# Design og implementering - 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 1**.

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

**Figur 1** 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 2**, 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 2** 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*-mønstret separerer logikken mellem forretningslogikken og datakilden, som en database. En illustration af dette ses i **Figur 3**. *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 (MSDN, 2015).



**Figur 3** Repository Pattern (MSDN, 2015)

*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, 2015).

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 Micsoft Sync Framework blevet valgt, da det er løsning der er mulig selv at implemente, uafhængigt af hvilken data provider der anvendes. En illustration af synkronisering mellem lokal database og ekstern database på en webserver ses på Figur 4.



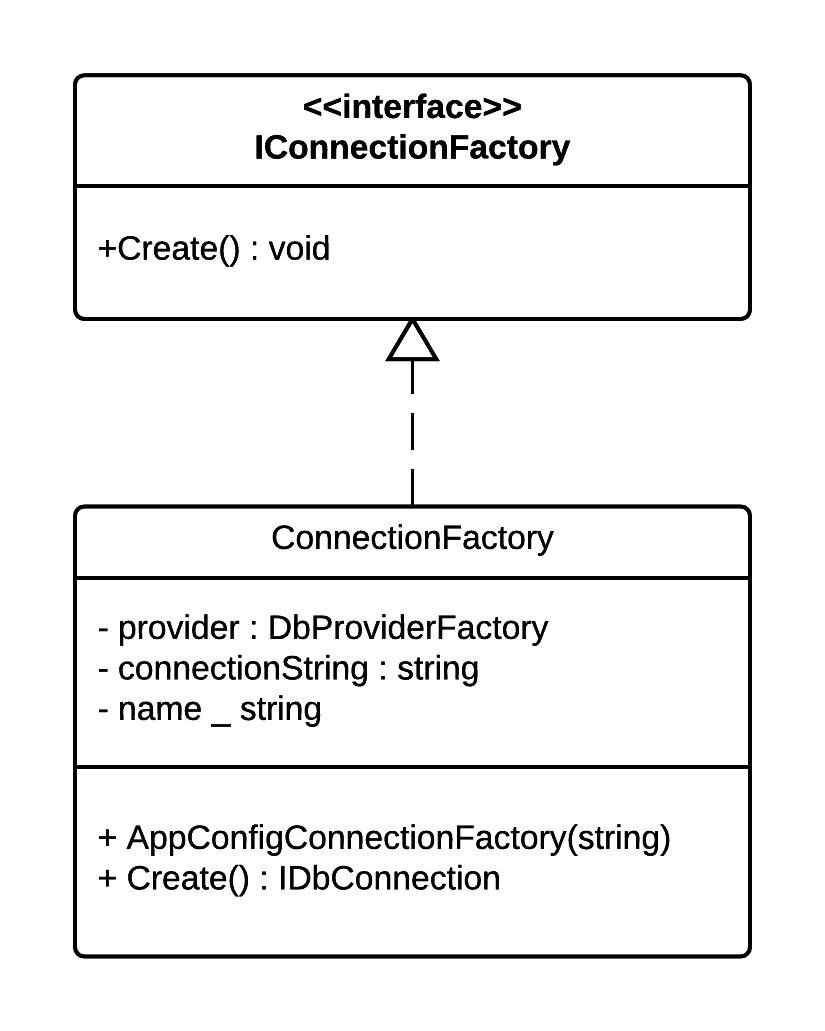
Figur Illustration af database synkronisering

#### Endeligt design

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

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.

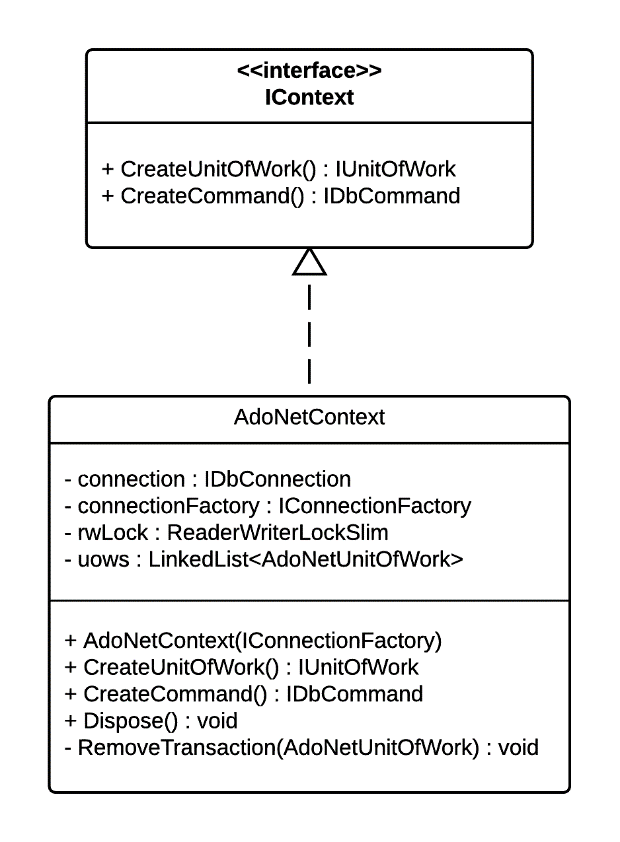
##### ConnectionFactory



Figur 5 ConnectionFactory

IConnectionFactory er et interface, som ses på Figur 5, 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.

##### AdoNetContext



Figur 6 AdoNetContext

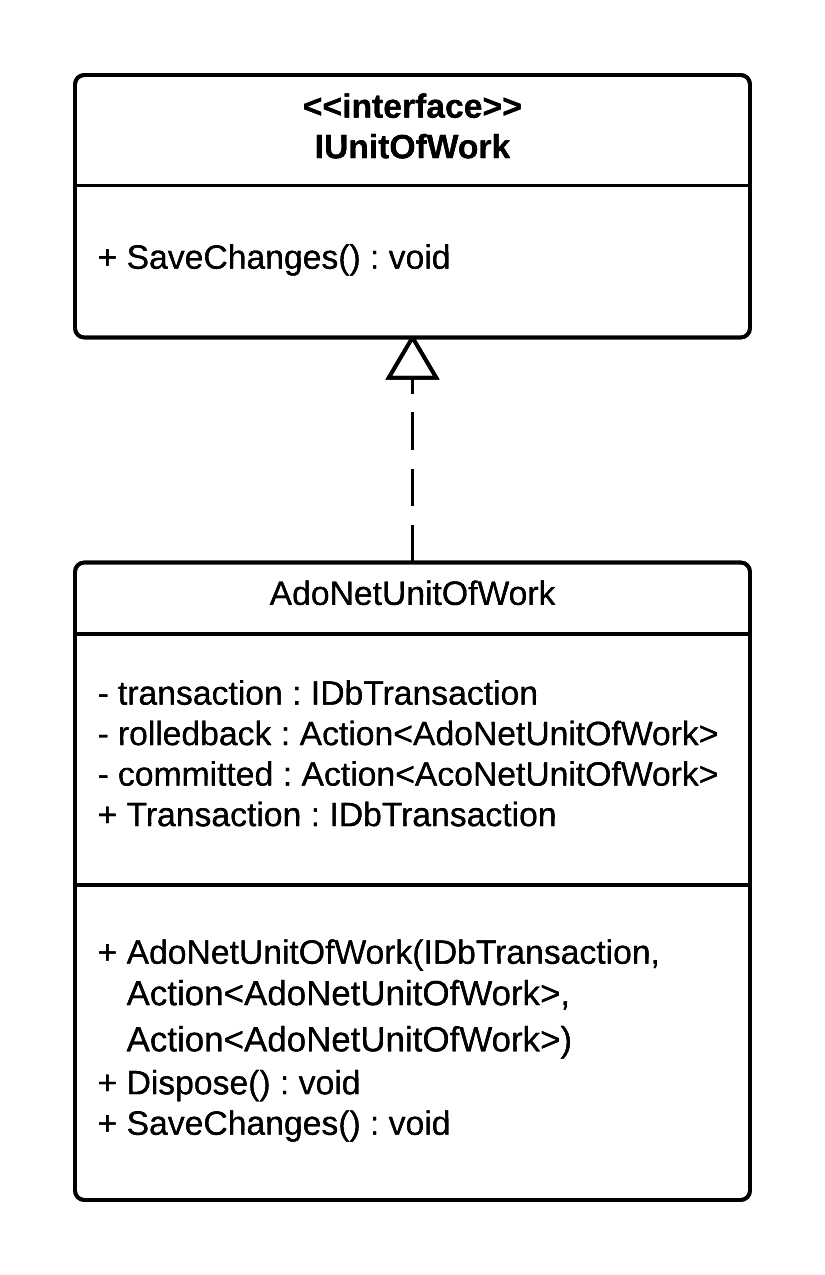
IContext er et interface, som ses på Figur 6 , svarer til DbContext i Entity Framework. Heri bliver forbindelsen til databasen opretholdt og man kan execute commands, hvor der kan anvendes et Repository. Her kan der også oprettes i Unit of Work, hvor man kan udføre sine databasetransaktioner.

##### AdoNetUnitOfWork

IunitOfWork er et interface, som ses på Figur 7, der er en af implementering af Unit of Work mønstret. Her er SaveChanges() dens commit funktionalitet. Herudover har AdoNetUnitOfWork, hvori der opretholdes en transaktionhistorik.

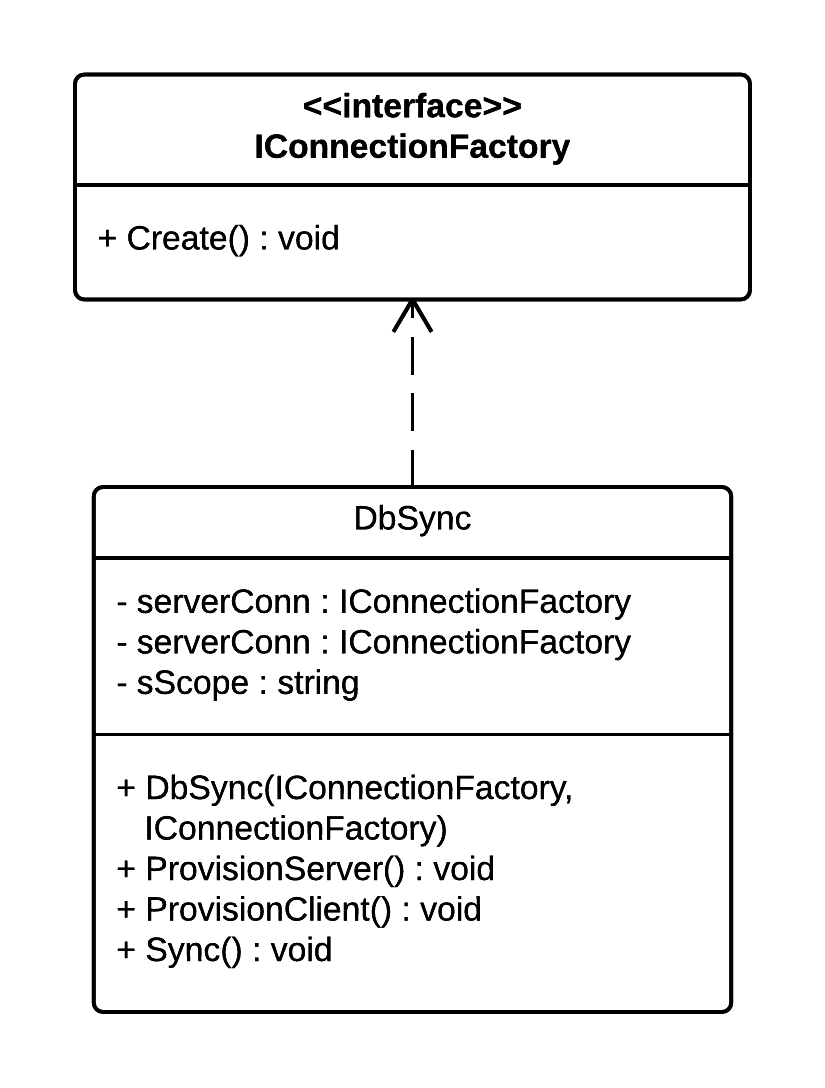
##### Repository

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



Figur 7 AdoNetUnitOfWork

##### DbSync



Figur 8 DbSync

DbSync, som ses i Figur 8, viser logikken for synkronisering af databaserne, hvorpå den anvender IconnectionFactory, til at åbne en forbindelse til databaserne. Et svekvensdiagram for synkroniseringsfunktionaliteten kan ses på Figur 9.



Figur 9 SD - Synkronisering

#### Anvendelse af DAL

Et sekvensdiagramm for anvendelse af DAL, kan ses på Figur 10. 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 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.

#### C:\Users\Mathis\Downloads\SD - Brug af DAL - New Page.png

Figur SD - 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 XX.

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;

}

Figur Kodeudklip fra AdoNetContext.cs

På Figur 11, ses et kodeudklip, 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 Figur 12. Her mappes ItemId over i et midlertidigt table, hvorpå det kan outputtes fra. Denne fremgangsmåde bliver også brugt for insert af List.

DECLARE @InsertItem TABLE ([ItemId] INT);

INSERT INTO [Items] ([ItemName],[StdVolume],[StdUnit])

OUTPUT [inserted].ItemId INTO @InsertItem

VALUES(@ItemName,@StdVolume,@StdUnit);

SELECT \* FROM @InsertItem";

Figur 12 Insert command fra ItemRepository.cs

#### ToList

Der udnytter arv til implementering af ToList, hvor den modtager en command og eksikverer og den og opretter objekter ud fra hvad den modtager. Anvendes i de fleste Read-metoder i de forskellige repositories.

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;

}

}

Figur 13 ToList fra Repository.cs

### Test

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

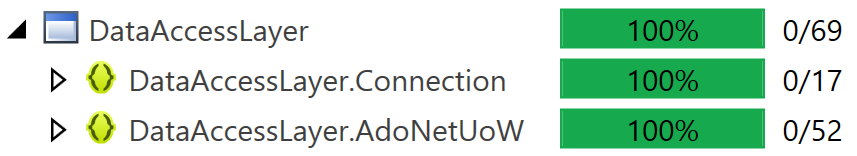
På Figur 14, ses et screenshot af testsuite for Fridge App’ens DAL. Her ses at alle tests var en succes, hvorpå funktionaliteten fungerer som forventet.

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.



Figur 14 Screenshot af testsuite for DAL

#### Coverage

På **Fejl! Henvisningskilde ikke fundet.** 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.

Figur 15 Screenshot af coverage for DAL

#### 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.



Figur 16 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

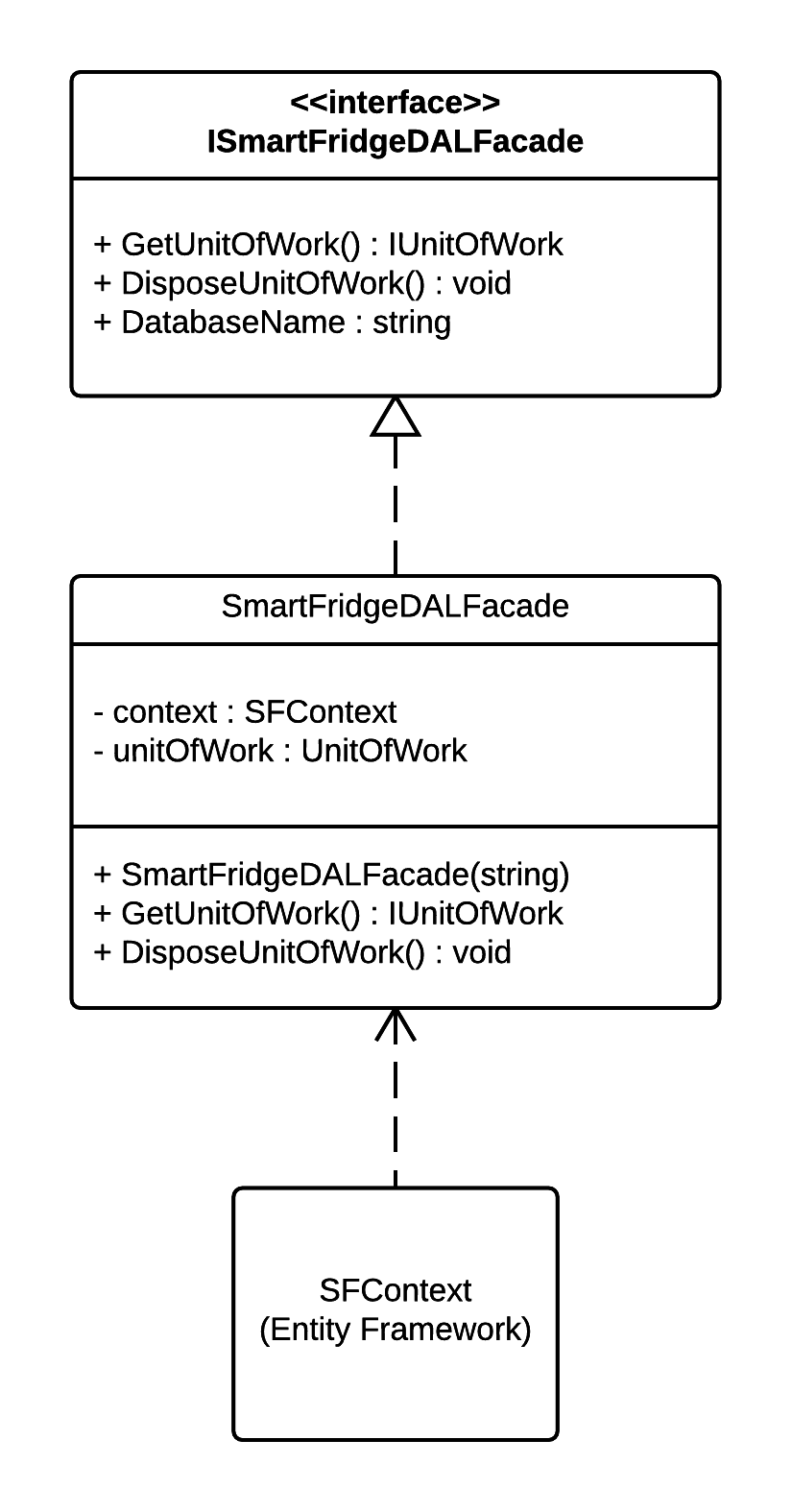
Façade er et design mønster, hvorpå man skaber et simpelt interface til et kompleks subsystem (dofactory, 2015). Et eksempel på det ses på Figur 17.



Figur Facade illustration (dofactory, 2015)

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

#### 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.

Figur SmartFridgeDALFacade

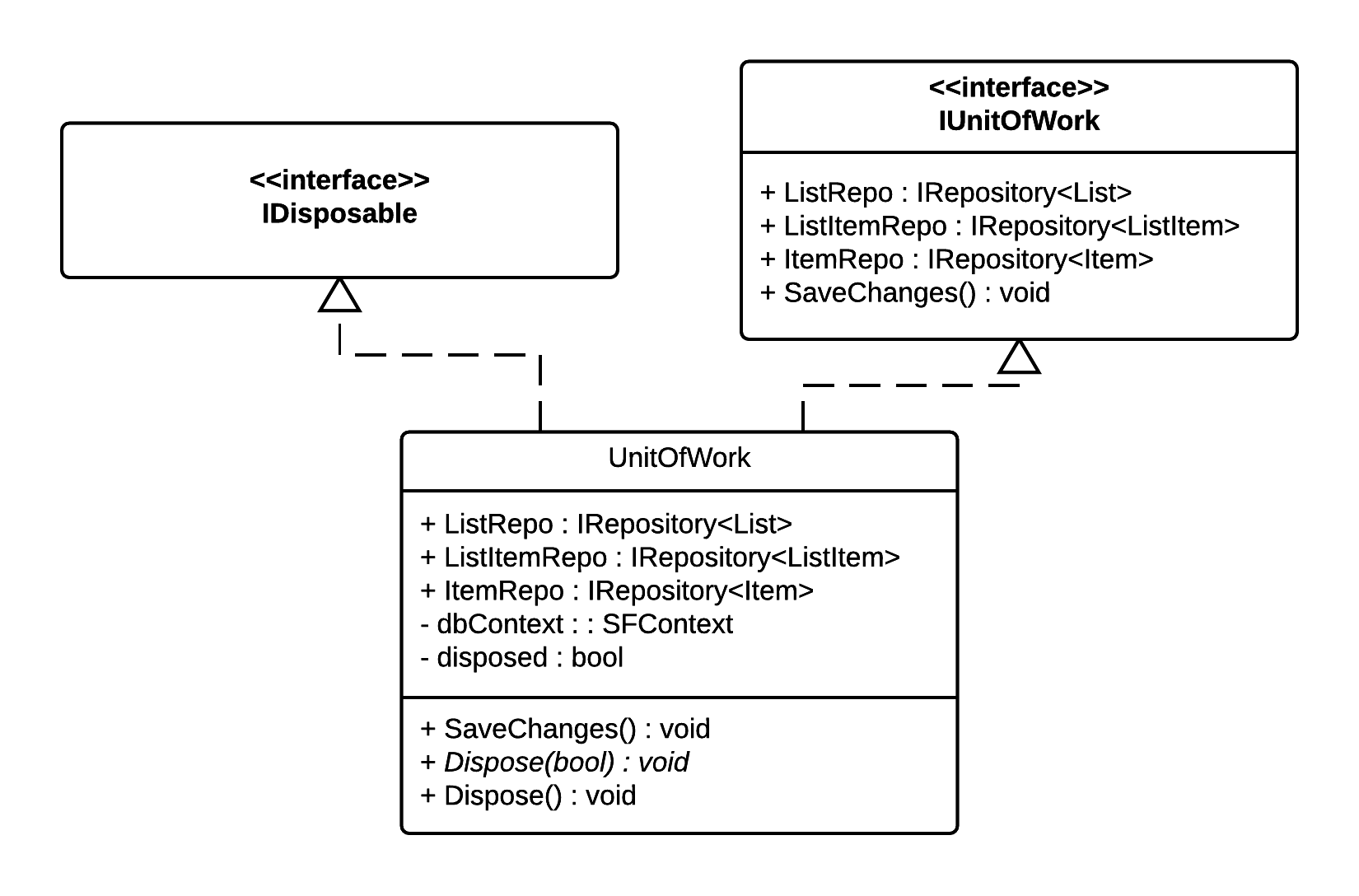
Klassediagrammet er udarbejdet over en iterativ proces, derved har det ikke været det endelige klassediagram fra start. I de følgende afsnit, vil de væsentlige dele af klassediagrammet blive beskrevet i ansvarsområder.

##### SmartFridgeDALFacade

ISmartFridgeDALFacade er et interface, som ses på Figur 18. 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

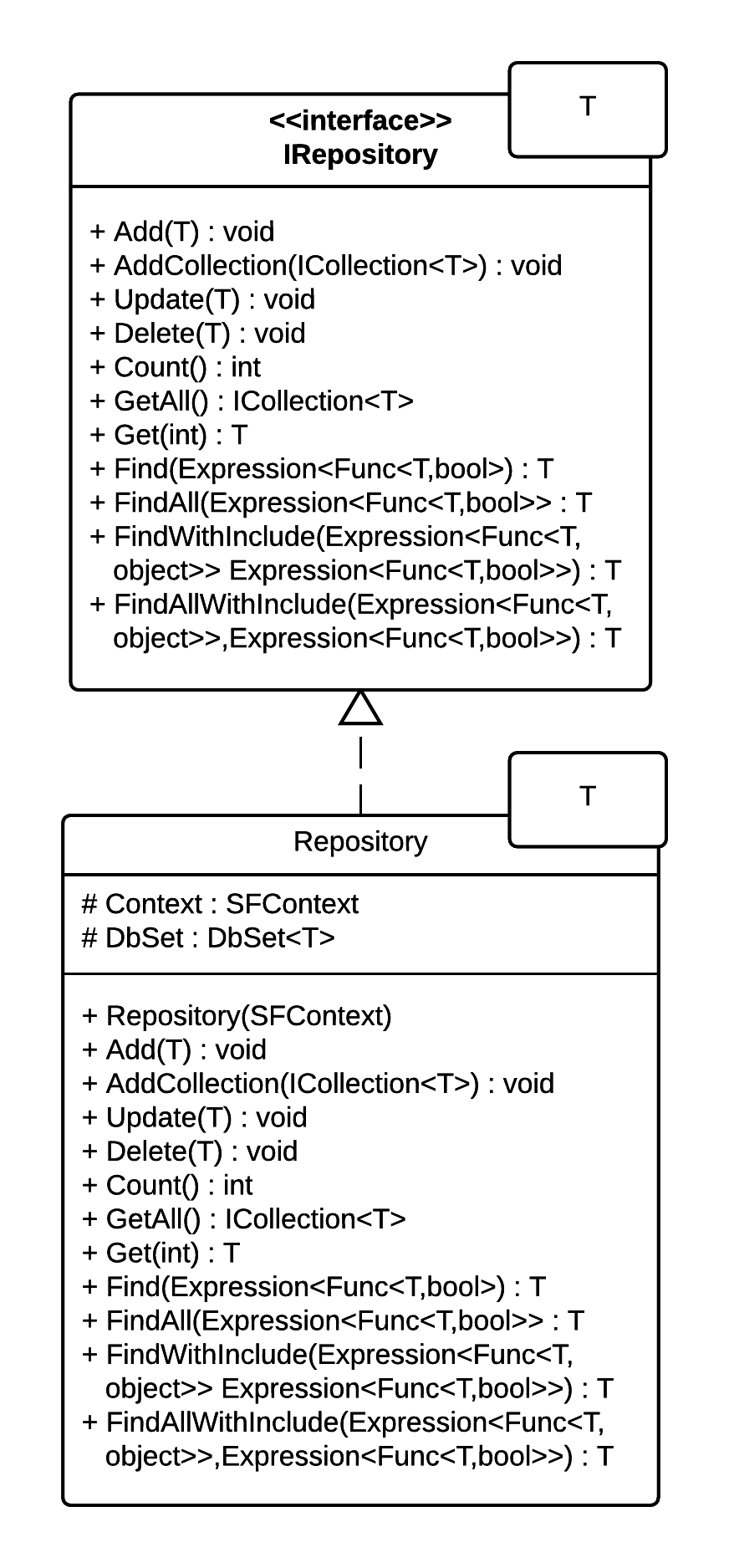
IunitOfWork er et interface, som på Figur 19, der er en implementering af Unit of Work mønstret. Til forskel for Unit of Work for Fridge App, oprettes repositories i et unit of work, derved er alt databasetilgang, transaktioner. UnitOfWork implementerer også Idisposable for at nedlægge Unit of Work for at nedlægge og frigive resourcer korrekt.



Figur 19 UnitOfWork

##### Repository

IRepository er et interface, som ses på Figur 20 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 Repository

#### Anvendelse af DAL

Et sekvensdiagram for anvendelse af DAL for Web app, ses på **Fejl! Henvisningskilde ikke fundet.**. 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 resourcer.



### 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 XX.

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

#### GetUnitOfWork

Figur Kodeudklip fra SmartFridgeDALFacade.cs

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;

}

På Figur 22 ses et kodeudklip, 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 22, 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.



Figur 22 Screenshot af testsuite for DAL for Web app

#### Coverage

På Figur 24, 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.



Figur 24 Screenshot af coverage for WebDAL

#### Statisk analyse

På Figur 25, 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 25 Screenshot af Code metrics for WebDAL