

I4 PRJ4

Forår 2015

SmartFridge

Projektdokumentation

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

Indholdsfortegnelse

Termliste	1
Fridge app	1
Web app.....	1
Kernefunktionalitet.....	1
Standard-varer	1
Opsætning.....	1
Kravspecifikation.....	2
Aktører	2
Aktørbeskrivelse	2
Bruger.....	2
Ekstern Database	2
Use cases.....	3
UC 1: Se varer.....	4
UC 2: Tilføj vare.....	5
UC 3: Rediger vare.....	6
UC 4: Fjern vare.....	7
UC 5: Synkroniser til ekstern database	8
UC 6: Notifikation om holdbarhedsdato	8
Udvidelser	9
Gammel vare fjernes.....	9
Tilføj opbevaringssted.....	9
Log in	9
Importer indkøbsliste.....	9
Scan vare	9
Vis ernæringsværdier.....	9
Find opskrift	9
Valg af tema	9
Juster temperaturalarm	9

I4 PRJ4, Gruppe 5

IKT

Find tilbud	9
MoSCow	10
Must	10
Should	10
Could	10
Would/Won't	10
Ikke-funktionelle krav	11
Systemarkitektur	12
Domænemodel	12
Fridge app	13
Web app.....	16
Design og implementering.....	18
Overordnet design	18
Database	19
Fridge app	19
Database – Webapp.....	28
Test.....	32
Fridge app	34
Design.....	34
Implementering	36
Test.....	53
Web app.....	54
Design.....	54
Design af views.....	55
Implementering	56
Test.....	70
Integrationstests	73
Coverage	75
Static analysis.....	75
Maintainability index	76

I4 PRJ4, Gruppe 5

IKT

Cyclomatic Complexity.....	76
Depth of Inheritance	76
Class Coupling	76
Accepttests.....	76
Test setup.....	76
Funktionelle krav.....	77
Ikke-funktionelle krav	83
Referencer.....	88
Bilag.....	89

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".

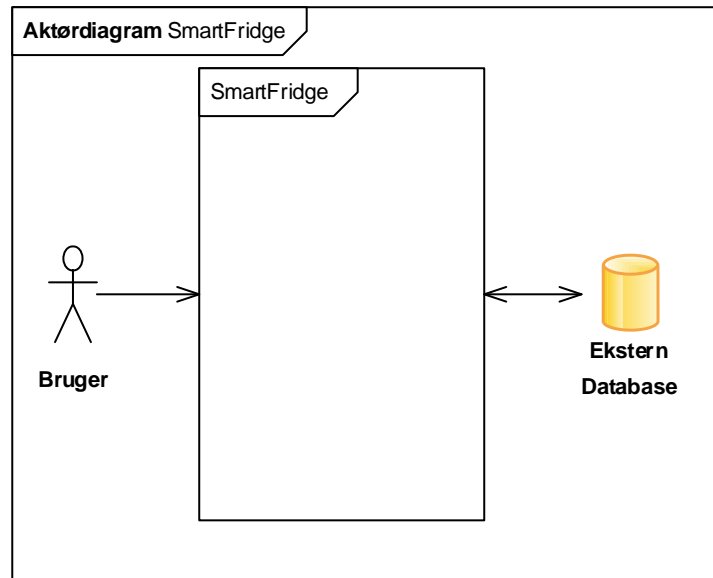
Opsætning

For at projektet *Fridge app* kan køre på PC, skal databasen først sættes op. For info om dette, henvises til brugermanualen i **bilag 02**.

Kravspecifikation

Aktører

Figur 1 viser use case-diagrammet med alle aktører og deres forhold til systemet SmartFridge.



Figur 1 Aktørdiagram over SmartFridge

Aktørbeskrivelse

Bruger

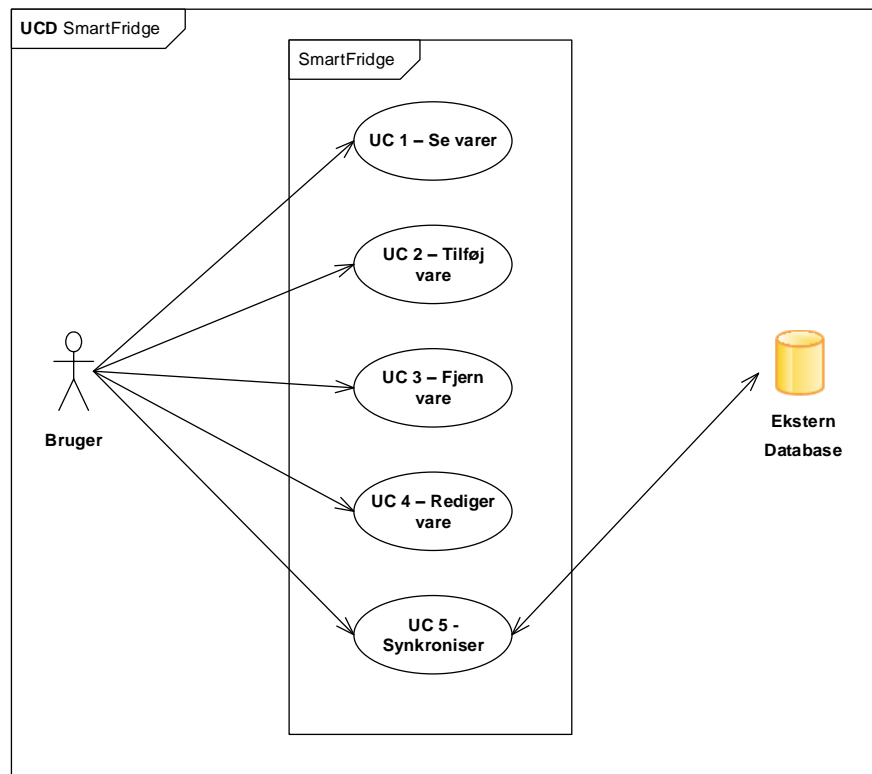
Aktørnavn:	Bruger
Alternativt navn:	
Type:	Primær
Beskrivelse:	Bruger er den primære bruger af systemet. Bruger interagerer med systemet gennem en brugergrænseflade.

Ekstern Database

Aktørnavn:	Ekstern Database
Alternativt navn:	Online Database
Type:	Sekundær
Beskrivelse:	Den Eksterne Database er en database, der ligger på en webserver. Den har kun forbindelse til det lokale system når der synkroniseres (se Use Case 7), og fungerer som direkte database for web-app'en. Databasen får tilføjet, fjernet og redigeret data ved synkronisering. Manipulationen kan også foregå fra systemets web app.

Use cases

For alle use cases gælder det, at i enhver undermenu, er der altid mulighed for at vende tilbage til hovedmenuen.



Figur 2 Use case-diagram over SmartFridge

I4 PRJ4, Gruppe 5

IKT

UC 1: Se varer

Use case 1 har til formål at lade brugeren få overblik over enten varebeholdningen, indkøbslisten eller listen over standard-varer.

Use case nr./navn	1 – Se varer
Mål	At få frembragt varerne på en liste.
Initialisering	Bruger
Aktører	- Brugere
Samtidige forekomster	1
Forudsætninger	Applikationen er startet
Resultat	Indholdet på den valgte liste vises på skærmen.
Hovedscenarie	<p>1. Brugere trykker på knappen "I køleskab". [Alternativt flow 1.a: Brugere trykker på "Indkøbsliste"] [Alternativt flow 1.b: Brugere trykker på "Standard-varer"]</p> <p>2. En liste over nuværende varer i køleskabet, samt mængden af disse, vises på skærmen.</p>
Alternativt flow	<p>1.a Brugere trykker på "Indkøbsliste". 1.a.1 En liste over nuværende varer på indkøbslisten, samt mængden af disse, vises på skærmen.</p> <p>1.b Brugere trykker på "Standard-varer". 1.b.1 En liste over nuværende standard-varer, samt mængden af disse, vises på skærmen.</p>

I4 PRJ4, Gruppe 5

IKT

UC 2: Tilføj vare

Use case 2 har til formål at lade brugeren tilføje varer til enten varebeholdningen, indkøbslisten eller listen over standard-varer.

Felterne "Antal" og "Volumen/vægt" udfyldes automatisk med standardværdier, når varetypen er valgt, men kan ændres efter behov.

Use case nr./navn	2 - Tilføj vare
Mål	At tilføje en vare til en liste.
Initialisering	Bruger
Aktører	- Bruger
Samtidige forekomster	1
Forudsætninger	UC1: Se varer
Resultat	Varen er tilføjet til en liste.
Hovedscenarie	<ol style="list-style-type: none"> 1. Bruger trykker på "Tilføj". 2. Varetype vælges. <p>[Undtagelse 2.a: Varetypen eksisterer ikke]</p> <ol style="list-style-type: none"> 3. Antal vælges. 4. Volumen/vægt vælges 5. Enhed vælges 6. Udløbsdato vælges eventuelt. 7. Bruger trykker på 'Tilføj og afslut, ' og varen tilføjes til listen. <p>[Alternativt flow 7.a: Bruger trykker på "Tilføj"]</p> <ol style="list-style-type: none"> 8. Der returneres til listen.
Alternativt flow	<p>7.a Bruger trykker på "Tilføj".</p> <p>7.a.1 Varen gemmes, og der fortsættes fra punkt 2.</p>
Undtagelser	<p>2.a Varetypen eksisterer ikke.</p> <p>2.a.1 Bruger indtaster den ønskede vare, og fortsætter fra punkt 3.</p>

I4 PRJ4, Gruppe 5

IKT

UC 3: Rediger vare

Use case 3 har til formål at lade brugeren redigere varer på enten varebeholdningen, indkøbslisten eller listen over standard-varer.

Use case nr./navn	3 - Rediger vare
Mål	At redigere en vare på en den nuværende liste.
Initialisering	Bruger
Aktører	- Brugere
Samtidige forekomster	1
Forudsætninger	UC1: Se varer
Resultat	Varen er blevet redigeret.
Hovedscenarie	<ol style="list-style-type: none"> 1. Brugere trykker på "Rediger". 2. Brugere retter vareinformation. [Alternativt flow 2.a: Brugere ændrer Varetype] [Alternativt flow 2.b: Brugere ændrer Antal] [Alternativt flow 2.c: Brugere ændrer Volumen/Vægt] [Alternativt flow 2.d: Brugere ændrer Enhed] [Alternativt flow 2.e: Brugere ændrer intet] 3. Brugere trykker på "Gem" og ændringerne gemmes i varen. [Alternativt flow 3.a: Brugere trykker på "Annuller"]

Alternativt flow	2.a	Bruger ændrer Varetype
	2.a.1	Bruger ændrer varens antal, og der returneres til punkt 2.
	2.b	Bruger ændrer Antal
	2.b.1	Bruger ændrer varens antal, og der returneres til punkt 2.
	2.c	Bruger ændrer Volumen/Vægt
	2.c.1	Bruger ændrer varens volumen/vægt, og der returneres til punkt 2.
	2.d	Bruger ændrer Enhed
	2.d.1	Bruger ændrer varens enhed, og der returneres til punkt 2.
	2.e	Bruger ændrer intet
	2.e.1	Bruger ændrer ikke noget, og der fortsættes fra punkt 3.
	3.a	Bruger trykker på "Annuller"
	3.a.1	Ændringer gemmes ikke.

UC 4: Fjern vare

Use case 4 har til formål at lade brugeren fjerne varer på enten varebeholdningen, indkøbslisten eller listen over standard-varer.

Use case nr./navn	4 – Fjern vare
Mål	At fjerne en vare fra en beholdning.
Initialisering	Bruger
Aktører	<ul style="list-style-type: none"> - Bruger - Database
Samtidige forekomster	1
Forudsætninger	UC1: Se varer
Resultat	Varen er fjernet fra beholdning
Hovedscenarie	1. Bruger trykker på "Fjern"-ikonet ud for en eksisterende vare, og varen fjernes fra GUI og database.

I4 PRJ4, Gruppe 5

IKT

UC 5: Synkroniser til ekstern database

Use case 5 har til formål at lade brugeren initiere en øjeblikkelig synkronisering mellem den lokale og den eksterne database.

Use case nr./navn	5 – Synkroniser til ekstern database
Mål	At synkronisere den lokale og den eksterne database.
Initialisering	Bruger
Aktører	<ul style="list-style-type: none"> - Bruger - Database
Samtidige forekomster	1
Forudsætninger	At Fridge app er forbundet til internettet.
Resultat	Den lokale og den eksterne database er synkroniseret.
Hovedscenarie	<ol style="list-style-type: none"> 1. Bruger trykker på "Synkroniser", og en øjeblikkelig synkronisering påbegyndes.

UC 6: Notifikation om holdbarhedsdato

Use case 6 har til formål at informere kunden om varer, som har overskredet sidste holdbarhedsdato, med en notifikation.

Use case nr./navn	6 – Notifikation om holdbarhedsdato
Mål	At notificere Bruger om at en vare har overskredet dens holdbarhedsdato.
Initialisering	Systeminitialiseret - en notifikation vises på GUI.
Aktører	<ul style="list-style-type: none"> - Bruger
Samtidige forekomster	1
Forudsætninger	UC1, UC3
Resultat	En notifikation vedrørende en given vares overskridelse af holdbarhedsdatoen har vist sig på brugerinterfacet.
Hovedscenarie	<ol style="list-style-type: none"> 1. Bruger trykker på notifikationslist-knappen og en pop-up vises med notifikationerne 2. Bruger trykker "Slet", og notifikationen slettes

Udvidelser

Følgende udvidelser vil i næste afsnit blive opdelt efter MoSCoW-metoden (DSDM CONSORTIUM, u.d.). De vil blive implementeret efter prioritet, efter at systemets kernefunktionaliteter er implementeret.

Gammel vare fjernes

Hvis holdbarhedsdatoen er overskredet ift. den dato som Bruger har angivet for en vare, notificeres Bruger. Notifikationen fjernes når Bruger indikerer til systemet at varen er fjernet, eller holdbarhedsdatoen på varen er ændret til en fremtidig dato.

Tilføj opbevaringssted

Bruger vælger tilføj opbevaringssted, og giver denne et navn. Bruger kan nu vælge dette sted som opbevaringsplads, når en ny vare tilføjes eller redigeres.

Log in

Når Bruger vil tilgå web-app'en, skal der først logges ind. Når bruger er logget ind, kan egen del af den eksterne database tilgås, og de sædvanlige funktioner vil være tilgængelige.

Importer indkøbsliste

Når Bruger har handlet ind, kan alle varer på indkøbslisten med ét tryk overføres til varebeholdningen.

Scan vare

En strekkodescanner tilsluttes systemet, og varer kan scannes. Varerne tilføjes til en valgfri varebeholdningsliste.

Vis ernæringsværdier

Bruger kan få oplyst ernæringsværdier for de enkelte varer.

Find opskrift

Bruger kan finde opskrifter baseret på de tilgængelige varer. Applikationen kan ud fra den valgte opskrift danne en indkøbsliste, med evt. manglende varer.

Valg af tema

Bruger får mulighed for at skifte grafisk tema på applikationen.

Juster temperaturalarm

Et termometer, som kan kommunikere med Fridge app'en, lægges i køleskabet, og Bruger sætter en max.- og en min.-temperatur. Kommer temperaturen uden for de satte værdier, advares Bruger.

Find tilbud

Bruger vælger "Find tilbud" på den færdige indkøbsliste. Applikationen finder nu selv de supermarkeder hvor der er tilbud på de varer, der er på indkøbslisten. De butikker hvor alle nødvendige varer sammenlagt kan købes billigst, kommer først.

I4 PRJ4, Gruppe 5

IKT

MoSCow

Must

Disse krav skal implementeres i det endelige produkt for at det er acceptabelt:

- Muligheden for at tilføje en vare.
- Muligheden for at fjerne en vare.
- Muligheden for at redigere en vares information.
- Muligheden for at se en liste over varer.
- Mulighed for at synkronisere med en ekstern database.
- Varer der ikke findes i "Køleskab", og som er tilføjet på "Standard-listen", tilføjes automatisk til indkøbslisten.

Should

Disse krav har, ligesom i must-sektionen, høj prioritet. Men kravene er ikke essentielle for at systemet fungerer og kan benyttes.

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

Could

Disse funktioner kunne være en del af systemet. Det er alle krav, hvor implementeringen er ret tidskrævende, og de er heller ikke essentielle for at systemet virker. De hører derfor ind under "Nice-To-Have"-kategorien.

- Mulighed for at se ernæringsværdier for fødevarerne på listerne.
- Mulighed for at se opskrifter, baseret på de varer, der befinder sig på køleskabs-listen.
- En pænere og mere interaktivt grafisk brugergrænseflade; f.eks. at køleskabslisten ses som "hylde" med drag-n-drop items.
- Indhentning af tilbud på de varer, der findes på indkøbslisten.

Would/Won't

Disse funktioner bliver ikke tilføjet til systemet pga. tid, penge og relevans. Det ville tage lang tid programmere drivere til enhederne, og koste penge at købe selve enhederne.

- En scanner, så der er mulighed for at scanne strekkoden på de nyindkøbte vare, og dermed tilføje dem listerne.
- En temperatursensor, der evt. gør brug af Bluetooth, så temperaturen i køleskabet kan overvåges.

Ikke-funktionelle krav

Alle krav er specificeret ud fra Lenovo Yoga 2 Pro (**bilag 03**) som platform.

1. System

- 1.1. Kernefunktionaliteterne skal kunne udføres i både *Web app* og *Fridge app*, med undtagelse af UC5 og UC6, som kun skal kunne udføres i *Fridge app*.

2. Databaser

- 2.1. Den lokale og den eksterne database skal automatisk synkroniseres hvert 10. minut.
- 2.2. I tilfælde af konflikter ved synkronisering, overskriver de nyest tilføjede data de ældste.

3. Fridge app

- 3.1. Ved opstart og nedluk, forsøges synkronisering mellem den lokale og den eksterne database.
- 3.2. Ændringer af data lagres straks i den lokale database.
- 3.3. En knap/et ikon på skærmen skal indikere status for synkronisering.
 - 3.3.1. Synkroniseret
 - 3.3.2. Ikke synkroniseret
- 3.4. Responstiden for navigation må maksimalt være to sekunder.
- 3.5. Skal kunne anvendes uden internetforbindelse.
- 3.6. Varer på standard-beholdning tilføjes automatisk til indkøbslisten ved mangel i køleskabet.

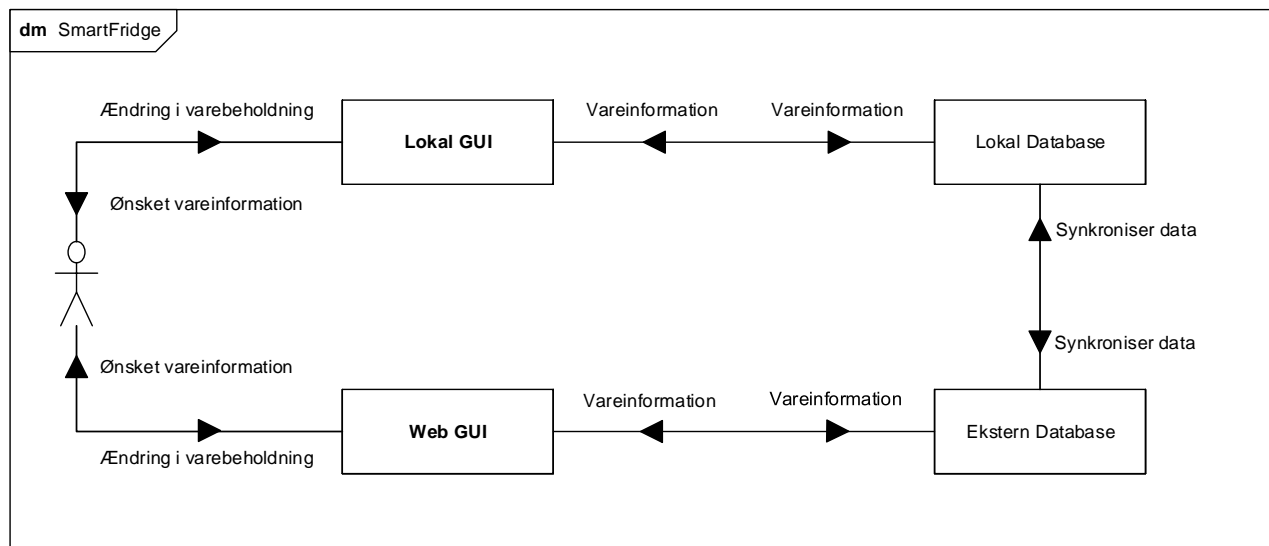
4. Web app

- 4.1. Ændringer af data lagres straks i den eksterne database.

Systemarkitektur

Domænemodel

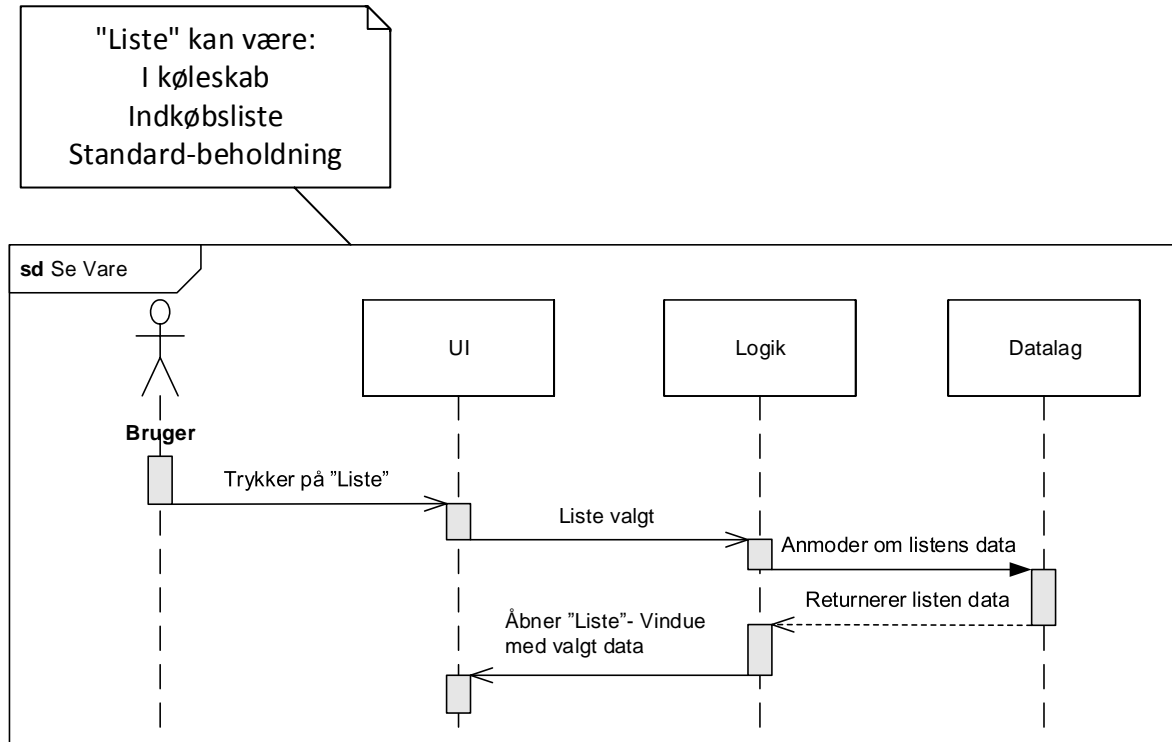
Domænemodellen (**Figur 3**) beskriver den overordnede kommunikation i systemet. Brugeren interagerer enten med den lokale GUI eller Web GUI'en. Den lokale GUI får sin information om varebeholdningen fra den lokale database, og har mulighed for at tilføje, fjerne og ændre i data'en. Web GUI'en får sin 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 3 Domænemodel af SmartFridge

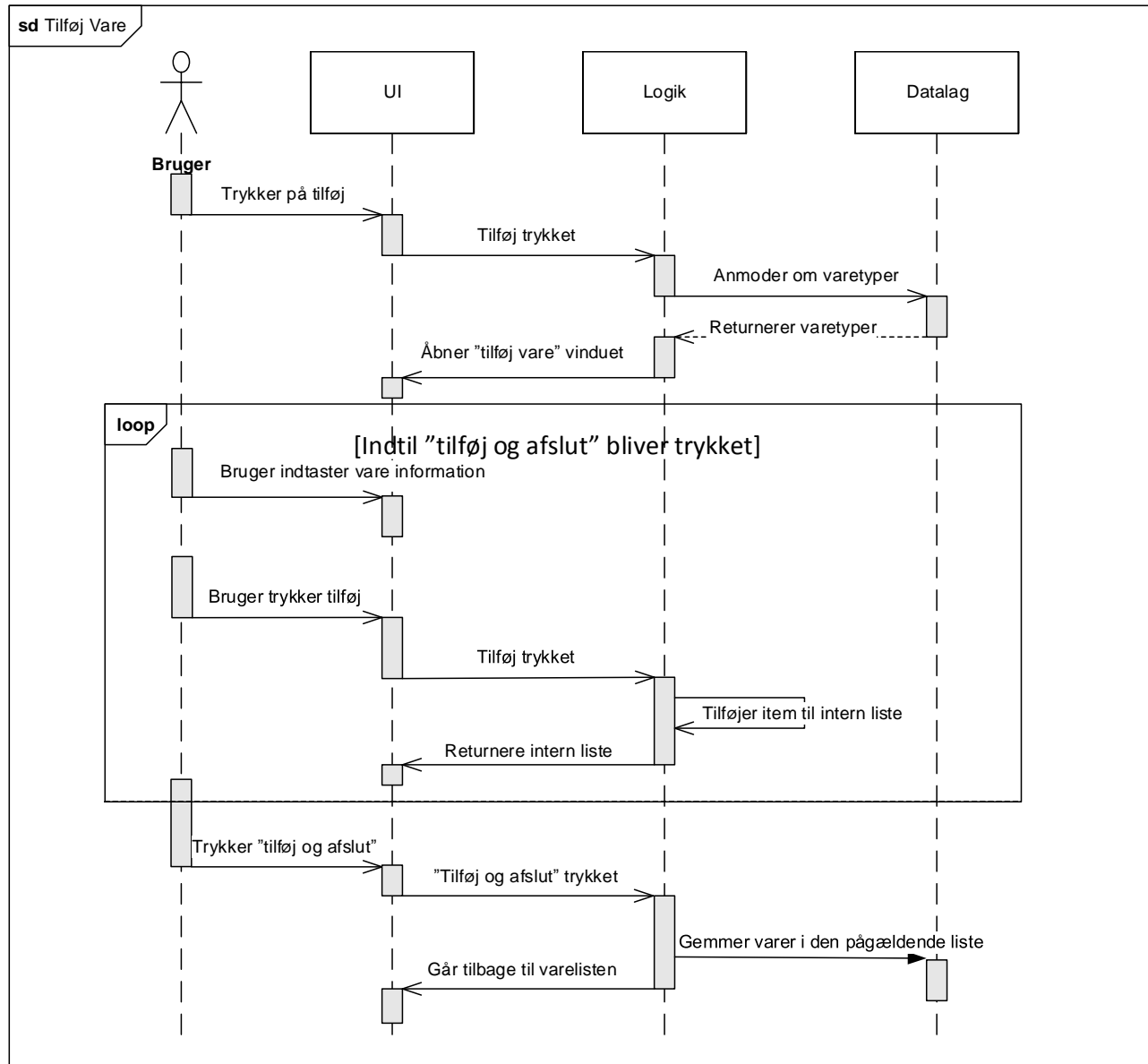
Fridge app

Følgende sekvensdiagrammer beskriver hver forløbet af en use case.



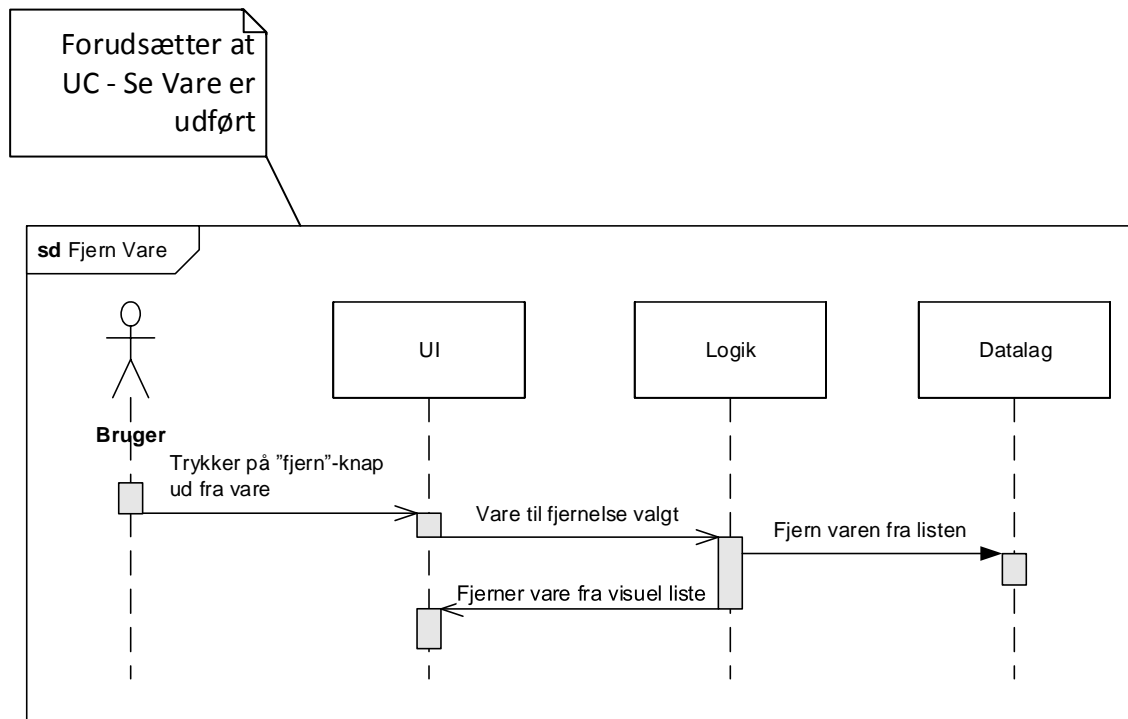
Figur 4 Sekvensdiagram for UC Se Vare

Sekvensdiagrammet på **Figur 4** beskriver hvordan de forskellige lag i applikationen kommunikerer med hinanden. Brugeren interagerer med UI, hvorefter Logik laget sørger for de rigtige data bliver vidst på UI'et. Logik-laget henter dens information fra datalaget. Se vare use casen dækker over flere lister. Det varierer hvilke data der bliver hentet til logik laget afhængigt af hvilke liste der vælges af brugeren.



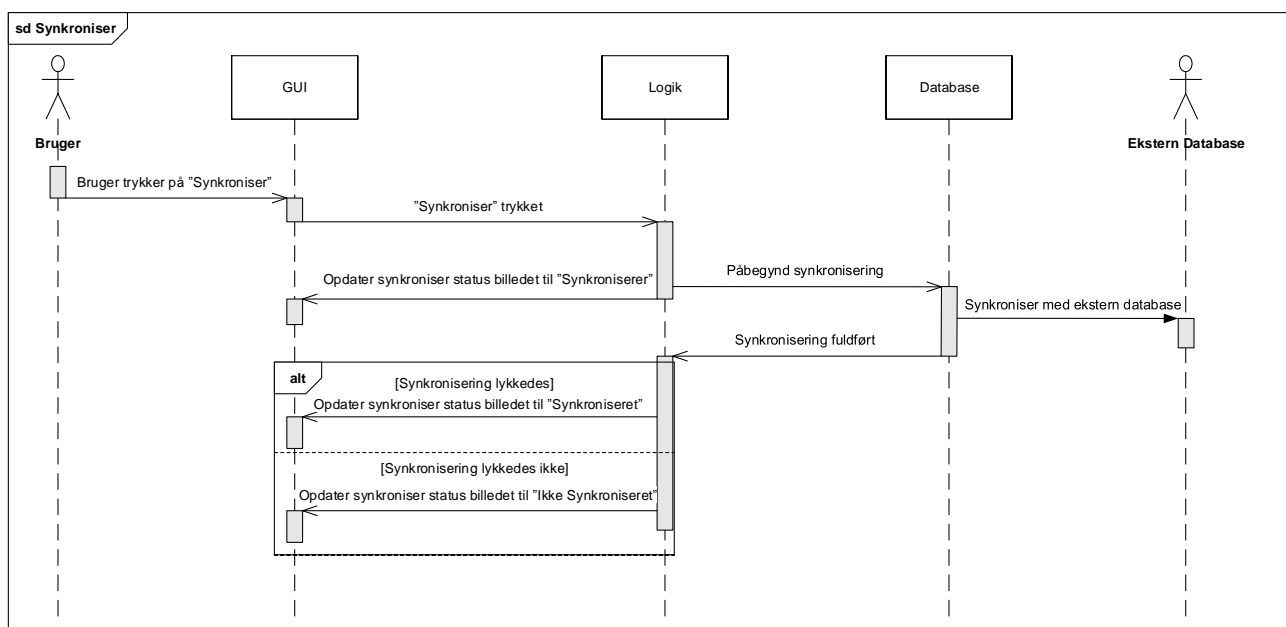
Figur 5 Sekvensdiagram for UC Tilføj Vare

Tilføj vares loop påbegyndes når Bruger indtaster vareinformationerne. Bruger kan tilføje lige så mange vare han/hun ønsker indtil der vælges "Tilføj og Afslut". De varer der tilføjes, før der trykkes på "Tilføj og Afslut", skal kunne ses i en midlertidig liste.



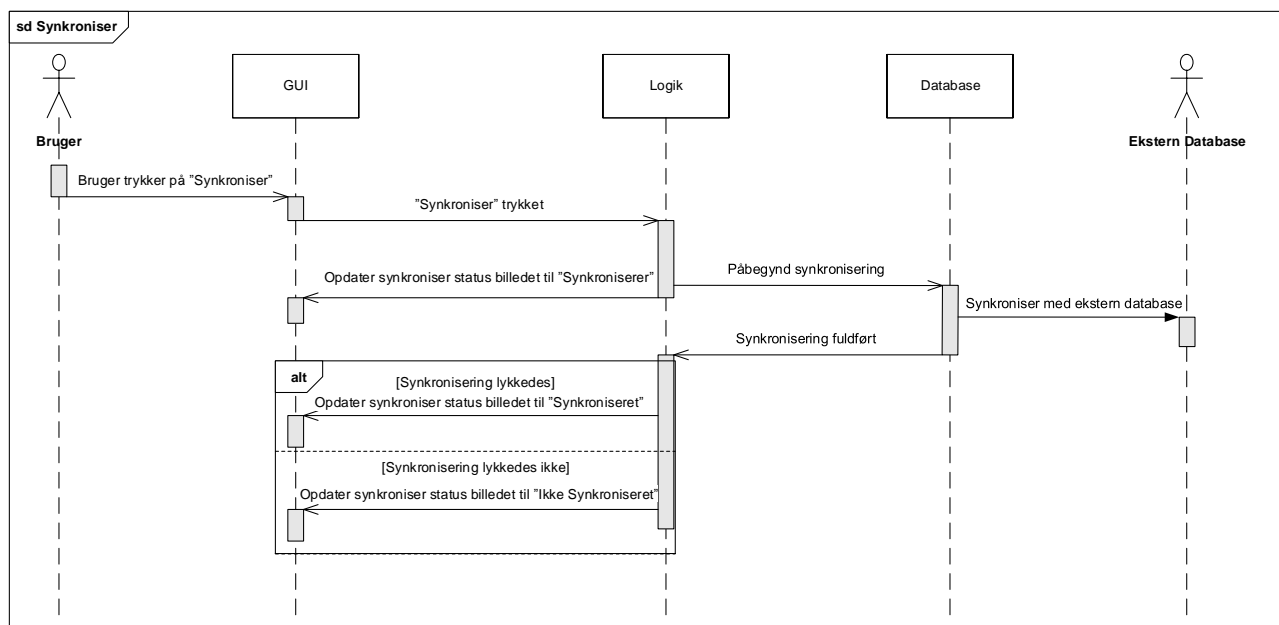
Figur 6 Sekvensdiagram for UC Fjern Vare

Figur 6 forudsætter at der allerede er tilføjet et item til den liste hvor item skal fjernes fra.



Figur 7 Sekvensdiagram for UC Synkroniser

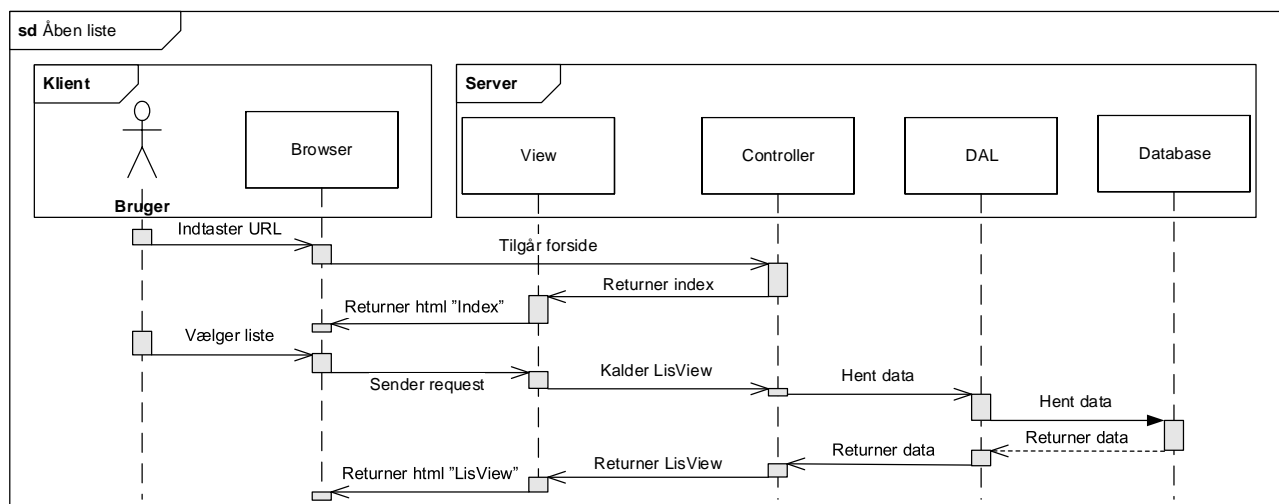
Synkroniseringen mellem den lokale- og den eksterne database, bliver i denne use case påbegyndt af brugeren, men sker også automatisk vha. et framework hvert 10. minut. Lykkedes den påbegyndte synkronisering ikke vil dette bliver indikeret i GUI'en.



Figur 8 Sekvensdiagram for UC Rediger Vare

Web app

Sekvens diagrammerne for web applikationen er opdelt i to. En klient-side hvor applikationen kan tilgås fra en enhed med internet forbindelse og server-side hvor applikationen kører på og returnere html kode til klient-sidens browser. Forbindelsen imellem klienten og serverne sker vha. http-protokollen. Databasen der refereres til i følgende sekvensdiagrammer symbolisere den eksterne database.

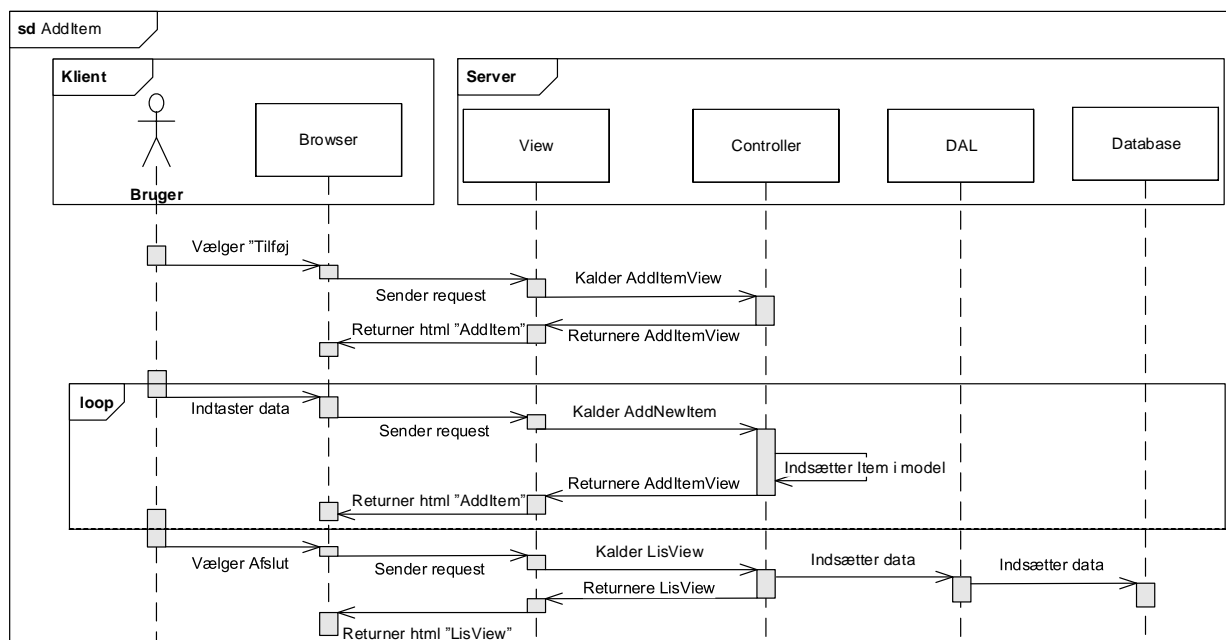


Figur 9 Sekvensdiagram for hvordan en liste åbnes

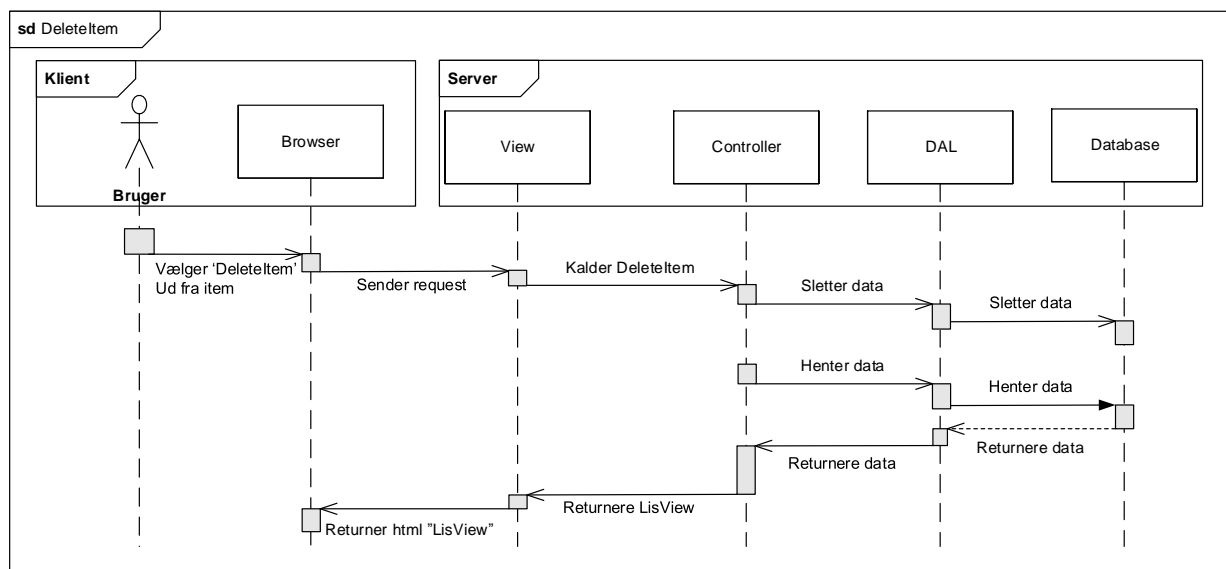
I4 PRJ4, Gruppe 5

IKT

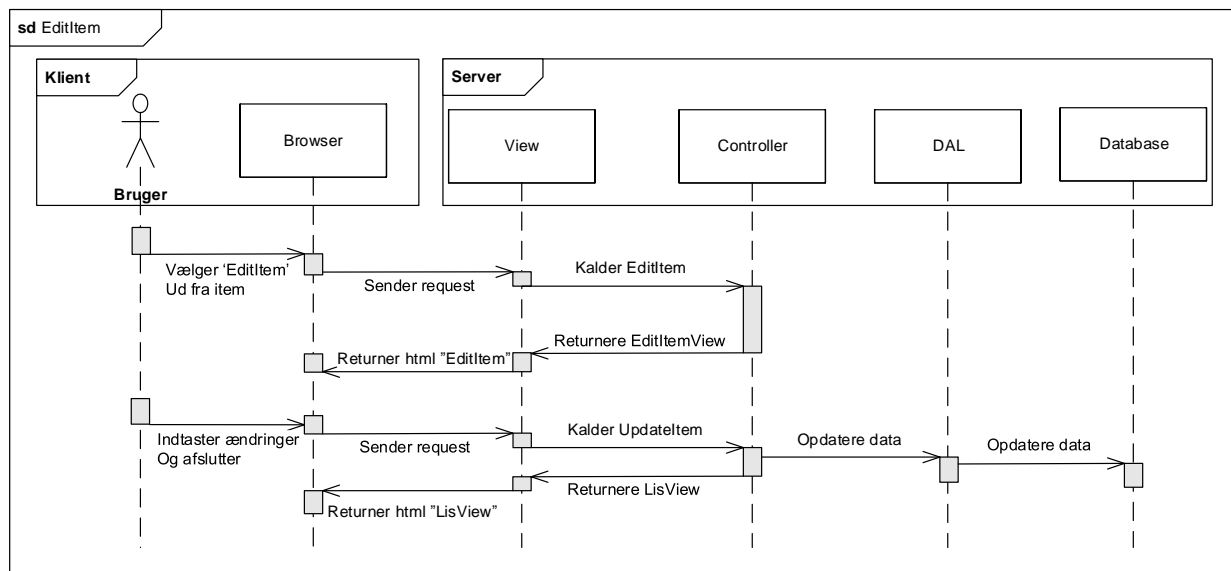
Sekvensdiagrammet 'Åben liste' skal være udført før sekvensdiagrammerne for AddItem, DeletItem og EditItem på henholdsvis **Figur 10**, **Figur 11** og **Figur 12**, kan udføres. Diagrammerne beskriver hvordan der kommunikeres imellem lagene.



Figur 10 Sekvensdiagram for UC Add Item



Figur 11 Sekvensdiagram for UC Fjern Item

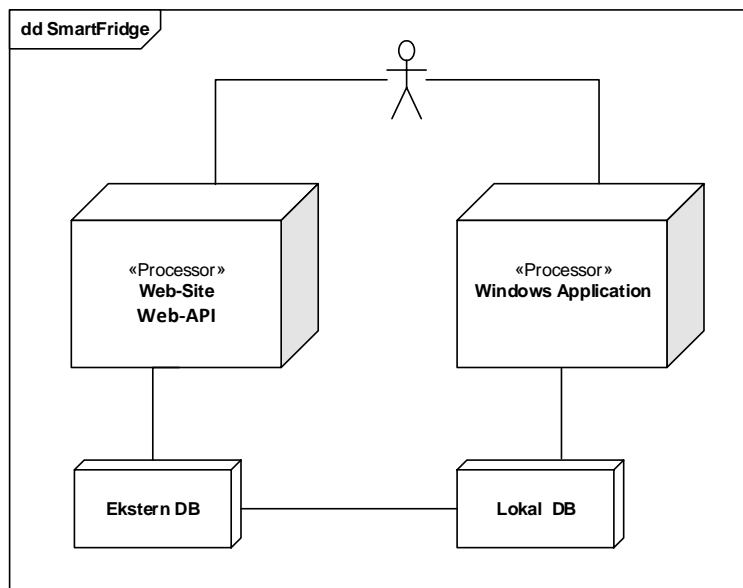


Figur 12 Sekvensdiagram for UC Edit Item

Design og implementering

Overoverdnet design

Deployment-diagrammet i **Figur 13** viser, hvordan de interne dele af Smart Fridge 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 13 Deployment Diagram for hele SmartFridge-systemet

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

Database

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

Fridge app

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

Design

I dette afsnit vil designprocessen blive beskrevet, samt hvilke designovervejelser der er gjort i forhold til *DAL* for *Fridge app*.

Teknologi

Før det var muligt at designe databasen, skulle der først vælges en teknologi til *DAL*, hvor der ville blive brugt en relationel database. På dette tidspunkt, var det oplagte valg *ADO.NET*, da der var blevet undervist i Database-kurset omkring dette. Den anden mulighed var *Entity Framework (EF)*. *EF* ville være klart lettere at arbejde med, men for læringens skyld blev det valgt at arbejde med *ADO.NET*, og der ville igen blive kigget på *EF* i forhold til *Web app*.

Objektmodel

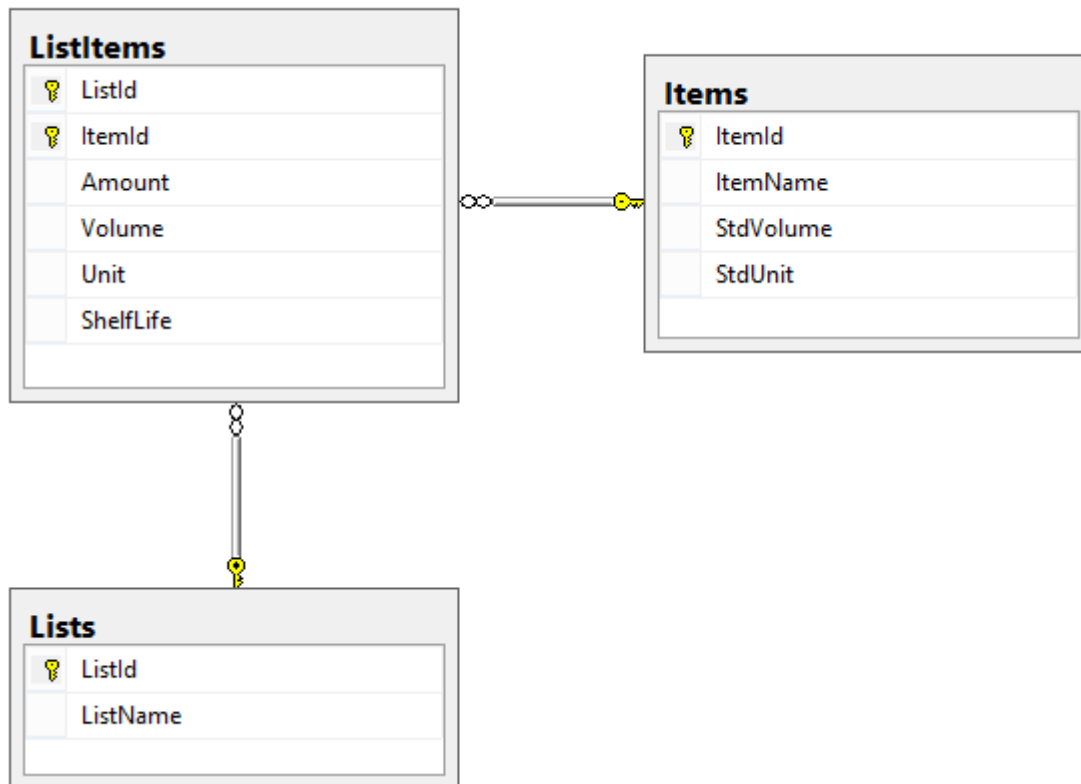
Der skulle findes en objektmodel til systemet, hvorpå data kunne gemmes i den relationelle database. Til dette blev *DDS-Lite* anvendt. Det var ønsket at kunne opretholde indtil flere lister, hvorpå disse lister indeholdt varer. Derudover var det ønsket at man kunne gemme varetyper som f.eks. mælk og æg i systemet, for at brugeren ikke kunne indskrive disse, hver gang der blev tilføjet en vare til en liste. På dette grundlag, blev der lavet en objektmodel, som ses i **Figur 14**.



Figur 14 Objektmodel

Her er der udnyttet et *Mange-til-Mange*-forhold mellem *List* (liste) og *Item* (varetype), hvori *ListItem* er selve den vare, der er på en liste. I **Figur 15**, ses en illustration af objektmodellen med attributter. Her ses det at man har lagret en varetype i tabellen *Items*, hvori man har standardværdier for mængde og unit. *ListItems*-tabellen indeholder attributterne for selve varen, og man ser at der er en *composite key*. Dette er for at kunne have flere af samme type varer, med forskellige holdbarhedsdatoer og i forskellige mængder/størrelser.

Da objektmodellen er fastlagt, er det nu muligt at overveje nogle designmønstre for *DAL*.

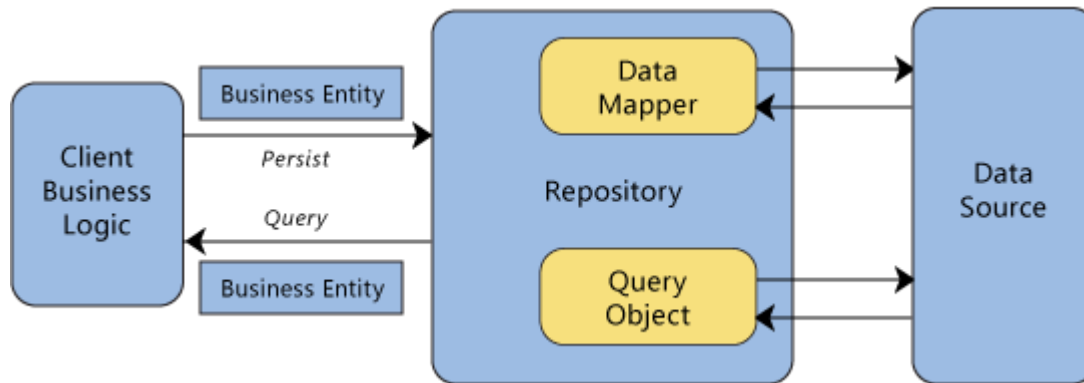


Figur 15 Objektmodel med attributter

Repository Pattern

I mange applikationer er det forretningslogikken, der tilgår data og databaser. Dog kan direkte tilgang introducere indtil flere problemer, som gentagen kode, højere risiko for programmeringsfejl, dårlig abstraktion og lav testbarhed.

For at kunne løse disse problemer, introduceres et *Repository pattern* til implementeringen. *Repository*-mønsteret separerer logikken mellem forretningslogikken og datakilden, som en database. En illustration af dette ses i **Figur 16**. *Repository*'et sørger derfor at tilgå datakilden for data og mapper data til forretningslogikken. Ved at logikken separeres, opnås der at alt data tilgang foregår igennem Repositoryet, der opnås en højere abstraktion. Herved har man opnået højere testbarhed af forretningslogikken (Microsoft).



Figur 16 Repository Pattern (Microsoft)

Repository-mønsteret tilbyder et sted, hvor man har sin datatilgang, hvor der kan gøres brug af ADO.NET data providers, hvilket gør oplagt at anvende sammen med ADO.NET. Det vil også skabe højere abstraktion, samt at gøre DAL langt mere testbart.

Anvendelse af mønstret vil blive beskrevet i implementeringsafsnittet for databasen.

Unit of Work

Unit Of Work er et design mønster, hvorpå man holder styr på hvilke transaktioner forretningslogikken laver til databasen. I et Unit Of Work, gøres alle de transaktioner som forretningslogikken ønsker at gøre, og derefter vil Unit of Work holde styr på hvad der skal ske på database-siden, og commit og rollback de ændringer, som forretningslogikken har gjort. På den måde opnås der noget logik, kan opretholde en liste af ændringer der skal ske og give det videre til databasen (Fowler).

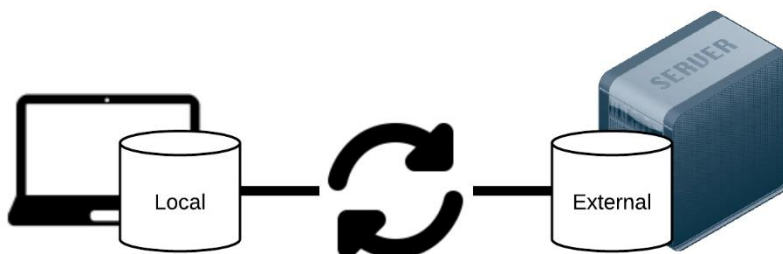
Unit of Work er oplagt at anvende sammen med Repository mønstret, hvorpå det er muligt at have et Unit of Work, hvor Repositoryet anvendes til database-transaktioner, hvorefter transaktionerne kan blive commitet.

Anvendelse af mønstret vil blive beskrevet i implementeringsafsnittet for databasen.

Synkronisering

For at muliggøre at der både er Fridge app'en og Web app'en i systemet, og sørge for at Fridge app er funktionel uden internetforbindelse, var det blevet en nødvendighed at implementere synkronisering af en lokal og en ekstern database.

Til dette formål, er Microsoft Sync Framework blevet valgt, da det er en løsning der er mulig selv at implementere, uafhængigt af hvilken data provider der anvendes. En illustration af synkronisering mellem lokal database og ekstern database på en webserver ses på **Figur 17**.



Figur 17 Illustration af database synkronisering

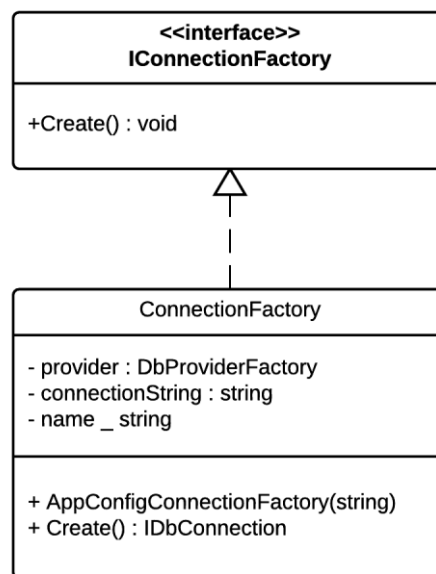
Endeligt design

Efter de designovervejelser der er blevet gjort i de forrige afsnit, er der blevet udarbejdet følgende klassediagram. Klassediagrammet kan ses i **bilag 05**.

Klassediagrammet er udarbejdet over en iterativ proces, derved har det ikke været det endelige klassediagram fra start. I de følgende afsnit, vil de væsentlige dele af klassediagrammet blive beskrevet i ansvarsområder.

ConnectionFactory

IConnectionFactory er et interface, som ses på **Figur 18**, der sørger for at skabe forbindelse til databasen med Create(). ConnectionFactory er en implementering af interfacet, hvori man giver den et navn til en connectionstring i app.config.



Figur 18 ConnectionFactory

AdoNetContext

IContext er et interface, som ses på **Figur 19**, svarer til DbContext i Entity Framework. Heri bliver forbindelsen til databasen opretholdt og man kan execute commands, hvor der kan anvendes et Repository. Her kan der også oprettes i Unit of Work, hvor man kan udføre sine databasetransaktioner.

AdoNetUnitOfWork

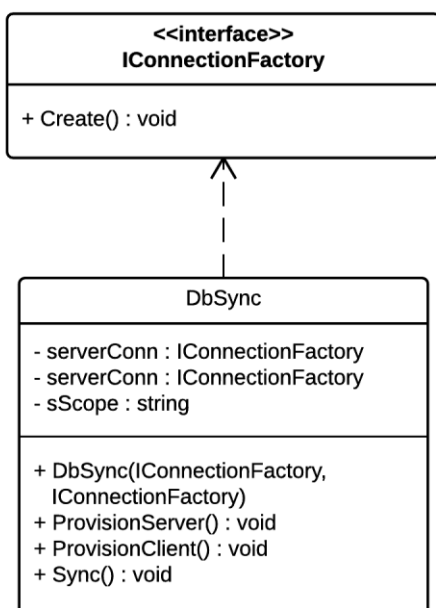
IUnitOfWork er et interface, som ses på **Figur 20**, der er en implementering af Unit of Work mønstret. Her er SaveChanges() dens commit funktionalitet. Herudover har AdoNetUnitOfWork propertyen Transaction, hvori der opretholdes en transaktionshistorik.

Repository

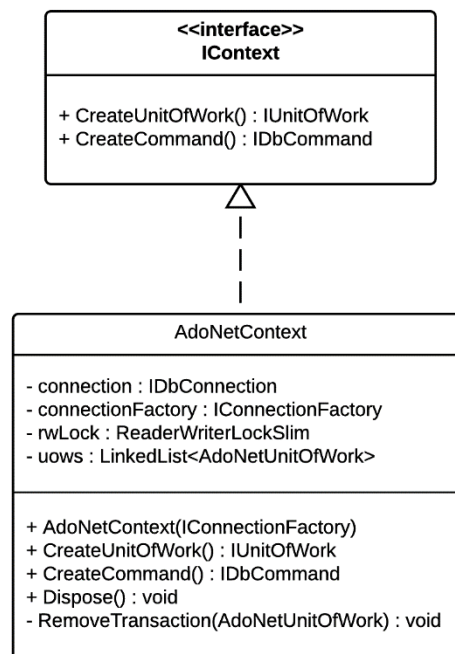
Repository kan ses på klassesdiagrammet i **bilag 05**. Dette er en implementering af Repository mønstret, hvor der anvendes arv. På den måde, er der et sted, hvor databasen tilgås.

DbSync

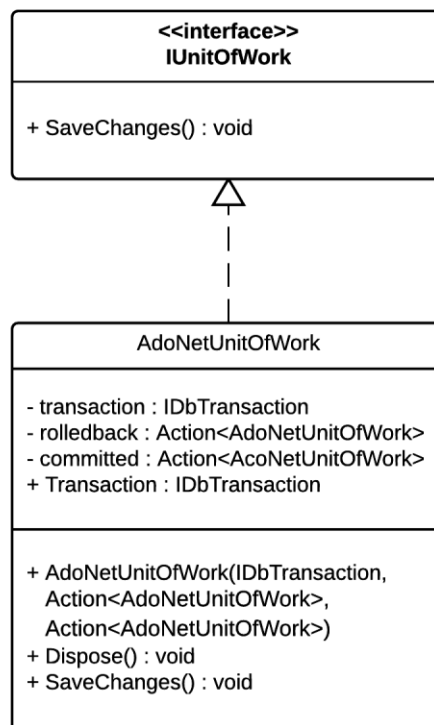
DbSync, som ses i **Figur 21**, viser logikken for synkronisering af databaserne, hvorpå den anvender IConnectionFactory, til at åbne en forbindelse til databaserne. Et sekvensdiagram for synkroniseringsfunktionaliteten kan ses på **Figur 22**.



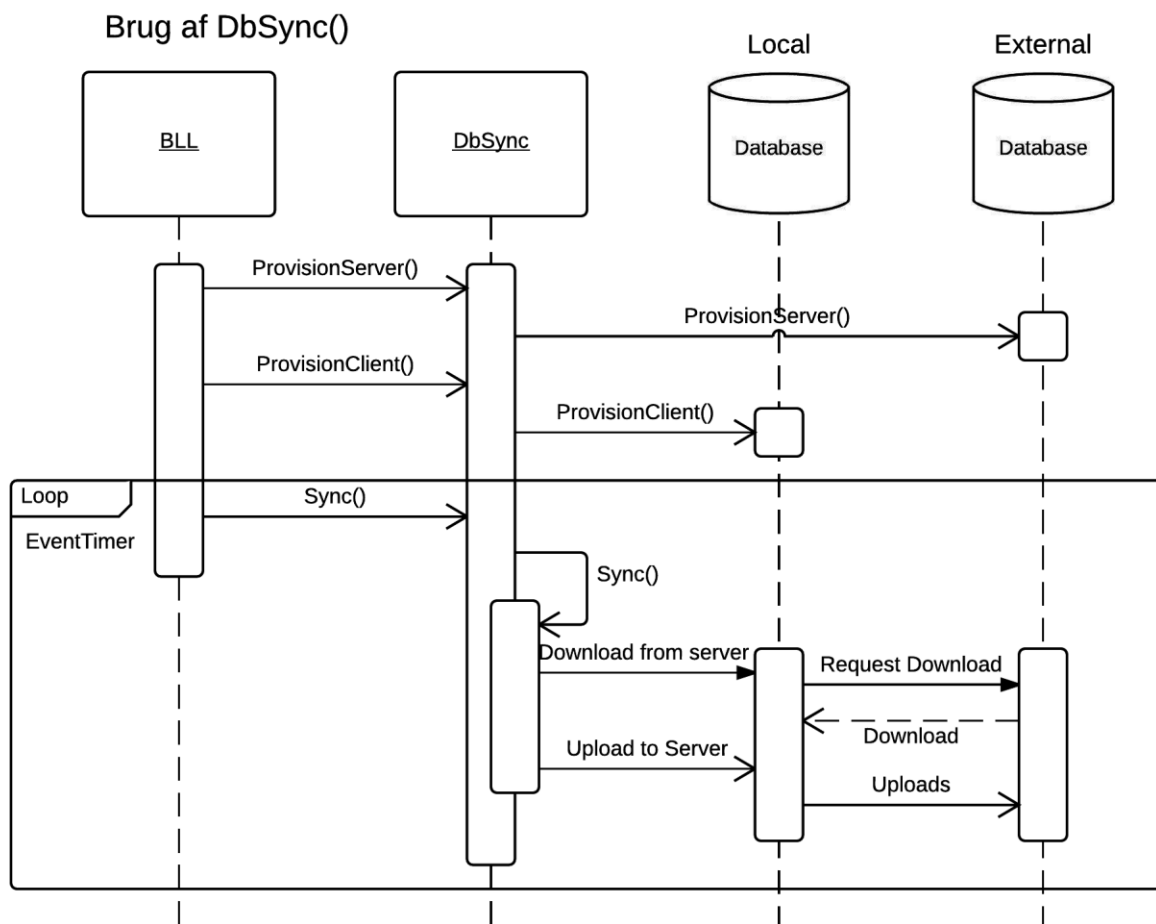
Figur 21 DbSync



Figur 19 AdoNetContext



Figur 20 AdoNetUnitOfWork

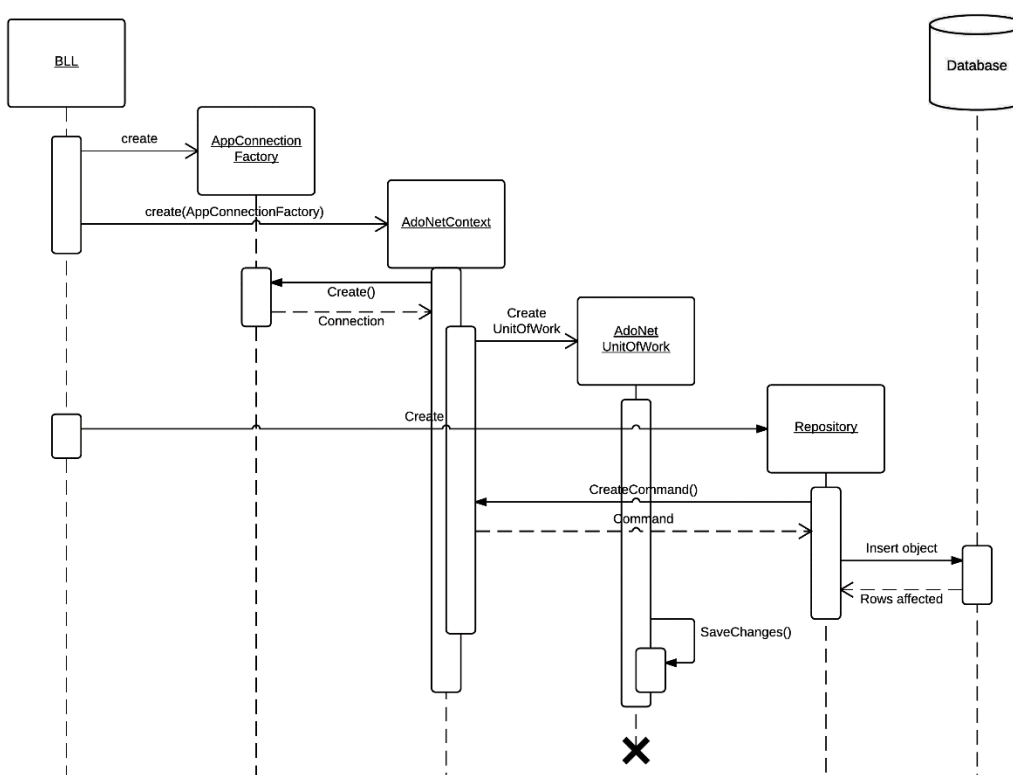


Figur 22 Sekvensdiagram for synkronisering

Anvendelse af DAL

Et sekvensdiagram for anvendelse af DAL, kan ses på **Figur 23**. Her illustreres hvordan BLL opretter de forskellige klasser, hvor Unit of Work og Repository mønstret anvendes til databasetransaktioner. Først skal der oprettes en *ConnectionFactory*, som beskrevet opretter en forbindelse fra en connectionstring i app.config. Herefter oprettes *AdoNetContext*, hvor databaseforbindelsen bliver injected. Derefter bliver der oprettet et *AdoNetUnitOfWork* af *AdoNetContext*, hvori der kan eksekveres databasetransaktioner. Her kan man oprette et *Repository*-objekt til den ønskede entity, man laver sine ønskede databasetransaktioner og herefter committes disse transaktioner til databasen. Herefter nedlægges *AdoNetUnitOfWork*.

Brug af DAL (insert)



Figur 23 Sekvensdiagram for brug af DAL

Implementering

I dette afsnit vil implementeringen af DAL blive beskrevet, samt væsentlig funktionalitet.

Alt kode er dokumenteret vha. XML comments og doxygen, som kan ses i **bilag 14**.

I de følgende afsnit vil væsentlig funktionalitet blive beskrevet, med eksempler.

CreateUnitOfWork

```
public IUnitOfWork CreateUnitOfWork()
{
    var transaction = _connection.BeginTransaction();
    var uow = new AdoNetUnitOfWork(transaction, RemoveTransaction, RemoveTransaction);

    _rwLock.EnterWriteLock();
    _uows.AddLast(uow);
    _rwLock.ExitWriteLock();
    return uow;
}
```

Kodestump 1 Kodeudklip fra AdoNetContext.cs

På **Kodestump 1**, ses et udsnit kode, hvorpå et UnitOfWork til databasetransaktioner bliver oprettet i AdoNetContext.cs. Her oprettes transaktionen, mens de gamle commits og rollbacks bliver fjernet med en Action delegate.

Insert command

Da der anvendes Sync Framework til synkronisering af databaser, kommer der forskellige triggers på SQL commands som insert, hvilket betyder at man ikke kan output inserted values, som id. Derved har det været nødvendigt med en insert command, som ses på **Kodestump 4**. Her mappes ItemId over i et midlertidigt table, hvorpå det kan outputtes fra. Denne fremgangsmåde bliver også brugt for insert af List.

ToList

Der udnyttes arv til implementering af ToList, hvor den modtager en command og eksekverer og den og opretter objekter ud fra hvad den modtager.

```
DECLARE @InsertItem TABLE ([ItemId] INT);
INSERT INTO [Items] ([ItemName],[StdVolume],[StdUnit])
OUTPUT [inserted].ItemId INTO @InsertItem
VALUES(@ItemName,@StdVolume,@StdUnit);
SELECT * FROM @InsertItem";
```

Kodestump 3 Insert command fra ItemRepository.cs

```
protected IEnumerable<T> ToList(IDbCommand command)
{
    using (var reader = command.ExecuteReader())
    {
        var items = new List<T>();
        while (reader.Read())
        {
            var item = new T();
            Map(reader, item);
            items.Add(item);
        }
        return items;
    }
}
```

Kodestump 2 ToList fra Repository.cs

Anvendes i de fleste Read-metoder i de forskellige repositories.

Test

I dette afsnit vil testning af DAL blive beskrevet, hvor coverage og statisk analyse vil blive dokumenteret.

På **Kodestump 5**, ses et screenshot af testsuite for Fridge App'ens DAL. Her ses at alle tests var en succes, hvorpå funktionaliteten fungerer som forventet.




Her ses også at Repository.cs, samt dens nedarvede klasser ikke er testet, hvilket er da 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. Udover det, er Sync heller ikke testet, da det er meget høj kobling til vores tabeller og man kan ikke mocke funktionaliteten ud, derved er dette heller ikke testet.

▲ ✓ {} SmartFridge.Tests.Unit (15 tests)	Success
▲ ✓ AdoNetContextIntegrationTest (5 tests)	Success
✓ CreateCommand_Called_ConnectionReceivesCall	Success
✓ CreateCommand_Called_ReturnsSqlCommand	Success
✓ CreateUnitOfWork_Called_ReturnsUnitOfWork	Success
✓ Dispose_Called_CalledFakeConnDispose	Success
✓ Dispose_CreateUnitOfWorkUsingScope_CallsDisposeWhenOutOfScope	Success
▲ ✓ AdoNetUnitOfWorkIntegrationTest (5 tests)	Success
✓ Dispose_CalledTwiceAndTransactionIsNull_DoesNotRollBackTwice	Success
✓ Dispose_Called_DoesDispose	Success
✓ Dispose_Called_DoesRollback	Success
✓ SaveChanges_CalledTwice_ThrowsInvalidOperationException	Success
✓ SaveChanges_Called_DoesCommit	Success
▲ ✓ AppConnectionFactoryIntegrationTest (5 tests)	Success
✓ Create_ConnectionNameIsSmartFridgeConn_ConnectionStringIsCorrect	Success
✓ Create_ConnectionNameIsSmartFridgeConn_ReturnsConnection	Success
✓ Create_ConnectionNameIsTest_ThrowsArgumentException	Success
✓ Ctor_ConnectionNameIsNull_ThrowsArgumentNullException	Success
✓ Ctor_ConnectionNameIsTest_Throws_ConfigurationErrorsException	Success

Figur 24 Screenshot af testsuite for DAL

Coverage

På **Figur 24** ses et screenshot af coverage resultatet for DAL-implementering. Her ses det at der er opnået 100% coverage, hvilket betyder at alt funktionalitet er testet.

▲  DataAccessLayer	100%	0/69
▶  DataAccessLayer.Connection	100%	0/17
▶  DataAccessLayer.AdonetUoW	100%	0/52

Figur 25 Screenshot af coverage for DAL

Statisk analyse

På **Figur 26**, ses et screenshot af code metrics for DAL. Her kan man se maintainability, hvor 20-100 er høj maintainability, hvilket viser at DAL-implementering kan vedligeholdes. Dog har DAL høj kompleksitet ved der kommer sig af høje koblinger i forhold til entities og repositories.

Hierarchy ▲	Maintainability Index	Cyclomatic Complexity
▲ C# DataAccessLayer (Debug)	79	84
▸ {} DataAccessLayer.AdoNet	88	16
▸ {} DataAccessLayer.Connec	83	5
▸ {} DataAccessLayer.Reposit	70	57
▸ {} DataAccessLayer.Sync	65	6

Figur 26 Screenshot af code metrics for DAL

Database – Webapp

I dette afsnit vil designprocessen, implementering samt test af database-delen for webapp'en blive beskrevet, samt de overvejelser der er blevet gjort for database-tilgang fra applikationen.

Design

I dette afsnit vil designprocessen af DAL for webapp'en blive beskrevet. Da objektmodel, 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 af Fridge app.

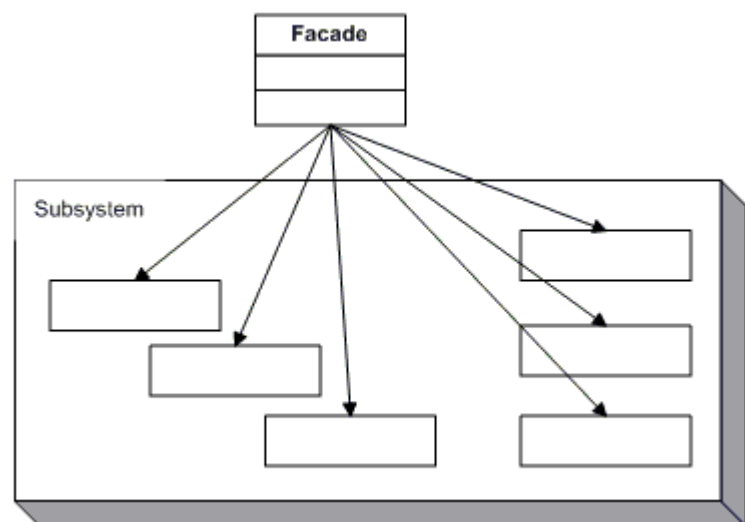
Teknologi

For læringens skyld, er der til DAL for web app'en blevet anvendt Entity Framework. Der er blevet anvendt samme objekt model som Fridge App, hvilket også gør synkronisering mulig mellem den lokale og eksterne database.

Façade mønster

Façade er et design mønster, hvorpå man skaber et simpelt interface til et kompleks subsystem (DoFactory, u.d.). Et eksempel på det ses på **Figur 27**.

Dette kan anvendes til at give BLL et simpelt interface til DAL, hvorpå vi kan skabe højere abstraktion, samt højere testbarhed for BLL.



Figur 27 Façade illustration (DoFactory, u.d.)

Endeligt design

Efter de designovervejelser der er blevet gjort i de forrige afsnit, samt anvendelsen af Repository og Unit of Work mønstret, er der blevet udarbejdet følgende klassediagram. Klassediagrammet kan ses i **bilag 07**.

Klassediagrammet er udarbejdet over en iterativ proces, derved har det ikke været det endelige klassediagram fra start. I de følgende afsnit, vil de væsentlige dele af klassediagrammet blive beskrevet i ansvarsområder.

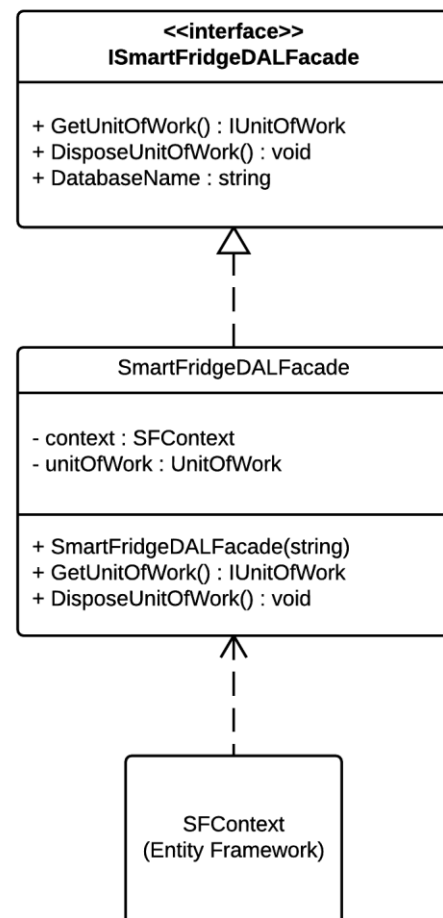
SmartFridgeDALFacade

ISmartFridgeDALFacade er et interface, som ses på **Figur 28**. Dette er en implementering af Façade mønstret, hvori man injecter et navn på en connectionstring i dens constructor, hvor den anvender en connectionstring fra web.config. Udover dette opretter den SFContext, som er DbContext for applikationen, implementeret vha. Entity Framework.

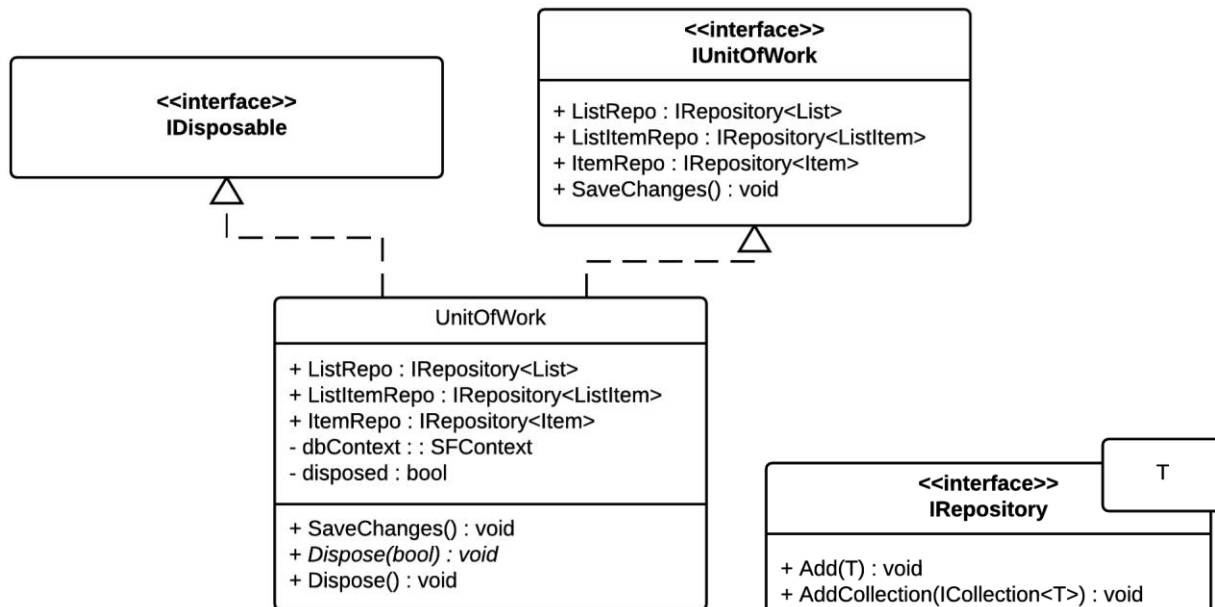
UnitOfWork

IUnitOfWork er et interface, som på **Figur 30**, der er en implementering af Unit of Work mønstret. Til forskel for Unit of Work for Fridge App, oprettes repositories i et unit of work, derved er alt databasetilgang, transaktioner.

UnitOfWork implementerer også IDisposable for at nedlægge Unit of Work for at nedlægge og frigive ressourcer korrekt.



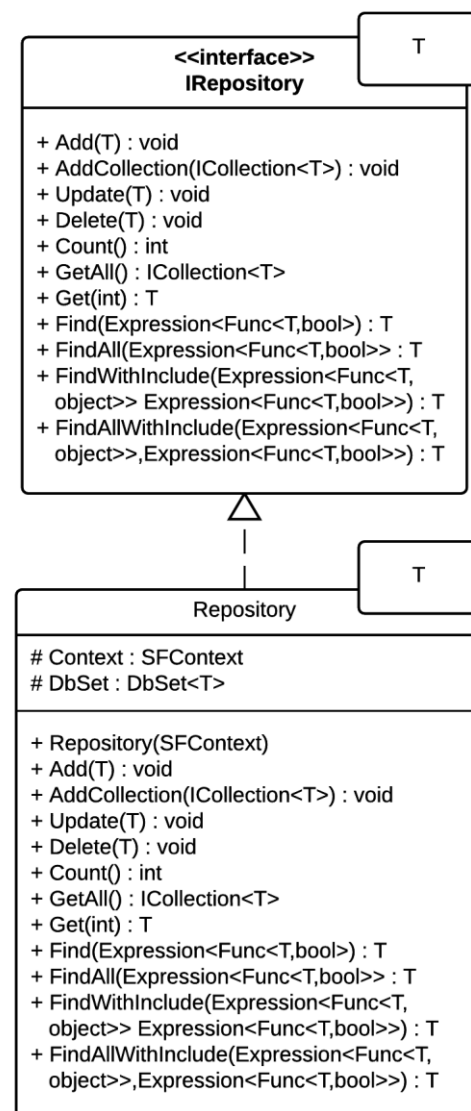
Figur 28 SmartFridgeDALFacade



Figur 30 UnitOfWork

Repository

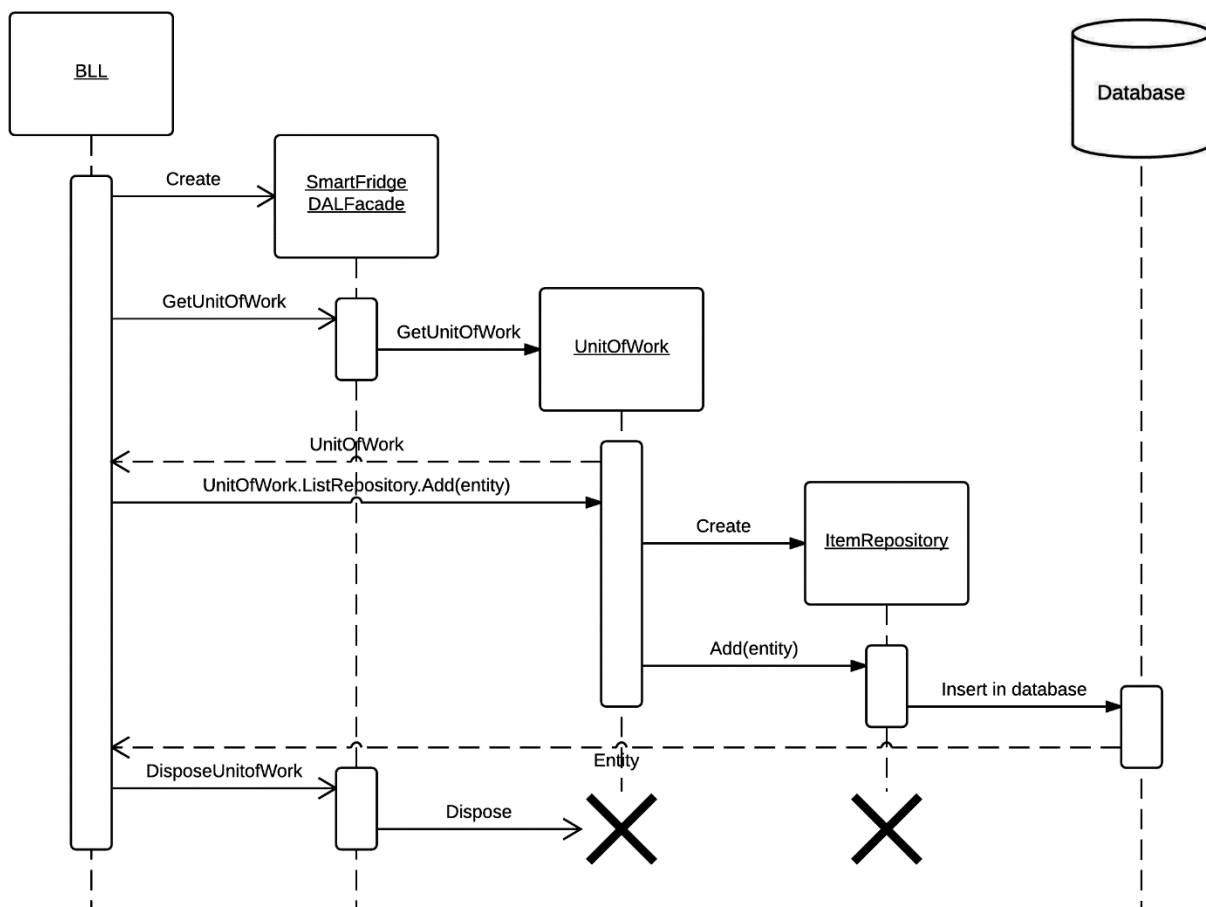
IRepository er et interface, som ses på **Figur 29** Dette er en implementering af Repository mønstret, hvor til forskel for Fridge app-implementering er dette implementeret som et generisk repository, i stedet for arv.



Figur 29 Repository

Anvendelse af DAL

Et sekvensdiagram for anvendelse af DAL for Web app, ses på **Figur 31**. Her ses det hvordan BLL kan oprette et SmartFridgeDALFacade objekt, hvor der kan oprettes et UnitOfWork med GetUnitOfWork. Heri kan man lave de ønskede transaktioner og derefter commite disse med SaveChanges(). Når man er færdig, kan man dispose UnitOfWork, hvorpå det nedlægges og man frigiver ressourcer.



Figur 31 Sekvensdiagram for anvendelse af DAL i Web app

Implementering

I dette afsnit vil implementeringen af DAL for Web app blive beskrevet, samt væsentlig funktionalitet.

Alt implementering er dokumenteret vha. XML comments og doxygen, som kan ses i **bilag 15**.

I de følgende afsnit vil væsentlig funktionalitet blive beskrevet, med eksempler.

GetUnitOfWork

```
public IUnitOfWork GetUnitOfWork()
{
    if (_unitOfWork != null)
        throw new InvalidOperationException("A Unit of Work is already in use.");

    _context = DatabaseName == null ? new SFContext() : new SFContext(DatabaseName);

    _unitOfWork = new UnitOfWork.UnitOfWork(_context);
    return _unitOfWork;
}
```

Kodestump 4 Kodeudklip fra SmartFridgeDALFacade.cs

I **Kodestump 4** ses et udsnit kode, hvor funktionaliteten for GetUnitOfWork() er implementeret. Her ses det hvordan der kan oprettes et Unit of Work, hvilket betyder det også kun er muligt at have en DbContext. Dette er fordelagtig, da der flere DbContexts kan skabe problemer i forhold til databaseforbindelse.

Test

I dette afsnit vil testning af DAL for Web app blive beskrevet, hvor coverage og statisk analyse vil blive dokumenteret. På **Figur 32**, ses testsuite for DAL i Web app. Som i DAL for Fridge App'en, er Repositoryet ikke blevet unit testet, da det igen er databasetransaktioner og det ikke er egentligt funktionalitet at teste. Udover det, er SFContext heller ikke testet, da det kommer fra Entity Framework, som må anses som gennemtestet.

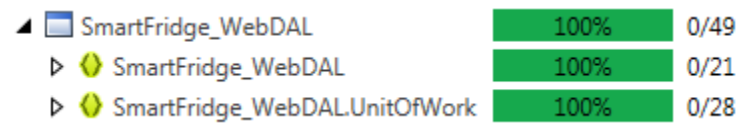
▲ ✓ <> DAL.Tests.Unit (15 tests)	Success
▲ ✓ SmartFridgeDALFacadeUnitTests (6 tests)	Success
✓ Ctor_parameterisTest_DatabaseNameIsTest	Success
✓ DisposeUnitOfWork_CalledAndUnitOfWorkIsNull_DoesNotThrowException	Success
✓ DisposeUnitOfWork_CallsGetThenDisposeThenGet_DoesNotThrowException	Success
✓ GetUnitOfWork_CalledAndDatabaseNameIsNull_ReturnsUoW	Success
✓ GetUnitOfWork_CalledAndDatabaseNameIsTest_ReturnsUoW	Success
✓ GetUnitOfWork_CalledTwice_ThrowsInvalidOperationException	Success
▲ ✓ UnitOfWorkUnitTests (9 tests)	Success
✓ Dispose_DisposedCalledTwiceSoDisposedIsTrue_DoesNotDisposeTwice	Success
✓ Dispose_DisposedIsFalse_DisposesContext	Success
✓ ItemRepo_GotItemRepo_ReturnsSameItemRepo	Success
✓ ItemRepo_itemRepoIsNull_CreatesNewItemRepo	Success
✓ ListItemRepo_GotListItemRepo_ReturnsSameListItemRepo	Success
✓ ListItemRepo_listitemRepoIsNull_CreatesNewListItemRepo	Success
✓ ListRepo_GotListRepo_ReturnsSameListRepo	Success
✓ ListRepo_listRepoIsNull_CreatesNewListRepo	Success
✓ SaveChanges_CallSaveChanges_ContextRecievesSaveChanges	Success

Figur 32 Screenshot af testsuite for DAL for Web app

Coverage

På **Figur 33** Screenshot af coverage for WebDAL ses coverage resultater for web DAL.

Her ses det, modsat Fridge app DAL, at der er opnået 100% coverage af den testede funktionalitet. Dette viser også at implementering er langt mere testbart, hvilket er en af fordelene ved anvendelsen af Entity Framework.

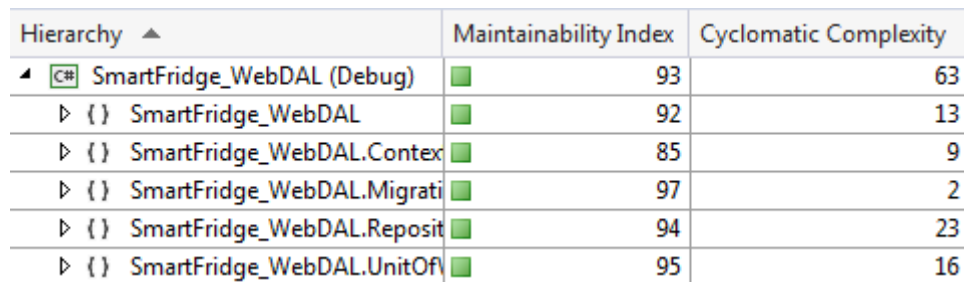


SmartFridge_WebDAL	100%	0/49
SmartFridge_WebDAL	100%	0/21
SmartFridge_WebDAL.UnitOfWork	100%	0/28

Figur 33 Screenshot af coverage for WebDAL

Statisk analyse

På **Figur 34**, ses code metrics af DAL for Web app. Her ses det at det er højere maintainability, samt lavere kompleksitet. Dette betyder at implementeringen af WebDAL har været mindre kompleks, hvilket Entity Framework har stor skyld for.



Hierarchy ▲		Maintainability Index	Cyclomatic Complexity
SmartFridge_WebDAL (Debug)		93	63
SmartFridge_WebDAL		92	13
SmartFridge_WebDAL.Context		85	9
SmartFridge_WebDAL.Migrati		97	2
SmartFridge_WebDAL.Reposit		94	23
SmartFridge_WebDAL.UnitOf		95	16

Figur 34 Screenshot af Code metrics for WebDAL

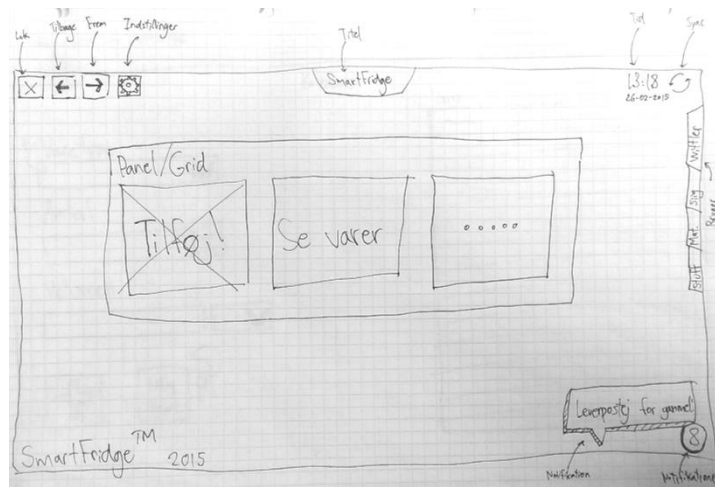
Fridge app

Design

Dette afsnit indeholder de grafiske og kodemæssige designovervejelser for de forskellige vinduer i Fridge App. Overvejelserne er baseret på en brainstorm (**bilag 09**). For større billeder, henvises til **bilag 10**.

Hovedmenu og ramme

Designet af hovedmenuen og rammen er blevet skitseret som i **Figur 35**.



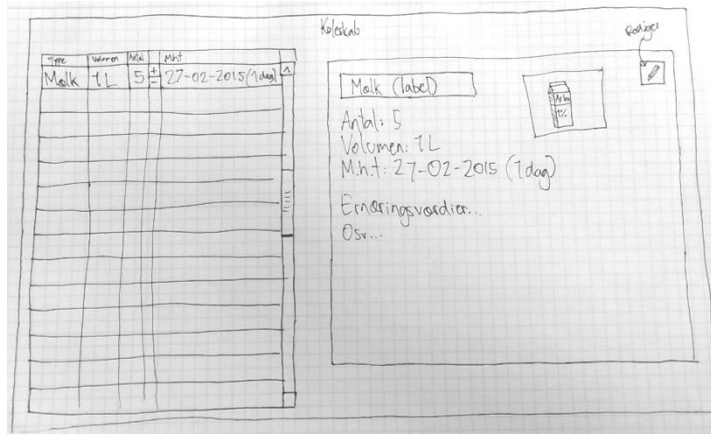
Figur 35 Skitse af hovedmenu og ramme

Rammen, som er alt omkring firkanten *Panel/Grid*, vil ikke ændres, når der ændres kontekst.

En senere ændring, i forhold til skitsen, har været at knapperne i *Panel/Grid* vil være direkte henvisende til de eksisterende lister, hvorfra det vil være muligt at tilføje, redigere og fjerne varer direkte.

Se varer

Designet af menuen for "Se varer" er blevet skitseret som i **Figur 36**.



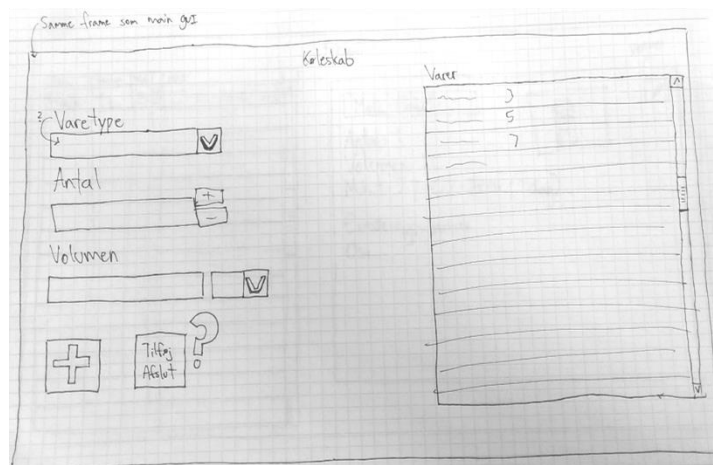
Figur 36 Skitse af "Se varer"

Siden skitsen, er desuden tilføjet en *Slet*-knap til venstre for hver vare, hvorfra det er muligt at fjerne en vare fra listen. Det skal desuden være muligt inkrementere/dekrementere antallet af vare med én ved tryk på '+/-'-knapper, som vil befinde sig under knappen *Rediger*.

Ernæringsværdier er et eksempel på hvor ekstra information fra eventuelle udvidelser vil befinde sig. Det vil altså ikke være en del af kernefunktionaliteten.

Tilføj vare

Designet af menuen for "Tilføj varer" er blevet skitseret som i **Figur 37**.



Figur 37 Skitse af "Tilføj varer"

Med implementering af mulighed for at indtaste holdbarhedsdato, er dette felt siden skitsen blevet tilføjet under *Volumen*-feltet.

I4 PRJ4, Gruppe 5

IKT

Implementering

[View Layer](#)

[Hovedvindue](#)

Øverste lag i *View Layer* er hovedvinduet (*MainWindow*), som består af en ramme omkring en *UserControl*, som det fremgår af **Figur 38**.



Figur 38 Hovedmenu

Rammen indeholder en mængde funktionalitet, som på et hvilket som helst givet tidspunkt i programmet vil være tilgængeligt.

Det røde kryds øverst til venstre repræsenterer muligheden for at afslutte programmet (*Close_Button_Clicked*).

De to orange pile tillader navigering frem (*Button_Forward_Clicked*) og tilbage (*Button_Back_Clicked*) blandt de tidligere besøgte sider. Selve funktionaliteten for disse funktioner er implementeret i *CtrlTemplate*-klassen, og vil blive uddybet i det følgende afsnit.

Det orange hus repræsenterer *Hjem*-knappen, som til enhver tid tillader brugeren at returnere til hovedmenuen; altså indlæser *UserControl*-klassen *ShowListSelection*.

Det blå tandhjul repræsenterer muligheden for at ændre på indstillinger. Da dette endnu ikke er implementeret, er den i første omgang blot en *placeholder*.

I øverste højre hjørne, ved siden af uret, som også styres internt i klassen (*timer_Tick*), ses status på synkronisering (*syncStatus*). Når alle lokale data er synkroniseret med den eksterne database, vises et grønt flueben i knappen. Hvert 10. minut forsøges en synkronisering automatisk (*eventT*), og i mellemtiden kan en synkronisering initieres ved et tryk på knappen (*SyncButton_OnClick*). I begge tilfælde skifter synkroniseringsknappens billede (*ChangeSyncImage*) til kun at indeholde to roterende pile. Hvis en synkronisering er fejlet, ændres billedet til at indeholde et rødt kryds.

Nederst til højre er en knap, som illustrerer antallet af notifikationer. Såfremt en vare er for gammel, vil der komme en notifikation om dette, samt et knap som tillader brugeren at fjerne notifikationen. Trykkes der et andet sted på skærmen, minimeres notifikationerne igen.

MainWindow
<ul style="list-style-type: none"> - timer: DispatcherTimer - clock: Clock - popup: Popup - Panel: StackPanel - eventT: EventTimer - syncStatus: SyncStatus - CtrlTemp: CtrlTemplate - UpdateNotificationButton(): void - DeleteNotification(object): void - TrySync(): void - Syncing(): void - ChangeSyncImage(): void - SyncButton_OnClick(object, RoutedEventArgs): void - Button_Home_Clicked(object, RoutedEventArgs): void - Button_Back_Clicked(object, RoutedEventArgs): void - Button_Forward_Clicked(object, RoutedEventArgs): void - Close_Button_Clicked(object, RoutedEventArgs): void - NotificationsButton_OnClick(object, RoutedEventArgs): void - AddNotificationsToPanel(List<Notification>, StackPanel): void - timer_Tick(object, EventArgs): void <hr/> <ul style="list-style-type: none"> + DeleteNotificationCommand: ICommand + UpdateNotificationsAmount(): void

Figur 39 Klassen *MainWindow*

Control Template

Indholdet i hovedvinduet *UserControl* styres af *UserControl*-klassen *CtrlTemplate* (**Figur 40**), som ved initialisering indlæser oversigten over tilgængelige lister. På samme tid oprettes en instans af objektet *BLL* (Business Logic Layer), som håndterer al ikke grafisk relateret *code behind*.

Constructoren for enhver af de implementerede *UserControls* tager imod en instans af klassen *CtrlTemplate*. Når et skifte ønskes, kaldes funktionen *ChangeGridContent* i *CtrlTemplate* med den ønskede *UserControl*, som indlæses i stedet for den nuværende, som set i **Kodestump 5**.

```
private UserControl _uc;

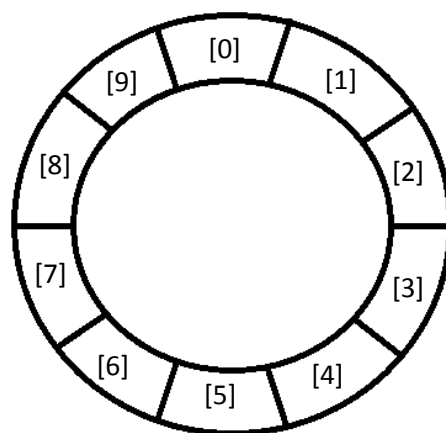
public void ChangeGridContent(UserControl uc)
{
    _uc = uc;
    CtrlTempGrid.Children.Clear();
    CtrlTempGrid.Children.Add(_uc);
    [...]
}
```

Kodestump 5 Skift af *UserControl* i *CtrlTemplate*

CtrlTemplate lagrer desuden de foregående *UserControls*, som programmet har vist, i en cirkulær buffer (**Figur 41**). Dette muliggør brugen af *Frem*- og *Tilbage*-knapperne, som ses øverst til venstre på **Figur 38**.

CtrlTemplate
<ul style="list-style-type: none"> - _uc: UserControl - NavHisColPos: int - NavHisColPosOrgPos: int
<ul style="list-style-type: none"> + _bll: BLL + NavHisCol: UserControl[] + ChangeGridContent(UserControl): void + NavigateBack(): void + NavigateForward(): void

Figur 40 Klassen *CtrlTemplate*



Figur 41 Cirkulær buffer

Den cirkulære buffer initieres som i **Kodestump 6**. Bemærk at attributternes navne er forkortet her af pladshensyn. For den fulde implementering, henvises til **bilag 11**.

```
public UserControl[] NavHisCol { get; private set; }
private int NavHisColPos;
private int NavHisColOrgPos;

public CtrlTemplate()
{
    [...]
    _uc = new CtrlShowListSelection(this);
    [...]
    NavHisCol = new UserControl[10];
    NavHisCol[0] = _uc;
    NavHisColPos = 0;
    NavHisColOrgPos = NavHisColPos;
}
```

Kodestump 6 Initiering af cirkulær buffer

NavigationHistoryCollection er selve arrayet, hvori bufferen lagres.

NavigationHistoryCollectionPosition holder styr på hvor den nuværende *UserControl* er placeret, mens *NavigationHistoryCollectionOriginalPosition* holder styr på den foregående *UserControl*. Disse benyttes at sikre at vi kan finde den korrekte *UserControl*, og at der ikke bevæges til *UserControl*, som ikke længere er gyldig.

Efter dette, lagres hver ny *UserControl* på den næste plads i bufferen, så snart den indlæses, og med funktionerne *NavigateBack* og *NavigateForwards*, benyttes *Frem-* og *Tilbage*-knapperne til at navigere i bufferen.

Show List Selection

Klassen *CtrlShowListSelection* (**Figur 42**) er den første *UserControl*, som indlæses i *CtrlTemplate*, når sidstnævnte initieres. Den indeholder en oversigt over de tilgængelige lister, og sørger for at oprette den rette *UserControl*, alt efter brugerens valg.

CtrlShowListSelection

```
- _ctrlTemp: CtrlTemplate
- BtnInFridge_Click(object, RoutedEventArgs): void
- BtnShoppingList_Click(object, RoutedEventArgs): void
- BtnStdContent_Click(object, RoutedEventArgs): void
+ CtrlShowListSelection(CtrlTemplate)
```

Figur 42 Klassen CtrlShowListSelection

Constructoren for *CtrlShowListSelection* tager imod *CtrlTemplate*-klassen, som har kaldt den, således at den kan benytte *ChangeGridContent*-funktionen til at skifte *UserControl*, når brugeren har trykket på en af knapperne, som set i **Kodestump 7**.

```
private CtrlTemplate _ctrlTemp;

private void BtnInFridge_Click(object sender, RoutedEventArgs e)
{
    _ctrlTemp.ChangeGridContent(new CtrlItemList("Køleskab", _ctrlTemp));
}
```

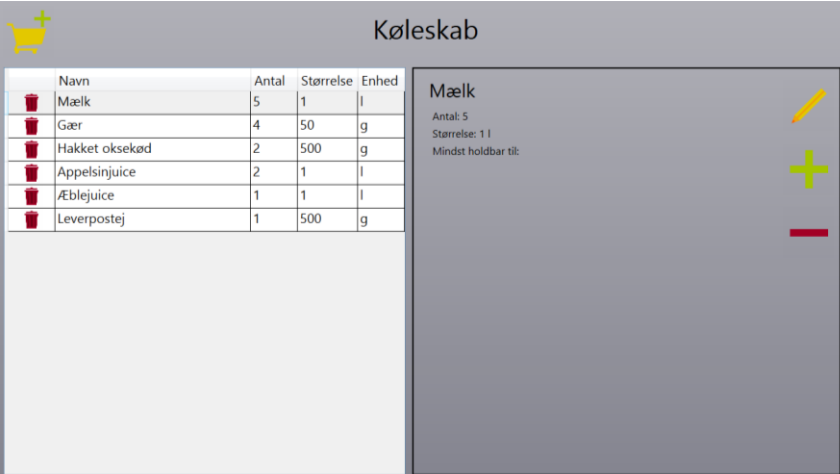
Kodestump 7 Skift af UserControl fra CtrlShowListSelection

Samme fremgangsmåde benyttes ved tryk på de andre lister.

Bemærk at listen identificeres på baggrund af en *string*. Denne umiddelbart hårde kodning er foretaget, med øje for muligheden for at udvide systemet til at lade brugeren oprette nye lister, som hver især kan identificeres på baggrund af deres navne. Alternativt kunne listerne oprettes som objekter, men da de alligevel ikke skulle indeholde andet end et navn, blev det valgt at en *string* var tilstrækkelig.

Item List

Figur 43 viser den grafiske repræsentation af *CtrlItemList*, uden den omkringliggende ramme.



	Navn	Antal	Størrelse	Enhed
	Mælk	5	1	l
	Gær	4	50	g
	Hakket oksekød	2	500	g
	Appelsinjuice	2	1	l
	Æblejuice	1	1	l
	Leverpostej	1	500	g

Mælk

Antal: 5

Størrelse: 1 l

Mindst holdbar til:

Figur 43 Listen "Køleskab"

Denne klasse (**Figur 44**) indeholder en del mere funktionalitet end de foregående, og gør derfor i stor stil brug af *Business Logic Layer*'et (BLL), som håndterer den bagvedliggende logik, mens *CtrlItem* selv håndterer den grafiske repræsentation.

CtrlItem viser indholdet af den valgte liste, baseret på den medsendte *string*, som kædes sammen med listen af samme navn i databasen.

Når *CtrlItem* oprettes, indlæses alle data (*LoadItemData*) fra den pågældende database i et *DataGrid*, og et klik på en række (*DataGridItems_SelectionChanged*) udløser nærmere informationer om den enkelte vare, som fremkommer i informationsvinduet i højre side af vinduet.

Til venstre for hver vare, repræsenterer den røde skraldespand muligheden for at slette en vare (*BtnDelete_Click*).

I informationsvinduet repræsenterer den gule blyant muligheden for at redigere den valgte vare (*BtnEdit_Click*), hvilket resulterer i at vareinformationerne omdannes til redigerbare tekstblokke, som illustreret i **Figur 45**. Efter redigeringen, er det muligt at bekræfte (*BtnAccept*) ved at trykke på det grønne flueben, hvorved ændringerne persisteres, eller annullere (*BtnCancel_Click*) ved at trykke på det røde kryds, hvorved ændringerne ignoreres.

Ved klik på det grønne plus er det muligt at øge (*BtnInc_Click*) eller formindske (*BtnDec_Click*) mængden af den valgte vare med én, uanset om der er trykket på *Rediger*. Disse funktioner syntes relevante at gøre let tilgængelige, for at gøre løbende forbrug, og registrering af ofte brugte varer, hurtigere at registrere.

Øverst til venstre i *CtrlItem* ses en gul indkøbsvogn med et grønt plus. Denne knap tillader brugeren at tilføje nye varer (*BtnAddItem_Click*), og fører til den sidste *UserControl*-klasse, *Additem*.

CtrlItem
<ul style="list-style-type: none"> - _ctrlTemp: CtrlTemplate - CtrlItem(string, CtrlTemplate) - selectedItemOld: GUIItem - selectedItem: GUIItem - GUIItems: ObservableCollection<GUIItem> - unitnames: List<string> - LoadItemData(): void - ShowButtonsAndTextBoxes(): void - HideButtonsAndTextBoxes(): void - GetUnitNames(): void - DataGridItems_SelectionChanged(object, SelectionChangedEventArgs): void - SelectedUnitCB_OnDropDownClosed(object, EventArgs): void - ButtonInc_Click(object, RoutedEventArgs): void - BtnDec_Click(object, RoutedEventArgs): void - BtnEdit_Click(object, RoutedEventArgs): void - BtnDelete_Click(object, RoutedEventArgs): void - BtnCancel_Click(object, RoutedEventArgs): void - BtnAccept_Click(object, RoutedEventArgs): void - BtnAddItem_Click(object, RoutedEventArgs): void <hr/> + ListType: string

Figur 44 Klassen *CtrlShowListSelection*



Figur 45 Rediger vare

Kodestump 8 viser hvordan der skiftes *UserControl* fra en liste til vinduet hvor varer tilføjes.

```
private void BtnAddItem_Click(object sender, RoutedEventArgs e)
{
    _ctrlTemp.ChangeGridContent(new AddItem(ListType, _ctrlTemp));
}
```

Kodestump 8 Skift af UserControl fra CtrlItemList til AddItem

CtrlTemplate-objektet, som blev sendt med og lagret i constructoren, benyttes til at kalde funktionen *ChangeGridContent*, hvori en ny instans af *AddItem* oprettes med listens egen type og det samme *CtrlTemplate*-objekt.

Kodestump 9 viser hvordan *BLL* benyttes til at håndtere funktionaliteten.

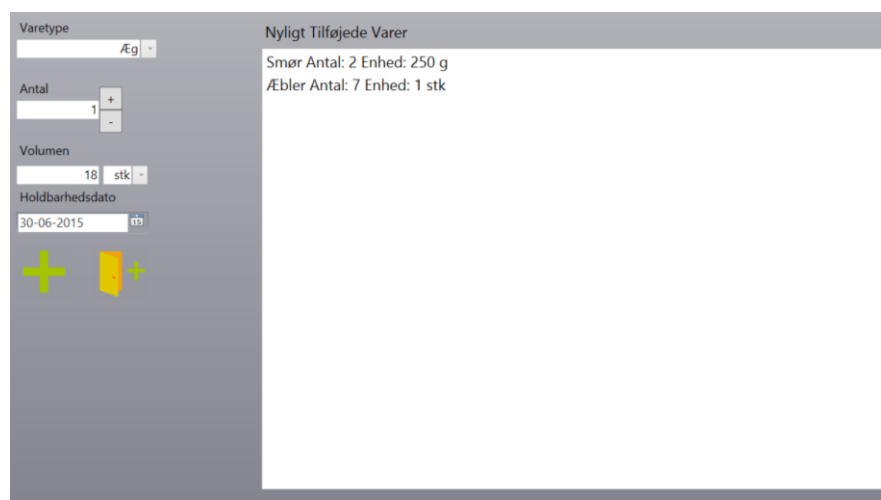
```
private async void BtnDelete_Click(object sender, RoutedEventArgs e)
{
    GUIItem itemDelete = DataGridItems.SelectedItem as GUIItem;
    GUIItems.Remove(itemDelete);
    _ctrlTemp._bll.DeleteItem(itemDelete);
    DataGridItems.UnselectAllCells();
    DataGridItems.SelectedIndex = 0;
}
```

Kodestump 9 Sletning af vare ved hjælp af Business Logic Layer

Mens funktionen selv håndterer den grafiske fjernelse af det pågældende objekt, benyttes *BLL*-objektet i *CtrlTemplate*-objektet til at slette varen fra databasen.

Add Item

Figur 47 viser den grafiske repræsentation af *AddItem*, uden den omkringliggende ramme.



Figur 46 Tilføj vare(r)

Som sine forgænger, *CtrlItemList*, kontrollerer denne klasse (**Figur 47**) meget funktionalitet, som opnås gennem funktionskald til klassen *Business Logic Layer (BLL)*, mens den grafiske repræsentation håndteres af klassen selv. Brugen af *BLL* foregår, som i *CtrlItemList*, gennem det medsendte *CtrlTemplate*-objekt.

AddItem tilbyder muligheden for at tilføje én eller flere varer til listen, hvorfra klassen blev kaldt.

Varens navn skrives ud for *Varetype*, antallet skrives ud for *Antal*, mængden og enheden skrives ud for *Volumen*, og holdbarhedsdatoen skrives under *Holdbarhedsdato*.

Det grønne plus repræsenterer muligheden for at tilføje varen (*AddButton_Click*) til den midlertidige liste (*newItems*) i højre side af vinduet. Ved klik på døren med det grønne plus, tilføjes og persisteres varen (*AddExitButton_Click*), sammen med varerne i den midlertidige liste, i databasen.

Varetype skrives i en *ComboBox*, hvor der ved klik kan ses forslag til eksisterende varer, som kan vælges hvis ønskes.

Enhed ud for *Antal* er også vist som en *ComboBox*, hvor det er muligt at vælge eksisterende enheder, eller en ny kan skrives.

Ud for *Antal* ses to '+'/'- knapper, som kan benyttes til at øge (*PlusButton_Click*) eller formindske (*MinusButton_Click*) mængden af den pågældende vare med én.

AddItem

```
- newItems: ObservableCollection<GUIItem>
- types: ObservableCollection<GUIItem>
- amount: uint
- selectedType: string
- _currentList: string
- _ctrlTemp: CtrlTemplate
- GetTypeItemFromName(string): GUIItem
- UppercaseFirst(string): string
- CreateNewItem(): GUIItem
- AddNewItem(GUIItem): void
- UpdateTextboxesFromType(GUIItem): void
- Exit(): void
- AddButton_Click(object, RoutedEventArgs): void
- AddExitButton_Click(object, RoutedEventArgs): void
- PlusButton_Click(object, RoutedEventArgs): void
- MinusButton_Click(object, RoutedEventArgs): void
- TextBoxAntal_LostFocus(object, RoutedEventArgs): void
- TextBoxVareType_OnLostFocus(object, RoutedEventArgs): void
- ComboBoxVareType_OnLostFocus(object, RoutedEventArgs): void
- ComboBoxUnit_OnDropDownClosed(object, RoutedEventArgs): void
+ AddItem(string, CtrlTemplate)
```

Figur 47 Klassen *AddItem*

Business Logic Layer

Constructor

I constructoren, der kan ses på **Kodestump 10** oprettes aller først den AdoNetContext som skal bruges til at oprette forbindelse til den lokale database med. Herefter oprettes de tre repositories, som skal bruges til at tilgå databasens forskellige tabeller.

```
public BLL()
{
    Context = new AdoNetContext(_connectionFactory);
    _itemRepository = new ItemRepository(Context);
    _listRepository = new ListRepository(Context);
    _listItemRepository = new ListItemRepository(Context);
}
```

Kodestump 10 Business Logic Layer constructoren

WatchItems

WatchItems er en property, der bruges til at hente en liste med alle de relevante GUIItems for en given liste til GUIen. Det første den gør er at indlæse hele databasens indhold med LoadFromDB(); funktionen, der er beskrevet senere i dette afsnit. Herefter finder den ud af om listen den skal fylde i eksisterede på i databasen. Hvis ikke listen eksisterede i forvejen oprettes den, dette kan ses på **Kodestump 11**.

```
List<string> temp = new List<string>();
foreach (var list in Lists)
{
    temp.Add(list.Name);
}

if (!temp.Contains(CurrentList))
{
    CreateList();
}
```

*Kodestump 11 Lists er en property,
der indeholder listerne.*

Efter det findes alle de ListItem's i databasen, der skal figurere på den nuværende liste. For hvert ListItem der bliver fundet oprettes et tilsvarende GUIItem, hvis type er givet af den Item, det pågældende ListItem er koblet sammen med.

Hele processen kan ses i **Kodestump 12**. Alle disse GUIItems ligges i en liste, som til sidst bliver returneret.

```
foreach (var dbListItem in _dblistItems)
{
    if (dbListItem.List.ListName == CurrentList)
    {
        foreach (var dbItem in _dbItems)
        {
            if (dbListItem.Item.ItemId == dbItem.ItemId)
            {
                GUIItem guiItem = new GUIItem();
                guiItem.Type = dbItem.ItemName;
                guiItem.Amount = (uint)dbListItem.Amount;
                guiItem.Unit = dbListItem.Unit;
                guiItem.Size = (uint)dbListItem.Volume;
                guiItem.ShelfLife = dbListItem.ShelfLife;
                guiItems.Add(guiItem);
            }
        }
    }
}
```

Kodestump 12 Her ses, hvordan alle ListItems, der skal figurere på den nuværende liste, kobles sammen med deres respektive item i et GUIItem

LoadFromDB

LoadFromDB indlæser alt fra databasen, og kommet indholdet i lokale objekter. Som det kan ses på **Kodestump 13**, gøres dette ved at kalde funktionen GetAll(); på samtlige repositories. Da denne funktion returnere en IEnumerable kaldes ToList(); for at lave dem til lister. Når alt ligger lokalt kaldes Mapper(); funktionen, der sørger for at de lokale referencer er som de skal være. Listerne fra databasen bruges til at oprette lokale GUIItemList objekter, der til forskel fra en List fra databasen har en indbygget liste af GUIItems.

```
List<List> lists = new List<List>();
using (var uow = Context.CreateUnitOfWork())
{
    lists = _listRepository.GetAll().ToList();
    _dbItems = _itemRepository.GetAll().ToList();
    _dblistItems = _listItemRepository.GetAll().ToList();
    _listItemRepository.Mapper(_dbItems, lists, _dblistItems);
}
```

Kodestump 13 Her indlæses databasens inhold

Når alt ligger lokalt og alle GUIItemListerne er oprettet sammenkobles alle ListItems med deres respektive Item, i et GUIItem.

Alle disse GUIItems ligger i deres tilhørende lokale GUIItemList, som det kan ses på **Kodestump 14**

```
foreach (var list in Lists)
{
    foreach (var listItem in _dblistItems)
    {
        if (listItem.List.ListId == list.ID)
            foreach (var dbItem in _dbItems)
            {
                if (listItem.Item.ItemId == dbItem.ItemId)
                {
                    GUIItem guiItem = new GUIItem()
                    {
                        Amount = (uint)listItem.Amount,
                        Type = dbItem.ItemName,
                        Size = (uint)listItem.Volume,
                        Unit = listItem.Unit,
                        ID = listItem.ItemId,
                        ShelfLife = listItem.ShelfLife
                    };
                    list.ItemList.Add(guiItem);
                    break;
                }
            }
    }
}
```

Kodestump 14 Her ligger alle ListItems og Items sammen til GUIItems, der fordeles ud på deres respektive lokale lister

CheckShelfLife

CheckShelfLife funktionen går alle items på en specifik liste igennem for at finde ud af, hvorvidt nogle af disse har overskredet deres sidste holdbarhedsdato. Den gør dette ved at sammenligne hvert GUIItems ShelfLife property, med dagens dato. For hvert GUIItem, der er blevet for gammel, tilføjes en notifikation til en liste, der til sidst returneres som, det kan ses på **Kodestump 15**.

```
var list = new List<Notification>();
foreach (var item in items.ItemList)
{
    if (item.ShelfLife != null)
    {
        if (item.ShelfLife.Date <= DateTime.Now)
        {
            string message = item.Type + " blev for gammel d. " + DateTime.Now.Date;
            list.Add(new Notification(message, DateTime.Now, item.ID));
        }
    }
}
return list;
```

I4 PRJ4, Gruppe 5

IKT

Kodestump 15 Her kan ses hvordan en notifikation med information omkring hvilken varer, der blevet for gammel, tilføjes for hver vare hvis holdbarhedsdato er overskreden.

AddItemsToTable

AddItemsToTable undersøger først, hvorvidt den liste funktionen blev fra eksistere, hvis den ikke før det bruges CreateList. Når listen eksistere findes den rigtige lokale liste dens id skal bruges når der skal tilføjes ListItems til databasen. Herefter løbes alle de GUIItems, der skal tilføjes igennem for at finde ud af om der er nogle af dem, som er af ikke i forvejen eksisterende type. Dette kan ses på **Kodestump 16**, her bruges IsNewItem til at finde ud af, hvorvidt et GUIItem er af en ny type eller ej. Bliver der tilføjet nye Items, kaldes LoadFromDB så de nye Items også ligger lokalt.

```
bool newItemAdded = false;
using (var uow = Context.CreateUnitOfWork())
{
    foreach (var newGuiItem in newGuiItems)
    {
        if (IsNewItem(newGuiItem))
        {
            Item dbItem = new Item()
            {
                ItemName = newGuiItem.Type,
                StdUnit = newGuiItem.Unit,
                StdVolume = (int)newGuiItem.Size
            };
            _itemRepository.Insert(dbItem);
            newItemAdded = true;
        }
    }
    uow.SaveChanges();
}
if (newItemAdded)
    LoadFromDB();
```

*Kodestump 16 Her tilføjes alle nye typer af Items,
hvorefter der loades fra databasen igen*

Når de nyeste opdateringer er hentet ned, tjekkes der på hver af de nye GUIItems om den kaldende liste allerede har et GUIItem af samme type. Findes der ikke et match, laves der et ListItem ud fra det nye GUIItems attributter, og det tilføjes til databasen. Eksistere der allerede et GUIItem af samme type sammenlignes der på deres størrelse/volume, enhed og udløbsdato, er der blot en af disse, som ikke stemmer overens med det i forvejen eksisterende tilføjes der et nyt ListItem.

Stemmer alle parametrene overens laves der et nyt ListItem, hvor antallet er summen af det gamle og det nye item. Herefter slettes det gamle ListItem, og det nye indsættes i databasen, hvilket kan ses på **Kodestump 17**.

```
[...]
if (dbItem.ItemName == newGuiItem.Type)
{
    foreach (var dbListItem in _dblistItems)
    {
        if (dbItem.ItemId == dbListItem.Item.ItemId &&
            dbListItem.List.ListId == currentGuiItemList.ID &&
            dbListItem.Unit == newGuiItem.Unit &&
            dbListItem.Volume == newGuiItem.Size)
        {
            int currentAmount = dbListItem.Amount;
            ListItem updatedListItem = new ListItem(((int)newGuiItem.Amount +
                                                    currentAmount),
                                                    (int)newGuiItem.Size,
                                                    newGuiItem.Unit,
                                                    dbListItem.List,
                                                    dbItem,
                                                    newGuiItem.ShelfLife);

            _listItemRepository.Delete(dbListItem);
            _listItemRepository.Insert(updatedListItem);
        }
    }
}
```

Kodestump 17 Her kan ses hvordan det håndteres når der bliver tilføjet et allerede eksisterende GUIItem

Når alle nye GUIItems er tilføjet til databasen på den ene eller den anden måde kaldes SaveChanges(); så ændringerne gemmes i databasen, og LoadFromDB(); kaldes for at få alle disse ændringer gemt lokalt også. Når det nyeste ligger lokalt kaldes Mapper(); der sikrer at alle referencer er som de skal være. Til sidst kaldes STDToShopListControl, som er beskrevet længere nede i dette afsnit.

Deleteltem

Deleteltem finder først den rigtige lokale liste, da listens id skal bruges til at finde den ListItem, der skal slettes fra databasen. Bagefter som vist på **Kodestump 18** findes og slettes det fundne ListItem. Når det er gjort laves et kald til `uow.SaveChanges()`, så ændringen også sker i databasen.

```
using (var uow = Context.CreateUnitOfWork())
{
    foreach (var dbListItem in _dblistItems)
    {
        if (dbListItem.Item.ItemName == GUIitemToDelete.Type
            && dbListItem.Amount == GUIitemToDelete.Amount
            && dbListItem.Unit == GUIitemToDelete.Unit
            && (uint)dbListItem.Volume == GUIitemToDelete.Size
            && dbListItem.ListId == currentGuiItemList.ID)
        {
            _listItemRepository.Delete(dbListItem);
            break;
        }
    }
    uow.SaveChanges();
}
```

Kodestump 18 Her kan ses hvordan et ListItem slettes fra databasen

Changeltem

Changeltem starte ligesom Deleteltem med først at finde den rigtige lokale liste da dennes id skal bruges til at finde det rigtige ListItem. Herefter tjekker den på om det kun er det fundne ListItem, der skal opdateres eller om den tilhørende Item også skal opdateres. Hvis Itemet skal opdateres gøres det som det kan ses på **Kodestump 19**.

```
[...]
if (oldItem.Type != newItem.Type)
{
    foreach (var dbItem in _dbItems)
    {
        if (dbItem.ItemName == oldItem.Type)
        {
            dbItem.ItemName = newItem.Type;
            _itemRepository.Update(dbItem);
        }
    }
}
[...]
```

Kodestump 19 Her opdateres et ListItems tilhørende item

Når det rigtige ListItem er fundet, bliver dette opdateret ved at det allerede eksisterende ListItem slettes, og der indsættes et nyt med de opdaterede værdier som det ses på **Kodestump 20**.

```
[...]
ListItem updatedListItem = new ListItem((int)newItem.Amount,
                                         (int)newItem.Size,
                                         newItem.Unit,
                                         dbListItem.List,
                                         dbListItem.Item,
                                         newItem.ShelfLife);
_listItemRepository.Delete(dbListItem);
_listItemRepository.Insert(updatedListItem);
uow.SaveChanges();
break;
[...]
```

***Kodestump 20** Her "opdateres" et ListItem ved først at slette det gamle ListItem, og derefter indsætte et nyt med opdaterede værdier*

STDToShopListControl

STDToShopListControl tjekker først på om den er blevet kaldt fra "Standard-beholdning" listen, da det kun er ved kalde derfra den skal gøre noget. Derfor returnere den bare, hvis kaldet kom fra en af de to andre liste. På **Kodestump 21** kan det ses, hvordan den opretter to lister af ListItems med varer der henholdsvis er i køleskabet, eller som Bruger har valgt altid skal være i køleskabet.

```
foreach (var dbListItem in _dblistItems)
{
    if (dbListItem.List.ListName == "Køleskab")
    {
        har.Add(dbListItem);
    }
    else if (dbListItem.List.ListName == "Standard-beholdning")
    {
        skalAltidHave.Add(dbListItem);
    }
}
```

***Kodestump 21** Her kan ses, hvorledes der oprettes en liste over ting, der er i køleskabet, og ting der skal være i køleskabet*

Listen over ting, der altid skal være i køleskabet kopieres over i en ny liste ved navn "mangler", og der kigges nu efter overensstemmelser mellem de to tidligere lister. Findes der et ListItem i køleskabet, hvis amount er større end eller lig med det fra standard-beholdningen, fjernes det fra mangler listen, ellers, hvis antallet er under bliver differensen mellem disse, antallet på "mangler" listen.

På **Kodestump 22** kan ses, hvordan dette er implementeret. Her efter løbes "mangler" listen igennem og samtlige ListItems kobles sammen deres tilhørende Item i et GUIItem, som tilføjes til en ny liste. Denne liste tilføjes til "Indkøbsliste" med AddItemsToTable funktionen.

```
List<ListItem> mangler = new List<ListItem>(skalAltidHave);
foreach (var STDListItem in skalAltidHave)
{
    foreach (var ownedListItem in har)
    {
        if (ownedListItem.Item.ItemName == STDListItem.Item.ItemName &&
            ownedListItem.Unit == STDListItem.Unit &&
            ownedListItem.Volume == STDListItem.Volume)
        {
            if (ownedListItem.Amount >= STDListItem.Amount)
            {
                mangler.Remove(STDListItem);
            }
            if (ownedListItem.Amount <= STDListItem.Amount)
            {
                STDListItem.Amount -= ownedListItem.Amount;
            }
        }
    }
}
```

Kodestump 22 Her fjernes alle vare som allerede er i køleskabet i et korrekt antal fra listen over varer, der mangler. Varer, hvor antallet er forkert, bliver stående med det manglende antal

Test

Som følge af det kodemæssige design, som i første omgang blev oprettet med ren *code behind*, og senere, med inkorporering af *Data Access Layer (DAL)*-laget, blev opdelt i *View* og *Business Logic Layer (BLL)*, er der opnået en ringe grad af testbarhed i *Fridge app*. Fokus har ligget på test af den bagvedliggende funktionalitet, frem for at bruge for meget tid på at omdesigne et, allerede på daværende tidspunkt, velfungerende system.

Da *Fridge app* er kodet uden *interfaces*, begrænser det ligeledes muligheden for modultests, da den høje kobling har medført at de fleste funktioner har måttet kastes direkte ud i integrationstests, idet de benytter sig af underliggende lag (*View* -> *BLL* -> *DAL*). Herudover oprettes der allerede i *BLL*'s constructor forbindelse til *DAL*, hvilket betyder at ingen tests vil kunne isoleres som modultests.

For mere om systemets tests, henvises derfor til afsnittet om **integrationstests, side 73**.

Web app

Design

Mappestrukturen (**Figur 48**) i web applikations projektet er generet af Visual Studio under oprettelsen af projektet. Den følger MVC mønstret og har derfor mapperne: Controllers, Views og Models. Models mappen er i dette projekt gjort overflødig da de klasser der burde ligge i denne mappe ligger i et class library ved navn: "SmartFridge_WebModels", som har taget udgangspunkt i modellaget fra den anden applikation.

App_Data mappen bruges ikke i dette projekt da al data der skal gemmes i denne applikation gemmes i en database.

App_Start indeholder klasser der henter scripts og bestemmer hvilken side der skal vises under opstart af applikationen. Disse er autogenerede og ikke ændret i.

Content mappen indeholder de .cs stylesheet, som hvert view bruger for at style siderne.

Images, fonts og Scripts mapperne indeholder henholdsvis de billeder, de scripts og de fonts, som bruges i applikationen.

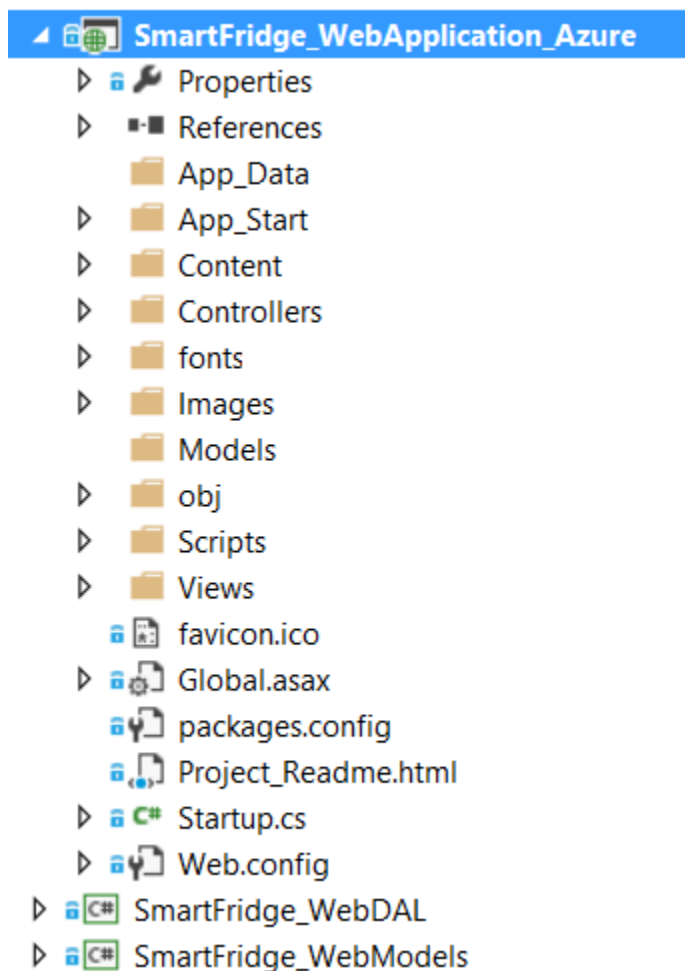
I Web.config (XML fil) indeholder information om hvilke assamblys applikationen skal hente, hvilke connectionsstrings den skal bruge og meget mere.

Global.asax (cs fil) kører de filer der ligger i App_Start under opstart af applikationen.

Hver controller har et View tilknyttet, hvor controlleren sørger for funktionalitet og Viewet sørger for at renderer html kode, som browseren kan læse og brugeren herved kan se i sin browser.

Dette projekt består af 4 controllers:

- HomeController
- ListViewController
- AddItemController
- EditItemController



Figur 48 WebApp mappestruktur. MVC pattern

Design af views

De individuelle views følger stort set samme design som deres modparter i Fridge-app, dog med et par forskelle. Da det er en web applikation, og den derfor vil køre i en browser, er luk knappen overflødig og derfor ikke taget med. En anden følge af at det kører i browser har været at uret fra Fridge-app har virket overflødigt da uanset, hvilken enhed den tilgås fra har let adgang til et ur på anden vis.

Udover disse to ting er "rammen" den samme som på Fridge-app og har altså stadig navigations knapperne. Da rammen er den samme vil denne ikke være med på de wireframes, der er i dette afsnit. Endnu en begrundelse for de mindre forskelle, der er på udseendet af de to applikationer, er at web applikationen, også skal kunne bruges på mindre enheder, hvor for mange ting i hvert view kunne besværliggøre brug.

Additem

Dette view svarer til "Tilføj varer" fra Fridge-app. Der er stort set ikke ændret på udseendet af dette view. Den eneste forskel er, at listen over nyligt tilføjede varer er placeret under selve tilføjelses menuen i stedet for ved siden af, som det ses ud fra det tilhørende wireframe på **Figur 49**.

Edititem

Det er ved edit item, at der er sket den største forandring. I Fridge-app var denne en del af det vindue, hvor man kunne se alle vare på en given liste, på web applikationen har den fået et vindue for sig selv. Grunden til dette var, at der ved brug af mindre enheder ville gå udover brugervenligheden, hvis der var for mange ting på et view. Som det kan ses på **Figur 50** Wireframe for EditItem, minder den rigtig meget om både boksen med vareinformation fra Fridge-apps List view, og om Add items view.

Index

Index svarer til hovedmenuen i Fridge-app, hertil har vi valgt at beholde det design, der var lavet der til.

Varetype

Antal

Volume Enhed

Sidste holbarhedsdato

Tilføj Tilføj & Afslut

Nyligt tilføjede varer

Vare information
Vare information
Vare information
Vare information
Vare information

Figur 49 Wireframe for Additem

Varetype

Antal

Volume Enhed

Sidste holbarhedsdato

Cancel Accept

Figur 50 Wireframe for EditItem

ListView

Som det fremgår af **Figur 51** har List viewet været igennem nogle ændringer. Det mest iøjefaldende er nok at boksen med vareinformation er væk. Dette skyldes at al information vedrørende en vare kunne vises i griddet, og at funktionaliteten vedrørende redigering af vareinformation blev flyttet ud i et view for sig. Udover dette er der ikke blevet ændret der store ved viewet udover at der i griddet til vareinformation er kommet en knap, der fører til EditItem viewet, til venstre for slet knappen.

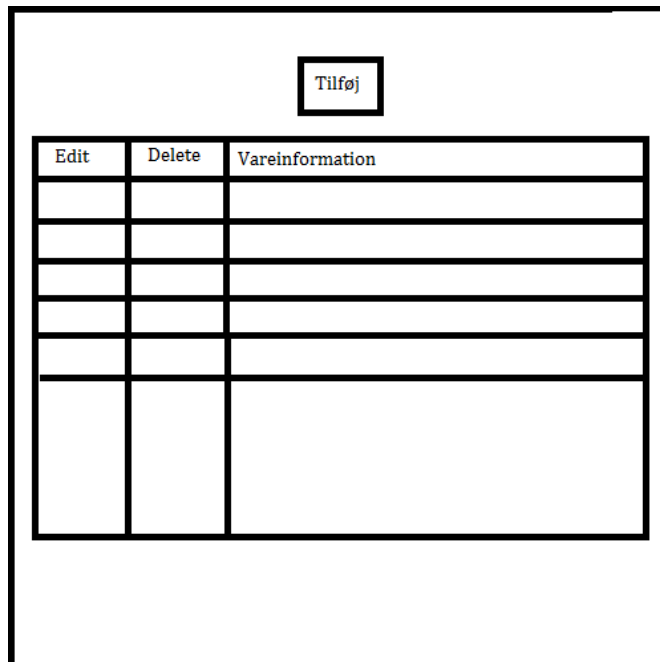
Implementering

Controllers

AddItemController

AddItem består af én constructor og 2 funktioner.

Constructoren indlæser items fra databasen vha. Unit Of Work, og tilføjer herefter alle items til ListGuiltems. model bliver sat til newGuiltems, som er de items der skal tilføjes i addNewItem-funktionen. Dette ses på **Kodestump 23**. Constructoren returnere et View, som sørger for at AddItemViewet bliver vist for brugeren.



Tilføj		
Edit	Delete	Vareinformation

Figur 51 Wireframe for List view

```
public ActionResult AddItem()
{
    var uow = Cache.DalFacade.GetUnitOfWork();
    ListGuiItemTypes.Add(new SelectListItem { Text = "Varetype", Value = "Varetype", Selected
    = true });
    foreach (var guiItemTypes in uow.ItemRepo.GetAll())
    {
        ListGuiItemTypes.Add(new SelectListItem { Text = guiItemTypes.ItemName
    });
    }
    model = newGuiItems;
    ViewBag.ListNewGuiItems = ListGuiItemTypes;

    uow.SaveChanges();
    Cache.DalFacade.DisposeUnitOfWork();
    return View(model);
}
```

Kodestump 23 AddItemController's constructor

Funktionen addNewItem tager imod de indtastede data og tilføjer dem til guiltemToAdd. Men først tjekkes der på om den holdbarhedsdato der medsendes er indtastet. Er denne ikke det, sættes det holdbarhedsdatoen til en default værdi. Ellers bliver den konverteret til den rigtige DateTime. Den skal konverteres da den modtager værdien som en string, men for at databasen kan forstå denne værdi skal den være af typen DateTime.

```
public ActionResult addNewItem(string Varetype, string Antal, string Volume, string
Enhed, string Holdbarhedsdato, string ItemImgClicked)
[...]  
    GUIItem guiItemToAdd = new GUIItem();  
    guiItemToAdd.ShelfLife = dblistItemDateTime; //AMOUNT READ FROM FIELD  
    guiItemToAdd.Amount = Convert.ToInt32(Antal); //Antal READ FROM FIELD  
    guiItemToAdd.Size = Convert.ToInt32(Volume); //Volume READ FROM FIELD  
    guiItemToAdd.Type = Varetype; //Varetype READ FROM FIELD  
    guiItemToAdd.Unit = Enhed; //unit READ FROM FIELD  
    foreach (var newGuiItem in newGuiItems)  
    {  
        if (newGuiItem.Type.Equals(guiItemToAdd.Type) &&  
            newGuiItem.Size.Equals(guiItemToAdd.Size) &&  
            newGuiItem.Unit.Equals(guiItemToAdd.Unit) &&  
            newGuiItem.ShelfLife.Equals(guiItemToAdd.ShelfLife))  
        {  
            newGuiItem.Amount += guiItemToAdd.Amount;  
            return null; //Viewet skal dog opdateres først  
        }  
    }  
    newGuiItems.Add(guiItemToAdd);  
[...]  
    model = newGuiItems;  
    ViewBag.ListNewGuiItems = ListGuiItemTypes;  
    return PartialView("~/Views/AddItem/AddItem.cshtml", model);
```

Kodestump 24 Udsnit af AddItemController's addNewItem funktion

På **Kodestump 24** ses hvordan der tjekkes om det item der skal tilføjes allerede eksistere. Hvis den allerede gør det, ændres der kun på dette items Amount i stedet for at tilføje et nyt. Findes item ikke i listen bliver det tilføjet. Når det nye item er tilføjet lægges det i den liste som skal vises (model) og viewet returneres. Den nye item er nu tilføjet og kan ses i tabellen.

I addNewItem funktionen tjekkes der også på parameteren 'ItemImgClicked'. Hvis denne er lig "Exit", betyder det at knappen 'Tilføj og Exit' er trykket på. Er dette tilfældet bliver funktionen Exit i AddItemController kaldt.

```
public ActionResult Exit()
{
    if (newGuiItems.Count == 0) {return null;}
    var uow = Cache.DalFacade.GetUnitOfWork();
    foreach (var newGuiItem in newGuiItems)
    {
        Item dbItem = uow.ItemRepo.Find(l => l.ItemName == newGuiItem.Type);
        if (dbItem == null)
        {
            dbItem = new Item(newGuiItem.Type){StdUnit = newGuiItem.Unit,
StdVolume = (int)newGuiItem.Size};
            uow.ItemRepo.Add(dbItem);
        }
        ListItem dbListItem = uow.ListItemRepo.Find(l =>
l.List.ListName == Cache.CurrentList.ListName &&
l.Item.ItemName == newGuiItem.Type &&
l.Unit == newGuiItem.Unit &&
l.Volume == newGuiItem.Size &&
l.ShelfLife == newGuiItem.ShelfLife);
        if (dbListItem != null)
            dbListItem.Amount += (int)newGuiItem.Amount;
        else if (dbListItem == null)
        {
            dbListItem = new ListItem();
            dbListItem.ShelfLife = newGuiItem.ShelfLife;
            dbListItem.Amount = (int)newGuiItem.Amount;
            dbListItem.Volume = (int)newGuiItem.Size;
            dbListItem.Unit = newGuiItem.Unit;
            dbListItem.ListId = Cache.CurrentList.ListId;
            dbListItem.ItemId = dbItem.ItemId;
            dbListItem.Item = dbItem;
            uow.ListItemRepo.Add(dbListItem);
        }
    }
    uow.SaveChanges();
    Cache.DalFacade.DisposeUnitOfWork();
    newGuiItems = new List<GUIItem>();
    model = newGuiItems;
    return RedirectToAction("ListView", "LisView");
}
```

Kodestump 25 Udsnit af AddItemController's Exit-funktion

Exit funktionen (på **Kodestump 25**) tjekker først og fremmest på om der er tilføjet nogen items. Er der ikke det returneres der null. Er dette ikke tilfældet skal der tjekkes på om hvert item der er tilføjet ligger i databasen. Der ledes først i item listen om det nye item eksistere. Herefter ledes der i ListItem for at se om det skulle lægge der. Findes de ikke bliver de tilføjet til databasen. Hvis det nye item ligger i ListItem,

skal det nuværende items Amount tælles op. Efter alle nye item er gennemgået gemmes ændringerne i databasen og listerne med items opdateres. Til sidst returneres ListView Viewet.

EditItemController

EditItem controlleren har 3 forskellige funktioner, den første af disse er EditItem funktionen, der til dels kan kaldes for en constructor for EditItem viewet. Her indlæses alt det data, der skal bruges i viewet, såsom varetyper fra cachén og de forskellige enheder, der kan bruges til at beskrive en varer. Udover dette ligges, den valgte vare i ViewData, og varetyperne og enhederne ligges i ViewBag. Det hele kan ses på **Kodestump 26**.

```
public ActionResult EditItem(GUIItem oldItem)
{
    _oldItem = oldItem;
    _types = new List<SelectListItem>();
    _types.Add(new SelectListItem { Text = "Varetype", Value = "Varetype",
                                    Selected = true });
    foreach (var item in Cache.DbItems)
    {
        _types.Add(new SelectListItem { Text = item.ItemName, Value=item.ItemName });
    }
    _units = new List<SelectListItem>(){new SelectListItem{Text = "l", Value = "l"},
                                       new SelectListItem{Text = "dl", Value = "dl"},
                                       new SelectListItem{Text = "ml", Value = "ml"},
                                       new SelectListItem{Text = "kg", Value = "kg"},
                                       new SelectListItem{Text = "g", Value = "g"},
                                       new SelectListItem{Text = "stk",
                                                           Value = "stk"}};
    ViewData.Add("oldItem", _oldItem);
    ViewBag.types = _types;
    ViewBag.units = _units;
    return View();
}
```

Kodestump 26 Viewets "constructor"

Den anden funktion, der dækker over den primære opgave for Controlleren er Updateltem. Denne funktion sørger for at de i viewet indtastede data bliver gemt ned i databasen, ved at opdatere enten det tilhørende ListItem, Item eller begge dele. Funktionen består af en serie af tjek som der kan ses et eksempel på, på **Kodestump 27**. tjekkene går alle sammen på at finde ud af hvad, der er blevet ændret og hvordan det skal håndteres. I tilfælde af en vare opdateres på en sådan måde, at det stemmer overens med allerede eksisterende vare på den liste, håndteres dette ved at tælle den originale vares antal op, og slette duplikatten.

```
if (listItem.Item.ItemId == _oldItem.ItemId)
{
    if ((listItem.Item.ItemName == _updatedGUIItem.Type))
    {
        listItem.Amount = Convert.ToInt32(_updatedGUIItem.Amount);
        listItem.Volume = Convert.ToInt32(_updatedGUIItem.Size);
        listItem.Unit = _updatedGUIItem.Unit;
        listItem.ShelfLife = _updatedGUIItem.ShelfLife;
        uow.ListItemRepo.Update(listItem);
        uow.SaveChanges();
        Cache.DalFacade.DisposeUnitOfWork();
        return RedirectToAction("ListView", "LisView");
    }
    [...]
}
```

Kodestump 27 Eksempel på Check i Updateltem funktionen

Den tredje funktion i kontrolleren er selectedUnit funktionen. Denne funktion sørger for at det er den valgte vares enhed, der vises i dropdownmenuen i stedet for det øverste element. Som det kan ses på **Kodestump 28**, gøres dette ved at finde enheden i listen over enheder, og sætte dens Selected property til true.

```
public static void selectedUnit(GUIItem guiItem)
{
    foreach (var unit in _units)
    {
        if (unit.Text == guiItem.Unit)
        {
            unit.Selected = true;
            break;
        }
    }
}
```

Kodestump 28 Den valgte items enhed findes, og sættes som den valgte med Selected propertyen

HomeController

HomeController controlleren er den første controller, der kommer i brug. Når siden tilgås kaldes dens Index metode, som returnere Index viewed, det er også i denne metode, hvor cachens data acces layer facade instantieres, cachen er beskrevet længere nede i dette dokument. Udover denne metode har den også SetCurrentList funktionen, der kaldes når brugeren har trykket på en af de tre mulige knapper i Index viewed. Funktionen der kan ses på **Kodestump 29**, gør brug af cachens DAL facade til at finde den liste i databasen, der svarer til den Bruger valgte, og sætte den i cachen, før den returnere det relevante view.

```
public ActionResult SetCurrentList(string listToEdit)
{
    List currentList = new List();
    var uow = Cache.DalFacade.GetUnitOfWork();
    currentList = uow.ListRepo.Find(l => l.ListName == listToEdit);
    Cache.DalFacade.DisposeUnitOfWork();
    Cache.CurrentList = currentList;
    return RedirectToAction("ListView", "LisView");
}
```

Kodestump 29 Her ses metoden, hvor den nuværende liste sættes i Cachen

LisViewController

LisViewController indeholder én constructor, 2 funktioner og en liste med de items der skal vises på ListView's view. Constructoren henter de items der ligger i Cache klassen ind og tjekker først på om denne liste er tom. På **Kodestump 30** tjekkes det både på CurrentListItems og DbItem om id'erne passe samme på hvert item. Hvis ListItem og Item's id matcher skal navnet fra Item og amount, volumen, unit og shelflife fra ListItem lægges sammen til ét nyt item og lægges over i TempData listen. Efter der er blevet tjekket på alle items i listerne bliver de lagt ind i model som herefter returneres sammen med viewet.

```
Cache.CurrentList = Cache.CurrentList;
List<GUIItem> tempData = new List<GUIItem>();
if(Cache.CurrentListItems.Any())
{
    foreach (var listItem in Cache.CurrentListItems)
    {
        foreach (var item in Cache.DbItems)
        {
            if(item.ItemId == listItem.ItemId)
            {
                GUIItem temp = new GUIItem(item.ItemName,
                (uint)listItem.Amount, (uint)listItem.Volume, listItem.Unit){ItemId = item.ItemId,
                ShelfLife = listItem.ShelfLife};
                tempData.Add(temp);
            }
        }
    }
    model = tempData;
    return View(model);
}
```

Kodestump 30 Udsnit af ListViewController's constructor.

Den første funktion i ListViewController er ToEditItem, hvis opgave er at få sendt det rigtige GUIItem over til EditItemController'en. Da det ikke var muligt at sende Id'et med fra viewet, så tjekker denne funktion på det medsendte GUIItem's attributter om de stemmer overens med et item på listen. Når dette bliver fundet bliver EditItem funktionen kaldt i EditItemControlleren, og det fundne item sendes med over.

Den anden funktion i ListViewController er DeleteSelectedItem. Ligesom ToEditItem funktionen får den et GUIItem med fra viewet og tjekker på om dette item eksistere i model listen. Når/hvis det medsendte item bliver fundet konverteres IEnumerable model om til en liste så den kan rettes i. Herefter slettes itemet fra listen og den nye liste, uden det slettede item, lægges i model igen. Herefter skal itemet slettes fra databasen, hvilket gøres ved at finde og slette den i Cache'ns CurrentListItems. Til sidst gemmes ændringerne, cache'ns adgang til databasen bliver disposed og constructoren, der returnere ListView viewet, bliver kaldt.

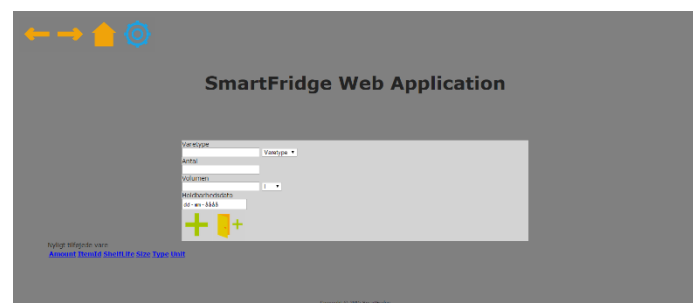
```
[...]
foreach (var item in model)
{
    if (item.Type == itemToDelete.Type && item.Amount == itemToDelete.Amount
    && item.Size == itemToDelete.Size && item.Unit == itemToDelete.Unit)
    {
        List<GUIItem> tempList = model.ToList();
        tempList.Remove(itemToDelete);
        model = tempList;
        foreach (var dblistitem in Cache.CurrentListItems)
        {
            if (dblistitem.ItemId == item.ItemId)
            {
                uow.ListItemRepo.Delete(uow.ListItemRepo.Find(l =>
1.ItemId == dblistitem.ItemId &&
1.ListId == dblistitem.ListId &&
1.Unit == dblistitem.Unit &&
1.Amount == dblistitem.Amount &&
1.ShelfLife == dblistitem.ShelfLife &&
1.Volume == dblistitem.Volume));
            }
        }
    }
}
uow.SaveChanges();
Cache.DalFacade.DisposeUnitOfWork();
return RedirectToAction("ListView", "LisView");
[...]
```

Kodestump 31 Udsnit af DeleteSelectedItem funktionen i LisViewController

Views

AddItem

Ligesom i SmartFridge-applikationen har AddItem fået sit eget view. På Figur 52 AddItem View ses en lysere grå indramning der indeholder de fire inputs: Varetype, Antal, Volumen og Holdbarhedsdato. Ydermere indeholder den 'Tilføj' knappen symboliseret af et plus og 'Tilføj og Exit' knappen symboliseret af en dør og et plus. Under den røde indramning er tabellen med de items der er tilføjet uden at afslutte.



Figur 52 AddItem View

Viewet indeholder en IEnumerable med de Guiltems der er tilføjet uden at afslutte. Ligesom de andre views bruger den samme layout (_Layout.cshtml) som wrapper til viewet. Disse ting ses på **Kodestump 32**.

```
@model IEnumerable<SmartFridge_WebModels.GUIItem>
@{
    ViewBag.Title = "AddItem";
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

Kodestump 32 Udsnit af additem.cshtml. Header

Alle inputs er inde i div klassen "NewItem" som ses på **Kodestump 33**. For alle inputs gælder det at det skal være inde i @using (Html.BeginForm()){. Dette er gjort for at fortælle programmet hvilken information den skal sende med til addItem funktionen. Varetypen har to muligheder for at få et input. Enten kan brugeren vælge at skrive det ind i <Input> objektet eller vælge det på den @html.DropDownList der er ved siden af inputtet.

```
<div class="NewItem" style="background-color: red; margin: 0 22em 0 22em">
    @using (Html.BeginForm("addItem", "AddItem"))
    {
        <label>Varetype</label>
        <br />
        <input type="text" name="Varetype" id="VaretypeText" required/>

        @Html.DropDownList("ListNewGuiItems",
(IEnumerable<SelectListItem>)ViewBag.ListNewGuiItems, new
        {
            @id = "VaretypeDropDown"
        })
        <br />

        <label>Antal</label>
        <br />
        <input type="number" name="Antal" min="0" required />
        <br />
    [...]
        <input type="image" src="~/Images/Add.png" alt="Submit" class="AddImg"
id="addItemImgClicked" name="ItemImgClicked" value="Add" />
        <input type="image" src="~/Images/AddAndExit.png" alt="Submit" class="AddImg"
name="ItemImgClicked" value="Exit" />
    }
```

Kodestump 33 Udsnit af AddItem.cshtml. Inputs

EditItem

Udover at det har fået sit eget view minder måden EditItem ser ud på rigtig meget om rediger vare fra Fridge-app. Varens information indlæses i tekstbokse, som kan redigeres i, og Bruger har mulighed for at vælge enten at gemme eller annullere de ændringer der er foretaget. Viewed, der kan ses på **Figur 53**, er ikke blevet helt som designet, hovedsagligt er det fluebenet og krydset, der ikke er kommet til at sidde som på designet. Dette skyldes bøvl med at få to forskellige Html forms til at være på samme linje. Udover dette skulle det egentlig bare have været centreret.



Figur 53 EditItem view

På **Kodestump 34** ses det allerførste, der sker i viewed. Den vare Bruger valgte i List viewed indlæses fra EditItem viewets ViewData, og der kaldes en funktion, der finder ud af, hvilken enhed, der skal være valgt i enheds dropdown menuen. Alle inputfelterne i viewed er lavet med Html helpers, på **Kodestump 35** kan der ses nogle eksempler på disse. Som det kan ses er der hovedsagligt brugt TextBox helperen, men i eksemplerne fra **Kodestump 35** kan det ses at der er manipuleret lidt med nogle af dem. Det er muligt at ændre på typen af TextBox helperen så den f.eks. kun tager imod tal eller datoer. Overholdes input typen ikke, gives der besked om dette, og man kan ikke gemme ændringerne før de alle sammen er overholdt. Alle Html helperne med vareinformationer i er i en Html form, der sender informationen videre til UpdateItem funktionen i EditItem controlleren.

```

@{
    ViewBag.Title = "EditItem";

    Layout = "~/Views/Shared/_Layout.cshtml";

    GUIItem dataSource = ViewData["olditem"] as GUIItem;
    EditItemController.selectedUnit(dataSource);
}
  
```

***Kodestump 34** Her ses, hvordan den valgte items data indlæses til brug i viewet*

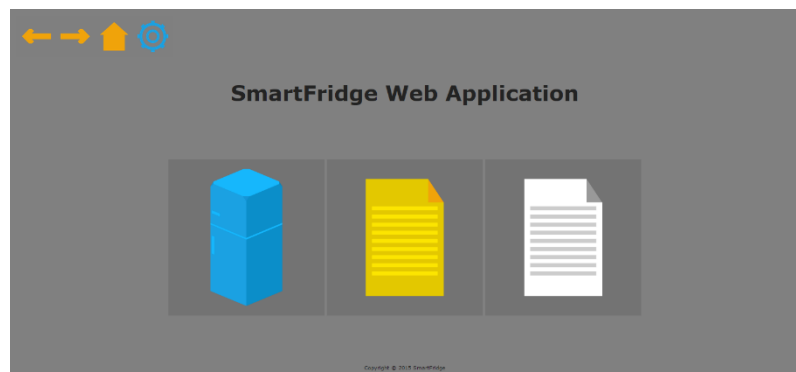
```

@Html.TextBox("Type", dataSource.Type, new
{
    @id = "VaretypeText"
})
@Html.DropDownList("types", (List<SelectListItem>)ViewBag.types , new
{
    @id = "VaretypeDropDown"
})
[...]
@Html.TextBox("Amount", dataSource.Amount, new { @type = "number", @min = 0 })
[...]
@Html.TextBox("Shelflife", dataSource.ShelfLife.ToString("dd-mm-yyyy"), new{@type =
"date", @autocomplete = "on"})
  
```

Kodestump 35 Eksempler på forskellige typer af Html helpers.

Index

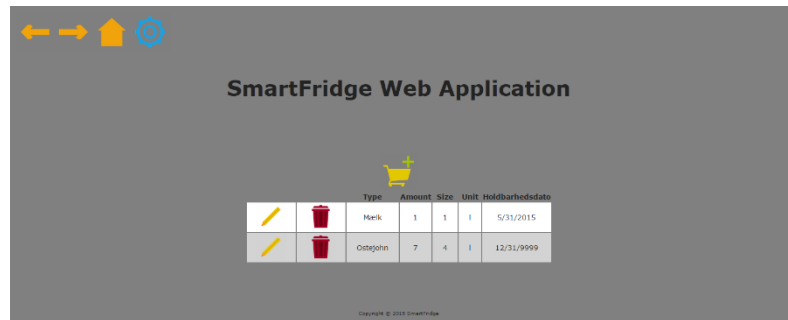
Index viewed der svarer til hovedmenuen fra Fridge-app er det simpleste af alle web applikationens views og består kun af tre knapper. Som det ses på **Figur 54**, ligner de to også hinanden rigtigt meget.



Figur 54 Index view web applikationens svar på en hovedmenu

ListView

ListView'et, der kan ses på **Figur 55**, indeholder en 'Tilføj Item'-knap til at navigere brugeren hen til AddItemView. Denne knap er symboliseret som en indkøbsvogn og et plus. Ydermere indeholder ListView en tabel med de items der er på den pågældende liste. Hvert item har to knapper ud fra sig. Et 'EditItem'-knap symboliseret af en blyant og en 'DeletItem'-knap symboliseret af en skraldespand. Disse knapper bliver tilføjet når griddet laves. På **Kodestump 36** ses det hvordan knappen er tilføjet til griddet. Cols, som griddet hedder, bliver lavet hver gang viewet bliver renderet. For hvert item i model listen bliver der lavet syv nye columns. De første to columns er redigér og slet knapperne. Herefter bliver hver attribut tilføjet, som det ses på **Kodestump 36** hvor Type bliver tilføjet.



Figur 55 ListView's View

Efter cols er blevet oprettet, sørger @grid.GetHtml for at renderere og style det på viewet.

```
[...]
IEnumerable<WebGridColumn> cols =
    new List<WebGridColumn> {
        new WebGridColumn{
            Format =
                @<text>
<a href="@Url.Action("ToEditItem", "ListView", new GUIItem(item.Type, item.Amount,
item.Size, item.Unit))"></a>
</text>
        }
        new WebGridColumn{
            Header="Type",
            Format = @<text>
                <span class="display">
                    <label id="lblType">@item.Type</label>
                </span>
            </text>
        },
    },
[...]
```

```
<div align="center">
    @grid.GetHtml(columns:cols,
        rowStyle:"oddRow", alternatingRowStyle:"evenRow")
</div>
```

Kodestump 36 Udsnit af ListView.cshtml

Models

Modellerne brugt i web-appen, er de samme klasser, der blev brugt i Fridge-app med undtagelse af GUIItem. Der er en model for hver type database entitet, der gøres brug af ved al database tilgang, og GUIItem klassen, som er den lokale repræsentation af en vare og derfor bruges i forbindelse med samtlige views. Modellerne er implementeret i et klassebibliotek for sig, da data access layered, der også ligger i et klasse bibliotek for sig, har behov for at kende til disse. Ellers er praksissen med at ligge modeller i et klasse bibliotek normalt kun noget, der gøres brug af i projekter med mange modeller.

Cache

Til web-appen er der blevet implementeret en cache klasse, der indeholder de relevante ting for de forskellige views. Alle cache klassens attributter er static. Især med facaden til data acces layered er det vigtigt at det er static, da flere instanser af denne kunne resultere i fejl, hvis flere instanser samtidigt forsøger at tilgå databasen. Som det fremgår af **Kodestump 37** CurrentList propertyens set metode, sørger for også at hente alle ListItems, der figurere på den nuværende liste og alle Items ned. Her hentes også samtlige items fra databasen ned. Grunden til at alle items hentes ned er at deres navne bruges som info i dropdown menuer i både EditItem og AddItem viewene, hvor det altså er relevant at få alle varettyper med, og ikke blot dem, der er tilknyttet ListItems på den nuværende liste

```
public static List CurrentList { get { return _currentList; }  
    set  
    {  
        _currentList = value;  
        var uow = DalFacade.GetUnitOfWork();  
        CurrentListItems = new List<ListItem>();  
        var tempList = uow.ListItemRepo.GetAll().ToList();  
        if (tempList.Any())  
        {  
            foreach (var Listitem in tempList)  
            {  
                if (Listitem.ListId == _currentList.ListId)  
                {  
                    CurrentListItems.Add(Listitem);  
                }  
            }  
        }  
        DbItems = uow.ItemRepo.GetAll().ToList();  
        DalFacade.DisposeUnitOfWork();  
    }  
}
```

Kodestump 37 CurrentList propertyen, hvis set metode også henter alle relevante ListItems på listen ned

Test

Som nævnt i rapporten er en af grundende til valget af MVC mønstret, at det er nemt at teste. Dette har vist sig ikke at være tilfældet, men det er alligevel

lykkedes at opnå en coverage på 100% på controllerne og Chache klassen i web applikationen. **Figur 56.**

Symbol	Coverage (%) ▲	Uncovered/Total Stmts.
▶ HomeControllerUnitTests	100%	0/14
▶ CacheUnitTests	100%	0/23
▶ LisViewControllerUnitTest	100%	0/31
▶ AddItemControllerUnitTest	100%	0/45
▶ EditItemControllerUnitTests	100%	0/104

Figur 56 Coverage rapport for controllers og Chache

AdditemController test

Til at teste AdditemController klassen er det oprettet 7 tests.

Setup funktionen sørger for at der bliver oprettet et nye objekter inden hver funktion bliver kørt. Nogle af disse objekter er; item, dalfacade, listitem og flere, som ses på **Kodestump 38.**

```
[SetUp]
public void Setup()
{
    var dalfacade = Substitute.For<ISmartFridgeDALFacade>();
    Cache.DalFacade = dalfacade;
    listitem = new ListItem(1, 1, "1", default(DateTime), currentList, item) {
ItemId = item.ItemId };
    dalfacade.GetUnitOfWork().ListRepo.GetAll().Returns(new List<List>()
{currentList});
    dalfacade.GetUnitOfWork().ItemRepo.GetAll().Returns(new List<Item> {item});
    dalfacade.GetUnitOfWork().ListItemRepo.GetAll().Returns(new List<ListItem>()
{listitem});
}
```

Kodestump 38 Setup til additem

Den første test vist på **Kodestump 39** tester på AddItem funktionen. Der oprettes et AddItemController objekt ved navn uut. Den kalder funktionen AddItem og konvertere det returnerede ActionResult, fra funktionen, til et ViewResult, som ligges ned i en variabel ved navn actResult. Herefter assertes der på om det returnerede actResult. Problemet ved denne test er at der ligger funktionalitet inde i funktionen, som ikke kan assertes på, da disse attributter er private. I denne test tjekkes der på om navnet på viewet er en tom string, hvilket den altid vil være når en funktion returnere 'return View(model);' kan den kun returnere navnet på viewet eller en tom string.

```
[Test]
public void AddItem_TestLoadingCorrectModel_ReturnCorrectModel()
{
    var actResult = uut.AddItem() as ViewResult;
    Assert.That(actResult.ViewName, Is.EqualTo(""));
}
```

Kodestump 39 AddItem test

De næste 5 funktioner tester addNewItem og disse tests er designet til at nå al koden i funktionen.

Funktionen, som ses på **Kodestump 40**, tilføjer et item og ender med at kalde en anden funktion i ListView. Derfor assertes der på om den rigtige funktion er blevet kaldet. Denne test giver derfor et mere sigende resultat sammenlignet med testen på **Kodestump 39**.

```
[Test]
public void addNewItem_AddItemAndExit_LisViewReturned()
{
    RedirectToRouteResult result = uut.addNewItem("TestItem", "1", "1", "1", "", "Exit") as RedirectToRouteResult;

    Assert.That(result.RouteValues["action"], Is.EqualTo("ListView"));
}
```

Kodestump 40 AddNewItem Test

HomeController test

HomeController var den nemmeste at teste, da den kun består af to metoder. Den første metode, Index, opretter Cache og returnere det View, der skal ses på forsiden. Der testes på om dette view returneres.

Den anden funktion skal kalde ListView funktionen i LisView Controlleren og sende en string med den rigtige liste med over. På **Kodestump 41** ses der hvordan der tjekkes på at den rigtige funktion bliver kaldt i LisView.

```
[Test]
public void TestSetCurrentListReturnsTheRigtView()
{
    string currentlist = "TestList";
    var actResult = uow.SetCurrentList(currentlist) as RedirectToRouteResult;

    Assert.That(actResult.RouteValues["action"], Is.EqualTo("ListView"));
}
```

Kodestump 41 HomeController test

LisViewController test

Ligesom i AddItemController og HomeController, blev der tjekket på den funktion der skal returnere viewet, returnere det rigtige view. Ydermere er der tre tests som alle har til formål at teste på om den rigtige metode bliver kaldt når henholdsvis ItemToEdit og DeleteSelectedItem bliver kaldt. De to tests, der tjekker på ItemToEdit metoden skal kalde EditItem metoden i EditItem controlleren. Og den test der tjekker på om DeleteSelectedItem bliver kaldt skal returnere samme view, men med en ændret model. Problemet disse tests er de samme som i AddItemController. Der bliver kun testet på slutresultatet og ikke den manipulation af det medsendte item.

EditItemController test

De tests der er lavet til til EditItemController har samme problem som beskrevet i AddItemController og LisviewController. Den største forskel, som set på **Kodestump 42** er at for at få en højere coverage % skal nogle ting initialiseres.

```
[Test]
public void UpdateItem_Item_EverythingButTypeAndNoShelfLifeUpdated()
{
    uut.EditItem(new GUIItem("test", 1, 1, "1"));
    FormCollection collection = new FormCollection();
    collection.Add("Type", "test");
    collection.Add("Amount", "3");
    collection.Add("Volume", "2");
    collection.Add("units", "kg");
    collection.Add("date", "");
    var actResult = uut.UpdateItem(collection) as RedirectToRouteResult;

    Assert.That(actResult.RouteValues["action"], Is.EqualTo("ListView"));
}
```

Kodestump 42 Test til EditItemController

Cache test

CacheUnitTests klassen indeholder tre tests og en setup. Disse tre test har alle til formål at tjekke på om den data der ligger i Cache. På **Kodestump 43** ses det at Chacen's currentList bliver sat til currentList, som indeholder data hentet vha. dalfacade i Setup funktionen. Herefter tjekkes der på om det rigtige listitem ligger i denne liste. Listitem initialiseres i Setup funktionen.

```
[Test]
public void CurrentListSet_ContainsListItem()
{
    Cache.CurrentList = currentList;
    Assert.That(Cache.CurrentListItems.Contains(listitem));
}
```

Kodestump 43 Cache test

De andre er opbygget på samme simple måde, men der assertes på hvad der ligger i de forskellige lister i klassen.

Integrationstests

På **Figur 60** ses klassen der indeholder tests af Business Logic Layeret.

BusinessLogicLayerTest (18 tests)	Success
✓ AddItemsToTable_1NewItem_SameItemAddedToKøleskab	Success
✓ AddItemsToTable_2NewItem_SameItemAddedToKøleskab	Success
✓ AddItemsToTable_AddTwoEqualItems_ItemsAreAddedTogetherAsOneInKøleskab	Success
✓ AddItemsToTable_AddTwoSimilarItems_ItemsAreStillSeparateInKøleskab	Success
✓ ChangeItem_ChangeItemFromMilkToCola_ItemChanged	Success
✓ ChangeItem_ChangeItemFromMilkToCola_ItemChangedAndAddedToExistingCola	Success
✓ CheckShelfLife_2ExpiredItems_2Notifications	Success
✓ CheckShelfLife_NoExpiredItems_NoNotifications	Success
✓ Compare_TwoNonSimilarItems_ReturnFalse	Success
✓ Compare_TwoSimilarItems_ReturnTrue	Success
✓ CreateNewItem_5ParametersInserted_Same5ParametersInNewItem	Success
✓ GetList_RequestFridge_FridgeReturned	Success
✓ GetList_RequestUnknown_NullReturned	Success
✓ STDToShopListControl_MilkOnSTDListAndInFridge_NothingChanged	Success
✓ STDToShopListControl_MilkOnSTDListButNotInFridge_MilkInShoppingList	Success
✓ STDToShopListControl_UnrecognizableListInserted_ExceptionThrown	Success
✓ Types_3ItemsInItemsList_3ItemsReturned	Success
✓ WatchItems_Add2ItemsToFridge_2ItemsReturned	Success

Figur 57 Test af BusinessLogicLayer

De tests der er at finde i BusinessLogicLayerTest er som udgangspunkt unittests, men bl.a. pga. manglende mulighed for at substituere 'Listen' ud, vil flere af disse tests agere som integrationstests. Stort set alle funktionaliteter bliver testet, på nær private funktioner. Blandt andet er der ikke skrevet en test for den private funktion 'CreateList', der bliver brugt til at oprette en liste hvis den ikke eksisterer i forvejen.

Testene er integrationstests, da der allerede i BLL'ens constructor bliver oprettet forbindelse til databasen.

Fra henholdsvis testfilen og BLL-filen:

<pre> public BLL uut; [SetUp] public void SetUp() { uut = new BLL(); uut.CurrentList = "Køleskab"; var items = uut.WatchItems; } </pre>	<pre> public BLL() { Context = new AdoNetContext(_connectionFactory); _itemRepository = new ItemRepository(Context); _listRepository = new ListRepository(Context); _listItemRepository = new ListItemRepository(Context); } </pre>
--	--

Kodestump 44 BLL test SetUp og BLL constructoren

Dette medfører at det er nødvendigt at rydde op efter udførsel af test. Her er et eksempel på en test:

```
[Test]
public void AddItemsToTable_1NewItem_SameItemAddedToKøleskab()
{
    var items = new ObservableCollection<GUIItem>();
    var rnd = new Random();
    string type = rnd.Next(int.MinValue, int.MaxValue).ToString();
    items.Add(new GUIItem(type, 5, 1, "Unit") { ShelfLife = DateTime.Now });
    string listName = "Køleskab";
    uut.AddItemsToTable(listName, items);

    //Find the list...
    GUIItemList list = null;

    foreach (var guiItemList in uut.Lists)
    {
        if (guiItemList.Name.Equals("Køleskab"))
            list = guiItemList;
    }
    int EqualItems = 0;
    foreach (var guiItem1 in items)
    {
        foreach (var guiItem2 in list.ItemList)
        {
            if (uut.Compare(guiItem1, guiItem2))
                EqualItems++;
        }
    }

    Assert.AreEqual(1, EqualItems);

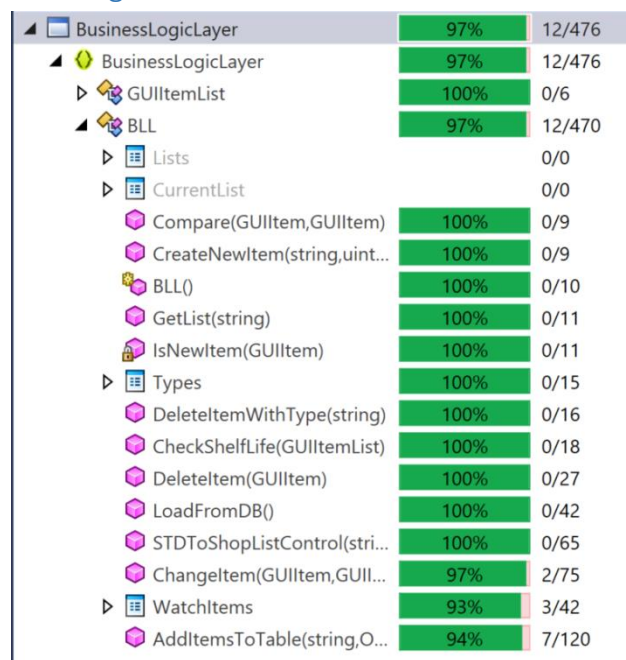
    foreach (var guiItem in items)
    {
        uut.DeleteItem(guiItem);
        uut.DeleteItemWithType(guiItem.Type);
    }
}
```

Kodestump 45 AddItemsToTable test

I testen oprettes en liste af GUIItems, hvor der tilføjes en vare til listen. Derefter kaldes *AddItemsToTable*, hvor den enkelte vare bliver tilføjet til *Køleskab*. Derefter findes *Køleskab*, og der tjekkes hvorvidt varen reelt er tilføjet til *Køleskab* eller ej. Dette gøres ved at udnytte *Compare*-funktionen, som er testet tidligere. Der bruges *EqualItems* for at sikre at den samme vare ikke er tilføjet flere gange end én. Efter assertes der på om testen gik godt og der ryddes op ved at slette det *ListItem* og *Item* som varen var knyttet til.

Generelt er testene opbygget sådan; Konstruktion af varer, test af metode, oprydning.

Coverage



Figur 58 Coverage af BusinessLogicLayer

På **Figur 61** ses et screenshot af Coverage-rapporten. Her er ikke helt opnået fuld coverage. WatchItems, Changeltem og AddItemsToTable har alle muligheden for at kalde 'CreateList'. Da BLL-testen i høj grad er en integrationstest, medfører det oprydning af de tests man foretager sig. Der er ikke skrevet en funktion for 'DeleteList', og derved kan Listen ikke opryddes ved test af disse specielle tilfælde. Dermed er de sidste procenter i de 3 nederste metoder i Figur 61 ikke testet, og giver BLL'en en coverage på 97%.

Static analysis

Her ses resultaterne for en statisk analyse foretaget af Visual Studio:

Code Metrics Results						
Filter: None		Min: 	Max: 	 		
Hierarchy ▲	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code	
⚠ One or more projects were skipped. C#						
▶ C# BusinessLogicLayer (Debug)	■ 74	137	1	24	257	
▶ C# DataAccessLayer (Debug)	■ 79	84	2	49	284	
▶ C# InterfacesAndDTO (Debug)	■ 88	26	1	10	37	
▶ C# ResourcesDictionary (Debug)	■ 100	0	0	0	0	
▶ C# SmartFridge.Tests.Unit (Debug)	■ 81	67	1	37	138	
▶ C# SmartFridgeApplication (Debug)	■ 84	73	9	72	186	
▶ C# SmartFridgeModellingLib (Debug)	■ 92	35	1	3	50	
▶ C# UserControlLibrary (Debug)	■ 78	90	9	53	314	

Error List

Output

Code Metrics Results

Figur 59 Static analysis, code metrics

De følgende vurderinger af resultaterne tager udgangspunkt i MSDN's retningslinjer (Microsoft, u.d.)

I4 PRJ4, Gruppe 5

IKT

Maintainability index

Giver indikation af hvor let det er at vedligeholde koden.

Da en værdi mellem 20 og 100 indikerer god maintainability, har samtlige projekter i vores solution en god maintainability, med et upper peak på 84, da SmartFrdigeModellingLib ikke er applikationskode.

Cyclomatic Complexity

Giver indikation af hvor kompleks kodetraverseringen er, altså om der er mange cyklusser.

Generelt denne værdi ikke enorm høj i solutionen, men BusinessLogicLayer har en semi-høj CC i forhold til DataAccessLayer. Dette kan skyldes et højt antal af hjælpefunktioner i BLL.

Depth of Inheritance

Giver indikation af arveheraki-kompleksiteten. Generelt lav da der ikke bliver arbejdet meget med arv i vores solution, da det ikke har været særlig relevant.

Class Coupling

Giver indikation af koblingen mellem klasser. Her er resultatet højt ved SmartFridgeApplication, da det er hovedprogrammet i vores solution. Med andre ord er det den store klasse der afhænger af en del små klasser. Her kunne man have valgt ligge et interface på flere klasser.

Accepttests

Test setup

Til udførelsen af accepttests til Fridge app er en computer med følgende specifikationer påkrævet:

- Minimum opløsning: 1920x1080.
- Tastatur og mus.
- Adgang til internettet.
- Adgang til SQL databasen smartfridgedb på en af følgende måder:
 - Websiden: <https://manage.windowsazure.com>.
 - Microsoft Visual Studio.
 - SQL Server Management Studio.

Ydermere skal computeren have kørt SmartFridge-SSDT projektet for at få den lokale database, og SmartFridge solution kørende på computeren inden accepttestene påbegyndes.

Til udførelsen af accepttests til Web app er en computer med følgende specifikationer påkrævet:

- Minimum opløsning: 1920x1080.
- Tastatur og mus.
- Adgang til internettet.
- Adgang til SQL databasen smartfridgedb på en af følgende måder:

I4 PRJ4, Gruppe 5

IKT

- Websiden: <https://manage.windowsazure.com>.
- Microsoft Visual Studio.
- SQL Server Management Studio.

Webapplikationen startes i Google Chrome ved at gå ind på siden:

<http://smartfridgeweb.azurewebsites.net/> og herefter kan accepttests påbegyndes.

Funktionelle krav

Det er en fælles prækondition for alle tests at systemet er tændt. Ydermere er det gældende for alle tests, der ændre på data i databasen, at databasen også skal inspiceres for disse ændringer. Dette vedrører test af UC2, UC3 og UC4

UC1: <i>Se varer</i>	Test	Forventet resultat	Resultat	Godkendt/ kommentar
Punkt 1: Bruger trykker på knappen " Køleskab".	Bruger trykker på knappen " Køleskab", og tester visuelt om den korrekte liste fremkommer.	En liste over nuværende varer i køleskabet, samt mængden af disse, vises på skærmen. Disse sammenlignes med dem i databasen.	Som forventet	Godkendt

UC1: <i>Alternativt flow</i>	Test	Forventet resultat	Resultat	Godkendt/ kommentar
Punkt 1.a: Bruger trykker på "Indkøbsliste".	Bruger trykker på knappen "Indkøbsliste", og tester visuelt om den korrekte liste fremkommer.	En liste over nuværende varer på indkøbslisten, samt mængden af disse, vises på skærmen. Disse sammenlignes med dem i databasen.	Som forventet	Godkendt
Punkt 1.b: Bruger trykker på "Standard- beholdning".	Bruger trykker på knappen " Standard- beholdning", og tester visuelt om den korrekte liste fremkommer.	En liste over nuværende standard-varer, samt mængden af disse, vises på skærmen. Disse sammenlignes med dem i databasen.	Som forventet	Godkendt

I4 PRJ4, Gruppe 5

IKT

Prækondition: UC1. Cola skal tidligere være tilføjet til systemet.

UC2: <i>Tilføj Vare</i>	Test	Forventet resultat	Resultat	Godkendt/ kommentar
Punkt 1: Bruger trykker på "Tilføj".	Bruger trykker på knappen "Tilføj. Og det testes visuelt at GUI konteksten skifter.	GUI'en viser nu "tilføj vare" konteksten.	Som forventet	Godkendt
Punkt 2: Varetype vælges	Cola fra dropdown- menuen vælges.	Den valgte varetype er nu sat til Cola.	Som forventet	Godkendt
Punkt 3: Antal vælges	Antallet 1 vælges.	Det valgte antal er nu sat til 1.	Som forventet	Godkendt
Punkt 4: Volumen/Vægt vælges	En volumen/vægt på 500 indtastes.	Den valgte Volumen/Vægt er nu sat til 500.	Som forventet	Godkendt
Punkt 5: Enhed vælges	Enheden ml vælges fra en dropdown menu.	Den valgte enhed er nu sat til ml.	Som forventet	Godkendt
Punkt 6: Bruger trykker på "Tilføj og afslut" og varen tilføjes til den valgte liste fra UC1	Bruger trykker på "Tilføj og afslut".	Varen er tilføjet til den valgte liste fra UC1. Værdierne sat fra punkt 2 til 5 tjekkes om de passer.	Som forventet	Godkendt
Punkt 7: Bruger returneres til den valgte liste fra UC1	Der testes visuelt at konteksten er skiftet.	Listen fra UC1 vises.	Som forventet	Godkendt

I4 PRJ4, Gruppe 5

IKT

Prækondition: UC1.

UC2: <i>Alternativt flow</i>	Test	Forventet resultat	Resultat	Godkendt/ kommentar
Punkt 5.a: Bruger trykker på "Tilføj".	Bruger trykker på knappen "Tilføj", og tester visuelt, om den indtastede vare fremkommer på listen over tilføjede varer.	Den indtastede vare figurerer på listen over tilføjede varer.	Som forventet	Godkendt

Prækondition: UC1.

UC2: <i>Undtagelser</i>	Test	Forventet resultat	Resultat	Godkendt/ kommentar
Punkt 3.a: Varetypen eksisterer ikke	Indtast Faxe Kondi manuelt i Varetype feltet	Varens varetype er sat til den manuelt indtastede	Som forventet	Godkendt

Prækondition: UC1.

UC3: <i>Rediger Vare</i>	Test	Forventet resultat	Resultat	Godkendt/ kommentar
Punkt 1: Bruger trykker på "Rediger".	Bruger trykker på knappen "Rediger" ud for Cola, og det testes visuelt at varens information kan rettes i vare infoboksen.	I infoboksen står alle varens informationer nu.	Som forventet	Godkendt
Punkt 2: Bruger retter vareinformation.	Alle alternative flows gennemgås	I infoboksen vises nu de nye informationer	Som forventet	Godkendt
Punkt 3: Bruger trykker på "Gem" og ændringerne gemmes i varen.	Bruger trykker på "Gem".	Varens informationer er ændret på listen. Og inputfelterne i infoboksen skjules	Som forventet	Godkendt

I4 PRJ4, Gruppe 5

IKT

Prækondition: UC1.

UC3: <i>Alternativt flow</i>	Test	Forventet resultat	Resultat	Godkendt/ kommentar
Punkt 2.a: Bruger ændrer varetype.	Varetypen ændres til Fanta, ved indtastning, og Bruger trykker på gem. Det testes visuelt i listen at varens type er ændret.	Varens type er ændret til Fanta	Som forventet	Godkendt
Punkt 2.b: Bruger ændrer antal.	Antal ændres til 3, ved indtastning, og Bruger trykker på gem. Det testes visuelt i listen at varens antal er ændret.	Varens antal er ændret til 3	Som forventet	Godkendt
Punkt 2.c: Bruger ændrer volume/vægt.	Volume/vægt ændres til 1, ved indtastning, og Bruger trykker på gem. Det testes visuelt i listen at varens volume/vægt er ændret.	Varens volume/vægt er ændret til 1.	Som forventet	Godkendt
Punkt 2.d: Bruger ændrer enhed.	Enheden ændres til L fra dropdown menu, og Bruger trykker på gem. Det testes visuelt i listen at varens enhed er ændret.	Varens enhed er ændret.	Som forventet	Godkendt

I4 PRJ4, Gruppe 5

IKT

Punkt 2.e: Bruger ændrer intet.	Intet ændres, og Bruger trykker gem. Det testes visuelt at varens informationer er som før.	Varen er som før.	Som forventet	Godkendt
Punkt 3.a: Bruger trykker på "Annuller".	Bruger trykker på "Annuller".	Varens informationer er som før, og input felterne i infoboksen skjules	Som forventet	Godkendt

Prækondition: UC1.

UC4: <i>Fjern Vare</i>	Test	Forventet resultat	Resultat	Godkendt/ kommentar
Punkt 1: Bruger trykker på "Fjern" ud for en eksisterende vare.	Bruger trykker på knappen "Fjern" ud fra Fanta, og det inspiceres visuelt at varen er blevet fjernet fra den i UC1 valgte liste.	Varen Fanta er slettet fra den i UC1 valgte liste.	Som forventet	Godkendt

UC5: <i>Synkroniser til ekstern database</i>	Test	Forventet resultat	Resultat	Godkendt/ kommentar
Punkt 1: Bruger trykker på "Synkroniser", og en øjeblikkelig synkronisering påbegyndes.	Bruger trykker på knappen "Synkroniser", og det testes visuelt at både den eksterne- og den lokale database indeholder det samme.	Indholdet af de to databaser er ens.	Som forventet	Godkendt

I4 PRJ4, Gruppe 5

IKT

Prækondition: UC1, og at der i UC2 er tilføjet en vare der er for gammel

UC6: <i>Notifikation om holdbarhedsdato</i>	Test	Forventet resultat	Resultat	Godkendt/kommentar
Punkt 1: Bruger trykker på notifikationslist-knappen og en pop-up vises med notifikationerne	Tryk på notifikationslist-knappen i nederste højre hjørne. Denne knap har værdien 1.	En liste af notifikationer for alle forældede varer vises. Værdien på knappen er 1.	Som forventet	Godkendt
Punkt 2: Bruger trykker "Slet", og notifikationen slettes	Tryk på slet, ud for en af notifikationerne. Værdien på knappen bliver 0.	Notifikationen er fjernet. Værdien på knappen er 0.	Som forventet	Godkendt

Ikke-funktionelle krav

1: <i>System</i>	Test	Forventet resultat	Resultat	Godkendt/kommentar
Punkt 1.1: Kernefunktionaliteterne skal kunne udføres i både Web app og Fridge app, med undtagelse af UC5 og UC6, som kun skal kunne udføres i Fridge app.	Det testes om de respektive accepttests for de funktionelle krav, også kan udføres på web-app'en.	De samme muligheder er tilgængelige på web-app'en, som på Fridge-app'en.	Som forventet	Godkendt

I4 PRJ4, Gruppe 5

IKT

Prækondition: UC1, og at der i UC2 er tilføjet en vare.

2: <i>Databaser</i>	Test	Forventet resultat	Resultat	Godkendt/ kommentar
Punkt 2.1: Den lokale og den eksterne database skal automatisk synkroniseres hvert 10. minut.	UC2 udføres, hvorefter der tages tid, og efter 10 minutter, åbnes web-app'en, hvorefter det testes visuelt om varen er tilføjet.	Den tilføjede vare er nu synlig gennem web-app'en.	Som forventet	Godkendt
Punkt 2.2.1: I tilfælde af konflikter ved synkronisering, overskriver de nyest tilføjede data de ældste.	UC2 udføres først på Fridge app, hvor antallet sættes til 1. Herefter udføres UC2 for samme vare på web app, hvor antallet sættes til 2. Til sidst udføres UC5, og antallet af varen testes visuelt på begge apps.	Antallet af varer er 2.	Som forventet	Godkendt
Punkt 2.2.2: I tilfælde af konflikter ved synkronisering, overskriver de nyest tilføjede data de ældste.	Ovenstående test udføres igen, men tilføjelserne udføres i omvendt rækkefølge. Antallet af varer testes visuelt på begge apps.	Antallet af varer er 1.	Som forventet	Godkendt

3: Fridge app	Test	Forventet resultat	Resultat	Godkendt/kommentar
Punkt 3.1.1: Ved opstart og nedluk, forsøges synkronisering mellem den lokale og den eksterne database.	Først UC5, og herefter UC2 udføres på Fridge app, hvorefter systemet lukkes. Herefter testes visuelt på web-app'en, om varen er tilføjet.	Varen er tilføjet.	Som forventet	Godkendt
Punkt 3.1.2: Ved opstart og nedluk, forsøges synkronisering mellem den lokale og den eksterne database.	UC2 udføres på web app, hvorefter Fridge app startes. Herefter testes visuelt på Fridge-app'en, om varen er tilføjet.	Varen er tilføjet.	Som forventet	Godkendt
Punkt 3.2: Ændringer af data lagres straks i den lokale database.	Først UC5, og herefter UC2 udføres på Fridge app, hvorefter der testes visuelt, at varen er tilføjet.	Varen er tilføjet.	Som forventet	Godkendt
Punkt 3.3.1: En knap/et ikon på skærmen skal indikere status for synkronisering.	UC5 udføres, og det testes visuelt, i hovedmenuen, om et ikon på skærmen indikerer at der er synkroniseret.	Et ikon indikerer at der er synkroniseret.	Som forventet	Godkendt

Punkt 3.3.2: En knap/et ikon på skærmen skal indikere status for synkronisering.	Internetforbindelse til computeren afbrydes. UC5 udføres, og herefter UC2 udføres, og det testes visuelt, i hovedmenuen, om et ikon på skærmen indikerer at der ikke er synkroniseret.	Et ikon indikerer at der ikke er synkroniseret.	Som forventet	Godkendt
Punkt 3.4: Responstiden for skift af kontekst i menuen må maksimalt være to sekunder. Ved tilgang til de forskellige lister gælder dette kun maksimalt for 42 varer.	Der trykkes på "Se varer", og tiden fra trykket til skift af kontekst måles med stopur.	Tidsmålingen overstiger ikke to sekunder.	Som forventet	Godkendt
Punkt 3.5: Skal kunne anvendes uden internetforbindelse.	Internetforbindelsen afbrydes, UC2 udføres, og det testes visuelt, om varen tilføjes.	Varen er tilføjet til listen.	Som forventet	Godkendt
Punkt 3.6.1: Varer på standard-beholdning tilføjes automatisk til indkøbslisten ved mangel i køleskabet.	En vare med typen <i>Test2</i> antal <i>1</i> , volumen <i>1</i> , unit <i>l</i> , holdbarhedsdato <i>15/03/1993</i> tilføjes til Standard-beholdningen.	Varen er tilføjet standard-beholdningen og indkøbslisten.	Som forventet	Godkendt

I4 PRJ4, Gruppe 5

IKT

Punkt 3.6.2: Varer på standard-beholdning tilføjes automatisk til indkøbslisten ved mangel i køleskabet.	En vare med typen <i>Test2</i> , antal <i>1</i> , volumen <i>1</i> , unit <i>l</i> , holdbarhedsdato <i>15/03/1993</i> tilføjes til <i>Køleskab</i> , og punkt 3.6.1 udføres én gang, hvor varens antal er 2	Varen er nu tilføjet standard-beholdning, og differencen mellem antallet af varer på <i>Indkøbsliste</i> og køleskab, tilføjes til antal på varen på indkøbslisten.	Som forventet	Godkendt
Punkt 3.6.3: Varer på standard-beholdning tilføjes automatisk til indkøbslisten ved mangel i køleskabet.	Varerne, tilføjet til indkøbslisten i punkt 3.6.1 og 3.6.2, fjernes. Varen fra punkt 3.6.1 tilføjes til køleskab med antal på 3. Herefter udføres punkt 3.6.1 igen.	Varen er nu tilføjet til standard-beholdningen, men ikke til indkøbslisten.	Som forventet	Godkendt
4: Web app	Test	Forventet resultat	Resultat	Godkendt/kommentar
Punkt 4.1: Ændringer af data lagres straks i den eksterne database.	UC2 udføres på web app, hvorefter der testes visuelt, at varen er tilføjet.	Varen er tilføjet.	Som forventet	Godkendt

Referencer

DoFactory. (u.d.). *Facade*. Hentet fra DoFactory: <http://www.dofactory.com/net/facade-design-pattern>

DSDM CONSORTIUM. (u.d.). *MoSCoW Prioritisation*. Hentet fra DSDM CONSORTIUM:
<http://www.dsdm.org/content/10-moscow-prioritisation>

Fowler, M. (u.d.). *Unit of Work*. Hentet fra Martin Fowler:
<http://martinfowler.com/eaCatalog/unitOfWork.html>

Microsoft. (u.d.). *Code Metric Values*. Hentet 2015 fra Microsoft Developer Network:
<https://msdn.microsoft.com/en-us/library/bb385914.aspx>

Microsoft. (u.d.). *The Repository Pattern*. Hentet fra MSDN-the microsoft developer network:
<https://msdn.microsoft.com/en-us/library/ff649690.aspx>

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>