benyttes *Web Forms*.

De *View*-filer, der benyttes, er *.cshtml*-filer i *Razor*-syntax (W3Schools, u.d.). Dette er valgt på baggrund af hvor nemt det er at bruge, hvis man har HTML-færdigheder. For at lave et *View*, skal der blot benyttes HTML, og hvis der skal tilføjes funktionalitet kan der kaldes en funktion i en controller, eller der kan skrives C#-kode direkte *Viewet* vha. '@'-annotationen. *Web Forms* bruger *drag'n'drop*-designmetoden i stedet for *HTML*-kodning. Dette gør det mere visuelt at designe en applikation, men har ikke været en nødvendighed under dette projekt.

MVC-mønsteret er primært valgt på baggrund af to ting. Den første har været, at det har skullet være nemt at køre tests på, og den anden har været den faglighed, som er opnået omkring mønsteret i både *I4SWT*- og *I4GUI*-fagene.

Til at lagre applikationens data, er *MSSQL* valgt over *MySQL*. Grunden til dette er muligheden for synkronisering mellem den online database og den lokale. Det er muligt at synkronisere mellem en *MySQL*-og en *MSSQL*-database, men da Windows-applikationen allerede benyttede sig af *MSSQL*, var det nemmere at lave tilsvarende til webapplikationen.

Det overordnede design for web-applikationen minder meget om designet for *Fridge app*, men der er dog et par små forskelle. Den væsentligste forskel fra designet af *Fridge app* er at *Rediger vare* er kommet ud i et *View* for sig, i stedet for at være en del af en *Item List*. For en mere detaljeret beskrivelse af designet af *Web app* henvises til projektdokumentationens side 54. En stor del af overvejelserne går på at web-applikationen skal kunne bruges på tværs af platforme. Da ikke alle enheder har lige store skærme, eller lige høj opløsning, har det været vigtigt, ikke at have for meget i hvert *View*. I nogle tilfælde har dette blot betydet en omstrukturering af det i forvejen eksisterende *View*, og i andre tilfælde, som med *Rediger vare*, er der blevet lavet et helt nyt *View*.

Implementering

Til oprettelsen af *Web app*-projektet er *Visual Studio* benyttet. *Visual Studio* har hjulpet med at oprette et projekt, der benytter sig af MVC-mønsteret, og som bruger Razor engine. Ydermere sættes det nemt op til *Azure* (Microsoft, 2010). Mappestrukturen bliver herved givet af *Visual Studio*. Mere om dette ses i projektdokumentation fra side 56.

Når data skal sendes fra et *View* til en *Controller*, benyttes *HTML Helpers*. Med disse hjælpere er det muligt at kalde en funktion i en *Controller* og sende information med. Dette kunne være information om et nyoprettet item. Ydermere findes der knapper og tekstbokse, som bliver renderet på Viewet som HTML-kode. Disse knapper hjælper også med at eksekvere kode i controlleren. Se et eksempel i **Kodestump 1**.

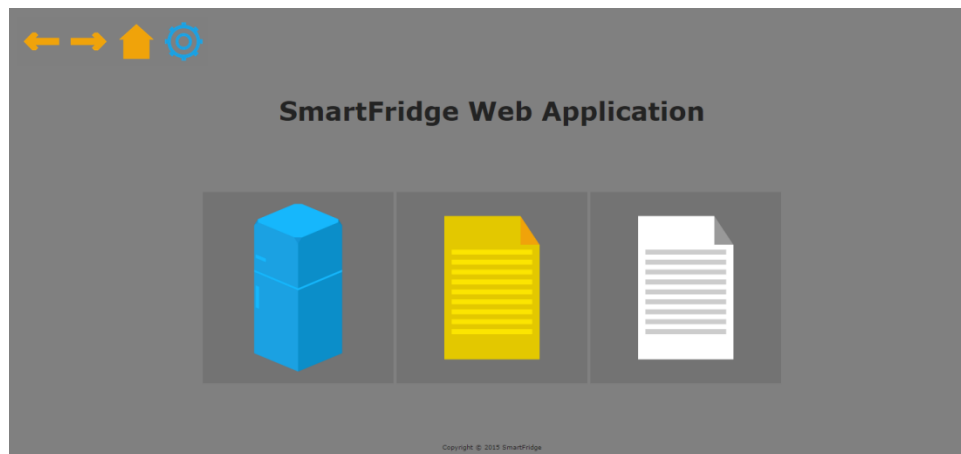
```
@Html.ActionLink("About this Website", "About")
```

Kodestump 1 Razor-syntax til en HTML Helper (W3Schools, u.d.)

View

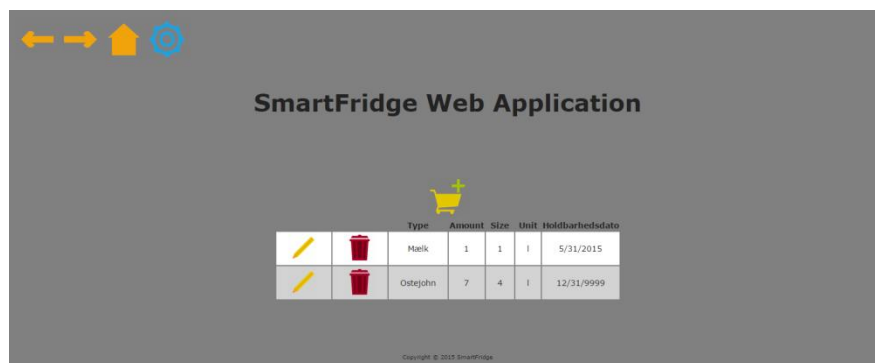
Grundet tidspres og faglig irrelevans er der ikke blevet fokuseret så meget på *View*-delen af web-applikationen. Til trods for dette, er *Views*'ne kommet til nogenlunde at ligne de wireframes der findes fra side 64 i projektdokumentationen. Der har dog primært været fokus på funktionalitet over udseende.

På **Figur 17** ses *Index*, der svarer til hovedmenuen i *Fridge app*, hvorfra en eksisterende liste vælges.



Figur 17 *Index*

Når en liste er valgt bliver brugeren præsenteret for et *View*, der minder om det, som kan ses på **Figur 18**. Dette *View* viser indholdet af den valgte liste, og giver mulighed for at navigere videre til enten *Tilføj vare* eller *Rediger vare*.

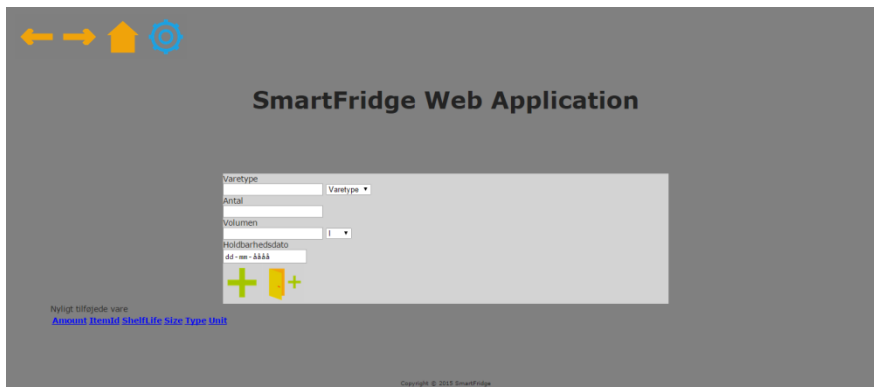


Figur 18 Listen "Køleskab"

I4 PRJ4, Gruppe 5

IKT

Vælges *Tilføj vare* bliver brugeren ført videre til vinduet på **Figur 19**, hvorfra der kan tilføjes varer til den nuværende liste.



SmartFridge Web Application

Varetype: Varetype:

Antal:

Volumen:

Holdbarhedsdato:

Nyligt tilføjede vare

Amount	ItemId	ShelfLife	Size	Type	Unit
--------	--------	-----------	------	------	------

Copyright © 2015 SmartFridge

Figur 19 Viewet "Tilføj vare"

Vælges *Rediger vare*, præsenteres brugeren for vinduet på **Figur 20**, hvor tekstboksene vil være fyldt med den nuværende vareinformation.



SmartFridge Web Application

Varetype: Varetype:

Antal:

Volumen:

Holdbarhedsdato:

Copyright © 2015 SmartFridge

Figur 20 Viewet "Rediger vare"

Foretages der ændringer, gemmes disse når der trykkes på fluebenet, hvorefter der returneres til den liste, som tidligere blev valgt. Trykkes der på krydset, returneres til listen, og vareinformationen er, som den var før.

Controllers

Samlet set udgør *controllerne* det, der svarer til *Business Logic Layer* fra *Fridge app*; de varetager kommunikationen med den eksterne database. Hver *Controller* har sit ansvar. *ListView* står for at indlæse og præsentere alle varer på en liste, og giver mulighed for at slette en udvalgt vare. Den står også for at omdirigere til de to *views*, *AddItem* og *EditItem*. Inden *controlleren* omdirigerer til *EditItem*, finder den først den vare, der skal sendes med til *EditItem*-controlleren, før der omdirigeres. *AddItem-controlleren* har til ansvar at tilføje varer til databasen. Finder den en tilsvarende vare på den nuværende liste, sørger den for blot at øge mængden af den eksisterende vare med det tilføjede antal, i stedet for at indsætte en dublet. *EditItem-controlleren* står for redigering og opdatering af en valgt vare. *EditItem* er implementeret således der ikke optræder dubletter på listen; i disse tilfælde håndteres det som i *AddItem*.

Cache

Til *Web app* er der blevet implementeret en *cache*. Denne *cache* indeholder alle de ting, som er fælles for alle *Controllers*. Klassen består kun af properties, der alle er statiske. Begrundelsen for dette er netop at det er fælles data, som alle controllers skal arbejde på. Det er også i *cache*n at facaden til *Data Access Layer* ligger, da der på alle tidspunkter kun må være én facade.

Test

Da webapplikationen er bygget op omkring MVC-mønsteret skal der kun testes på *Controller*- og *Model*-klasserne i projektet, da det er her, funktionaliteten i applikationen ligger. Pga. den lave kobling, som dette mønster giver, er *unit tests* nemme at lave. Udfordringen i dette projekt har været at koble den logik, som er lavet til *Fridge app*, over til *Web app*. For at sikre kontinuitet mellem de to, er den overordnede designovervejelser til logik blevet genbrugt.

Et af problemerne ved den måde, hvorpå funktionerne i applikationen er opbygget, er at mange af dem returnerer et *View* eller et *RedirectToAction*, som set **Fejl! Henvisningskilde ikke fundet.**, efter et item er gennemgået anden logik i samme funktion. Et eksempel på dette kunne være i *EditItemController* hvor funktionen *UpdateItem* er implementeret. Denne funktion tager imod en *FormCollection* hvori de nye værdier, der skal ændres i det gamle item, ligger. Logikken i funktionen ændrer det gamle item til de nye værdier, men returnerer det *View*, som skal vises efter *item*'et er ændret. Dermed er det ikke muligt blot at sammenligne returværdien fra funktionen med hvad der forventes, at det pågældende item er ændret til. I stedet skal der testes på om der eksisterer et *item* i listen med *items*, der har de rette værdier. Dette går igen med alle de funktioner, der har *ActionResult* som returparameter.

```
return RedirectToAction("ListView", "LisView");
```

Kodestump 2 - Udsnit af hvad der returneres fra UpdateItem-funktionen i EditItemController.cs

Tests af funktionaliteten i *Web app* har ikke været et af vores store fokusområder. Selv om en af grundene til valget af MVC-mønsteret var at det skulle være langt mere testbart, blev selve implementeringen af funktionaliteten prioriteret højere. De test der er udført har haft til formål at berøre alt koden, og derved give 100% coverage.

Resultater og diskussion

Systemet er blevet implementeret med alle dets kernefunktionaliteter. SmartFridge er defineret som et digitalt køleskab med muligheden for at se, hvilke varer der er i brugerens fysiske køleskab, og dette mål er opnået. Gruppen har ikke nået at implementere alle de udvidelser, der er blevet foreslået, men det har heller ikke været hensigten at implementere samtlige udvidelser. Synkroniseringen er blevet implementeret, og virker med mange ting, men ikke ved synkroniseringen af to vare-instanser med samme type. Løsningen er et redesign af databasen, og det er blevet vurderet, at der ville være for meget refaktoreringsarbejde i denne iteration.

Undervejs er der blevet foretaget nogle valg, der gjorde at produktets udvikling blev sværere. Blandt andet er der valgt *ADO.NET* frem for *Entity Framework*. Dette valg har gjort det sværere at ændre objektmodellen af databasen. Herudover lærte vi alt for sent at vi skulle have brugt *MVVM pattern* helt fra starten af til design af *Fridge app*. Valget af 3-lags-modellen har medført at meget af koden ikke er testbart, da der stadig er en del kode i *code behind*. Dog er der flyttet funktionalitet ud af *UserControl*-biblioteket og ind i *Business Logic Layer*, som er blevet grundigt testet.

Generelt har vi haft problemer med at de ønskede funktionaliteter ofte er kommet forud for undervisning. Som eksempel er der brugt meget tid på at undersøge og lære at benytte *ASP.NET MVC*, før den nødvendige undervisning i emnet blev påbegyndt i faget I4GUI.

På trods af ulige timing af undervisning kontra implementering, er produktet endt med at være tilfredsstillende, hvor alle krav er blevet godkendt i accepttesten.

Den nuværende iteration indeholder:

- *Fridge app* til Windows-platform.
- *Web app* til brug uden for hjemmet.
- Lokal database til persistering af data.
- Ekstern database(Azure) til at kunne se varer på *Web app*.
- Synkronisering af lokal og ekstern database.
- Notifikationer vedrørende forældede varer på *Fridge app*.
- Sammenligning af varer på standard-liste og køleskabs-liste, for at tilføje differencen på indkøbslisten. Implementeret på *Fridge app*.

Udviklingsværktøjer

Herunder gives en kort beskrivelse af relevante udviklingsværktøjer benyttet i projektet.

Git

Til organisering af filer, både kode og dokumentation, er der blevet benyttet et Git-repository fra GitHub. Dette er et system til fildeling, hvor der samtidig tages højde for filhistorik. Det har været muligt at sidde flere personer på samme fil på en gang, for derefter at 'merge' indholdet sammen til én fil.



Microsoft Visual Studio 2013

Udviklingsværktøjet er blevet brugt til at skrive stort set alt kode. Databasen, *Fridge app* og *Web app* er alle blevet udviklet i Visual Studio. Alle *frameworks* nævnt herunder er 'udvidelser' til dette værktøj.



Microsoft Visio 2013

Et tegneværktøj, som er benyttet til de fleste UML-diagrammer for systemet.



LucidChart

Et tegneværktøj, der kan bruges til at lave diverse UML-diagrammer til databasen.



DDS-Lite

Et værktøj, der gør det muligt at designe en relationel database, og herefter generere de nødvendige scripts til at bygge databasen. Er benyttet til at designe den lokale database.



Entity Framework

Et bibliotek, der gør det muligt at arbejde i et objektorienteret .NET-miljø, som kan mappes direkte til en relationel database.



Microsoft Sync Framework

Et bibliotek, der hjælper med synkroniseringsprocessen mellem den lokale og den eksterne database.



NUnit

Framework benyttet til systematiske tests, for at sikre kodens kvalitet ved ændringer.



NSubstitute

Framework brugt i forbindelse med tests. Bruges for dynamisk at substituere undermoduler i det modul som testes, frem for manuelt at skrive *fakes*.



Doxygen

Værktøj benyttet til at generere overskuelig dokumentation for programkode.



Opnåede erfaringer

Kristoffer Lerbæk Pedersen

Det har været en fornøjelse at arbejde på dette projekt, og en ny oplevelse at arbejde på et system som udelukkende har været softwareorienteret. Det har, for mig, været det mest givende projekt at være en del af, selv om der har været både fordele og ulemper i forhold til tidligere semesterprojekter.

Den sværeste ved dette projekt, har været at jeg ikke har kunnet have fingrene i alt software-relateret, som det tidligere har gjort sig gældende. Systemet har simpelthen været for omfattende, og jeg har måttet give slip fra fordybelsen i nogle områder, for at kunne arbejde mere fokuseret på nogle andre. Hvor jeg, under udviklingen af *WPF*-applikationen kunne være rigtig meget med, og spillede en stor del i designet og den grundlæggende konstruktion, har jeg ikke haft meget med databaselaget at gøre. På samme måde var jeg med i starten af udviklingen af web-applikationen, men idet det lå så sent i forløbet, og jeg endte med det primære ansvar for konstruktionen af rapporten, har jeg også her været nødt til at prioritere min tid.

Fordelene ved projektet har heldigvis været mange. Jeg føler at de fag, vi har haft på 4. semester, har været de mest brugbare, i forhold til at føle mig klar til at komme ud i en virksomhed og være en del af et erfarent team, og jeg har virkelig kunnet mærke i dette projekt, at der har været brug for den viden, jeg har tilegnet mig i løbet af semesteret.

Som ved alle andre semesterprojekter, kunne det være sjovt at prøve at lave det forfra igen, med den viden, og de erfaringer, som jeg har draget mig siden projektets påbegyndelse. Jeg er sikker på at udviklingen ville gå markant hurtigere, og at kvaliteten af produktet ville være kraftigt øget, idet diverse mønstre og principper denne gang ville blive taget i brug. Det til trods, er jeg rigtig godt tilfreds med det produkt, vi har fået udviklet.

Jeg synes at gruppen har fungeret godt, og haft en god dynamik. Til møder og gruppearbejde har der i perioder været tendens til at miste fokus lidt for ofte, men som med alle andre udfordringer, har vi været gode til at komme på løsninger på dette, og det blev aftalt af Rasmus og jeg ville være opmærksomme på dette, og sørge for at bringe samtalen tilbage på sporet igen.

Alt i alt har det været en positiv oplevelse, og jeg glæder mig til at præsentere produktet til eksamenen.

Mathias Siig Nørregaard

Dette projekt har for mig været det mest relevante projekt, i forhold til at studere til IKT-ingeniør. Sidste projekt var blandet Elektro og IKT, men det projekt tog meget udgangspunkt i embedded software. Det har sådan set været fint, men for mig har det været det vigtigere at få erfaring inden for et felt, som jeg personligt har tænkt mig at arbejde med som færdiguddannet ingeniør.

Det kan være svært at træffe den rigtige beslutning fra starten. Nogle gange, trods teknologiundersøgelser, er det ikke altid til at vide at der findes et bedre alternativ. Jeg føler vi har lært

I4 PRJ4, Gruppe 5

IKT

meget ved nogle gange at tage forkerte beslutninger, som eksempelvis at projektet skulle have været opbygget med et MVVM-pattern. Jeg sidder med følelsen af, at hvis vi skulle lave det samme projektet forfra nu, vil vi kunne arbejde meget nemmere samt levere et markant bedre produkt. Jeg synes at efter man har arbejdet med projektet og fulgt semestrets forskellige fag, er man klart mere parat til at komme ud i en virksomhed og arbejde sammen med professionelle softwareudviklere.

Som gruppe synes jeg, vi har arbejdet fint sammen. Jeg var glad for vi skulle være seks mand, i stedet for otte som sidste semester. Den mindre gruppestørrelse medførte at man fik mere ansvar, som jeg ser som en prioritet.

Det har fungeret godt at have et taskboard og en tidsplan at følge. Modsat andre semestre, har vi været bedre til at afgrænse projektet og sætte mere tid af til rapportskrivning, hvilket jeg ved er utrolig vigtigt. Det var dog ærgerligt at vi aldrig fik Jenkins (continuous integration) op at køre, trods de mange timer, der er blevet brugt på det. Jeg tror muligvis, det kunne have givet et skub i den rigtige retning ift. tests og integration af database og applikation.

Opsummerende har det været et spændende og lærerigt projekt – jeg vil tage erfaringen med mig videre på de næste semestre.

Mathias Schmidt Østergaard

Dette projekt har været det mest relevante for mig i løbet af min studietid. Opgaven har været meget fri, hvilket har givet mig god mulighed for at lave noget forskelligt. Lige fra en WPF-applikation hvor XAML og C# har været i førersædet, til en web-applikation, hvor HTML og C# blev blandet godt sammen. Selvfølgelig er det, ligesom de tidligere projekter, en læringsproces hvor vi hele tiden er blevet klogere på hvordan projektet skulle laves. Hvis noget skulle være lavet om, havde det nok været WP-applikationen. Den kunne have fulgt et pattern som var nemmere at teste. Men resultatet blev godt alligevel.

Gruppearbejdet er gået rigtig godt. Vha. Scrum-værktøjet på vores Redmine-board har jeg ikke været i tvivl om hvad jeg skulle lave på noget tidspunkt. Kommunikationen i gruppen har primært foregået over Facebook eller i en dialog til møderne, hvilket har fungeret rigtig godt. Der har fra tid til anden også være en smule useriøsitet, men dette er ikke noget der har været et problem for mig eller arbejdet i gruppen.

Hvis jeg skal sætte fingeren på noget der måske kunne forbedres, så ville det nok være på et enkelt område. Nemlig databasen. Mathis har klaret denne opgave meget godt, men vi andre har også været meget afhængige af hans arbejde. Skulle dette projekt laves på ny, skulle vi nok have haft en enkelt mand eller to mere på denne opgave. Jeg har selv været på mange forskellige opgaver i de forskellige lag i projektet, hvilket har været fedt, at prøve lidt af det hele.

Alt i alt er jeg tilfreds med min egen præstation, og mener vi har haft et godt gruppearbejde.

Mathis Malte Møller

Dette semesterprojekt har været det mest interessante af de fire projekter, jeg har lavet. Dette har været et rent softwareprojekt, hvor jeg siddet som softwareudvikler i et projektteam.

I projektet har jeg siddet meget alene og arbejdet med database-delen af projektet, hvor jeg har tilegnet mig en masse viden indenfor især *SQL*, *ADO.NET*, *Entity Framework*, som vi har lært i database-kursuset, og også *Microsoft Sync Framework*.

Jeg har set det som utroligt spændende at lå lov til at fordybe mig i en del af et projekt, så man kunne stå med et godt *Data Access Layer*, dog har det på andre punkter været ærgerligt ikke at arbejde sammen med andre om DAL, da man ikke havde nogen at sparre med.

Jeg mener at projektstyringen, som at have en agil tilgang, med elementer fra Scrum, har været en effektiv form, hvorpå folk altid har haft noget at gå i gang med, og det var sjældent at folk var helt færdige med deres arbejde, hvilket ikke gjorde noget, da det tilføjede noget ekstra pres, så folk blev hurtigt færdige.

Da vi har haft en fungerede gruppekontrakt, har der for meste været høj disciplin under møder og gruppemøder, men hvis folk kom til at pjatte for meget, blev dette også rettet op på. Generelt synes jeg at folk har arbejdet seriøst, når de skulle.

Jeg synes at gruppedynamikken har været god, også selvom jeg ikke har haft mange opgaver, hvor jeg arbejdede sammen med folk. Hvis der har været et problem, har folk hjulpet hinanden, og folk var gode til at snakke om mulige løsninger på tekniske problemer. Folk har virket tilfredse omkring arbejdet, og hvis folk var utilfredse, blev det løst, hvilket jeg mener er vigtigt for et godt samarbejde.

Mikkel Koch Jensen

Dette projekt har for mig været det mest interessante, især fordi at det af de fire semesterprojekter også har været det mest relevante. Den agile tilgang til projektet har efter min mening fungeret rigtig godt, da det har medført at der hele tiden var stor sikkerhed omkring hvad hver enkelt medlem skulle lave. Det har også betydet at der næsten hele tiden har været en fungerende udgave af produktet, hvilket jeg synes har været meget motiverende. Generelt set har jeg i det hele taget syntes at den agile tilgang har fungeret rigtig godt.

Alle har haft mulighed for selv at vælge deres opgaver, og derfor har jeg valgt at sidde på forretningslogikken for både WPF-applikationen og ASP.NET-applikationen. Begge applikationer er blevet gennemført på en tilfredsstillende måde, men set i bakspejlet skulle WPF-applikationen nok være bygget op efter *MVVM*, da måden den er bygget op på lige nu har besværliggjort tests. Til trods for dette har udviklingsprocessen af denne stadigvæk været utroligt lærerig, og kvaliteten af ASP.NET-applikationen er helt klart blevet højnet på grund af dette.

I4 PRJ4, Gruppe 5

IKT

Til projektet er der blevet brugt GIT, for at lette udviklingsprocessen, hvilket generelt set har fungeret rigtig godt. Der er dog en gang imellem sket nogle fejl, der har gjort at noget data er gået tabt. Dog har disse fejl for det meste været relativt lette at fikse, og jeg mener helt klart at fordelene ved GIT har vejet mere end rigeligt op for de småproblemer, der har været.

Alt i alt har gruppen fungeret rigtig godt og alle har virket engageret. Der er blevet arbejdet seriøst, men der har også været plads til at have det sjovt. Når der engang imellem opstod nogle problemer, var alle i gruppen flittige til at hjælpe, så vi hurtigst muligt kunne komme videre i processen.

Rasmus Witt Jensen

For mig har dette semesterprojekt været det mest lærerige og succesfulde igennem mit studie. Det her ligeledes været det mest relevante for mine mål som softwareudvikler. Arbejdet med WPF, ASP.NET og JavaScript har været utrolig udfordrende og spændende.

Jeg har lært vigtigheden af at have opbygget sin kode efter en ordentlig arkitektur, og opfylde SOLID principperne. Uden dette har vi haft utroligt svært ved at teste programmet.

På trods af at jeg føler at der har været mangler i forhold til at opfylde disse principper, synes jeg at det endelige resultat er tilfredsstillende. Gruppen har som helhed formået at bære opgaven, og få lavet et godt produkt. Hvert medlem har ydet en god indsats, og leveret et godt stykke arbejde.

Projektstyringen, hvor der har været stor vægt på en agil tilgang, og med en del elementer fra Scrum, har vist sig at være yderst effektiv. Det er sjældent sket at folk har ventet på at andre blev færdige med noget, for at kunne komme videre med deres eget. Hvis dette har været tilfældet, er det der har stået i vejen blevet prioriteret højt, og derved hurtigt blevet løst.

Der har til tider været meget gøgl i gruppen, hvilket har kunnet hæmme arbejdet. Dette blev der dog hurtigt rettet op på, ved at indføre at Kristoffer og jeg selv skulle sørge for at folk holdt fokus under vores gruppearbejde. Dette har effektiviseret arbejdet, og jeg føler ikke et lige så stort tidspres i forhold til projektet her i slutningen, i forhold til hvad jeg har gjort i de forrige semesterprojekter.

Personligt føler jeg gruppen har fungeret godt, og har haft et godt sammenhold. Alle virker tilfredse, og har folk været utilfredse med noget, er det blevet sagt og rettet op på.

Fremtidigt arbejde

For at finde de første mål for fremtidigt arbejde, ses på MoSCoW-opdelingen i projektdokumentationens side 10, hvor de første ikke-implementerede use cases er at finde under *Should*:

- En påmindelse om manglende vare(r) på en af listerne.
- Muligheden for at tilføje flere skabe.
- Et log-in-system, så må kan være flere brugere om samme system, samt af sikkerhedsmæssige årsager.

Hernæst vil det være en idé at foretage en evaluering af eventuelt manglende udvidelser, såsom en mere dynamisk indkøbsliste, med mulighed for at krydse varer af, som brugeren har lagt i kurven, mens der handles.

Implementering af funktionaliteterne under *Could* vil gøre produktet mere gennemført, og vil derfor være næste skridt i udviklingen, hvis det antages at produktet fortsat kan udvikles uden lancering.

Antages der derimod en deadline for produktlancering, vil en højere prioritet være at se på de afgrænsninger, som er sat, i særdeleshed at *Fridge app* udelukkende er udviklet til at køre i opløsningen 1920x1080 pixels.

Til sidst bør der kigges på *Would/Won't*-delen. Disse dele mangler stadig at blive teknologiundersøgt tilstrækkeligt til at der kan foretages en vurdering om hvorvidt, det ville være rentabelt at implementere disse funktioner.

Konklusion

Projektet er endt ud med en WPF-applikation (*Fridge app*), en web-applikation (*Web app*) og tilhørende databaser. Den ene database kører lokalt, og den anden kører på en Azure-server sammen med *Web app*. De to databaser synkroniserer automatisk, eller ved at brugeren aktivt beder om det ved et tryk på brugerinterfacet på *Fridge app*. Der er implementeret flere funktionaliteter på *Fridge app* end på *Web app*, da det er her, gruppens største fokus har været. Arbejdet med *Fridge app* startede før introduktionen til MVP, MVC og MVVM. Konsekvensen af dette er manglende afkobling af XAML, *code behind* og *Business Logic Layer (BLL)*. Dette har gjort det svært at teste, og derfor er der heller ikke skrevet så mange *unit tests* og integrationstests som ønsket.

Databasetilgangen er i *Fridge app* implementeret med ADO.NET, hvilket har givet problemer i forhold til *mapping*, og at bevare data-integriteten i selve databasen.

I *Web app* har vi haft fokus på at få designet og struktureret programmet bedre end *Fridge app*. Her har vi brugt MVC, og haft fokus på at overholde dette designprincip. En af grundene til at bruge MVC har været at undgå nogle af de udfordringer vi stødte på under udviklingen af *Fridge app*, og dermed udvikle en velstruktureret web-applikation. Databasetilgangen i *Web app* er lavet med *Entity Framework (EF)*,

I4 PRJ4, Gruppe 5

IKT

og mange funktionaliteter og *mapping* sker derfor automatisk. Dette har gjort *Web app* væsentligt mere læsbar og vedligeholdelsesvenlig end *Fridge app*.

I projektet er der brugt *continuous integration* i form af et *Git*-repository. Brugen af *Git* har gjort det simpelt at arbejde i den samme *Visual Studio*-solution på samme tid, og rette de fejl, der kommer, når nye funktionaliteter implementeres. Der har dog været nogle problemer i form af *merging*, når der er blevet ændret i præcis de samme filer, imellem hvert push/pull. Ved disse merges er der nogle få gange gået data tabt, som skulle have været bevaret. På trods af dette opvejer fordelene klart de ulemper, der har måttet være uden brug af *Git*.

Git var ligeledes en nødvendighed, da automatisering af tests i form af *Jenkins* var et ønske, og sammenkoblingen af *Git* og *Jenkins* var blevet introduceret i I4SWT. Efter at have brugt meget tid på at få det sat op, måtte vi dog konstatere at det ikke var tiden værd. Uden adgang til selve *Jenkins*-serveren kunne der ikke laves de nødvendige rettelser for at få *Jenkins* til at compilere projektet. De første par fejl blev løst over et par uger i samarbejde med Troels Fedder Jensen (underviser i I4SWT samt I4SWD på ASE), men da der blev ved med at dukke nye problemer op, gik vi fra brugen af *Jenkins*.

Selve gruppearbejdet har været udført med en agil tankegang, og har primært været inspireret af *Scrum*, i form af stå-op-møder, retrospektmøder, sprints og et taskboard. Den agile udviklingsproces har gjort det muligt hele tiden at have et fungerende produkt, som kunne vises til vejlederen, og løbende at tage de vigtigste opgaver. Det har også muliggjort at alle i projektgruppen hele tiden har været klar over hvor langt projektet har været på et givet tidspunkt, og har været i stand til at søge hjælp med det samme, hvis et medlem har været forhindret i at arbejde videre.

Alt dette har resulteret i et godt slutresultat, med et projekt som opfylder de krav, der blev sat i starten af projektet. Det har dog ikke været muligt at implementere mange af de overvejede ekstra tilføjelser, grundet tidspres. Dog er der blevet stiftet bekendtskab med nye teknologier, som fx Sync Framework, hvilket der ikke er blevet undervist i.

Alt i alt er projektarbejdet gået godt, produktet er blevet udviklet på tilfredsstillende vis og accepttesten er gennemført uden fejl.

Referencer

Hambrick, P. (2013, december 5). *Seque Technologies - Custom Software Development, Professional Website Design, Information Technology*. Retrieved from .NET Web Forms vs. MVC: Which is Better?: <http://www.seguetech.com/blog/2013/12/05/dotnet-web-forms-vs-mvc-which-better>

Microsoft. (2010, februar 1). *Microsoft Azure*. Retrieved from Microsoft Azure: Skybaseret databasebehandlingsplatform: <http://azure.microsoft.com/da-dk/>

Sukesh, M. (2014, september 26). *WebForms vs. MVC*. Retrieved from CodeProject - For those who code: <http://www.codeproject.com/Articles/528117/WebForms-vs-MVC>

W3Schools. (n.d.). *ASP.NET MVC - HTML Helpers*. Retrieved from W3Schools Online Web Tutorials: http://www.w3schools.com/aspnet/mvc_htmlhelpers.asp

W3Schools. (n.d.). *ASP.NET Razor - C# and VB Code Syntax*. Retrieved from W3Schools Online Web Tutorials: http://www.w3schools.com/aspnet/razor_syntax.asp

Bilag

Bilag forefindes på CD-rom.

<i>Bilag 01</i>	<i>Projektbeskrivelse.pdf</i>	<i>(Dokument)</i>
<i>Bilag 02</i>	<i>Brugermanual.pdf</i>	<i>(Dokument)</i>
<i>Bilag 03</i>	<i>Lenovo_YOGA_2_Pro-13_Nordic_Unit.pdf</i>	<i>(Dokument)</i>
<i>Bilag 04</i>	<i>Deployment_Diagram.pdf</i>	<i>(Dokument)</i>
<i>Bilag 05</i>	<i>DAL_FridgeApp_Klassediagram.png</i>	<i>(Billede)</i>
<i>Bilag 06</i>	<i>DAL_FridgeApp_Sekvensdiagram.png</i>	<i>(Billede)</i>
<i>Bilag 07</i>	<i>DAL_WebApp_Klassediagram.png</i>	<i>(Billede)</i>
<i>Bilag 08</i>	<i>DAL_WebApp_Sekvensdiagram.png</i>	<i>(Billede)</i>
<i>Bilag 09</i>	<i>Brainstorms</i>	<i>(Dokumentsamling)</i>
<i>Bilag 10</i>	<i>Skitser</i>	<i>(Billedsamling)</i>
<i>Bilag 11</i>	<i>Kode_FridgeApp</i>	<i>(VS2013 solution)</i>
<i>Bilag 12</i>	<i>Kode_WebApp</i>	<i>(VS2013 solution)</i>
<i>Bilag 13</i>	<i>Setup_Lokal_Database</i>	<i>(VS2013 solution)</i>
<i>Bilag 14</i>	<i>Kodedokumentation_FridgeApp</i>	<i>(Doxygen HTML)</i>
<i>Bilag 15</i>	<i>Kodedokumentation_WebApp</i>	<i>(Doxygen HTML)</i>
<i>Bilag 16</i>	<i>Mødeindkaldelser</i>	<i>(Dokumentsamling)</i>
<i>Bilag 17</i>	<i>FridgeApp_Executable</i>	<i>(Program)</i>
<i>Bilag 18</i>	<i>Retrospektmødereferater</i>	<i>(Dokumentsamling)</i>
<i>Bilag 19</i>	<i>Task Board</i>	<i>(Dokumentsamling)</i>
<i>Bilag 20</i>	<i>Tidsplaner</i>	<i>(Dokumentsamling)</i>
<i>Bilag 21</i>	<i>Mødereferater</i>	<i>(Dokumentsamling)</i>
<i>Bilag 22</i>	<i>Turnusordning.pdf</i>	<i>(Dokument)</i>
<i>Bilag 23</i>	<i>Git_log.pdf</i>	<i>(Dokument)</i>
<i>Bilag 24</i>	<i>Gruppekonspekt.pdf</i>	<i>(Dokument)</i>
<i>Bilag 25</i>	<i>User_stories.pdf</i>	<i>(Dokument)</i>
<i>Bilag 26</i>	<i>Review.pdf</i>	<i>(Dokument)</i>
<i>Bilag 27</i>	<i>Teknologiundersøgelser</i>	<i>(Dokumentsamling)</i>