## Design, implementering og test - Database

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

For mere en detaljeret gennemgang, henvises til projektdokumentationens **side XX**.

### Fridge app

#### Design

##### Teknologi

Før der har kunnet udarbejdes et design, har det været nødvendigt først at bestemme hvilken teknologi der har skullet anvendes til at implementere *DAL*. Her har de oplagte valg stået mellem *ADO.NET* og *Entity Framework*, da det har været de teknologier, der er blevet nævnt i Database-kurset. Her er det blevet valgt at anvende ADO.NET for læringens skyld, foruden at der ville blive overvejet at anvende *Entity Framework* til *Web app*.

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

##### Objektmodel

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

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

Efter disse entities er blevet identificeret, er et ER-diagram blevet udarbejdet, som ses på **Figur 1**, vha. *DDS-Lite.*

C:\Users\Mathis\Desktop\SmartFridgeV1.jpg

**Figur 1** ER diagram

Der er udnyttet et *Mange-Til-Mange*-forhold mellem *List* og *Item*, hvor den svage entitet *ListItem* forbinder de to. På denne måde har det været muligt at have den konkrete vare og varetypen adskilt på en fordelagtig måde. Objektmodellen med attributter kan ses på **Figur 2**.



**Figur 2** Objekt model med attributter

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

##### Repository pattern

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

##### Unit of Work

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

##### Synkronisering

Da det er et krav at systemet skal kunne fungere lokalt uden internet, har det været nødvendigt at implementere synkronisering mellem en lokal database og en ekstern database. En illustration af dette ses på **Figur 3**. Til formålet er der anvendt *Microsoft Sync Framework*, da dette er en løsning som man selv kan implementere, frem for synkroniseringsværktøjer.



**Figur 3** Illustration af synkronisering

På denne måde sørges der for at indholdet af den lokale og den eksterne database altid er ens, hvorved brugeren kan anvende systemet på både *Fridge app* og *Web app*. Et sekvensdiagram for synkronisering kan ses på **Figur 4**, hvor det illustreres hvordan der laves en provision af server-databasen og den lokale database, hvorefter databaserne synkroniseres.



**Figur 4** Sekvensdiagram for synkronisering

##### Endeligt design

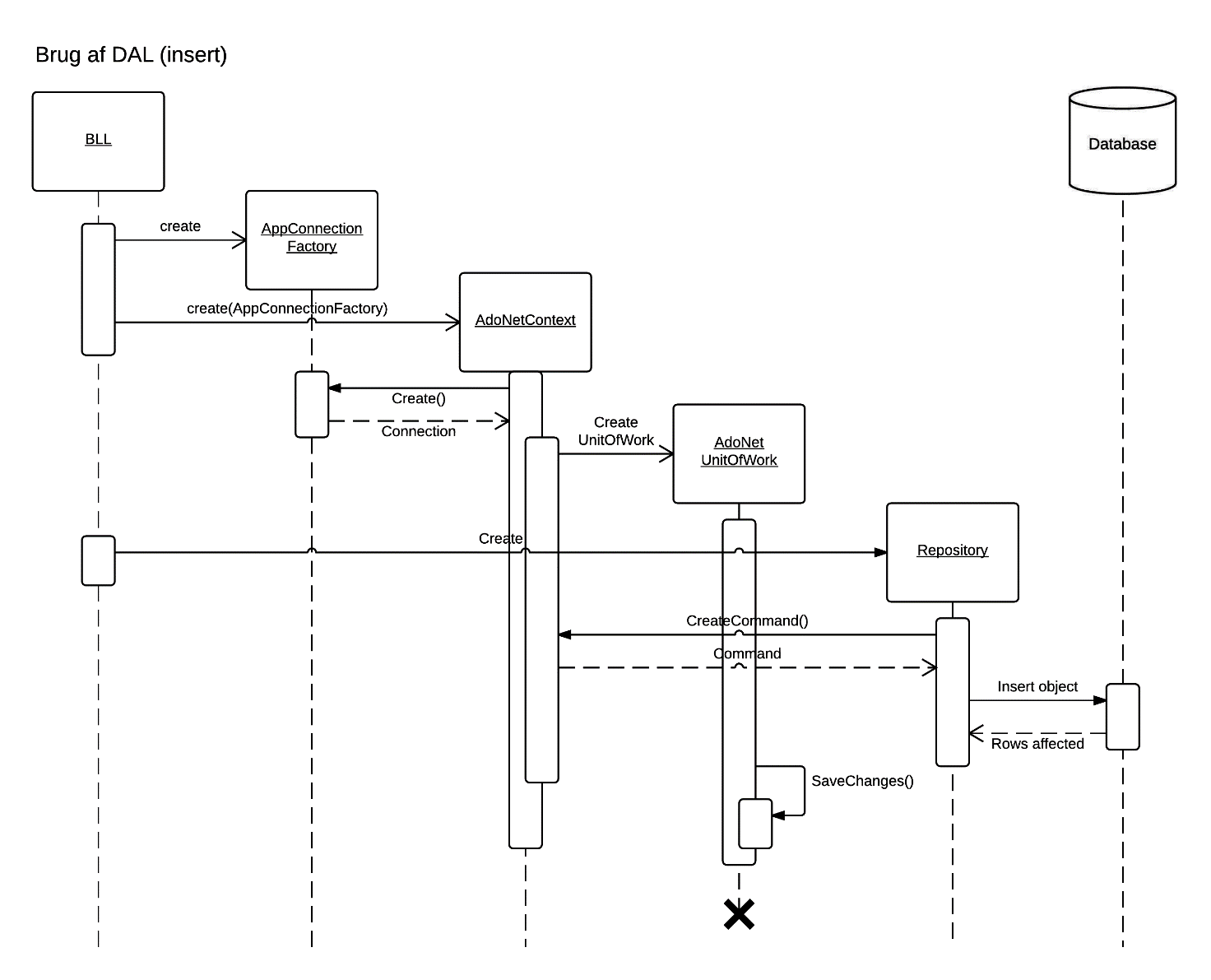
Efter de designovervejelser, der er blevet gjort i de forrige afsnit, er der blevet udarbejdet et klassediagram. Klassediagrammet kan ses i **bilag XX**, og uddybes i projektdokumentationen fra **side XX**.Klassediagrammet er udarbejdet over en iterativ proces, og har dermed ikke været det endelige klassediagram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

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

##### Anvendelse af DAL

**Figur 5** viser et sekvensdiagram for en *insert operation* ved brug af *DAL*. Dette beskriver hvordan *Business Logic Layer* (*BLL*) anvender *DAL* til at indsætte en entitet i databasen.

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



**Figur 5** Sekvensdiagram for brug af DAL

#### Implementering

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

#### Test

I dette afsnit vil tests af *DAL* blive beskrevet, samt *coverage* og *statisk analyse*.

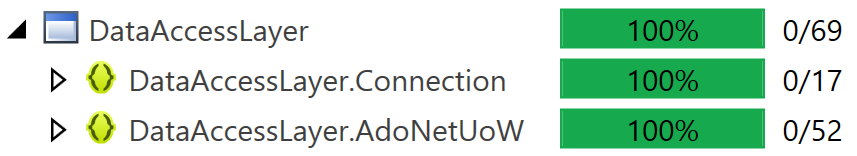
**Figur 6** viser et screenshot af *testsuite* for *DAL* i *Fridge app*. Som det fremgår, har alle tests været en succes, hvorpå funktionaliteten fungerer som forventet. Her ses også at *Repository.cs*, samt dens nedarvede klasser, ikke er testet, hvilket skyldes at disse klassers ansvar er at skrive til databaser med *SQL commands*, hvilket ikke er særlig testbart. Desuden vil disse tests heller ikke give så meget, da det kun er databasetilgang. Herudover er Sync heller ikke testet, da der er meget høj kobling til tabellerne, og da man ikke kan *mock*’e funktionaliteten ud, er dette heller ikke testet.



**Figur 6** Testsuite af DAL

##### Coverage

**Figur 7** viser et screenshot af *coverage*-resultatet for DAL-implementering. Her fremgår det at der er opnået 100% coverage, hvilket betyder at al kode er blevet berørt.



**Figur 7** Coverage af DAL

##### Statisk analyse

**Figur 8** viser et screenshot af *code metrics* for *DAL*. Her fremgår *maintainability*, hvor 20-100 er høj *maintainability*, hvilket viser at *DAL*-implementering kan vedligeholdes. Dog har *DAL* høj kompleksitet, der kommer sig af høje koblinger i forhold til *entities* og *repositories*.



**Figur 8** Code metrics for DAL

### Web app

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

#### Design

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

##### Teknologi

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

##### *Façade* pattern

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

##### Endeligt design

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

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

##### Anvendelse af DAL

Et sekvensdiagram for anvendelse af *DAL* for *Web app*, ses på **Figur 9**. Her ses det hvordan *BLL* (eller en *Controller*) kan oprette et *SmartFridgeDALFacade*-objekt, hvor der kan oprettes et *UnitOfWork* med *GetUnitOfWork*. Heri kan de ønskede transaktioner kan laves, hvorefter disse *commit*’es med funktionen *SaveChanges*. Efter brug kan *UnitOfWork* *dispose*’s, hvorpå det nedlægges og resourcer frigives.



**Figur 9** Sekvensdiagram for brug af DAL på Web app

#### Implementering

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

#### Test

I dette afsnit vil test af *DAL* for *Web app* blive beskrevet, og *coverage* og *statisk analyse* vil blive dokumenteret.

**Figur 10** viser *testsuite* for *DAL* i *Web app*. Som i *DAL* for *Fridge app*, er repositoryet ikke blevet modultestet, da det igen er databasetransaktioner, og det ikke er egentligt funktionalitet at teste. Herudover er *SFContext* heller ikke testet, da det kommer fra *Entity Framework*, som må anses som gennemtestet.

Disse test var lettere at skrive, da brugen af interfaces, som ikke har været så meget anvendt i *DAL* for *Fridge app*, gjorde det lettere *mock*’e funktionaliteten ud.



**Figur 10** Test af DAL for Web app

##### Coverage

På **Figur 11**, ses *coverage*-resultater for *web DAL*. Her ses det at der er opnået 100% coverage af den testede kode.



**Figur 11** Coverage af DAL for Web app

##### Statisk analyse

**Figur 12** viser *code metrics* af *DAL* for *Web app*. Her ses det at der er højere maintainability, samt lavere kompleksitet. Dette betyder at implementeringen af *WebDAL* har været mindre kompleks, hvilket *Entity Framework* tilbyder.



**Figur 12** Code metrics af DAL for Web app