# Design, implementering og test - Database

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

For mere detaljeret gennemgang, henvises til Design, implementering og test for databasen i projektdokumentation.

## Fridge app

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

### Design

I dette afsnit vil designprocessean blive beskrevet, hvor de designovervejelser der er gjort i forhold til DAL for Fridge App.

#### Teknologi

Før at der kunne udarbejdes et design, var det nødvendigt først at bestemme hvilken teknologi der skulle anvendes til at implementere DAL. Her stod de oplagte valg mellem ADO.NET og Entity Framework, da det var de teknologier der var blevet nævnt i database-kursuset. Her blev det valgt at anvende ADO.NET for lærings skyld, samt at der ville blive overvejet at anvende Entity Framework til Web app’en.

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

#### Objekt model

Da der arbejdes med relationelle databser, skulle der udarbejdes en objekt model, hvor at data kunne gemmes korrekt. Her blev det diskuteret hvilke objekter der var nødvendige at kunne gemme i databasen. Her blev følgende entities identificeret:

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

Da disse entities var identificeret, kunne der udarbejdes en objekt model, som ses på Figur 1, der er udarbejdet vha. DDS-lite. Der er udnyttet et Mange-til-mange mellem ’List’ og ’Item’, hvor weak entityen ’ListItem’ er imellem. På den måde var det muligt at have den konkrete vare og varetypen adskilt på en fordelagtig måde. Objektmodellen med attributter kan ses på Figur 2.

Som beskrevet, er ’ListItem’ en weak entity, hvilket introduceret flere udfordringer. ListItems update funktion er ikke implementeret, samt det har været en nødvendighed at implementere en mapper for at binde objekt modellen sammen. Dette ville Entity Framework kunne håndtere, så brug af Entity Framwork havde været fordelagtigt.

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

Figur ER diagram



Figur Objekt model med attributter

#### Repository mønster

Som en del af designet, er Repository mønstret blevet anvendt. Repository mønstret giver mulighed at have ét sted, hvor at alt databasetilgang foregår. Formålet med dette er at vi seperare databasetilgangen for forretningslogikken, der er med til at gøre systemet lettere at vedligeholde, samt lettere læsligt. Udover det, giver Repository mønstret også mere testbarhed, da det vil være muligt at mocke repositoryet ud og derved kan dette gøre mere forretningslogik testbart. For en mere detaljeret beskrivelse af repository mønstret, henvises til Repository mønster-afsnittet i projektdokumentationen.

#### Unit of Work

Til designet, sammen med Repository mønstret, er Unit of Work blevet anvendt. Formålet med dette, er at alle databasetransaktioner foregår i et Unit of Work, hvorpå det er muligt at commit og rollback, derved er transaktionerne kontrolleret. Dette er fordelagtigt for at sørge for at databasetransaktioner foregår med færre fejl, samtidig med at alle transaktioner bliver comitted på samme tid, så databasen tilgås på en gang. For en mere detaljeret beskrivelse af Unit of Work, henvises til Unit of Work-afsnittet i projekt-dokumentationen.

#### Synkronisering

Da det er et krav at systemet skal kunne fungere lokalt uden internet, har det været nødvendigt at implementere synkronisering af 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 man selv kan implementere.

På den måde bliver der sørget for at indholdet af den lokale og den eksterne database altid er ens, derved kan brugeren anvende systemet på både Fridge App og Web app. Et sekvensdiagram for synkronisering på kan ses på Figur 4, hvor det illustreres hvordan man laver en provision af Server database og lokal database og derefter synkroniserer databaserne.



Figur 3 Illustration af synkronisering

#### C:\Users\Mathis\Downloads\DbSync SD - New Page.png

Figur 4 SD - Synkronisering

#### Endeligt design

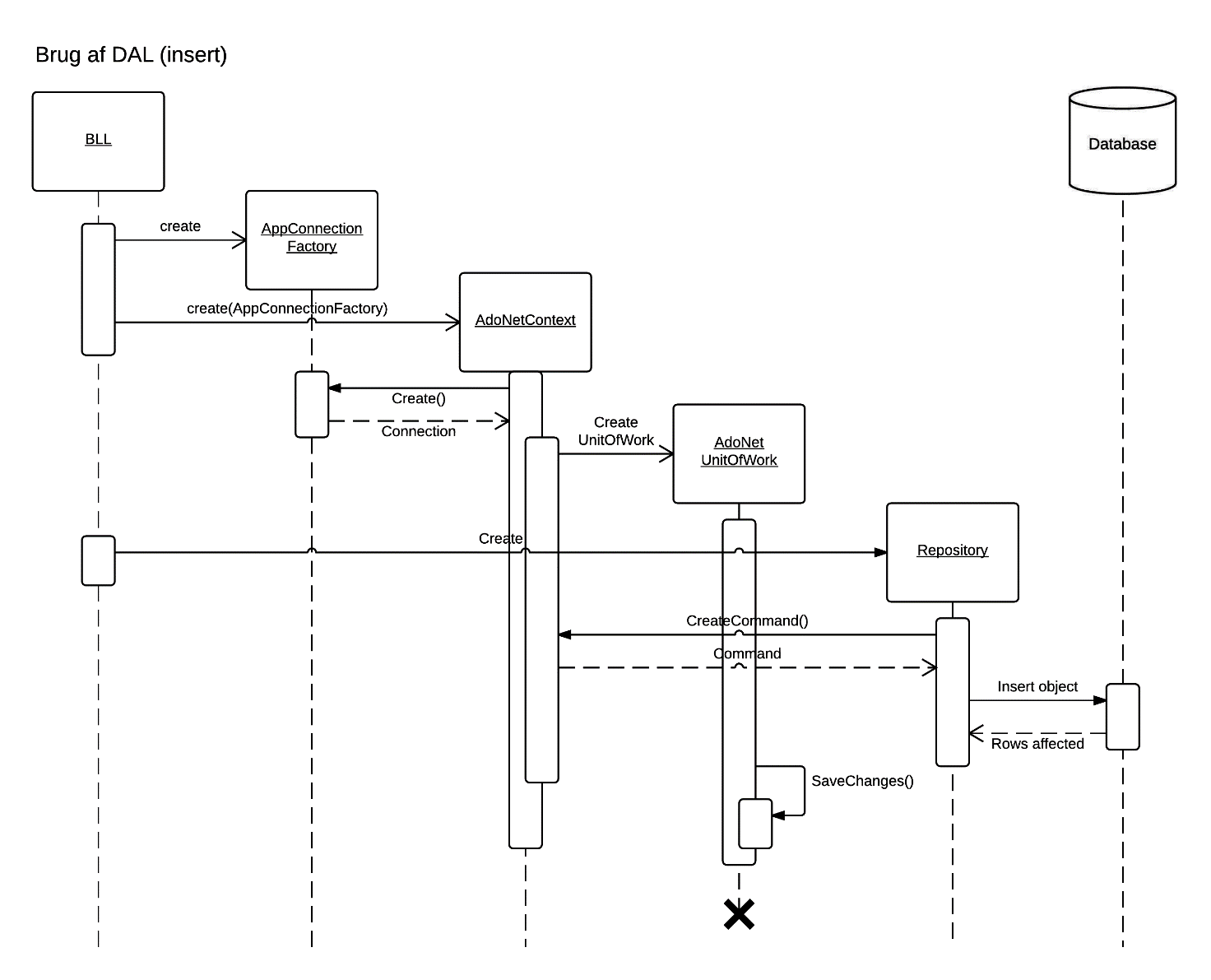
Efter de designovervejelser der er blevet gjort i de forrige afsnit, er der blevet udarbejdet et klassediagram. Klassediagrammet kan ses i bilag XX. Klassediagrammet er udarbejdet over en iterativ proces, derved har det ikke været det endelige klassediagram fra start.

Opbygningen af DAL, som klassediagrammet, er at der er en AppConnectionFactory, hvis ansvar er at oprette forbindelse til databasen, igennem en connectionstring fra app.config. Denne forbindelse injectes i AdoNetContext, som svarer til Entity Frameworks DbContext, hvorpå at man kan oprette et Unit of Work og execute commands på databasen. I et unit of work, kan man anvende repositoryet til databasetilgang, som er transaktionstyret. Når man er færdig med sine databaseoperationer, kan man commite og nedlægge Unit of Work. For en mere detaljeret gennemgang, henvises til projektdokumentationen.

#### Anvendelse af DAL

På Figur 5, ses et sekvensdiagram for en insert operation ved brug af DAL. Dette beskriver hvordan at BLL anvender DAL til at kunne inserte en entity i databasen.

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 eksikveres databasetransaktioner. Her kan man oprette et Repository-objekt til den ønskede entity, man laver sine ønskede databasetransaktioner og herefter commites disse transaktioner til databasen. Herefter nedlægges AdoNetUnitOfWork.



Figur SD - 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 projektdokumentation.

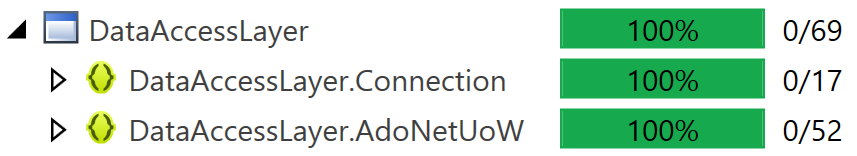
### Test

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

På 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.



#### Coverage

På Figur 15 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.

#### Statisk analyse

På Figur 16, ses et screenshot af code metrics for DAL. Her kan man se maintainability, hvor 20-100 er høj mainability, 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.

## Web app

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

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 udenom de tidligere beskrevet problemer med brug af ADO.NET, såsom abstraktion og testbarhed. Udover dette, er det også for læringens skyld.

#### Façade mønster

Udover anvendelse af Repository mønstret og Unit of Work, er der også anvendt Façade til DAL for Web App. Formålet med dette mønster er for controllerne har et sted, hvor de tilgår alt database, hvilket giver højere abstraktion, samt høj testbarhed. For mere detaljeret gennem af Façade mønster, henvises til projektdokumentation.

#### 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 XX. Klassediagrammet er udarbejdet over en iterativ proces, derved har det ikke været det endelige klassediagram fra start.

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

#### Anvendelse af DAL

Et sekvensdiagram for anvendelse af DAL for Web app, ses på Figur 6. Her ses det hvordan BLL (eller en controller) 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 resourcer.



Figur 6 SD - 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 dokumentatiom, samt dokumentation af væsentlig funktionalitet, henvises til projektdokumentation.

### Test

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

På Figur 7, 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.

Disse test var lettere at skrive, da brugen af interfaces, som ikke har så megen anvendt for DAL for Fridge App, gjorde det lettere mocke funktionaliteten ud.



Figur 7 Test af DAL for Web app

#### Coverage

På Figur 8, ses coverage resultater for web DAL. Her ses det at der er opnået 100% coverage af den testede funktionalitet.

Figur 8 Coverage af DAL for Web app

#### Statisk analyse

På Figur 9, 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.



Figur 9 Code metrics af DAL for Web app