# Intelligent profilstyring i MARS
# Implementeret i .NET 2.0 C#

Informatik og Matematisk Modellering

Danmarks Tekniske Universitet

Forfatter
Kåre Rune Christensen, s012351

Vejleder

Mads Nyborg
DTU IMM

Jens Christian Fassel
Infopaq International A/S

IMM – B.Eng. – 2007 - 51

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

IMM – B.Eng. – 2007 - 51

# Abstrakt

Design og implementering af profilstyringsværktøj til det eksisterende Medie Analyse Registrerings System (*MARS*) i .NET 2.0 C#.

*MARS* er bygget til at registrere medier/artikler via *elementerne subject* og *measure*. Disse *elementer* skal håndteres via profiler og kategorier.

Profiler består af kundespecifikke, globale og virtuelle *elementer*, hvorved *element*typer kan deles kunderne og profilerne imellem. *Elementerne* bruges til at validere og indeksere indholdet af medier og deres gennemslagskraft. Der findes flere typer af *elementer*, hvor nogle beskriver mediernes indhold mens andre deres gennemslagskraft både positivt og negativt, til dette bruges Infopaqs egen enhedsskala *Paq Value*. Selve mediet/artiklen er også beskrevet i *MARS* med de tilhørende attributter, der indeholder tilstrækkelig information, således gennemslagskraften kan udregnes.

Oprettelse og redigering af kunder, profiler og *elementer* implementeres i en *n-tier*-struktur som en del af en eksisterende entreprise løsning, og skal derved implementeres med passende *OO Design Patterns*.
Profilstyringsværktøjet skal kunne styres gennem en GUI og via WebServices ned til databasen der ligger bag det eksisterende *MARS*.

Det primære mål er at kunne oprette og redigere kunde- og profilopsætninger i *MAR*S.
Det sekundære mål med værktøjet er, at guide brugerne til at sætte profilerne op på fornuftig vis ved hjælp af logiske hjælpemidler i form af forslag, intelligente advarsler samt statistiske oplysninger om *elementernes* brug.

Nøgleord
- Profilstyring
- Entreprise Solution
- N-tier model
- Design patterns
- Microsoft .NET 2.0 C#
- WebService
- Databaser
- Infopaq
- Medieanalyse

# Indholdsfortegnelse

Abstrakt ... iii
Indholdsfortegnelse ... iv
Nomenklatur ... vii
Forord ... ix

1. Introduktion ... 1
  1.1 Forudsætninger ... 1
2. Analyse ... 3
  2.1 Opbygningen af MARS ... 3
  2.2 Analyse af MARS Admin ... 4
    2.2.1 Domæne Model ... 5
    2.2.2 Use Case ... 6
    2.2.3 Data Overensstemmelsesproblemer ... 14
    2.2.4 Resumé ... 16
3. Design ... 17
  3.1 UML Design ... 17
    3.1.1 Use Case Package opdeling og *n-tier* model ... 17
    3.1.2 Sequence Diagram ... 18
    3.1.3 Class Diagram og design patterns ... 20
  3.2 Klasse Opbygning ... 24
    3.2.1 Entiteter/*elementer* ... 24
    3.2.2 Interfaces ... 24
    3.2.3 Nedarvning ... 25
    3.2.4 Enumerationer ... 25
  3.3 GUI Beskrivelse ... 25
  3.4 Resumé ... 27
4. Implementering ... 28
  4.1 Valg af udviklingsmiljø ... 28
    4.1.1 Udgivelse i virksomhedsmiljø ... 28
    4.1.2 Generelt om variable ... 29
  4.2 Navnekonvention ... 29
  4.3 Solution opbygning i VS2005 ... 30
  4.4 Funktionaliteter og Design Patterns ... 32
    4.4.1 Properies ... 32
    4.4.2 Interfaces ... 33
    4.4.3 Nedarvning ... 34
    4.4.4 Transactions ... 35
    4.4.5 Stored Procedure ... 36
    4.4.6 Beskedsystem og exceptions ... 37
  4.5 Resultat ... 38
  4.6 Resumé ... 40
5. Test ... 41
  5.1 Introduktion til test baggrund ... 41
  5.2 Opdeling af testområder ... 41
  5.3 Implementering af test ... 41
    5.3.1 Node base *element* test ... 42
    5.3.2 Entity Dirty Flag test ... 42
    5.3.3 MainProvider SetProfile test ... 43

5.3.4 MainLogic transaction test....45
5.3.5 Get Entity from GUI test....46
5.4 Resumé....47
6 Udvidelser....48
7. Konklusion....49
8. Litteraturliste....51

9. Bilag ... A
9.1 CD indhold ... A
9.2 Projektplan ... B
9.3 Element hierarki ... C
9.4 GUI Udkast ... D
9.4.1 Hoved Form i abstrakt stadie ... D
9.4.2 Valg af Entitet ... E
9.4.3 Oprettelse af *element* ... F
9.4.4 Flyt og Kopier Dialogboks ... F
9.4.5 Visning af *element* detaljer ... G
9.4.6 *Element*-forslag Form ... H
9.5 MARS Database model ... I
9.6 UML Use Case Diagrammer ... J
9.6.1 Use Case Diagram Top Level ... J
9.6.2 Use Case Diagram 1. Customer handling ... K
9.6.3 Use Case Diagram 2. Profile handling ... L
9.6.4 Use Case Diagram 3. Category and Subject handling ... M
9.6.5 Use Case Diagram 4. Measure handling ... N
9.6.6 Use Case Diagram 5. Create Entity ... O
9.6.7 Use Case Diagram 6. Update Entity ... P
9.6.8 Use Case Diagram 7. Alter Binding ... Q
9.6.9 Use Case Diagram 8. Validate Entity ... R
9.6.10 Use Case Diagram 9. Suggestion ... S
9.6.11 Use Case Diagram 10. Showing ... T
9.7 UML Sequence Diagrammer ... U
9.7.1 Opret *elementer* abstrakt ... U
9.7.2 Select Subject ... V
9.7.3 Edit Category ... W
9.8 UML Class Diagram Concept ... X
9.8.1 Frontend Overview Concept ... X
9.8.2 Frontend GUI Concept ... Y
9.8.3 Frontend Domain Data Concept ... Z
9.8.4 Frontend Domain Utility Concept ... Æ
9.8.5 Frontend DALC Concept ... Ø
9.8.6.Backend Concept ... Å
9.8.7 Backend Service Concept ... AA
9.8.8 Backend Domain Logic Concept ... BB
9.8.9 Backend Domain Data Concept ... CC
9.8.10 Backend DALC ... DD
9.9 UML Class Diagrams ... EE
9.9.1 Frontend.GUI ... EE
9.9.2 Frontend.Data ... GG
9.9.3 Frontend.DALC ... HH
9.9.4 Domain.Data ... II
9.9.5 Backend.Service ... LL
9.9.6 Backend.Logic ... MM
9.9.7 Backend.DALC ... NN
9.10 Test Class Diagrams ... OO
9.10.1 Accessors (UnitTestProject) ... OO
9.10.2 Test Classes (UnitTestProject) ... PP

# Nomenklatur

Dette afsnit indeholder udvalgte fagudtryk eller udtryk specifikke for netop dette projekt. Det kan bestå i virksomheden interne udtryk.
Denne rapport vil indeholde notationen for nomenklatur-ord i kursiv gennem hele rapporten. Overskrifter er undtaget dette format.

| **Ord** | **Beskrivelse** |
|---|---|
| *backend* | Et delsystem af *MARS Admin n-tier* løsning. |
| *bindingsværdier* | De værdier et *element* kan have udover de *grundværdier* det holder. Det repræsenterer værdierne, som er specifikke for den virtuelle del af et *element*. |
| *category* | *Element* og en kategori i *MARS* hvori *subjects* skal forekomme. |
| *concurrency control* | Kontrol af *concurrency issues*. |
| *concurrency issues* | Overensstemmelsesproblemer mht. data tilgang. Ofte brugt i forbindelse med databasetilgang. |
| *customer* | Øverste *element* i *element*-hierarkiet. Dækker over begrebet kunde i medieanalyseterminologien. |
| *DALC* | Data Access Layer Component, hvilket er det lag der tager sig af den mere direkte forbindelse til data, og bliver tit brugt uden den store foretningslogik. |
| *design patterns* | Design mønstre der skal bruges til at designe kildekode med. Der findes mange forskellige typer for *design patterns*, flere af dem vil blive nævnt videre i rapporten. |
| *enterprise solutions* | Dette dækker IT-løsninger i større virksomheder, hvor det kan være nødvendigt at have flere forskellige computere til at løse enkelte opgave som f.eks. at WebServers bruger databaseservers. De kan flyttes rundt som separate enheder. Der er også tit brugt *n-tier* modeller i sådanne løsninger. |
| *frontend* | Et delsystem ad *MARS Admin n-tier* løsning. |
| *grundelement* | Den del af et *element* der indeholder *grundværdierne*. *Bindingsværdierne* er den anden del af *elementet*. |
| *grundværdier* | De værdier det er indeholdt i et *grundelement*. Disse værdier kan blive delt af flere bindinger, men hver af disse virtuelle *element*dele indeholder individuelle *bindingsværdier*. |
| *MARS* | Media Analysis Registration System. Et internt softwareprogram i Infopaq, der bliver brugt til at registrere medieanalyser. |
| *MARS Admin* | Dette projekt, her tænkes på selve implementeringen og brugen deraf. I korte træk er det et administrationssystem til det eksisterende *MARS*. |
| *Measure* | *Element* som skal holde værdier for hvorledes et medie/artikel er vurderet, og skal senere i *MARS* bruges til at indsætte vurderingsværdier til beregning af *Paq Value*. |
| *multiple inheritance* | Nedarvning af flere klasser til én enkelt klasse. |
| *n-tier* | Flerdelt lagmodel af software tit brugt i *enterprise solutions*. Denne model er også gennemgået grundigt i rapporten. |
| *OO* | Objekt Orientering, hvilket er en teknisk måde at programmere på, hvilket gør det lettere for en programmør at overskue kildekode, grundet den mere virkelighedsnære repræsentation af |

| | |
|---|---|
| | data. |
| *Paq Value* | En virksomheds internt beregnet værdi, der ved hjælp af en række parametre skal udregne en gennemslagskræft for kendte artikler eller medier. |
| *profile* | Et *element* i *MARS* som bruges til at holde *category* og *measure*. |
| *sub customer* | Når der tales om *sub customers*, menes der *customers* som er opdelt i et hierarki af 2 niveauer og *sub customer* er det nederste. |
| *subject* | *Element* i *MARS* som skal vise om medier/artikler indeholder bestemte metadata. |
| *tier* | Et lag i en *n-tier* model. |
| *Unified Process (UP)* | Det er en procesmetode til at udvikle software med. Beskrevet yderligere i rapporten. |

# Forord

Dette projekt er udarbejdet som et bachelorprojekt ved IMM DTU hos virksomheden:

Infopaq International A/S
Kongens Nytorv 22
1050 København K
DK Danmark

Projektet er skrevet i samme forløb med undertegnedes praktikophold i virksomheden. Projekt perioden har strakt sig fra midt i juni til sidst i august 2007. Praktikopholdet har indeholdt forskellige aspekter af virksomheden medieanalyse-afdeling, både interne og eksterne problemstillinger for produkter og produktion. Der er i praktikopholdet udarbejdet et medieanalyseværktøj til at registrere artikler/medier med. Dette værktøj kunne opsættes med kunder og profiler. Til at udføre dette har undertegnede i samarbejde med IMM DTU og Infopaq defineret hvilke rammer et sådan profilstyringsværktøj skal have.

Det meste af projektarbejdet er foretaget i virksomheden, og undertegnede takker derfor mange gange for brugen af virksomheden materialer og viden, samt den sociale omgang med kollegaer.

Projektets omfang er beskrevet i denne rapport, og det kræver ikke nødvendigvis yderligere indsigt i virksomheden for forståelse af projektet. Interne udtryk og begreber er forklaret i nomenklaturen, mens almene tekniske udtryk ikke er, fordi målgruppen af læsere er tænkt på som ligeværdige fagfolk.

Projektplanen findes i bilag 9.2 Projektplan.

# 1. Introduktion

Dette projekt er udarbejdet og implementeret hos medieovervågningsvirksomheden Infopaq International A/S. I selve praktikperioden har undertegnede været med til at designe og udvikle et internt medieanalyseværktøj hvori medieanalyseafdelingen evaluerer, validerer og redigerer forskellige artikeldata fra det eksisterende produktionssystem. Intet af kildekoden i dette projekt er taget fra eksisterende systemer. Det eneste genbrug i projektet en den obligatoriske database som er indlejret i det eksisterende *MARS*. Dette forklares senere i dette afsnit.
Det intelligente profilstyringssystem i *MARS*, som dette projekt omhandler, har hovedfunktion i at oprette og redigere *elementer* i *MARS*. Internt hos Infopaq kaldes dette projektet for *MARS Admin*.
Før dette projekts direkte indhold forklares, vil en kort gennemgang af forudsætningerne for projektet beskrives.
Dette medieanalyseværktøj, som kaldes for Media Analysis Registration System, også forkortet *MARS,* er, designet til at registrere medieanalyser. For at kunne registrere disse analyser, skal *MARS* indeholde nogle grundsten. Disse grundsten skal opsættes til hver medieanalyse-kunde i Infopaq for at kunne udarbejde en medieanalyse for den givne kunde. De såkaldte grundsten består af *elementerne customer, profiles*, *categories*, *subjects* og *measures*.
I *MARS* er disse *elementer* delt op i et hierarki, og beskrevet i underafsnittet 1.1 Forudsætninger på side 1. Denne struktur sammen med dens data er designet og implementeret i en dedikeret database internt hos Infopaq.

Dette projekt vil beskrive hvordan disse *elementer* kan redigeres og oprettes i *MARS*-database således at *MARS* kan registrere medier ud fra disse såkaldte profiler. Disse profiler er beskrevet i næste underafsnit 1.1 Forudsætninger.

Alt i denne rapport er udarbejdet af undertegnede i projektperioden hvis ikke andet er direkte beskrevet.

## *1.1 Forudsætninger*

Før analyse og design fasen starter, er der nogle forudsætninger der skal nævnes for læseren.

*Elementerne* som *MARS* logisk består af, er ordnet i et bestem hierarki. Dette hierarki er skitseret i bilag 9.3 Element hierarki. Hierarkiet er på nuværende tidspunkt fastlagt i *MARS*. De forskellige *element*typer bliver kort beskrevet i næste afsnit 2. Analyse.

*Unified Process(UP)* er en udviklingsmetode til at udvikle software med. Den udspringer fra design-notationssproget UML.
Det bygger på iterationer hvori alle aspekter af en procesudvikling gennemgås. Udviklingen af hvert aspekt i iterationerne varierer jo længere projektet skrider frem. F.eks. er der mere analyse og design i de første iterationer, mens der senere er mere implementering og test. Et alternativ kan være Scrum. Uden at komme ind på dette kan dertil siges at Scrum er kommunikationsproces for styring af flere programmerings-teams. Sammen med Scrum bruges programmeringsprocessen Extreme Programming (XP), som kort beskrevet er simpel, dynamisk og mere praktisk orienteret. Det vil sige at selv langt henne i en projektproces kan

designændringer forekomme. I modsætning til *UP*, som er en del tungere at arbejde med, skal planlægning i XP baseres på erfaring og tit med brugeren i fokus.

# 2. Analyse

Til at designe projektet, er der brugt UML[1] notation for at understøtte den før omtale *Unified Process (UP)* arbejdsmetode, hvor dette projekt vil indgå som 1. iteration. Grundet den internationale virksomhed, som Infopaq er, beskrives UML på engelsk, og dette projekt er ingen undtagelse. Både begreber og diagrammer vil være på engelsk, mens forklaringer i denne rapport vil fremgå på dansk.
Når *MARS Admin* skal analyseres, bliver det gjort på baggrund af følgende tre faktorer:

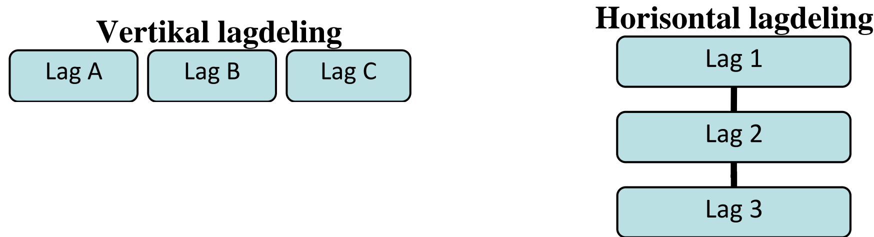
- brugen af projektet hos Infopaq
- de eksisterende systemer
- arkitektur samt den almene fremgangsmetode i virksomheden

Under disse faktorer findes argumentgrundlaget hyppigt under Objekt Orienteret (*OO)* arkitektur som design mønstre (*design patterns*) og entreprise løsninger (*entreprise solutions*), som er passende for virksomheden og dette projekts brug, både nu og i fremtidige udvidelser.
Det skal hertil siges, at brugen af disse redigeringer og oprettelser ikke nødvendigvis skal bruges dagligt i Infopaq, men skal kunne bruges af alle Infopaq's internationale medieanalyseafdelinger. Det er imidlertid sjældent at landene imellem skal redigere overlappende data, grundet det nuværende kundebehov samt virksomhedens almene procedure.

## *2.1 Opbygningen af MARS*

Det eksisterende *MARS* er opbygget som en *n-tier* struktur bestående af 3 hovedsystemer. Disse 3 systemer bliver ofte kaldt for *frontend*, *backend* og database. Ofte bliver der refereret til en *n-tier* model, når der bruges 3 eller flere lag. Der bliver i virksomheden gjort brug af denne 3-lagsmodel i flere eksisterende systemer. Det gode ved netop denne model skal findes i den fleksible distribution af de forskellige systemer og undersystemer. Med dette menes, at systemerne fysisk kan ligge på forskellige computere, men behøver det ikke. De er opdelt i horisontale ansvarsområder, hvilket vil sige, at data altid skal igennem alle lag (*tier(s)*) for at komme til målet, som kunne være fra klient til database. I en horisontal model er det let at uddele ansvarsområder, der påvirker alle oven- og underliggende systemer, da alle data skal igennem alle lag. Denne model kan ses på Figur 1 Lag-modeller. Det vil også gøre det enkelt at opdatere og ændre interne metoder som de forskellige lag bruger, fordi dette ansvarsområde kun findes én gang i en horisontal model. Det mest rammende argument i *OO* design vil uden tvivl være genanvendeligheden af de forskellige lag. Dette gør det også muligt at udskifte lag, som kunne være grundet i teknologiskift.
Derimod er en vertikal model at foretrække hvis der skal findes mange forskellige måder at operere på, hvor få eller ingen overlap er. Denne model kan også ses på Figur 1 Lag-modeller. Dette gør at de vertikale lag parallelt har adgang til mange af de samme eksterne lag, og kan derfor vælge at tilgå dem på forskellige måder.

[1] UML findes i flere versioner, f.eks. 1.0 1.5 og 2.0. Til dette projekt er der brugt version 2.0 i design værktøjet, men fordi 2.0 mest af alt bygger uden på den eksisterende notation af 1.0, kan dette projekt ses som UML 1.0 notation.




**Figur 1 Lag-modeller**

Disse modeller kan selvfølgelig også kombineres med hinanden, hvilket ofte kan være en rationel bedre løsning, grundet de forskellige lags ansvarstyper, som f.eks. at have forskellige typer af klienter (lag 1A-B-C) til den samme *backend* (lag 2) og database (lag 3). Alternativt skulle der designes nye lag til både *backend* (lag 2) og database (lag 3), for hver klient (lag 1) der implementeres.
I kombinations-modeller kan de vertikale lag endda tilgå andre forskellige horisontale lag, som også har parallelle ansvarsområder, for at få differentieret kommunikationen og logikken, således at der f.eks. findes flere måde at tilgå det samme service-lag på.

Fordi *MARS* er opbygget som en *n-tier* struktur, skal der være nogle fundamentale krav opfyldt.

- Den overordnede struktur skal være horisontal
- Systemet skal være i mere end ét lag

N'et i *n-tier* står for antallet af lag systemet og kan variere fra 1 til n, hvor n vil stå for mængden af de naturlige tal[2].

For at vise *MARS* det eksisterende systems datadesign, er database-modellen vist i bilag 9.5 *MARS* Database model, hvilken ikke er udarbejdet i projektperioden. Dette design vil ikke blive yderligere kommenteret i dette projekt, fordi dens relevans ikke er inden for projektets direkte analysefelt.

## 2.2 Analyse af MARS Admin

Fordi *MARS Admin* skal være et parallelt værktøj til det eksisterende *MARS* vil der blive taget udgangspunkt i *MARS n-tier* model som beskrevet ovenfor. Det skal til dette siges at den interne struktur i *MARS Admin*'s forskellige lag selvfølgelig individuelt bliver designet efter dens egne behov.

Der skal tænkes over hvilke funktionaliteter der er brug for i projektet, og på hvilken måde det er hensigtsmæssigt at opdele disse, i forhold til brugen af dem.
For at udnytte viden fra både brugeren og den tekniske side, er der arbejdet parallelt med UML Use Case diagrammer og udkast til brugergrænsefladen (GUI). Denne metode vil gøre det lettere at finde de Use Cases projektet skal bestå af, fordi det giver en stærkere visuel oplevelse for en bruger. Brugeren kan også lettere bemærke hvilke funktioner programmet skal kunne udføre, ved at tænke hvad der skal ske, når den interagerer med GUI. I dette projekt er denne teknik mest brugt til at overveje hvilke

[2] Notation som er almindeligt kendt fra lineær algebra.

funktioner der skal udvikles ud over basis funktionerne. Den egentlige brug af og finpudsning af GUI opbygningen er overladt til afsnit 3. Design.

Det har fra starten været meget klart hvilke basisfunktioner der, fra medieanalyse-afdelingen hos Infopaq, har været behov for, og det er disse der er taget udgangspunkt i.

Redigering og oprettelse af følgende *elementer* samt deres bindingsværdier:

- *Customer*
- *Profile*
- *Category*
- *Subject*
- *Measure*

*Bindingsværdierne* er de værdier der er specifikke for netop den binding mellem virtuelle *elementer*. Dette skal ses i forhold til det førnævnte hierarki *elementerne* har sig imellem. Fordi den eksisterende database til *MARS*, som *MARS Admin* også opererer på, indeholder *grundværdier* for de enkelte *elementer*, kan der også finde flere bindinger i hierarkiet, der udspringer fra samme *grundelement*.

| Eks. | | | |
|---|---|---|---|
| Profil2 | | Profil4 | |
| | Measure1 | | Measure1 |

I det ovenstående eksempel kan det ses hvorledes det samme *measure-element* binder på forskellige profiler. Hver af bindingerne vil indeholde individuelle *bindingsværdier*, mens *grundelementets* værdier bliver delt imellem bindingerne. Hver *element*type har forskellige typer af *bindingsværdier*.

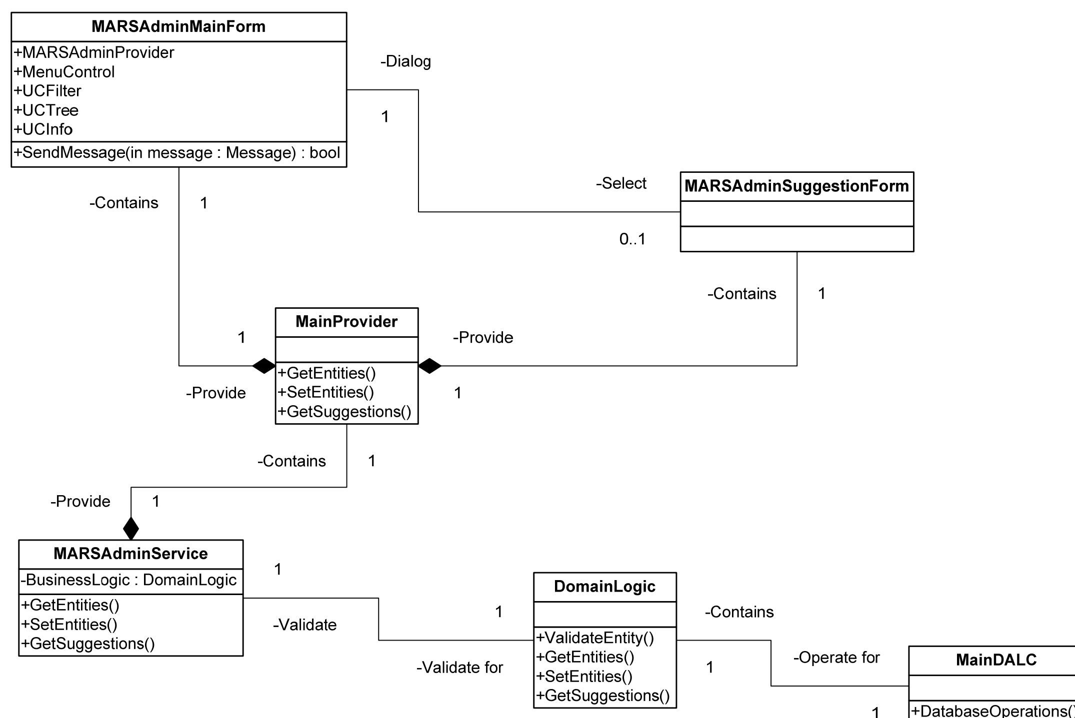
Fordi flere af de funktionaliteter *MARS Admin* skal indeholder ikke helt skal virke på samme måde som *MARS*, vil det ikke være hensigtsmæssigt at genbruge direkte modeller fra *MARS*. Alternativt skal modeller og klasser deles imellem de to projekter, hvilket vil gøre det mere vanskeligt, grundet den meget forskellige brug af data.

### 2.2.1 Domæne Model

Nedenfor på Figur 2 Domæne Model kan der ses et UML Domain Diagram, som i korte træk kan beskrives ligesom et konceptuelt Class Diagram.
Det kan på modellen ses, at den er opdelt i lag. Først vil *frontend* delen beskrives som består af de tre øverste klasser, derefter er de tre nederste klasser beskrevet som *backend* delen.
MARSAdminMainForm og MARSAdminSuggestionForm skal indeholde forskellige kontroller, der er opdelt efter funktionalitet. I dette tilfælde er der en kontrol, UCTree, som skal fungere som hierarkisk fremvisning og *element*-vælger. UCInfo er den kontrol der skal vise den detaljerede information for det givende *element* samt dens *bindingsværdier*. Disse to Forms vil holde en MainProvider, der skal kommunikere videre ned til det underliggende lag i den såkaldte *n-tier* model.
*Backend* delen vil bestå af den obligatoriske service MARSAdminService. *Backend* lagets primære mål er at kommunikere med databasen, men for at dette kan håndteres tilfredsstillende, har det fra tidligere projekter, som *MARS*, været en fordel at styre

disse databaseforbindelser med noget logik. Denne logik kan både operere på data, men mest af alt på at styre de forskellige forbindelser i den sidste konceptklasse MainDALC. *DALC*, som er datatilgangskomponenten, er den direkte operator på databasen.

De stærke bindinger der f.eks. findes mellem MainProvider og MARSAdminService, skal vise at MainProvider faktisk ikke kan eksistere uden en MARSAdminService, da det ikke vil give mening uden.
Denne domænemodel skal fremvise de mere obligatoriske punkter som er konkluderet ud fra den førnævnte beskrivelse af *n-tier* modellen. MARSAdminSuggestionForm er indsat i denne model, fordi der på en eller anden måde skal forekomme et valg fra brugeren ud af eventuelle intelligente valideringer.




**Figur 2 Domæne Model**

## 2.2.2 Use Case

For at opbygge et grundlag for hvilke funktioner *MARS Admin* skal understøtte, og hvilke den ikke skal, bør der ses på hvilke funktionaliteter der oftest er brug for ifølge medieanalyseafdelingerne i virksomheden. De tidligere nævnte basisfunktioner er obligatoriske for systemet. Ud over disse funktionaliteter, findes der også et generelt problem i de gamle registreringssystemer, som ikke er *MARS*, at grund*elementerne* kan blive oprettet flere gange med variationer i evt. navnet. Det er på grund af dette problem, at den intelligente profilstyring kommer ind i billedet. For at undgå denne redundans og misvisning samt den meget svære dataanalyse, er datavalidering altid et godt våben.
Der er selvfølgelig mange måder at designe og implementere intelligens på. Projektets spinkle omfang af kunstig intelligens gør, at simplere metoder skal tages for øje. Ydermere skal der også konkluderes ud fra, at der er valgt en *n-tier* model med

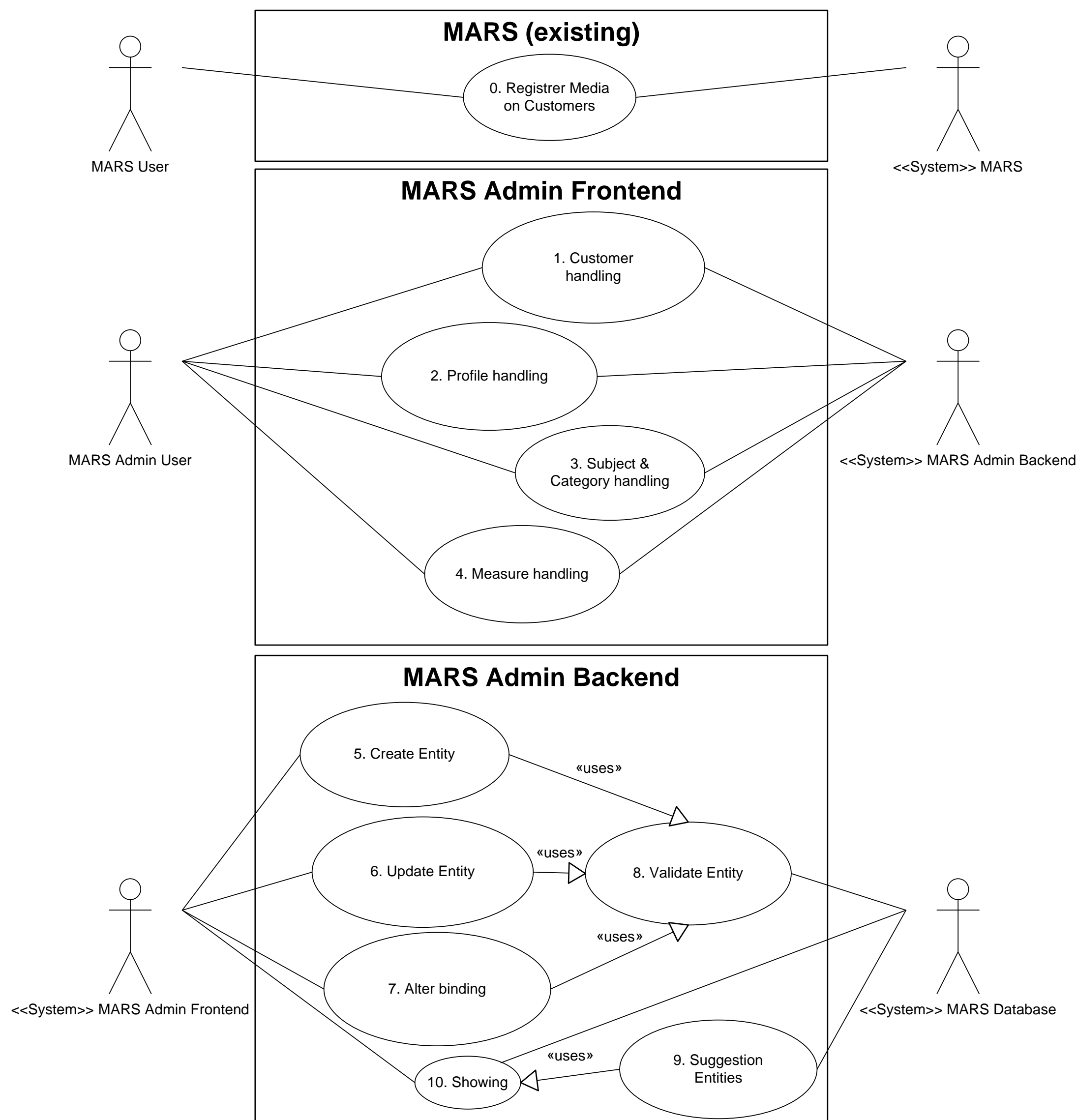
indeholdende *OO* design, vil det være relativt let at udskifte procedurerne for de intelligente grundlag i projektet.

For at beskrive de Use Cases der er udarbejdet fra førnævnte grundlag, er de delt op i henhold til den beskrevne *n-tier* model i en *frontend* og en *backend*.
Der er kun taget udgangspunkt i enkelte, men udvalgte diagrammer og skemaer for at give en mere simpel forståelse af *MARS Admin*. Alle Use Case diagrammer findes i bilag 9.6 UML Use Case Diagrammer.
Der er ikke gjort direkte brug af selve Use Case skemaer, grundet den meget omstendige procedure, men mest af alt, fordi størstedelen af informationerne skal findes i funktionaliteter og opbygning, hvilket de fleste modeller, der er gjort brug af kan give. Der vil f.eks. senere blive beskrevet sekvensdiagrammer, hvilket også ville have udgjort en større del af et Use Case skema.
Nogle af de ting et Use Case skema kunne have tilføjet, kunne være prioritet, alternativt flow (branching), og evt. kritiske udfald.

**Topmodel**
Først en kort gennemgang af opdelingen og hvorfor netop denne opdeling.
På Figur 3 Use Case Diagram Top Level, kan ses 3 systemer:

- Eksisterende MARS, som ikke er direkte beskrevet her.
- MARS Admin Frontend, dette repræsenterer klient applikationen, som MARS Admin bruger til at opsætte kunder med. Denne består at forretningslogikken.
- MARS Admin Backend, dette system er selve databehandlingen, samt data logikken.

.




**Figur 3 Use Case Diagram Top Level**

For at kunne navigere lettere rund i Use Case diagrammerne, er der givet numre og bogstaver til hver af de viste Use Cases. Dette er gjort fordi antallet af Use Cases er relativt stort, og derfor opdelt i underdiagrammer. Denne nummerering skal gøre det lettere at se hvor og hvad der bliver refereret til. I diagrammet, som ses på Figur 3 Use Case Diagram Top Level, er Use Cases nummereret med tal. I underdiagrammerne bliver der brugt bogstaver således at 1. Customer handling, fra top modellen indeholder et underdiagram med nummereringen 1A, 1B osv.

På Figur 3 Use Case Diagram Top Level skal der lægges mærke til at aktørerne både fremstår som aktører, men også som system-rammer. Både *frontend* og *backend* bliver visualiseret på denne måde. Notationen er både understøttet af UML 1.0 og 2.0. Fordi det viser flere abstraktioner af samme aktør/system i samme diagram, kan det vise hvordan de forskellige lag kommunikerer internt og at de har den samme fælles grænseflade til de omkringliggende aktører/systemer.

Fordi UML værktøjer der er udlevet i virksomheden ikke understøtter forskellige notationer for humane aktører og system aktører, er der brugt en litterær notation <<System>> ved system aktører.

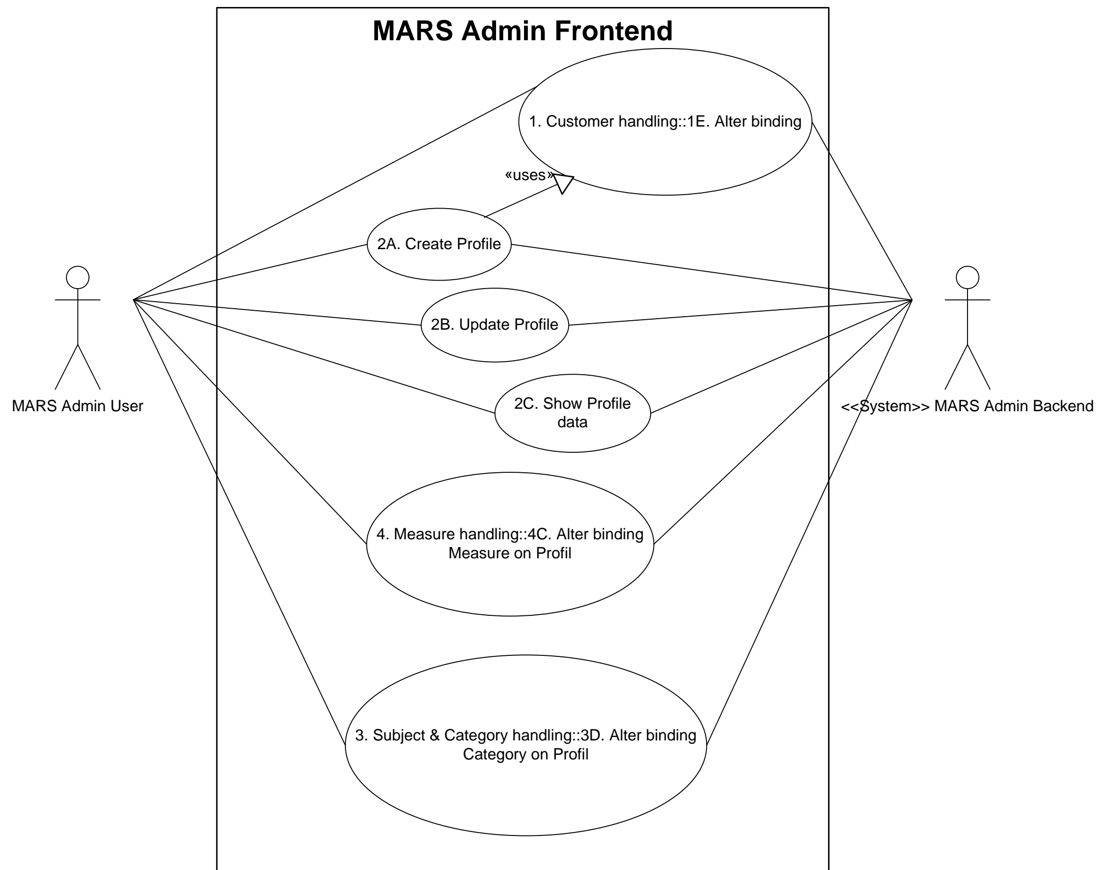
En kortfattet punktbeskrivelse af de enkelte topmodel-Use Cases som ses på Figur 3 Use Case Diagram Top Level.

0 Eksisterende *MARS* (ikke beskrevet yderligere her)
1. Customer handling, omhandler emner for opsætning af *customer*. vist i bilag 9.6.2 Use Case Diagram 1. Customer handling.
2. Profile handling, omhandler emner for opsætning af *profile*, vist i bilag 9.6.3 Use Case Diagram 2. Profile handling.
3. Subject & Category handling, omhandler emner for opsætning af *subject* og *category*, vist i bilag 9.6.4 Use Case Diagram 3. Category and Subject handling.
4. Measure handling, omhandler emner for opsætning af *measure*, vist i bilag 9.6.5 Use Case Diagram 4. Measure handling.
5. Create Entity,oprettelse af *elementer* som beskrevet i MARS Admin Frontend top level punkt 1-4, vist i bilag 9.6.6 Use Case Diagram 5. Create Entity.
6. Update Entity, opdater *elementer* beskrevet i MARS Admin Frontend til level punkt 1-4, vist i bilag 9.6.7 Use Case Diagram 6. Update Entity.
7. Alter Binding, rediger *bindingsværdier*, vises i bilag 9.6.8 Use Case Diagram 7. Alter Binding og er yderligere beskrevet i under diagrammer, eks. 1E, 4C og 3E. som henholdsvis er vist i bilag:
   a. 9.6.2 Use Case Diagram 1. Customer handling
   b. 9.6.5 Use Case Diagram 4. Measure handling
   c. 9.6.4 Use Case Diagram 3. Category and Subject handling
8. Validate Entity, validering af *elementer*, skal finde mulige fejl og mangler ved entiteter, fx. dubletter eller lignende (også om data er korrekte, hvis dette er nødvendigt). På dette sted kunne håndteringen af data overensstemmelses problemer også håndteres, dette emne bliver omtalt i afsnit 2.2.3 Data Overensstemmelsesproblemer.
9. Suggestion Entities, find forslag til et givent *element*, grundet bindinger, navn, forekomster eller lignende. Denne Use Case er en grundsten i den intelligente profilstyring. Underdiagrammet kan ses i bilag 9.6.10 Use Case Diagram 9. Suggestion.
10. Showing, visning af *elementer* samt resultaterne fra 9. Kan ses i bilag 9.6.11 Use Case Diagram 10. Showing.

**Frontend**
For at tage udgangspunkt i hvorfor Use Case diagrammerne er opbygget på netop denne måde, skal man betragte hvilke top model Use Cases der findes i *frontend* systemet, på Figur 3 Use Case Diagram Top Level. Her kan ses de 5 *element*typer der blev nævnt i starten af dette kapitel. Hver af disse *element*typer har fået sit eget Use Case diagram, hvilket gør det mere overskueligt at finde forskelle *elementerne* imellem. Der er, som der også blev nævnt i starten af dette kapitel, et meget stort overlap i funktionaliteten af de forskellige *elementer*. Dette betyder at der kommer en stor redundans af ensartede Use Cases. Selvom det giver en del mere arbejde at beskrive disse ensartede Use Cases, bliver det tydeligere at se, om der skal forekomme funktionaliteter *element*typerne imellem, hvorpå disse skal differentieres.

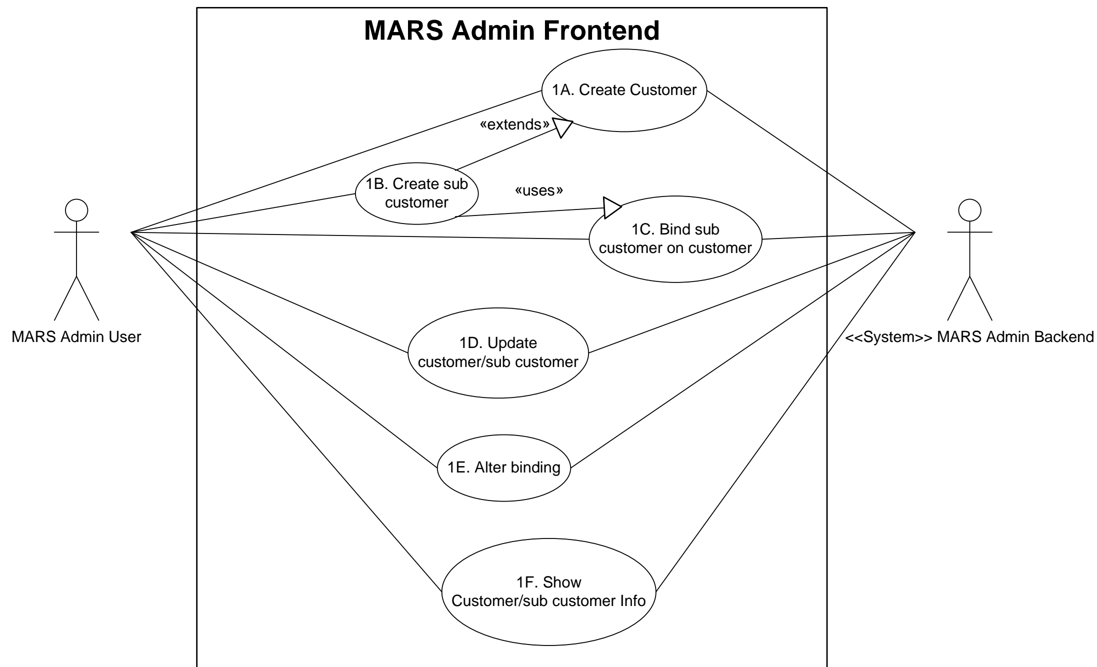
Alternativt til denne opdeling kunne alle *element*typernes Use Cases være opdelt efter hvilke operationer de skulle udføre. Denne opdeling har fordele i at reducere redundansen i Use Cases, men gøre det sværere at se forskelle *element*typerne imellem, ikke des mindre muligt.

Mest af alt giver den brugte model et bedre overblik over de små forskelle *element*typerne har, hvilket f.eks. kan ses via deres associationer. Dette kan ses på f.eks. Figur 4 *Profile* håndtering og Figur 5 *Customer* håndtering. Begge diagrammer gør brug af basisfunktionerne som beskrevet tidligere. Det er også tydeligt at se at begge diagrammer har yderligere Use Cases og bindinger til disse som f.eks. *sub customer* Use Cases 1B og 1C på Figur 5 *Customer* håndtering. Der gøres endda brug af <<extends>> associationen for Use Case 1B fra 1A.
Derudover har diagrammet på Figur 4 *Profile* håndtering endnu en forskel som skal ses på Use Case 3 og 4. Der findes nemlig flere bindinger på en profil end andre *element*typer.




**Figur 4 *Profile* håndtering**

- 1E. beskrevet i bilag 9.6.2 Use Case Diagram 1. Customer handling, skal forekomme når der oprettes en ny *profile*, således en *profile* altid binder på en *customer*.
- 2A. Oprettelse af en *profile*.
- 2B. Opdatering af en *profile* (kan inkludere 1E).
- 2C. Visning af de forskellige *profile* data.
- 4C. beskrevet i bilag 9.6.5 Use Case Diagram 4. Measure handling.
- 3D. beskrevet i bilag 9.6.4 Use Case Diagram 3. Category and Subject handling..

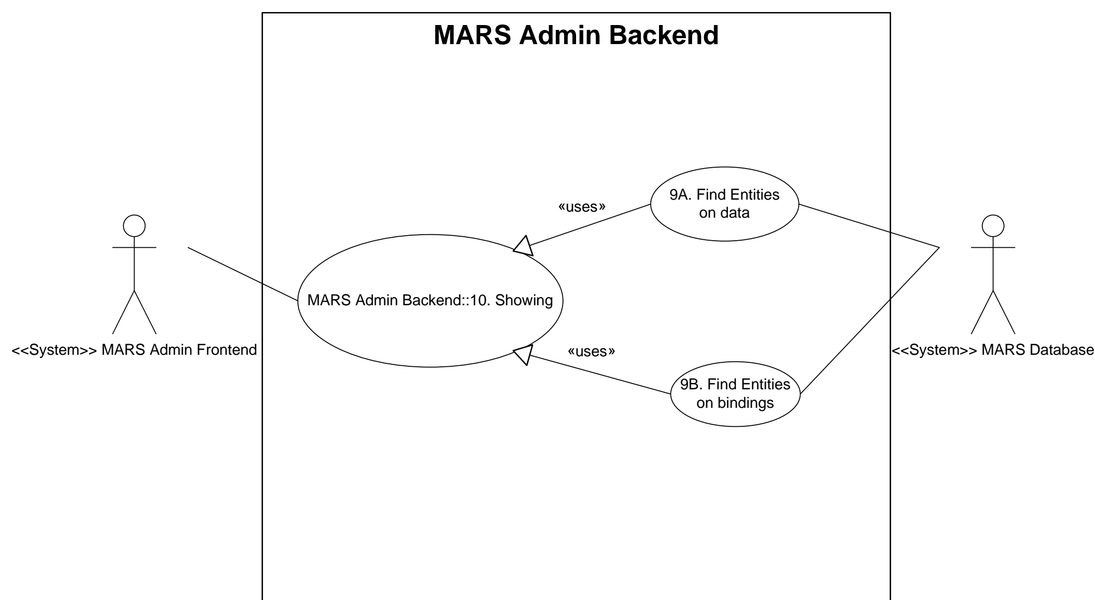



**Figur 5 *Customer* håndtering**

- 1A. Oprette en *customer* og systemet opretter den så videre ned i systemet.
- 1B. Udvidelse af 1A. som opretter *customer* som *sub customer*. Kan oprette *sub customer*. (samme som kunde bare med en forælder).
- 1C. Bliver altid brugt når en *sub customer* 1B. oprettes. Kun 1 binding (ikke virtuel). Binder *sub customer* på en eksisterende *customer*.
- 1D. Opdatere *customer*. (kan inkludere 1C)
- 1E. Håndtering af bindinger fra en *customer* til en *profile*. En *profile* kan binde på flere *customers*, men bliver ikke brugt. (kan inkludere 1C).
- 1F. Visning af *customer* data, så *MARS Admin* kan håndtere disse.

**Backend**
I denne del af systemet, gøres der brug af den alternative måde at opdele Use Cases i, som omtalt i afsnittet ovenfor. Diagrammerne er opdelt efter hvilke operationer systemet skal udfører, og ikke efter *element*typerne. Dette gøres fordi *backend* systemet mere skal fungere som en service for andre systemer, og derfor er det vigtigt at skitsere hvilke operationer der skal udføres og dermed hvilke Use Cases disse skal gøre brug af. Dette kan også være medhjælpende til at bestemme og designe klasse-diagrammet for *backend* systemet som beskrevet i afsnit 3.1.3 Class Diagram. Hele argumentet for valget af denne model type skal ses i opbygningen af data-gennemstrømningen. Disse diagrammer skal hjælpe med at vise hvilke operationer, der skal kunne udføres, men mest af alt hvordan de er associeret.

På Figur 6 Suggestion, vises to ensartede operationer, og deres date-gennemstrømning i systemet. De gør begge brug af den samme Use Case 10, for at komme videre til *frontend* systemet.

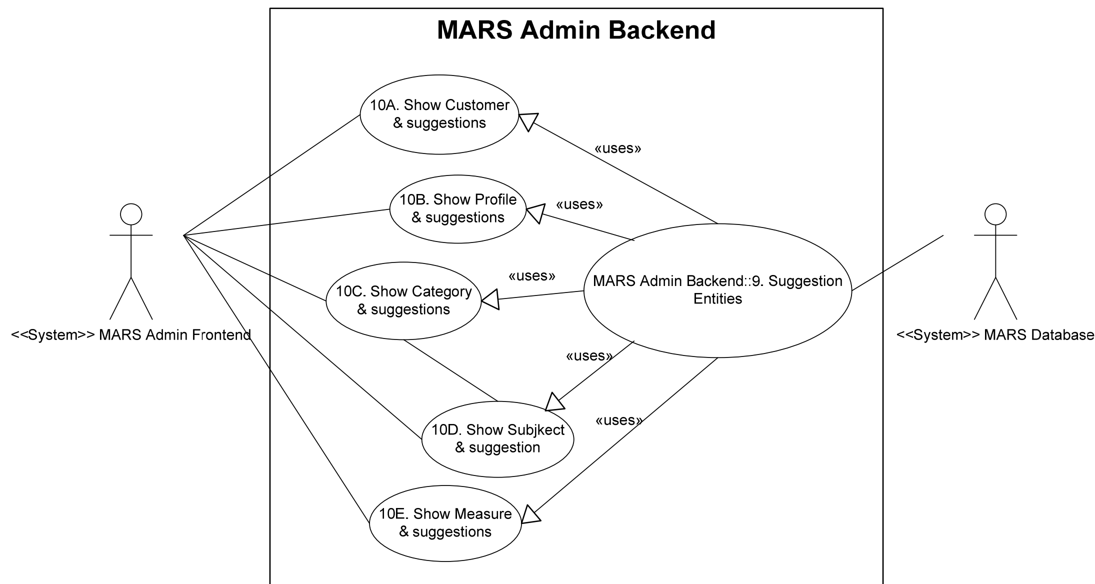



**Figur 6 Suggestion**

Dette diagram viser måderne hvorpå systemet skal finde lignende entiteter der skal foreslås til visning videre op i systemet.

- 9A. Find *elementer* der har lignende dataværdier.
- 9B. Find *elementer* der har lignende bindinger, som fx. er i samme *category*, eller findes i samme *profile* eller *customer*
- 10. Visning bruges altid af 9x. - og viser resultatet videre til *frontend*. Beskrevet i bilag 9.6.11 Use Case Diagram 10. Showing.

Også på Use Case 10 (underdiagram til Figur 6 Suggestion Use Case 10), som vist på Figur 7 Showing, vises hvordan alle de forskellige *element*typers Use Cases er samlet. Sættes disse to diagrammer sammen, kan det tydeligt ses, hvorledes data strømmer fra databasen, op igennem *backend* systemet og videre til *frontend* systemet, hvilket også er hensigten.




**Figur 7 Showing**

Dette diagram skal tolkes som et valg mellem de forskellige <<Uses>> bindinger, og ikke at alle <<Uses>> bindinger er aktive på samme tid.
Højre side er taget fra Use Case 9 som vist i bilag 9.6.10 Use Case Diagram 9. Suggestion, og skal ses som en samlet Use Case for at skabe et bedre overblik.

- 10A. Visning af *customer elementer* samt eventuelle forslag.
- 10B. Visning af *profile elementer* samt eventuelle forslag.
- 10C. Visning af *category elementer* samt eventuelle forslag.
- 10D. Visning af *subject elementer* samt eventuelle forslag.
- 10E. Visning af *measure elementer* samt eventuelle forslag.
- 9. beskrevet i bilag 9.6.10 Use Case Diagram 9. Suggestion.

Som før nævnt er de resterende Use Case diagrammer i bilag 9.6 UML Use Case Diagrammer.

### 2.2.3 Data Overensstemmelsesproblemer

Der findes mange måder at løse overensstemmelsesproblemer (*concurrency issues*) på. Emnet er så stort i sig selv, at det kunne fylde hele projektet. Grundet projektets mål, vil emnet kun blive analyseret, og rationelt set ville det henholde projektets mål. Grundet til at analysen stadig forbliver en del af projektet, skal findes i *MARS* kundedatabasens stigende datamængde, og dermed også en fremtidig øgelse af *concurrency issues*.

Der findes umiddelbart 2 former for overensstemmelseskontrol (*concurrency control*); optimistisk og pessimistisk.

**Optimistisk**
Den optimistiske bygger på, at alle kan få deres egen kopi af data hvori der er information om hvilken version det er på læsningstidspunktet. Derefter kan alle prøve at gemme data, men kun den der kommer først vil kunne gøre dette, fordi den originale version stadig er ens med kildens. Det vil ikke lade sig gøre fordi de resterende kopier, grundet den nye version i kilden og den gamle version i kopierne, som de resterende har, ikke er ens.

**Pessimistisk**
Den pessimistiske er kort, simpel og er bygget på at data bliver meldt ud, og dermed låst så andre ikke kan få adgang til disse. Data vil først blive lukket op, når den pågældende bruger indikerer at data er meldt ind igen og ikke bruges mere.

Generelt set bygger de henholdsvis på konflikt-opdagelse og konflikt-nægtelse.

Begge typer *concurrency issues* har fordele og ulemper.
Den optimistiske model gør data let tilgængeligt og med god mulighed for brugere at arbejde med alle dataelementer. Derimod kan det skabe versionskonflikter og disse skal der tages højde for, ved f.eks. sammensmeltning af data eller i værste tilfælde forkastelse.
Den pessimistiske model gør adgang til data besværlig og andre brugere må vente til data igen bliver tilgængelig. Det kan derimod være fornuftigt at bruge den pessimistiske model, grundet data-konfliktproblemerne er fjernet.
Det vil være logisk at bruge optimistiske modeller når *concurrency issues* er sjældne, fordi det giver data bedre frihed til at flyde rundt i systemet. Pessimistiske modeller kan bedre bruges i situationer hvor der forekommer mange *concurrency issues*, hvor den kan være med til at forhindre disse.

I *MARS Admin*, som er et flerebrugersystem, findes *concurrency issues* også. Det er imidlertid ikke særligt sandsynligt at *concurrency issues* er af større betydning, fordi systemet ikke vil blive brugt så frekvent og kun af få brugere. Dette gør det umiddelbart bedst at løse *concurrency issue* med en optimistisk model, grundet den sjældne konflikt. Det kan derimod blive svært at sammensmelte data fra flere *elementer*, fordi de består af f.eks. booleske værdier, decimal værdier, navne og beskrivelser og nøgleord. Talværdierne er ikke umiddelbart lette at sammensmelte, fordi de enten skal være det ene eller det andet og ikke et gennemsnit eller en anden form for relativ beregning. Navnet skal give mening, og vil som værdierne også baseres på den ene eller den anden version af data for at give mening.

| Eks. | |
|---|---|
| Ver. 1: | Ver. 2: |
| Navn: Fjernsyn | Navn: TV |

Det kan ikke umiddelbart ses hvilken en version der skal gemmes. I dette tilfælde skal der mere information til, i form af regelsæt for sammensmeltningen. I dette eksempel er betydningen ens, men det kan være navnet også skal læses at englændere. Dermed kan en regel være at slå op i en ordbog og finde det navn der lyder mest sandsynligt for det gældende sprog. Der kunne også gøres brug af semantiske ordbøger til bestemmelse af ordenes betydning, for dermed at analysere hvorledes ordene var

relateret til hinanden og deres betydning, hvorom der kunne konkluderes hvilke ord der kunne være relevante at smelte sammen. I ovenstående eksempel ville en semantisk ordbog kunne genkende versionernes type til at være overlappende, og dermed ved en forudbestemt regel; overskrive eller forkaste ordet.

I *MARS Admin* projektet ville dette være for stor en opgave i 1. iteration, og dermed ikke være en fornuftig løsning i skrivende stund.

Til en senere iteration vil der relativt let kunne implementeres en simpel *concurrency control*.
Den letteste løsning ville helt klart være den pessimistiske model, hvor data vil blive låst, da dette kan gøres ved hjælp af et flag i databaserækkerne. Dette flag skal sættes inden data læses, og skal huskes at sænkes efter data er skrevet.
Skal den optimistiske model implementeres kan der gøres brug af hash-funktioner som rækkefelter i databasen. Disse hash-værdier skal genereres på baggrund af rækkens yderligere værdier og dermed virke som en slags versionsnøgle, der kunne læses af alle. Denne hash-nøgle skal gemmes i original form når data læses. Før den nye data gemmes, kan den originale hash-nøgle valideres op imod den originale på den pågældende tabelrække. Dette kan sikre at kun den første kan gemme data, af dem der havde læst den originale hash-nøgle, hvorefter den nye gemte hash-nøgle vil overskrive den originale i databasen.

### 2.2.4 Resumé

Der bliver gjort brug af en *n-tier* model, som set udefra ligner det eksisterende i *MARS*. Opdelingen af konceptuelle klasser som ses på Figur 2 Domæne Model, skal vise at kommunikationen mellem klasserne skal ske via ensartede mønstre, hvilket understøtter en horisontal model.
Use Cases er beskrevet via diagrammer som senere suppleres med blandt andet sekvens diagrammer.
Opdelingen af Use Case diagrammerne er differentieret efter hvilken information der skal indsamles i de individuelle systemer (*frontend* og *backend*).
*Concurrency control* er ikke umiddelbart rationelt at designe i 1. iteration, og analysen vil derfor kun være beskrevet og ikke designet videre på. I analysen er der dog beskrevet flere måder at løse problemet på.

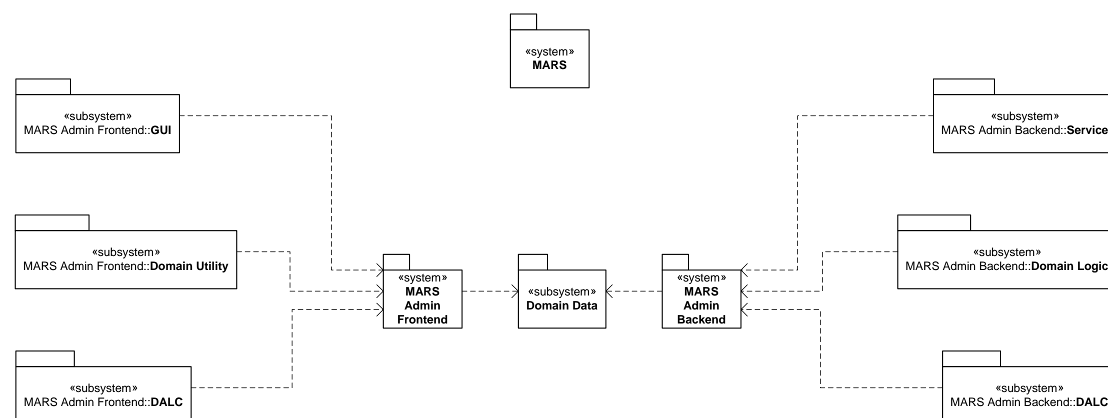
# 3. Design

Til at beskrive *MARS Admin* designet, vil der blive gjort brug af kendte *design patterns* og diagrammer i form af UML.
Der vil blive gennemgået en række konceptuelle klasse-diagrammer og deres sammensætning. Dette bliver gjort i sammenhæng med generelle design funktionaliteter i forbindelse med implementeringsovervejelser i generelt *OO* programmering.
For at kunne implementere dette design er der også lagt op til et visuelt GUI design. Dette design bliver omtalt længere nede i dette afsnit.

## *3.1 UML Design*

### 3.1.1 Use Case Package opdeling og *n-tier* model

Øverst på Figur 8 Package og System Diagram er <<system>> *MARS* sat for sig selv. Det resterende diagram viser opdelingen af *MARS Admin*, som her er opdelt efter logiske packages og subsystems. Diagrammet er vist på <<subsystems>> niveau, grundet flere af de viste <<subsystems>> deler flere packages, og <<subsystems>> opdelingen viser bedre den interne struktur, der videre skal bruges til implementeringen. En anden, men mindre grund, er også at det brugte UML værktøj ikke let kan visualisere delte packages i <<system>> objekter.
Hver af <<system>> objekterne på figuren, dækker over én eller flere packages.
Der er i dette design lagt op til at hvert af <<system>> objekterne, *frontend* og *backend*, er yderligere opdelt i <<subsystems>>. Denne interne opdeling skal hjælpe med at strukturere ansvarsområder, som ikke nødvendigvis kan forklares ud fra Use Case diagrammernes opdelingsmodeller fra afsnit 2.2 Analyse af MARS Admin. Denne opdeling har en anden abstrakt vinkel, som bygger på en mere overordnet lagdeling af *n-tier* modellen, men også af de fælles objekter/*elementer* systemerne har imellem sig.
Fordi opbygningen af *n-tier* modellen er en fundamental brik i designet, er dette diagram endnu en skridt tættere på abstraktionen og forståelsen af en *enterprise solutions* struktur. Det er på diagrammet let at se hvad systemerne deler imellem sig, og derfor også lettere at bruge i implementering øje med.

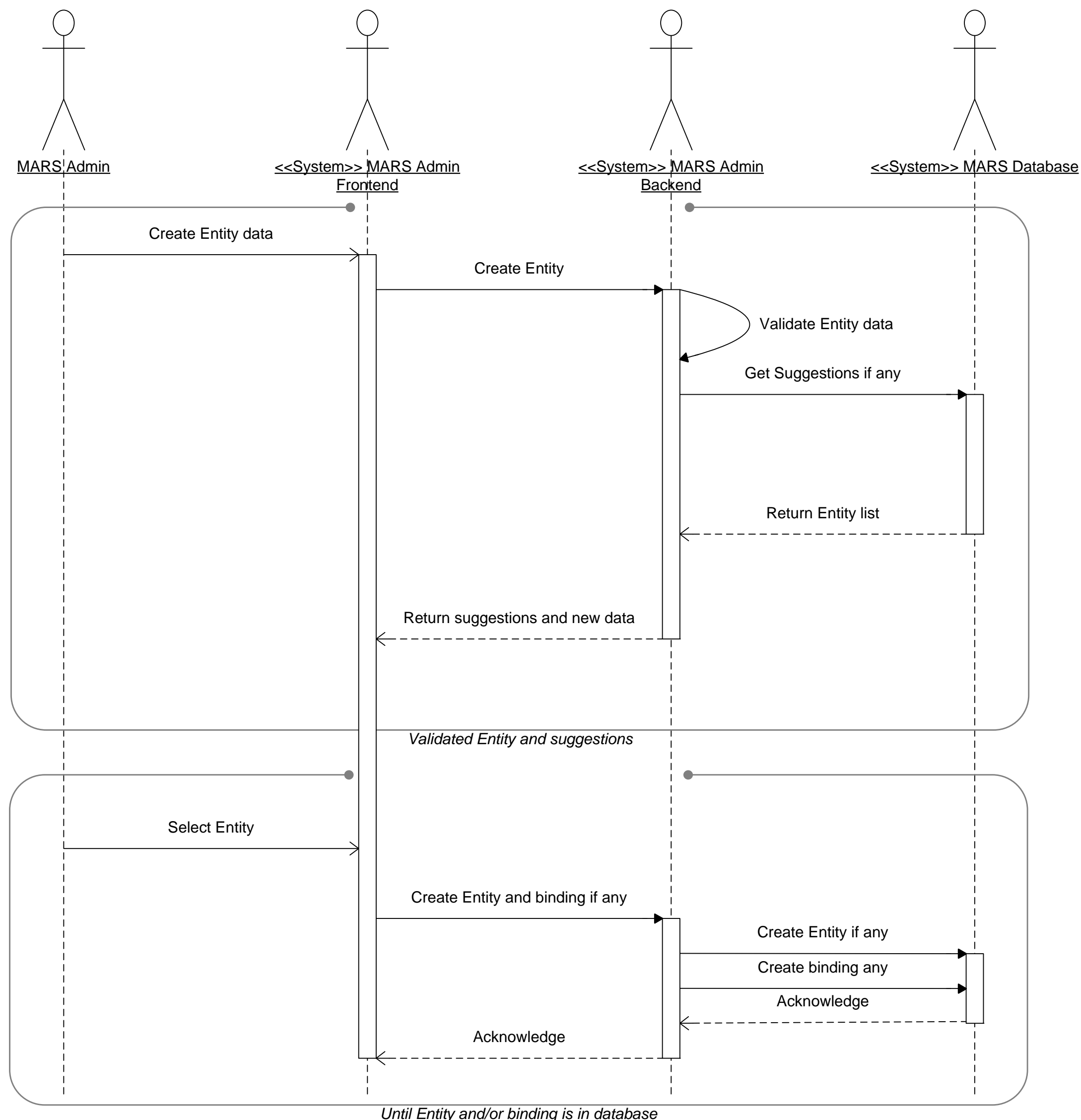



**Figur 8 Package og System Diagram**

Figur 8 Package og System Diagram viser i venstre siden *frontend* delen, som består at et internt GUI lag. Sammen med dette lag ligger <<subsystems>>, *DALC*, som skal sørge for at kommunikere med *backend* delen. Til at gøre dette skal der bruges forskellige værktøjer for at konvertere service formater. Dette er en meget kendt problemstilling fordi WebServices objekter ikke almindelig vis kan bruges uden for service-laget, som i dette tilfælde er i *frontend* delen. WebServices vil ikke eksponere metoder, men kun variable, og derfor skal de konverteres til lokale versioner af klasser. Dette er også grunden til at begge systemer, *frontend* og *backend*, skal dele dataklasser og typer, der skal kommunikeres imellem disse. På denne måde gøres det sikkert at objekttyper og egenskaber er ens i begge systemer.

### 3.1.2 Sequence Diagram

For at opnå en forståelse af hvordan data skal flyde igennem systemerne, kan der gøres brug af UML sequence diagrammer. Fordi disse diagrammer er beregnet til at få en forståelse af hvad og hvordan data skal flyde igennem systemerne, er der lavet nogle få og udvalgte diagrammer for at se datagennemstrømningen i større træk for at opnå opdelingen af det videre design af klasserne og deres funktionaliteter. Det er f.eks. vigtigt at se hvordan *elementer* bliver oprettet i systemet, fordi dette er en grundlæggende procedure, hvor alle systemerne kommer til sin ret på det konceptuelle niveau.




**Figur 9 Sequence Diagram opret *element***

Dette diagram viser sekvenserne som bliver foretager ved oprettelse af et *element*. Diagrammet viser en abstrakt sekvens af hvordan oprettelser af *elementer* skal ske i *MARS Admin* systemer. Aktørerne, der bliver brugt, kan genkendes fra tidligere brug i afsnit 2. Analyse.

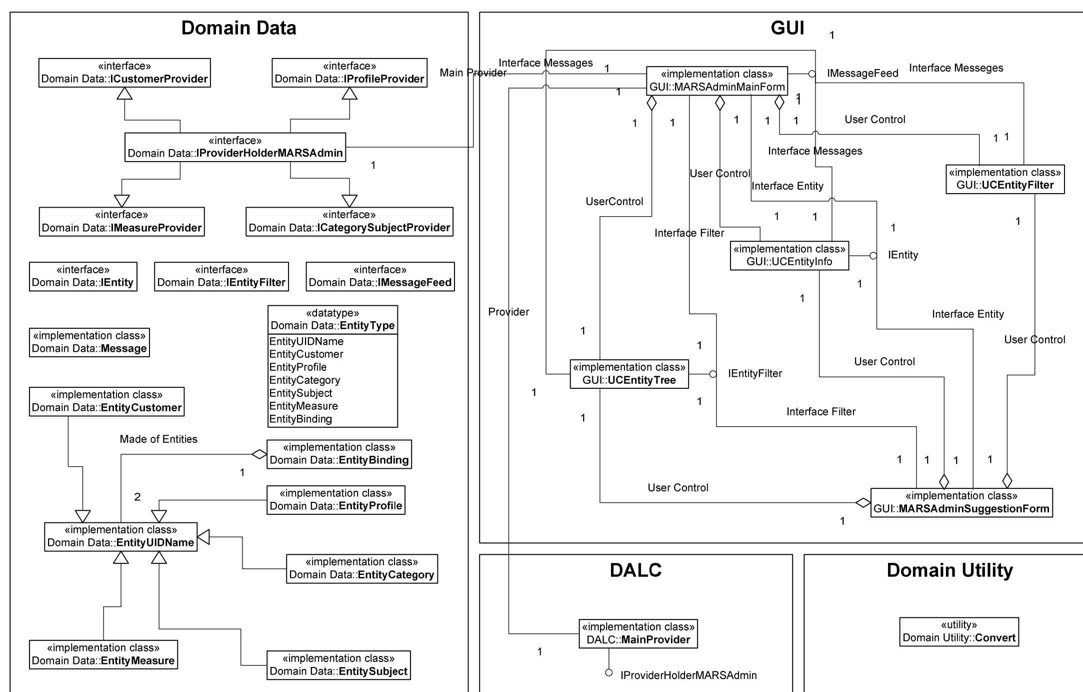
- Øverste sektion viser hvordan de rigtige *element*data vælges.
- Der bliver indtastet *element* data af *MARS Admin* bruger i *MARS Admin frontend*, hvorefter systemet forespørger *MARS Admin backend* om data er valide.
- *MARS Admin backend* spørger *MARS* databasen om *elementer* af lignende form, for ikke at oprette *elementer* der allerede findes i systemet.
- I *MARS Admin frontend* vil vise *elementer* og evt. data i forhold til deres indbyrdes bindinger, således det tydeliggøre *elementer* fra hinanden.
- Denne ovenstående sekvens er indkapslet som en færdig sekvens, men for at *elementet* bliver oprettet skal nedenstående sekvens også gennemløbes.

- Når *MARS Admin* bruger har valgt en foreslået eller det nye *element*, kan der oprettes et *element* i databasen alt efter *element*typen. Dette vil gøre forskellen om *elementet* bærer en binding eller om det kun er et ny virtuel *element* fra et allerede eksisterende *element*. I dette tilfælde vil det kun være dindingen der bliver oprettet.
- *MARS Admin frontend* vil oprette et *element* samt bindingen til det evt. overliggende *element*, hvis dette er nødvendigt. Begge oprettelser skal valideres, og fejler én skal forrige oprettelser eller redigeringer fortrydes.
- *MARS Admin backend* opretter *elementet* og binding via databasen, som bekræfter begge operationer, og ligeledes bekræfter *backend* til *frontend* således *MARS Admin* bruger ved at handlingen er sket korrekt.

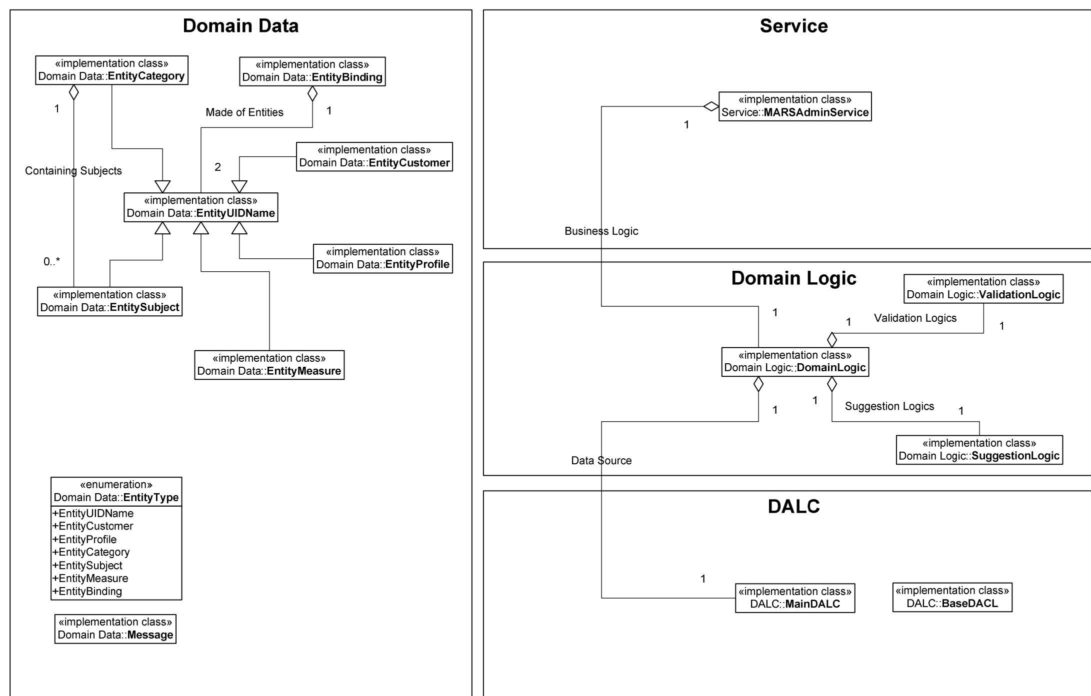
De resterende sequence diagrammer er vedlagt i bilag 9.7 UML Sequence Diagrammer.
Det kan på diagrammet ses tydeligt at der skal flere forbindelser til databasen, når der skal oprettes *elementer*. Dette kan skabe problemer, grundet evt. fejl under én eller flere af oprettelserne. I sådanne tilfælde vil dette føre til fejldata i databasen, fordi partielle data vil forefindes. Disse datafejl skal kunne forhindres, lige meget om der grunden til fejlen er grundet i datafejl eller forbindelsesfejl mm. Den direkte implementeringsløsning vil blive beskrevet i afsnit 4. Implementering.

## 3.1.3 Class Diagram og design patterns

For at vise hvordan konceptet i klasserne skal foreligge, vil en gennemgang af *frontend* og *backend* delen først blive gennemgået.
For at opridse de to systemers overordnede hensigter skal der refereres til Figur 10 Class Diagram Frontend Koncept og Figur 11 Class Diagram Backend Koncept.




**Figur 10 Class Diagram Frontend Koncept**




**Figur 11 Class Diagram Backend Koncept**

Disse to diagrammer har forskellige opbygninger, som nævnt før i afsnit 2.2 Analyse af MARS Admin. Dette er gjort klart i henhold til deres funktion i hele *MARS Admin* systemet.
*Design pattern* som er en stor del af designet og implementeringen, vil blive beskrevet herunder. *Design patterns* er meget vigtig for en *enterprise solution*, fordi funktionaliteter tit er spredt ud forskellige steder i virksomheden og skal let kunne flyttes rundt, sammensættes med lignende projekter eller kunne arbejde med andre udvidelser. Hvis der bliver beskrevet og implementeret *design patterns* vil der tit kunne arbejdes simultant på projekter af flere personer, hvilket tit kan være en fordel. Det kan også lettere ses i det abstrakte billede, hvordan bestemte områder af systemet skal og vil opføre sig. Det kan ydermere give en lettere forståelse til samarbejdspartnere der er involveret i systemets adfærd.

**Frontend**
På Figur 10 Class Diagram Frontend Koncept kan ses struktur fra GUI til *DALC* med sideliggende domain dataklasser og en static utility klasse. Denne konstruktion er designet efter *design patterns* som;

- Fundamental patterns
  - Proxy pattern
  - Functional pattern
- Structure patterns
  - Composite pattern
  - Adapter pattern
- Creational petterns
  - Lazy initialization pattern
  - Builder pattern

Hensigten med denne del er at styre visuelle data på en struktureret og informativ måde. Grundet dette er de brugte *design patterns* forklaret herunder.
Alle disse *design patterns* er valgt for det overordnede princip efter Functional pattern, som er en Fundamental pattern type, hvilket vil sige, at hver moduldel har sit helt eget ansvarsområde, og prøver at holde sig til dette.
Dette *design pattern* bliver eksempelvist brugt i domain data området, hvor hver Entity-klasse holder på sine egne data og vil holde styr på om data er blevet redigeret siden oprettelsen af objektet, eller hvordan objektet bliver repræsenteret i forhold til objektets indhold.

Proxy pattern, som også er et Fundamental pattern, bliver brugt i sammenhæng med de Interfaces som skal sørge for, at de forskellige GUI klasser kan kommunikerer med hinanden. Proxy pattern går kort sagt ud på at give Interfaces via klasser til en anden klasse, således en mere simpel abstraktion er givet. Dette *design pattern* er valgt fordi det overordnede *design pattern* påskriver en stram ansvarsdeling, og for at forsikre en forudbestemt kontrakt klasser imellem.

Composite pattern er et Structure pattern, som i korte træk sørge for, at klasser i samme træstruktur implementerer samme Interface. Dette vil medføre at alle klasser i dette hierarki kan bruges til samme Interface kontrakt. Dette er især vigtigt når dette skal bruges med Proxy pattern, som nævnt ovenfor.

Lazy initialization pattern som tilhører Creational patterns, sørger for at objekter og data kun bliver kalkuleret eller hentet, når der er brug for disse. Dette pattern er tænkt til de dele, der skal kommunikere med *backend* delen. Dette skal sørge for at kommunikationen kun skal forekomme på et minimum af data for ikke at belaste *backend* systemet unødigt.

Builder pattern er også et Creational pattern, hvilket er skabt til at separere konstruktionen af komplekse objekter fra dens repræsentation, således at den sammen konstruktionsproces kan bygge forskellige repræsentationer. Netop dette *design patter* er valgt fordi alle *elementerne* skal vises og redigeres i GUI-delen, hvilket giver meget overlappende GUI-komponenter, grundet de meget ens værdityper, men alle *element*typer har tilføjet individuelle og forskellige værdier til sig. Dette betyder at et sådant *design pattern* vil kunne opbygge de forskellige GUI detaljer alt efter hvilken *element*type der skal vises eller redigeres.

Adapter pattern som også er et Structure pattern, bliver brugt i sammenhæng med den provider-struktur *frontend* delen skal implementere. Denne provider struktur går i korte træk ud på at give hver klasse sin egen lokale provider. Denne provider er givet fra et centralt sted, hvilket gør det sikkert at alle klasser kommunikerer videre ned i systemet på samme måde og med samme provider-objekt.
Adapter pattern går ud på at videregive partielle Interfaces, som en given klasse kan håndtere. På denne måde kan klassen bruge den samme provider, men har kun tilgang til udvalgte dele.
Netop dette *design pattern* skal sørge for, at de forskellige GUI -klasser selv kan håndtere deres ansvarsområde med data fra *backend* delen uden at få tilgang til andre klassers ansvarsområder for provider delen.
Et eksempel på dette kunne være at have forskellige GUI-klasser til de forskellige *element*typer, men hver af disse klasser må kun håndtere bestemte funktionaliteter fra

*backend* systemet. Oven i dette skal det være klart at de bruger det samme *backend* system.
Bruges dette *design pattern* ikke, kan der forekomme uoverensstemmelser for hvilken provider der bruges, og restriktionen på hvilke funktionaliteter der må tilgås. Nå der implementeres uden sådanne *design patterns*, kan det lettere forekomme, at det før nævnte overordnede Functional pattern ikke bliver implementeret korrekt. Hvis dette er tilfældet, kan dataoverlap forekomme og usikkerheden om hvilke klasser der har ansvaret, vil være mere uklart.

**Backend**
Denne del af *MARS Admin* er som før nævnt et service lag, hvilket vil sige, at det også vil kunne bruges af andre systemer, og at denne del skal være selvforsynende med fornødne informationer eller vide hvor manglende kan forefindes. Det er også på grund af dette, at der også intern i *backend* delen bliver brugt en horisontal lag-deling, hvilket sikre en ensartet tilgang til datakilden, som i dette tilfælde er den eksisterende *MARS* database.
I *backend* delen er der brugt lidt af de samme *design patterns*, men nogle ændringer er taget i brug, grundet den anderledes ansvarsfunktion i systemet.

- Fundamental patterns
  - Functional pattern
- Structure patterns
  - Pipes and filter pattern
  - Façade pattern
- Behavioral pattern
  - Memento pattern
  - Interpreter pattern

Functional *design pattern* som er beskrevet i *frontend* delen vil ikke yderligere blive beskrevet, fordi argumentationsgrundlaget er enslydende.

Pipes and filter pattern er et Structure pattern. Dette *design pattern* bliver brugt i sammenhæng med den meget ligefrem horisontale struktur *backend* delen er opbygget af. Fordi dette *design pattern* gør brug af 'output til input' sammenkædning, hvilket betyder, at en klasses retur-værdier bliver brugt til en andens metode-argumenter. Dette skaber nogle groft optegnede linjer gennem *backend*-designets lag, og viser at data faktisk kun løber vertikalt i den horisontale model hvilken også var meningen som beskrevet i afsnit 2. Analyse.

Façade pattern er også et Structure pattern, og er mere et generelt *design pattern* i dette projekt. Det bliver nævnt i denne del fordi det her er en tydelig del. Dette *design pattern* bliver brugt til at afskærme komplekse processer ved kun at fremvise udvalgte og simple indgange til disse. Brugen af dette *design pattern* kan meget tydeligt ses i Service laget af *backend* delen, hvilket også er meningen med hele *backend* systemet. Det kan tydeligt ses i bilag 9.8.7 Backend Service Concept at indgangene i denne klasse er simple, i forhold til de resterende lag i *backend* delen.

Memento pattern som tilhører Behavioral patterns, skal gøre designet mere robust med hensyn til associeret databehandling. Dette *design pattern* bruges til at udføre et 'rollback', hvilket fører data tilbage til det oprindelige stadie, hvis nødvendigt. Det er brugt til at designe datas logiske associationer ned mod databasen således at associerede data kun kan redigeres eller oprettes sammen, selvom der opereres på flere forskellige datarækker, tabeller og databaser. Dette *design pattern* bliver ofte brugt når datarækker har logiske binding, og kun giver mening når de redigeres sammen. Ved evt. fejl kan disse logiske bindinger garanteres at alle er redigeret eller ingen er redigeret.
Alternativt vil ubrugelige og logiske partielle data ligge i databasen uden associationer og bindinger. Større og uventede fejl vil kunne opstå i den forbindelse, og det skal dette *design pattern* være med til at forhindre.

Interpretor pattern som også er et Behavioral pattern, er mest af alt brugt til at vise hvordan kildekode er uddelt til andre optimerede dele i systemet. Dette *design pattern* bliver brugt til at uddelegerer specialiserede opgaver i andre sprog, der er optimeret til disse opgaver. I dette projekt skal det dække over måden selve databaseoperationerne bliver kørt på. Selve databaseoperationerne kan blive kørt direkte af databasen selv, og er dermed med til at optimere køretiden af de forskellige operationer. Disse små scripts (stored procedure, som bliver omtalt i afsnit 4. Implementering), vil blive kaldt fra *backend* delen, hvilket vil understøtte dette *design pattern*.

**Generelt**
*Design patterns* som nærmest er obligatorisk i .NET vil ikke blive omtalt yderligere, fordi de ikke er dedikeret og designet til netop dette projekt. Sådanne *design* patterns kunne eksempelvist være Observer pattern som er et Behavioral pattern. Dette *design pattern* vises i event baseret programmering, hvilket .NET WindowsForm FrameWork fundamentalt er baseret på.
De resterende konceptuelle Class diagrammer over individuelle dele og klasser er vist i bilag 9.8 UML Class Diagram Concept.

Fordi dette design kun er af konceptuel karakter, er flere hjælpe-klasser, metoder og variable udeladt, grundet en nedtoningen af komplekse klasser vil øge overblikket. Dette gælder også de resterende diagrammer i bilag 9.8 UML Class Diagram Concept.

## *3.2 Klasse Opbygning*

### 3.2.1 Entiteter/*elementer*

De forskellige *element*typer er designet til at indeholde basisværdier samt dens virtuelle *bindingsværdier*. Der er også tænkt på at give elementerne funktioner til at udføre simple operationer til at holde styr på sig selv. Dette kan væres den evne til at ændre sit 'Dirty' flag, som viser om data i *elementet* har været redigeret siden oprindelsen. Dette skal dermed indikere om den er forskellig fra den version der blev hentet i databasen, og på den måde vise, om det er nødvendigt at opdatere den.

### 3.2.2 Interfaces

Interfaces er kendt fra mange programmeringssprog. Det er en slags kontrakt imellem klasser, hvilket gør det mere sikkert at vide hvilke metoder og variable en given

klasse indeholder. Interfaces kan også være med til at understøtte fænomenet *multiple inheritance*, hvilket ikke bliver understøttet i mange sprog, grundet den meget forvirrende sammensmeltning af overlappende funktionaliteter.

I dette projekt bliver Interfaces brugt til at kommunikere klasser imellem. Der er også designet hierarkier af Interfaces for at skabe en ansvarsstruktur som beskrevet tidligere i afsnit 3.1.3 Class Diagram og design patterns.
Fordi Interfaces er skabt til at håndtere kontrakter imellem klasser er de tit nødvendige komplekse *enterprise solutions*.

### 3.2.3 Nedarvning

Der bliver flere steder gjort brug af nedarvning. På steder hvor ensartede strukturer eller adfærdsmønstre forefindes, kan der tit være brug for nedarvning.
I dette projekt bruges det til at designe Entity-hierarki i Domain Data delen som ses på f.eks. Figur 10 Class Diagram Frontend Koncept. Dette er gjort på grund af de meget ensartede *elementers* opbygning, og giver dermed en lettere tilgang til rettelser og et centralt sted at implementere dens ansvarsområde-funktioner.
Også i GUI-delen på samme figur, bliver der brugt nedarvning til at håndtere både basis GUI-klasser, men også til ensartede GUI-klasser.

### 3.2.4 Enumerationer

Enumerationer er brugt til værdier der har forudbestemte grænser. Der er både fordele og ulemper ved at benytte enumerationer.
Fordelene skal ses i at de fastsatte værdier kan få tilføjet navne som typer. Dette kan gøre det entydigt hvad de valgte værdier dækker over på designniveau. De kan også sørge for at kun bestemte værdier kan vælges, således valideringer af denne ikke behøves for at vide om den har værdier inden for de bestemte rammer.
Bagsiden af medaljen kunne være runtime-oprettelser af bestemte værdier, hvilket kunne give konverteringsproblemer for værdier der skal fortolkes til enumerationer. Dette er kendt fra flere programmeringssprog. Det er imidlertid ikke umuligt, men bare besværligt taget operationens kompleksitet i betragtning.

## *3.3 GUI Beskrivelse*

GUI-opbygningen er generelt opdelt i tre hoved elementer:

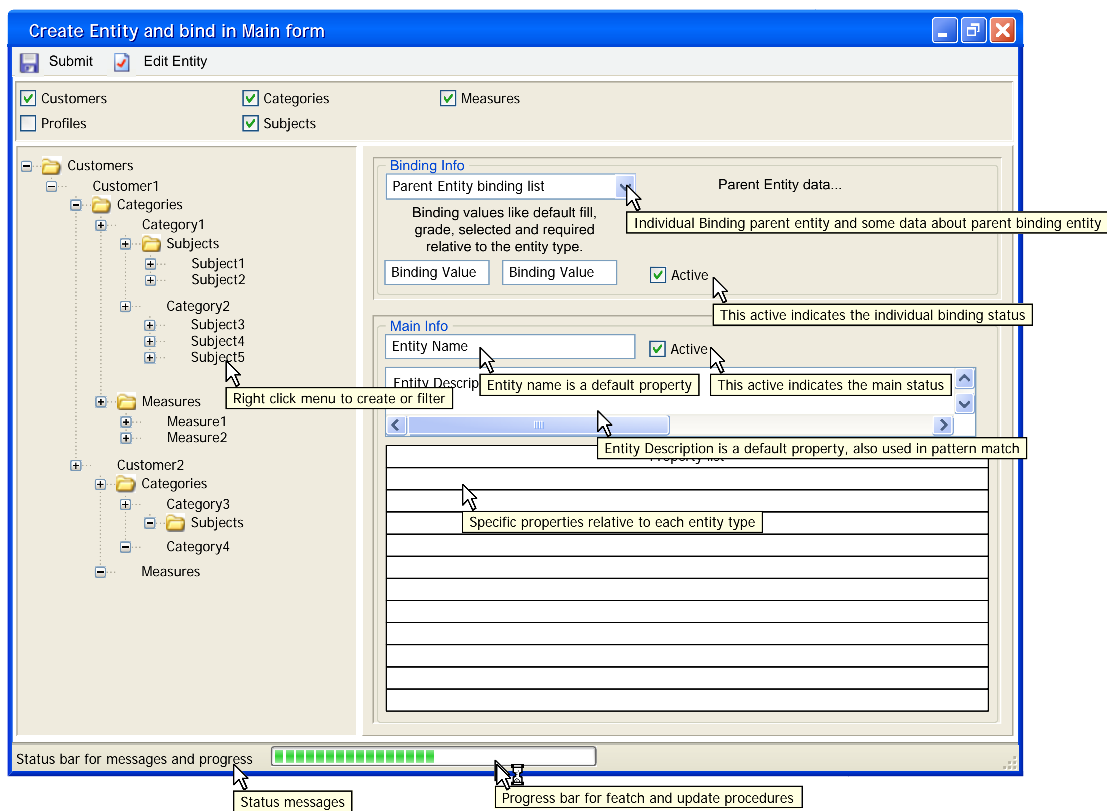
- Valg og navigation af *elementer*
- Filtrering af visning af *element* hierarkiet
- Visning og redigering af *element* data

Denne opdeling er designet på baggrund af analysen af de grundlæggende funktionaliteter som er omtalt i afsnit 2. Analyse.

På Figur 12 GUI abstrakt stadie, kan denne trevejs-opdeling ses. Øverst ligger filtreringskontrollen. Under denne ligger i venstre side navigationen af *elementer* og hele hierarkiet. Og til højre for dette er visningen og redigering af *element*data.

Den øverste kontrol skal filtrere *element*typer i navigationstræet, for at skabe et bedre overblik over bindinger *elementerne* imellem. Et eksempel på dette kunne være at brugeren ville se alle de *measure elementer* for en given *customer*. Dette kan være svært hvis den givne *customer* har flere *sub customers* eller/og flere *profiles*. I dette tilfælde vil *measures* blive vist i flere forskellige grene af navigationstræet. Filtreringen skal gøre dette billede lettere overskueligt, ved at samle alle *measures* under samme *customer* uden at differentiere mellem de forskellige *profiles* de er bundet til.

Den overordnede opdeling er relativ let at forstå, og der er taget udgangspunkt i de førnævnte funktionaliteter og behov. Der er taget høje for at hver GUI del varetager sig eget område, og gør det derfor lettere at samle funktionaliteter som det resterende design er opbygget af. Måden GUI skitserne er opbygget på, gør det også let at genbruge delene i andre sammenhænge. Dette kunne f.eks. være at opbygge en Wizard-guide, hvor brugeren bliver guidet gennem de forskellige typer af funktionaliteter systemet kan foretage. Dette vil kunne gøres ved at bruge de forskellige GUI dele hver for sig, men tildele dem et bestemt stadie som er tilpasset efter brugerens forrige valg.




**Figur 12 GUI abstrakt stadie**

De resterende GUI design tegninger er vist i bilag 9.4 GUI Udkast.

## *3.4 Resumé*

Den før omtalte *n-tier* model bliver designet til at varetage de forskellige roller og ansvarsområder hele *MARS Admin* systemet skal indeholde. Dette bliver gjort ved at se på sequence diagrammer, som viser datagennemstrømningen i systemerne. Disse diagrammer kan vise information om begrundelserne for nogle af de valgte *design patterns*.
Der er også beskrevet de konceptuelle linjer for Class diagrammerne i begge systemdele, *frontend* og *backend* 3.1.3 Class Diagram og design patterns. Til disse diagrammer er der tilknyttede forskellige *design pattern*, som er aktuelle for netop disses funktionalitet og ansvarsområde.
Der er gjort rede for udvalgte og generelle designklasser og deres funktionaliteter, og hvorfor de er brugt til dette projekt 3.2 Klasse Opbygning.
For at designe den visuelle GUI del, er der optegnet skitser til de forskellige dele, samt gennemgået hvordan og hvorfor delene skal bruges.

# 4. Implementering

Først beskrives nogle generelle retningslinjer for projektets implementeringsfase og udviklingsmiljø, hvorefter beskrivelser af udvalgt kildekode vil blive gennemgået og reflekteret i forhold til designet, og her tænkes især på *n-tier* model og *design patterns*. Alle implementerede Class diagrams er lagt i bilag 9.9 UML Class Diagrams.
I de givende kildekodeeksempler længere nede i dette afsnit, kan disse Class diagrams bruges til at give et bedre overblik og klassernes opbygning og relationer.

## *4.1 Valg af udviklingsmiljø*

Implementeringen til dette projekt er der valgt at gøres brug af Microsofts WindowsForms og ASP.NET fra MS .NET 2.0 C# udviklingsmiljøet. Dette valg er taget på baggrund af måden de eksisterende og tilhørende applikationer er implementeret på i virksomheden. *MARS* og applikationerne omkring er alle lavet i C# og med samme implementeringsbaggrund. Den største forskel på skrivende stund er *MARS backend*, som opererer på en MS SQL 2000 Database Server, hvorimod *MARS Admin* opererer på en MS SQL 2005 Database Server. De eneste forskelle der er bemærket specielt for dette projekt, er SQL2005's meget strikse måde at tildele rettigheder til brugere på.
Der bliver brugt WindowsForms til *frontend* systemet, hvilket er valgt på baggrund af den fleksible framework. WindowsForms er lettere at debugge, hvilket alternativt skal gøres via trace i ASP.NET. Denne løsning er kun valgt grundet den restriktive interne brug i virksomheden. Skal det tilgås GUI systemer fra kunder eller 3.parter bliver de ofte implementeret i klassisk ASP eller ASP.NET fordi de er mere uafhængige af OS platform (hvis der ikke bliver brugt for mange ActiveX-scripts. .NET er som sagt udviklet af Microsoft ® og er derfor mere stabilt på et Microsoft ® OS som Windows.

### 4.1.1 Udgivelse i virksomhedsmiljø

Til at gøre *MARS Admin* løsningen som en rigtig *enterprise solution* med *n-tier* model, er det også vigtigt at kunne holde styr på klient-versionerne, *frontend*. Til at styre dette er der brugt Microsoft ® egen ClickOnce løsning som er en del af Visual Studio 2005 Team Edition for Software Developers pakke. Denne løsning er god til at holde styr på versioner fra både brugeres og udviklerens synspunkt, fordi den henter oplysninger et centralt sted om opdateringer. Er den installerede version forældet, vil ClickOnce spørge om en opdatering, hvormed de nye filer vil blive downloadet fra den givende FTP-server og kan installeres men kun de nødvendige sikkerhedsrettigheder til forskel fra Windows Installer.
Udvikleren kan direkte udgive nye versioner af softwaren til den pågældende FTP- og http-Server. Http-serveren skal holde styr på hvilke versioner der bruges, mens FTP-serveren skal indeholde filer til download.
Selve *backend* systemet er opbygget af en ASP.NET WebService med de underliggende lag af logik- og *DALC*-klasser. Fordi *backend*-systemets servicelag er implementeret som en WebService, skal den ligge på en web-server(http). Til dette, som mange andre projekter, er WebServicen lagt på en Microsoft ® Internet Information Server (ISS), hvilken selvfølgelig let kan håndterer ASP.NET software. Der vil ikke blive beskrevet yderligere detaljer om ASP.NET valget grundet den indlysende sammenhæng til *frontend* systemet. Der kan alternativt bruges andre

WebService sprog som WSDL, hvilket selvfølgelig også er understøttet af .NET. ASP.NET gør normalt vist brug af SOAP protokollen når der tales om WebServices.

### 4.1.2 Generelt om variable

Alle steder i kildekoden er der brugt VS2005 måde at opdele kildekode i regioner på. Dette kan ses ved f.eks. Properties er indkapslet i #region tags som vist herunder:

```
...
#region Properties
private EntityType _type;

        public EntityType Type
        {
            get { return _type; }
            set
            {
                _type = value;
                //when EntityType is set, generate new Icon
                this.GenerateNodeType();
                this.Dirty = true;
            }
        }
...
#endregion
...
```

På denne måde er det let at navigere rundt i kildekodens, ved kun at kunne se de regioner der arbejdes med.

## *4.2 Navnekonvention*

Til dette projekt er der blandt andet brugt den kendte Hungarian Type Notation som kort bliver beskrevet i dette afsnit.
Først skitseres fordele og ulemper ved navnekonventioner.
Fordi der i *enterprise soultions* tit bliver arbejde på projekter af flere mennesker, og overtagelser af projekter også kan forekomme, vil det være oplagt at bruge forudbestemte notationer for forskellige elementer i kildekoden. Mange notationer er ikke sprogbestemte hverken af talt eller implementeret sprog.
Områder hvor det er oplagt at kunne se forskel direkte på navne er f.eks. access specifiers (private, protected og public) samt på lokale variable. Andre områder kunne være brugen af navne i forhold til typen af objektet. Et eksempel på dette kunne være at se forskel på metoder og variable kun ved hjælp at navnets format.
Fordi mange navne indeholder flere ord skal disse også let kunne læses separeret, hvilket gør navnet lettere at læse for mennesker. Måden Hungarian Type Notation kan gøre dette på, er at skrive startbogstaver med stort, og ikke at skrive noget yderligere separeringstegn imellem. Der er yderligere måder at starte navnet på som vist i eksemplet herunder.

Eks.

| **Pascal case** | **Camel case** |
|---|---|
| **BackColor, DataSet** | **numberOfDays, isValid** |

I dette projekt vil der blive brugt Camel case til variable og parametre/argumenter mens metoder, datatyper, Interfaces, namespaces og klasser vil bruge Pascal case. Namespaces er forklaret senere i dette afsnit.
Private og protected variable er vist medstartende '_' tegn. Dette gør dem tydelige at se, og derfor også hurtigere at se hvordan de bliver brugt i klasserne.

Exceptions vil blive vist som i eksemplet neden for. Denne notation bliver brugt fordi .NET selv bruger **e** navnet til event-parameter i f.eks.

```
public delegate void MouseEventHandler(object sender, MouseEventArgs e);
```

Eks.

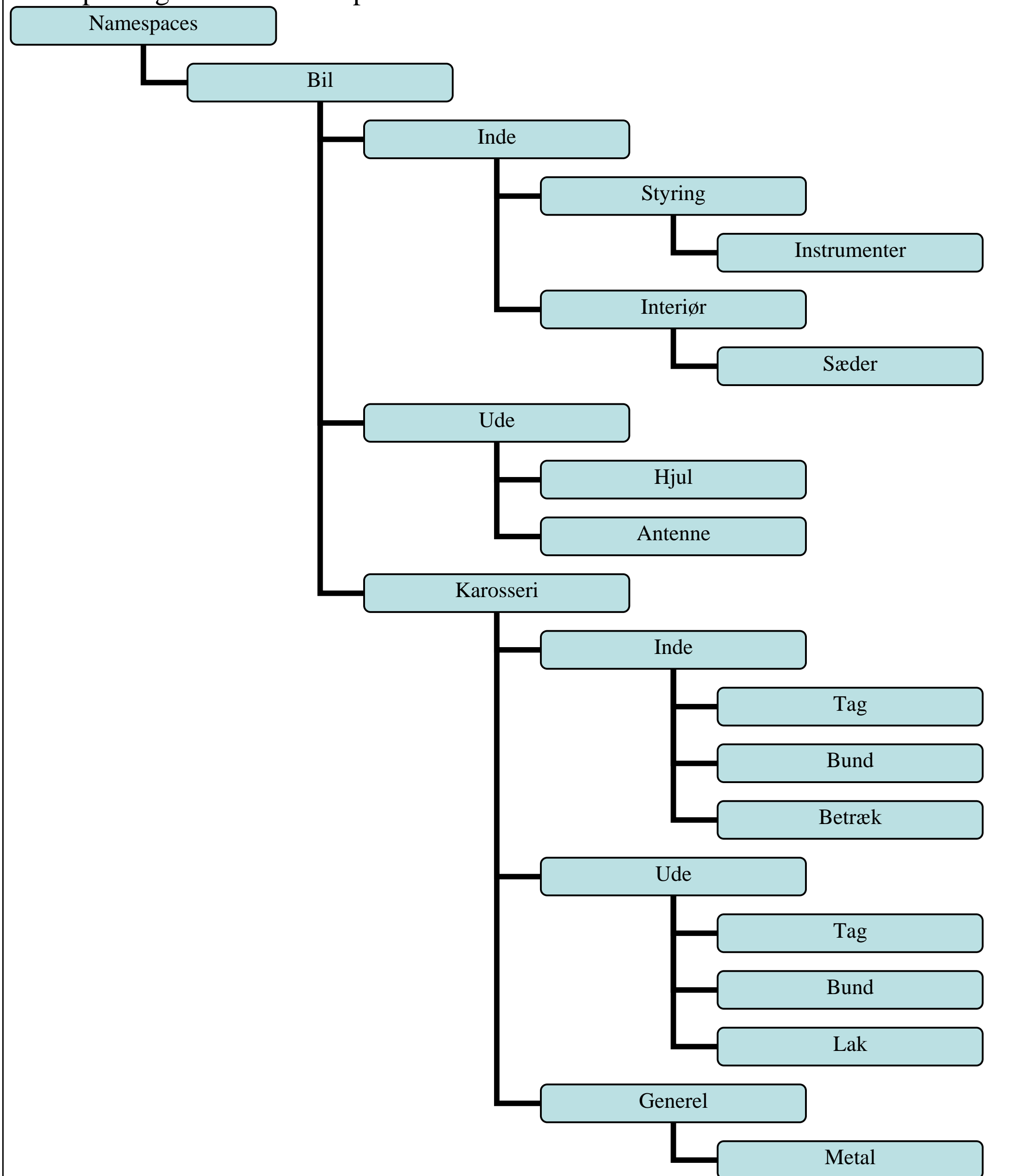
```
catch (Exception ex)
{
// Handle Exception
}
```

Flere mener at Hungarian Type notation eller lignende standarder ikke behøves i moderne IDE miljøer. Denne mening bygger på IDE miljøers evne til at indikere over for programmøren hvilke objekttyper der vises med f.eks. farver og ikoner. Dette synes nu ikke at være en skudsikker begrundelse for unødvendigheden af navnekonventioner. Det er ikke i alle tilfælde let at se hvordan en IDE vil visualisere navne og typer, og der findes gennemgående ingen forskrevne notationer på hvordan de skal vises i en IDE. Dette er helt op til producenten af disse. Med navnekonventioner kan der ses ud over disse problemer, og selv i almindelige tekst-redigerings-programmer kan forskellene let ses, hvis det skulle være en nødvendighed i praksis. I yderste tilfælde kan alternative visualiseringer ikke nødvendigvis erstatte litterære. Dette kan vises ved at sætte en farveblind foran farve-fremhævet kildekode, og se om forskelle tydeligt kan ses, når der ikke bliver brugt navnekonventioner. Til det må svaret være retorisk nej, og med lidt eftertanke giver 'både og' også mere mening når der vil være flere mennesker om at læse den samme kildekode.

## 4.3 Solution opbygning i VS2005

Der er gjort overvejelser med hensyn til opbygningen og struktureringen af kildekoden i MS Visual Studio 2005. Grundet brugen af hele projektet, er det besluttet at lave hele implementeringen i en enkelt solution fil. Det kunne der i mod også være opdelt i de 2 logiske solutions; frontend og backend, for at få den større afstand mellem klient og server ved implementeringen, men også ved senere brug. Dette er ikke hensigten på nuværende tidspunkt, og det vil derfor være mere hensigtsmæssigt at samle koden i samme struktur for dermed at arbejde på hele projektet som en helhed og ikke delprojekter.

Der skal selvfølgelig benyttes namespaces i henhold til opdelinger af logiske klasser. Opdelingen skal ske, både gennem klassernes brug, men også gennem deres type. Dette kan forklares ved et eksempel.




**Figur 13 Namespaces**

Dette eksempel viser en måde at opdele namespaces på, som forklaret ovenfor. Det kan ses i eksemplet, at selve karosseriet er på samme niveau som opdelingen af Ude og Inde namespaces. Ude og Inde namespaces er opdelt efter brugen af klasserne, mens karosseriet er opdelt efter typen.
Dette viser at der ikke entydigt kan opdeles mellem de sideliggende namespaces Ude og Inde.
I dette eksempel er den videre opdeling af karosseri opdelt i de namespaces som igen er opdelt af brugen.

Denne form for opdeling gør det lettere at bruge netop det namespace der passer til den funktionalitet en klasse skal indeholde. Denne opdeling er valgt på grund af det før omtalte *design pattern*, Functional pattern, som er ledetråd for det overordnede

princip. Dette gør det lettere at bruge færre namespaces i hver klasse og dermed også nedsætte abstraktionsniveauet for at få en mere simpel implementeringsforståelse.
En generel regel i virksomheden kunne være formatet:

Eks.
**CompanyName.TechnologyName[.Feature][.Design]**

På den måde vil navne overlappe på klasser, datatyper og Interfaces også minimeres eller elimineres helt.

## *4.4 Funktionaliteter og Design Patterns*

I kildekode er der brugt en notation for at indikere at det er et uddrag fra en kontekst. Denne notation er tre punktummer efter hinanden '`...`', og viser at der finde kildekode foran eller/og bagefter, alt efter hvor notationen er brugt.

Der er generelt i hele projektet brugt .NET Docs til at kommentere metoder med som beskrevet herunder i 4.4.3 Nedarvning.

### 4.4.1 Properies

Properties er en almindelig kendt *OO* metodetype til at tilgå private variable i klasser på uden at give direkte tilgang til disse. De bruges ofte til ændre eller forhindre data i at blive læst eller gemt via dens `get` og `set` kald. Disse kald gør det muligt at bruge metoden som en variabel ved f.eks. at skrive:

Eks.

```
private int _number;

public int Number
{
      get { return value; }
      set { _number = value; }
}
...
Number = 3;                   //brug af set
int currentNum = Number;      //brug af get
```

I den underliggende kildekode kan det også ses hvordan de i implementeringen af projektet er brugt til at implementere Functional pattern ved at sætte et element til selv at holde og sætte værdier om ændringer (Dirty flag).
I `set` kaldet er der indsat et andet property `set` kald på objektets Dirty flag, hvilket vil sige at alle der har tilgang til denne property kan se om objektet er blevet ændret.

```
...
namespace Infopaq.MediaAnalysis.MARSAdmin.Domain.Data
{
    public class EntityCategory : EntityUIDName
    {
        #region Properties

        private EntityBindCategory _bind;

        public EntityBindCategory Bind
        {
            get { return _bind; }
            set
            {
                _bind = value;
                this.Dirty = true;
            }
        }
...
```

### 4.4.2 Interfaces

Interfaces er i nedenstående kildekode brugt til at udnytte Adapter pattern. Det er her vist i hoven WindowsForm'en som instantierer klasser som implementerer Interfaces der skal kommunikere klasserne imellem. Her er også vist de før omtalte provider-klasser der implementere grundlaget for kommunikationen mellem *frontend* og *backend*. Disse klasser gives videre til underliggende WindowsForms UserControls, således at de selv kan gøre brug af disse. Dette mønster er brugt fra Proxy pattern. Videre ned i kildekoden kan beskedsystemets Interface-klasser ses, også disse bliver givet vide til de underliggende klasser, for at kunne kommunikere.
De to næste linjer som er taget fra nedenstående kildekode, viser hvorpå parallelle klasser, som navigationstræet og visning/redigeringen skemaet (kendt fra GUI skitserne), kan kommunikere. Dette er implementeret grundet *elementernes* skiftende status og værdier, som begge UserControls skal kunne opretholde.

```
this.ucEntityTree.Entity = this.ucEntityInfo;
this.ucEntityInfo.EntityUpdate = this.ucEntityTree;
```

```
...
namespace Infopaq.MediaAnalysis.MARSAdmin.Frontend.GUI
{
    /// <summary>
    /// This Class is a Windows Form used to hold various UserControls
    /// for the purpose of showing level of information and direct data for several
    /// Domain Entities. These UserControls also include edit functions.
    /// </summary>
    public partial class MARSAdminMainForm : Form, IMessageFeed
    {
...
                //provider system
                this.ucEntityTree.EntityAllProvider = this._MARSAdminProvider;
                this.ucEntityInfo.EntityInfoProvider = this._MARSAdminProvider;
                this.ucEntityInfo.EntitySuggestionProvider = this._MARSAdminProvider;
                //message system
                this.ucEntityTree.MessageFeed = this;
                this.ucEntityInfo.MessageFeed = this;
                this.ucEntityFilter.MessageFeed = this;
                //entity system
                this.ucEntityTree.Entity = this.ucEntityInfo;
                this.ucEntityInfo.EntityUpdate = this.ucEntityTree;

                //filter system
                this.ucEntityFilter.Filter = this.ucEntityTree;
...
```

### 4.4.3 Nedarvning

Der kan ses i kildekoden herunder, hvordan brugen af metodedokumentation er udnyttet. Denne dokumentation er både god til at se hvad metoden skal bruges til, hvad den skal bruge af parametre og hvad den returnerer. I VS2005 samt tidligere versioner af Visual Studio bliver denne dokumentation også vist når metoden bruges på design niveau i IDE'en. Det gør det lettere at navigere i metoderne, hvis beskrivelserne er fyldsgørende.

Øverst i underliggende kildekode kan det ses at klassen `EntitySubject` nedarves fra klassen `EntityUIDName`. Der kan ses i metoden `Initialize` hvordan det sidste metodekald har et **`base`**-præfiks som i C# viser at baseklassen(også kendt som superklassen[3]) bliver kaldt, hvilket skal initialisere base-værdierne som klassen har nedarvet fra.

```
...
namespace Infopaq.MediaAnalysis.MARSAdmin.Domain.Data
{
    public class EntitySubject : EntityUIDName
    {
...
        /// <summary>
        /// Initialize, used by constructors to set Entity values
        /// </summary>
        /// <param name="uid">UID</param>
        /// <param name="name">Name</param>
        /// <param name="active">Active</param>
        /// <param name="description">Description</param>
        /// <param name="type">Type of Entity</param>
        /// <param name="dirty">Dirty Flag</param>
        /// <param name="defaultSelected">Default selected flag</param>
        /// <param name="customerExclusive">Can only be use by this customer</param>
        /// <param name="bind">Subject Bind Infomation</param>
        protected void Initialize(
            int uid,
            string name,
            bool active,
            string description,
            EntityType type,
            bool dirty,
            bool defaultSelected,
            int customerExclusive,
            EntityBindSubject bind
            )
        {
            this.DefaultSelected = defaultSelected;
            this.CustomerExclusive = customerExclusive;
            this.Bind = bind;
            //run last to initiate dirty flag correct
            base.Initialize(uid, name, active, description, type, dirty);
        }
...
```

Herunder kan der ses et uddrag fra kildekoden fra filen **`'UCEntityInfo.cs'`**. For at vise hvordan det førnævnte *design pattern* Builder pattern er implementeret i projektet. Det er implementeret ind i en .NET UserControl klasse, hvilken skal styre og bygge andre UserControl klasser alt efter hvilket *element* den skal repræsentere. Dette bliver gjort ved en switch-løkke, hvor hvert *element*type bliver fanget og en UserControl vil blive bygget og placeret i den base-klasseholder, `this.UCEntityBase`, den

[3] Super notationen bliver brugt i Java, fordi der bliver refereret til superklassen. I C3 hedder denne klasse baseklassen.

overordnede Builder-klasse besider. Derefter vil base-klasseholderen blive instantieret med de krævede værdier og Interface-klasser som tidligere omtalt.

```
...
namespace Infopaq.MediaAnalysis.MARSAdmin.Frontend.GUI.UserControls
{
    public partial class UCEntityInfo : UCEntityBaseInfo, IEntity
    {
...
public override bool ShowEntity(EntityBinding binding)
        {
            if (UCEntityBase != null)
                this.UCEntityBase.Dispose();

            switch (binding.MainEntity.Type)
            {
                case EntityType.Customer:
                    this.ucEntityCustomerInfo = new UCEntityCustomerInfo();
                    this.UCEntityBase = this.ucEntityCustomerInfo;

                    this.Controls.Add(this.ucEntityCustomerInfo);
                    break;
...
protected void ucEntityBaseInit(EntityBinding binding)
        {
            //generating name corresponding to type
            this.UCEntityBase.Name = "uc" + binding.MainEntity.Type.ToString() +
"Info";
            this.UCEntityBase.Dock = DockStyle.Fill;
            //give providers and message systems
            this.UCEntityBase.EntityInfoProvider = this.EntityInfoProvider;
            this.UCEntityBase.EntitySuggestionProvider =
this.EntitySuggestionProvider;
            this.UCEntityBase.MessageFeed = this.MessageFeed;
            this.UCEntityBase.EntityUpdate = this.EntityUpdate;
        }
    }
}
```

### 4.4.4 Transactions

Det før omtalte Memento pattern fra afsnit 3.1.3 Class Diagram og design patterns i *backend* delens Domain.Logic, vil blive implementeret via `TransactionScope` klassen. Denne klasse udfører automatisk et 'rollback' hvis der forefindes en exception inden klassen `Complete()` metode bliver kaldt i `TransactionScope using` rammen.
Der bliver her kaldt to *DALC* metoder inde i før omtalte ramme, og hvert af disse metodekald indeholder redigeringer i databasen, hvilke er associeret logisk til hinanden. Disse kald repræsenterer henholdsvis *grundværdier* og *bindingsværdier*. *Grundværdierne* skal oprettes først for at returnere evt. indeksnøgler (UID) værdier som bliver omtalt i 4.4.5 Stored Procedure.

```
...
namespace Infopaq.MediaAnalysis.MARSAdmin.Backend.Logic
{
    public class MainLogic
     {
...
public EntityMessage SetMeasureInfo(EntityMeasure measure, EntityBinding binding, bool
create)
        {
            EntityMessage message = new EntityMessage();

            try
            {
                using (TransactionScope scope = new TransactionScope())
                {
                    int identity = _DALC.Measure.SetMeasureBase(measure, binding,
create);
                    message = _DALC.Measure.SetMeasureBind(measure, binding, create,
identity);

                    scope.Complete();
                }
                return message;
            }
            catch (Exception ex)
            {
                return new EntityMessage(ExceptionMessage.GenerateMessage(ex),
ImportanceLevel.Highest, "Transaction Measure", StatusType.Error);
            }
        }
...
```

### 4.4.5 Stored Procedure

For at gøre brug at det valgte Interpreter pattern som beskrevet i afsnit 3.1.3 Class Diagram og design patterns, er der brugt Stored Procedures, hvilket er databaseoptimeret scripts direkte implementeret på databasen.

Herunder er vist kildekoden til en oprettelse af basis elementet measure. Fordi dette kun opretter basis værdierne, skal det returnere det ny indsatte indeksnøgle (UID).

```
...
ALTER PROCEDURE [dbo].[MarsAdminDAL_CreateMeasureBase]
      @name varchar(50),
      @active bit,
      @description varchar(500)

AS

INSERT INTO
      dbo.Measure

      (
      Name,
      Active,
      Description
      )
      VALUES
      (
      @name,
      @active,
      @description
      )

SELECT SCOPE_IDENTITY() AS [SCOPE_IDENTITY]

grant execute on [MarsAdminDAL_CreateMeasureBase] to SpExecutor
```

```
...
ALTER PROCEDURE [dbo].[MarsAdminDAL_GetCustomerSuggestion]
      @token varchar(50)
AS

set @token = '%' + @token + '%'

SELECT
      Customer_UID AS UID,
      Name,
      Active,
      Description

FROM
      dbo.Customer

WHERE
      Name LIKE @token OR Description LIKE @token

Return 0
grant execute on [MarsAdminDAL_GetCustomerSuggestion] to SpExecutor
```

### 4.4.6 Beskedsystem og exceptions

Beskedsystemet gør brug af førnævnte Interface system, hvilket gør det let at kommunikere videre op i systemet som f.eks. exceptions gør det. Bagdelen ved at gøre det direkte med exceptions er de evt. kædereaktioner der kan medfølge, og dermed gøre det uoverskuelig at se hvad der er gået galt. Derudover kan interne beskedsystemer også bruges til informationer eller advarsler som ikke nødvendigvis bliver kastet som en exception.

Herunder kan der ses kildekode, hvor dette Interface (MessageFeed) er brugt i forbindelse med en exception.

```
...
namespace Infopaq.MediaAnalysis.MARSAdmin.Frontend.GUI.UserControls
{
    public partial class UCEntityUIDNameInfo : UCEntityBaseInfo, IEntity,
IEntitySelect
    {
...
catch (Exception ex)
            {
                this.MessageFeed.SendMessage(new EntityMessage(ex, "Dirty Flag",
StatusType.Alert));
                return false;
            }
...
```

Næste kildekode er taget som et eksempel på hvordan exceptions bliver behandlet i beskedsystemet. Der tages højde for inner exceptions, og hvis disse forekommer, vil de også blive en del af beskeden. Denne funktionalitet er indbygget i besked konstruktøren, således exception handleren kun skal bekymre sig om at sende en exception videre i systemet, og ikke hvad den indeholder eller ikke indeholder. Denne funktionalitet understøtter også fint Functional *design pattern*, som er beskrevet i afsnit 3.1.3 Class Diagram og design patterns.
Generelt er der brugt exceptions, og især til klasser der er tæt på lag der kommunikere eksternt, som eventhandlers i GUI, eller WebService-metoder. Fordi disse er de øverste lag, vil de kunne fange exceptions som andre metoder ikke selv fanger eller som kastes videre op i systemet.

```
...
namespace Infopaq.MediaAnalysis.MARSAdmin.Domain.Data
{
    public class ExceptionMessage
    {
        public static string GenerateMessage(Exception ex)
        {
            string error = ex.Message;
            if (ex.InnerException != null)
                error += " " + ex.InnerException.Message;
            return error;
        }
    }
}
```

## 4.5 Resultat

Størstedelen af det designede projekt er implementeret og fungerer efter hensigten. Dette vil blive gennemgået nærmere i afsnit 5. Test.
Nogle af de sekundære mål er ikke færdig implementeret, men er implementeret i 'proof of koncept' stil, hvilket vil sige at der er implementeret én af *elementernes* intelligente suggestion løsninger, for at vise det kan lade sig gøre. De resterende *elementers* kildekode på dette område, vil på en stor del af kildekoden være ens. Der vil også senere kunne implementeres yderligere intelligente suggestion-algoritmer.
En anden implementering del, der er udeladt i projektperioden er filtreringen af *element*typer. Selve UserControl '**UCEntityFilter.cs**' er implementeret efter hensigten, men det tilhørende Interface den skal gøre brug af, til at kommunikere ned til navigationstræet **'UCEntityTree.cs'**, er ikke implementeret i sidstnævnte

klasse. **'UCEntityTree.cs'** filen indeholder forberedelse til implementering af Interfacet, men er ikke implementeret.
Enkelte *element*værdier er ikke færdig implementeret, men ved nærmere gennemgang af implementeringen, vil det ikke være af større arbejdsbyrde at udvide disse værdier i næste iteration af *UP* forløbet.
Kildekoden er vist herunder, og det kan ses at systemet tilbagemelder med en meddelelse i det før omtalte beskedsystem.

```
#region IEntityFilter Members

        public bool FilterEntity(List<EntityType> filterTypes)
        {
            this.MessageFeed.SendMessage(new EntityMessage("Not Implementet Yet!",
ImportanceLevel.Normal, "Filter Entities", StatusType.Alert));
            return false;
        }

        #endregion
```

Til at implementere en sådan filtreringsmetode, kan der bruges hjælpemetoder, som skal finde *elementernes* bindinger, mens andre hjælpemetoder skal holde styr på hvilke bindinger der skal springes over eller ej, når de vises i navigationstræet.
En anden måde at implementere det på kunne være at de eksisterende byggemetoder, som bygger træstrukturen op, skal udelade visse *elementtyper*.

Kildekoden til hele projektet kan ses på den vedlagte CD-ROM hvis indhold er beskrevet i bilag 9.1 CD indhold. Der er udleveret er midlertidig WebService som kører på test data i virksomheden. Denne service er tilgængelig i DMZ og er derfor tilgængelig for alle. Den herunder viste URL peger på denne service, som også er implementeret som en WebReference i kildekoden.

**http://213.150.59.121:1212/MARSAdminService.asmx**

Test af selve WebService kildekoden kan også foretages ved at installere den medfølgende database-fil på en tilgængelig MSSQL2005Server, som kunne være den Express version i Visual Studion 2005, og derfra kører hele projektet lokalt.

Fordi der også i *backend.DALC* laget er brugt Enterprise Library for .NET Framework 2.0 - January 2006, skal disse også installeres og refereres til i C# projektet. Linket til denne installation er:
http://www.microsoft.com/downloads/details.aspx?FamilyID=5A14E870-406B-4F2A-B723-97BA84AE80B5&displaylang=en

Der skal også installeres Microsoft ® .NET Framework v2.0 for at kunne eksekverer kildekoden når den er kompileret. Dette er selvfølgelig en fundamental del af VS2005.

## *4.6 Resumé*

Gennemgangen af implementeringsregler dækkede over navnekonventioner og opbygning af Visual Studio solution via namespaces. Disse opdelinger i solution filen, hvilket er funderet i namespaces, er taget fra argumentgrundlaget fra afsnit 3.1.3 Class Diagram og design patterns.
Der er vist udvalgte kildekode-linjer for at vise nogle af de toneangivende *design patterns*, og hvordan de er implementeret i projektet.
Selve implementeringen er forløbet planmæssigt hvis der ses bort fra udvalgte sekundære mål, som filtrering af navigationstræet. Konceptet for intelligent profilstyring er vist implementeret, men mangler yderligere implementering af alle *element*typer.
Enkelte *element*værdier kan ikke redigeres, men fundamentet for implementering af disse er allerede lagt.
Kildekoden er lagt med og kan ses på vedlagte CD sammen med Stored Procedures og test-databasefilen.

# 5. Test

Hele projektet og implementeringen er gennemgået, og flere aspekter af kildekoden skal derfor testes for funktionalitet og typiske faldgrubber. I de følgende afsnit vil testprojektet blive gennemgået, og der vil blive vist flere eksempler via kildekode. Selve testprojektet er en del af kildekoden på den vedlagte CD, `'UnitTestProject.csproj'`.
Der er også udarbejdet Class Diagram for UnitTest-klasserne. Disse diagrammer kan findes i bilag 9.10 Test Class Diagrams.

## 5.1 Introduktion til test baggrund

Der er til dette projekt valgt at gøre brug af UnitTest metoden. Denne test type er en måde at validere individuelle dele af kildekoden. Når der tales om units, er det den mindste del af en applikation der logisk kan testes på. Fordi dette er implementeret i *OO*-sprog er en unit tænkt på som en klasse. Hvis der havde været tale om procedural-sprog, kunne unit være programmer, funktioner eller procedure.
Unit test er normalt ikke brugt som GUI test, men kan ved alternativt brug godt udfylde enkelte direkte GUI test, som at ændre forskellige kontrol eller form parametre. Hvad UnitTest ikke kan er direkte at klikke på de enkelte knapper eller kontroller ved f.eks. at simulere 'drag and drop' procedure. Det skal der bruges andre typer af testprogrammer til.

Valget er naturligt faldet på UnitTest projektet fordi det er en del af VS2005 og flere tidligere versioner. Det er som ovenfor omtalt brugt til at teste klasse med. I fagsproget bliver sådanne projekter ofte kaldt for 'test harness' eller 'automated test framework'.
Valget af UnitTest testtype er også et godt valg, fordi implementeringen er skrevet efter det omtalte Functional *design pattern*, hvilket holder funktionaliteter bundet ind til de respektive klasser. Dette stemmer godt overens med UnitTest hensigt, som er at isolere hver del af programmet til validering. Og gør dermed hver unit/klasses ansvarsområde korrekt.

## 5.2 Opdeling af testområder

Testprojektet er delt op i klasser som hver især skal validere forskellige typer af objekter og klasser i *MARS Admin*.

## 5.3 Implementering af test

Der er udvalgt nogle testtyper fra testprojektets kildekode. De efterkommende eksempler skal vise konceptet bag UnitTest i dette projekt, og hvordan abstraktionen af de forskellige klasser opfører sig når de instantieres som objekter.
Der bliver brugt samme notation for kildekoden, som nævnt i afsnit 4. Implementering.
Alle implementerede UnitTest-klasser er valideret positivt, og dette vil derfor ikke blive yderligere kommenteret i kommende underafsnit.

### 5.3.1 Node base *element* test

For at teste *elementerne* som i implementeringen kaldes for **Entity**, er der i kildekoden herunder vist hvordan en node, som er de objekter navigationstræstrukturen består af, bliver oprettet ud fra base*elementet* **EntityUIDName**. **EntityUIDName** er det objekt alle andre *elementer* er nedarvet fra. Denne funktionalitet er nødvendig grundet repræsentationen af *elementerne* fra *backend* systemet, som netop også bruger de fælles Domain.Data **Entities**. `EntityTreeNode` er en implementeret klasse i dette projekt, som er nedarvet fra standard klassen **TreeNode**

Først oprettes et nyt base-objekt, hvilket tildeles værdier, og derefter bruges som argument til `EntityTreeNode` konstruktøren. Denne nye `EntityTreeNode` valideres derefter op imod de originale værdier fra base*elementet*.

```
...
[TestClass]
    public class EntityTreeNodeTest
    {
        [TestMethod]
        public void TestMethodUIDNameConstuctors()
        {
            EntityUIDName testUIDName = new EntityUIDName();
            testUIDName.Name = "Test Name";
            testUIDName.Type = EntityType.Profile;
            testUIDName.UID = 42;
            testUIDName.Description = "Testing";
            testUIDName.Active = true;
            testUIDName.Dirty = false;

            //test EntityUIDName constructor
            EntityTreeNode testNodeUIDName = new EntityTreeNode(testUIDName);

            //test node text value
            Assert.AreEqual(testUIDName.Name, testNodeUIDName.Text, false);
            //test uid value
            Assert.AreEqual(testUIDName.UID, testNodeUIDName.UID);
            //test dirty
            Assert.AreEqual(testUIDName.Dirty, testNodeUIDName.Dirty);
            //test active
            Assert.AreEqual(testUIDName.Active, testNodeUIDName.Active);
            //test type
            Assert.AreEqual(testUIDName.Type, testNodeUIDName.Type);
        }
...
```

### 5.3.2 Entity Dirty Flag test

Her under er kildekoden vist for UnitTest af 'Dirty Flag' funktionaliteten af en `EntityTreeNode`. Dette objekt er konstrueret meget lig base*elementet* som før sagt bliver brugt til at nedarve fra, for alle andre *element*typer.

Først i testen bliver en `EntityTreeNode` oprettet med navn og type. Dette vil sætte variablen 'Dirty' til false, hvilket også er forventet og fornuftigt, fordi objektet ikke har været ændret siden det blev oprettet af sin konstruktør.
Efter denne test ændres en variabel i `EntityTreeNode`, og 'Dirty' variablen bliver igen testet, og skulle så vise at objektet er ændret siden oprindelsen. Endnu en ændring bliver foretaget, og igen bliver der testet at objektet viser ændring.

```
...
[TestMethod]
        public void TestMethodDirtyFlag()
        {
            EntityTreeNode testNode = new EntityTreeNode("testNode",
EntityType.Measure, true);

            //test init dirty flag
            Assert.IsTrue(testNode.Dirty == false);
            //change node base values (properties)
            testNode.Description = "test dirty flag";
            //test dirty flag again
            Assert.IsTrue(testNode.Dirty == true);
            //change node base values (properties) again
            testNode.DirectoryNode = false;
            //test dirty flag again
            Assert.IsTrue(testNode.Dirty == true);
        }
...
```

### 5.3.3 MainProvider SetProfile test

Til denne test er det vigtigt at databasen værdier ikke er ændret siden skrivende stund, fordi værdierne UnitTest klassen validerer mod skal stemme overens med pågældende data. Disse data er skrevet direkte i testkildekoden, hvilket kan være en farlig procedure, men nødvendigt fordi alternative metoder ikke nødvendigvis kan garantere korrekte resultater. Dette kunne være at skrive et SQL-script direkte i testprojektet, som skulle hente eller sætte de samme data, som selve applikationen skal.

Først kan der ses i nedenstående kildekode, at der oprettes et *profile element* (**`EntityProfile`**). Dette bliver fyldt med værdier inkl. *Bindingsværdier* (**`EntityBinding`**). Derpå oprettes den forventede besked, som skal repræsentere de værdier metodekaldet skal returnere.
For at kunne teste om værdier kan ændres af Provideren (**`target`**), vil de originale værdier først blive hentet. Derpå ændres én af dens variable og ligges i det **`EntityProfile`** som nu skal gemmes, 'Dirty Flag' sættet til 'uændret' værdi. Derefter foretages kaldet, som gemmer **`EntityProfile`**, og den returnerede besked gemmes, hvorefter at blive valideret mode den forventede værdi.
Igen hentes **`EntityProfile`** fra Provider, og værdierne skal valideres mod den nye og ændrede **`EntityProfile`**.

Via denne UnitTest bliver flere sider af Provider-strukturen testet, og konceptuelle områder som beskedsystem og Provider gemme og hente metoder. Det kan ikke ses i denne test at metodekaldet;
`target.SetProfileInfo(profile, binding, create);`
gør brug af flere stored procedures som er en del af *backend* systemet Memento pattern. Hvilket vil sige at alle Stored Procedure kald i den tilhørende *DALC* skal gennemføres korrekt før ændringerne slår igennem. Denne funktionalitet er beskevet i næste afsnit.

```
...
    [TestClass()]
    public class MainProviderTest
    {
...

        /// <summary>
        ///A test for SetProfileInfo (EntityProfile, EntityBinding, bool)
        ///</summary>
        [TestMethod()]
        public void SetProfileInfoTest()
        {
            MainProvider target = new MainProvider();

            //init agument values
            EntityProfile profile = new EntityProfile();
            profile.BillingDetails = "0";
            profile.Bind = new EntityBindProfile();
            profile.Bind.Active = true;
            profile.Description = "Test Profile";
            profile.Name = "demo -Test Profile 2";
            profile.Type = EntityType.Profile;
            profile.UID = 6;

            EntityBinding binding = new EntityBinding();
            binding.MainEntity = profile as EntityUIDName;
            binding.BindEntity.UID = 3;
            binding.BindEntity.Type = EntityType.Customer;
            binding.BindEntity.Name = "ZZ Test";

            //update state
            bool create = false;

            //init expected values
            EntityMessage expected = new EntityMessage();
            expected.ImportanceLevel = ImportanceLevel.Low;
            expected.Text = "Profile Updated Correct!";
            expected.Status = "6";
            expected.StatusType = StatusType.Info;

            //get original values
            EntityProfile profileOrg = target.GetProfileInfo(binding);

            //set active to oppersit value to test value is updated
            //init dirty flag
            profile.Active = !profileOrg.Active;
            profile.Dirty = false;

            //get actual values
            EntityMessage actual;
            actual = target.SetProfileInfo(profile, binding, create);

            //test values
            Assert.AreEqual(expected.Text, actual.Text);
            Assert.AreEqual(expected.StatusType, actual.StatusType);
            Assert.AreEqual(expected.Status, actual.Status);
            Assert.AreEqual(expected.ImportanceLevel, actual.ImportanceLevel);

            //get updated values
            EntityProfile profileNew = target.GetProfileInfo(binding);

            //test values
            Assert.AreEqual(!profileOrg.Active, profileNew.Active);
            Assert.AreEqual(profile.BillingDetails, profileNew.BillingDetails);
            Assert.AreEqual(profile.Bind.Active, profileNew.Bind.Active);
            Assert.AreEqual(profile.Description, profileNew.Description);
            Assert.AreEqual(profile.Dirty, profileNew.Dirty);
            Assert.AreEqual(profile.Name, profileNew.Name);
            Assert.AreEqual(profile.Type, profileNew.Type);
            Assert.AreEqual(profile.UID, profileNew.UID);
        }
    }
}
```

### 5.3.4 MainLogic transaction test

Denne test er baseret på transactions, hvilket er den klasse der bliver brugt til at implementere Memento pattern.
Det første der sker i denne testdel, er at argumenterne til at hente de originale database værdier bliver initialiseret.
Derefter bliver de originale værdier hentet fra databasen via *Backend.DALC*, som den testede klasser (**`target`**) gør brug af, og derefter gemt lokalt.
Testen går ud på at opdatere nogle værdier i database via transactions, og derfor oprettes det objekt der skal gemmes med nye værdier. En af de nye værdier i objektet `EntityBinding` bliver sat til en forkert værdi, hvilket vil sige at opdateringen vil fejle. Herunder kan se at indeksnøglen (UID) er sat til 0, og derfor ikke er valid.

```
EntityUIDName entityMainFail = new EntityUIDName(0, "Testing
category", EntityType.Category);
```

Forventet fejlbesked bliver genereret, og opdateringsmetoden bliver kørt. Den returnerede fejlbesked (`"No Category Bind Match"`) viser at metoden ikke kan finde den valgte indeksnøgle.
Data fra databasen hentes igen og de skal stemme overens med de originale data, og er dermed ikke opdateret selvom den ene af de to objekter er korrekt opdateret.

```
...
[TestClass()]
    public class MainLogicTest
    {
...
[TestMethod()]
        public void SetCategoryInfoTest()
        {
            MainLogic target = new MainLogic();

            //init argument binding values for orginal category
            EntityUIDName entityMainOrg = new EntityUIDName(1, "Farvandsvæsenet",
EntityType.Category);
            EntityUIDName entityBindOrg = new EntityUIDName(1, "Farvandsvæsenet",
EntityType.Customer);
            EntityBinding bindingOrg = new EntityBinding(entityMainOrg,
entityBindOrg);
            //init actucal values for later test
            EntityCategory expectedCategoryOrg = target.GetCategoryInfo(bindingOrg);

            //init argument values for failed update
            EntityCategory category = new EntityCategory();
            category.Active = true;
            category.Bind = new EntityBindCategory(true, true);
            category.CustomerExclusive = 0;
            category.Description = "description test";
            category.Dirty = true;
            category.Name = "Testing category";
            category.Type = EntityType.Category;
            category.UID = 1;

            //init bind values
            //entityMain UID is set to 0 to force invalid update
            EntityUIDName entityMainFail = new EntityUIDName(0, "Testing category",
EntityType.Category);
            EntityUIDName entityBindFail = new EntityUIDName(1, "Farvandsvæsenet",
EntityType.Customer);
            EntityBinding bindingFail = new EntityBinding(entityMainFail,
entityBindFail);

            //update state, only invalid uid on update generates an error
            //creation would generate new identity UID
            bool create = false;

            //init expected values
```

```
            EntityMessage expected = new EntityMessage("No Category Bind 
Match",ImportanceLevel.Highest, "Transaction Category", StatusType.Error);
            EntityMessage actual;

            //get actual values
            actual = target.SetCategoryInfo(category, bindingFail, create);
            //test values
            Assert.AreEqual(expected.Text, actual.Text);
            Assert.AreEqual(expected.StatusType, actual.StatusType);
            Assert.AreEqual(expected.Status, actual.Status);
            Assert.AreEqual(expected.ImportanceLevel, actual.ImportanceLevel);

            //test that category base info is NOT updated
            //test orgiginal values with actual values after failed update

            //get actual values
            EntityCategory actualCategory = target.GetCategoryInfo(bindingOrg);
            //test old values with new
            //before and after update values should be the same
            Assert.AreEqual(expectedCategoryOrg.UID, actualCategory.UID);
            Assert.AreEqual(expectedCategoryOrg.Type, actualCategory.Type);
            Assert.AreEqual(expectedCategoryOrg.Required, actualCategory.Required);
            Assert.AreEqual(expectedCategoryOrg.Name, actualCategory.Name);
            Assert.AreEqual(expectedCategoryOrg.Dirty, actualCategory.Dirty);
            Assert.AreEqual(expectedCategoryOrg.Description, 
actualCategory.Description);
            Assert.AreEqual(expectedCategoryOrg.CustomerExclusive, 
actualCategory.CustomerExclusive);
            Assert.AreEqual(expectedCategoryOrg.Bind.Active, 
actualCategory.Bind.Active);
            Assert.AreEqual(expectedCategoryOrg.Bind.Required, 
actualCategory.Bind.Required);
            Assert.AreEqual(expectedCategoryOrg.Active, actualCategory.Active);
        }
    }
}
```

### 5.3.5 Get Entity from GUI test

Det sidste udvalgte UnitTest-eksempel skal vise om base*element* objektet kan hentes fra GUI UseControl'erne, hvilket skal bruges til at udtrække værdier som brugeren har indtastet som *element*værdier. For at kunne tilgå objekters protected og privat members gør UnitTest brug af accessors, som kan operere på disse.
Dette gøres ved at give testobjektet til accessor-klassen, hvorefter accessor-klassen bruges som testobjekt.

Først initialiseres valgte værdier til UseControl'en hvorefter flere af dens værdier ændres, og til sidst hentes via `GetEntityCategoryFromUI()` metode.
De hentede værdier valideres op imod de indsatte.

```
...
[TestClass()]
    public class UCEntityCategoryInfoTest: IMessageFeed
    {
...
         /// <summary>
        ///A test for GetEntityCategoryFromUI ()
        ///</summary>
        [DeploymentItem("Infopaq.MediaAnalysis.MARSAdmin.Frontend.GUI.exe")]
        [TestMethod()]
        public void GetEntityCategoryFromUITest()
        {
            UCEntityCategoryInfo target = new UCEntityCategoryInfo();


UnitTestProject.Infopaq_MediaAnalysis_MARSAdmin_Frontend_GUI_UserControls_UCEntityCate
goryInfoAccessor accessor = new
UnitTestProject.Infopaq_MediaAnalysis_MARSAdmin_Frontend_GUI_UserControls_UCEntityCate
goryInfoAccessor(target);

            EntityCategory actual;

            EntityUIDName expectedCustomer = new EntityUIDName();
            expectedCustomer.Name = "testing correct";
            expectedCustomer.UID = 56;


accessor.entityUIDNameBindingSourceCustomerExclusive.Add(expectedCustomer);
            accessor.chkBxBindRequired.Checked = true;
            accessor.chkBxRequired.Checked = false;

            actual = accessor.GetEntityCategoryFromUI();

            Assert.AreEqual(expectedCustomer.UID, actual.CustomerExclusive);
            Assert.AreEqual(accessor.chkBxBindRequired.Checked, actual.Bind.Required);
            Assert.AreEqual(accessor.chkBxRequired.Checked, actual.Required);
        }
...
```

## 5.4 Resumé

UnitTest er en simpel, men værdifuld måde at teste *OO* applikationer på, og passer godt til det overordnede Functional *design pattern*, som bliver brugt i dette projekt. Alle implementerede test er valideret korrekt efter hensigten, og dermed kan testprojektet vurderes som succesfuldt.
Selvom dette kun er et lille udvalg af de mange UnitTest der er implementeret i projektet, giver det alligevel et overblik over hvordan UnitTest virker og hvordan de forskellige *design patterns* testes.

# 6 Udvidelser

Herunder er opskrevet små som store udvidelser eller rettelser til de efterkommende iterationer i den før om talte *UP* projektmetode.

- Fordi en visning i navigationstræet i GUI niveau kun viser basis-*element*-værdierne, vil et logisk OG ved egenskaben 'active' i både binding og basis *element* vise om denne binding er aktiv eller ej. Derpå kan der vises visuelt i træet om den er aktiv. PT vises kun basis *elementets* aktiv flag, hvilket helt sikkert viser når den ikke er aktiv da det er logisk OG. Men hvis bindingsværdien 'active' ikke er sat vil den stadig vises som 'active' i træet uden dog at være det (Det kan stadig ses i *element*data GUI).
- Checkbox treenodes kunne være en måde at lave 'active' synlig på direkte på træet. Dette ville gøre det lettere at se og redigere *elementers* 'active' flag.
- *Concurrency control* som er nævnt i afsnit 2.2.3 Data Overensstemmelsesproblemer.
- Udvidelse til filtrering af navigations-GUI; søg i *elementer* via navn og description eller andre basale *element*data.
- Suggestion funktioner kan udvides med ordbøger, som retstavning på flere sprog eller semantik. AI til indlæring af hele eller dele af strukturer via f.eks. semantik for de udtryk givne strukturer måtte indeholde.
- Copy/paste funktionalitet i treeview, samt evt. drag and drop.
- Højrekliks-menu med copy/paste, clone, create, refresh eller andre kontekstbestemte funktioner.
- Brugermanual, fordi systemet kun skal bruges internt i virksomheden, er denne del af projektet udsat til senere implementering, og det kunne evt. implementeres direkte i systemet via .NET Help klasser.
- Tråde, f.eks. brug af design pattern Object pool som er et Creational pattern. Dette kunne bruges i forbindelse med den provider struktur der er designet og implementeret. En let måde at implementere dette på ville være at bruge en .NET BackgroundWorker klasse som er et Thread Framework, det let kan sættes i gang og løbende rapportere status. Denne understøtter også afbrydning, hvilket er smart i forbindelse med Lazy initialization pattern der skal sørge for minimum af arbejde. Tråde kunne hjælpe med til at forbinde til *backend* systemet og dermed databasen uden at brugeren skal vente på GUI klassernes initialisering, hvilket kan få renderingen til at 'hænge'.
- Der kan med de eksisterende GUI klasser opbygges en wizard som guider brugeren gennem systemet som forklaret i afsnit 3.3 GUI Beskrivelse. Dette kræver lidt tilpasning af GUI klasserne, men bliver generelt understøttet grunde det valgte Functional *design pattern* som beskrevet i afsnit 3.1.3 Class Diagram og design patterns.
- Vise statistiske oplysninger om brugen af de forskellige *elementer*, for at hjælpe brugeren til at se mønstre i brugen af *elementerne*.
- Brugerrettigheder, hvorved brugeren skal identificeres og der på den måde kan gemmes hvem der har redigeret eller oprettet forskellige data.

# 7. Konklusion

Det har fra starten været en stor udfordring at designe et relativt stort projekt som det, men med en gennemgang af en række analyse trin som er beskrevet i 2. Analyse, blev behov og struktur hurtigt optegnet.

Hele projektet, *MARS Admin*, er forløbet inde for den givne tidsramme, og der er designet og implementeret de primære mål for projektet. Dette skal ses i lyset af, at en stor del af projektet er lagt i at designe en *enterprise solution* som skal overholde en række *design patterns*. En stor del af arbejdet har ligget i at få det fundamentale *n-tier* framework til at fungere med de tilhørende *design patterns* der er valgt.
Fordi dette fundament er designet og implementeret vil mange af de før omtalte udvidelser relativt let kunne designes og implementeres.

Der er en lille del af projektets sekundære mål, som ikke er implementeret. Fordi fundamentet for disse manglende metoder er lagt, vil dette ikke kræve de store ændringer af hverken design eller kildekode.

Der er vist at *MARS Admin* systemet er funderet på baggrund af en *n-tier* model via *design patters*, hvilket flere steder har vist sig at fungere rigtigt godt. Mange af eksemplerne, som er trukket frem i afsnit 4. Implementering, viser hvor godt *design patterns* er i praksis for forståelsen af komplekse modeller.
Det har også været en fordel, at bruge et over ordnet Funcional *design pattern*, hvilket har hjulpet med at indkapsle klassers ansvarsområder, og dermed simplificere den abstrakte forståelse af systemet.

Projektet har en lang række af mulige udvidelser, og deres hensigt er at fremme funktionaliteten og brugervenligheden af *MARS Admin*. Projektet er funderet på basale behov, men er designet til at håndtere en lang række udvidelser. Dette er langt hen af vejen designets skyld.

Denne rapport er udarbejdet af undertegnede:

S012351 Kåre Rune Christensen

# 8. Litteraturliste

| | |
|---|---|
| **Titel:** | Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. |
| **Forfatter:** | Craig Larman |
| **Udgave:** | Second Edition 2002 |

| | |
|---|---|
| **Titel:** | UML Distilled<br>Applying the standard object modeling language |
| **Forfatter:** | Martin Fowler with Kendall Scott |

| | |
|---|---|
| **Titel:** | Patterns of Enterprise Application Architecture |
| **Forfatter:** | Martin Fowler with contributions |

| | |
|---|---|
| **Titel:** | Introduction To Algorithms |
| **Forfatter:** | Thomas H Cormen<br>Charles E. Laiserson<br>Ronald L. Rivest<br>Clifford Stein |
| **Udgave:** | Second Edition 2001 |

| | |
|---|---|
| **Titel** | MDSN |
| **Forfatter:** | Microsoft ® Corporation |
| **Udgave** | Online http://msdn2.microsoft.com<br>Lokal MSDN Library for Visual Studio 2005 |

# 9. Bilag

## 9.1 CD indhold

- Denne rapport `'MARSAdminRapport.pdf'` i digitalt format.
- .NET C# kildekode med solution- og projekt-filer findes i mappen `Infopaq.MediaAnalysis.MARSAdmin`
- Stored Procedure kildekode er i mappen `StoredProcedures` i tekst format.
- MSSQL2005 Databasefil `'AnalysisDB_MARSDev.bak'`

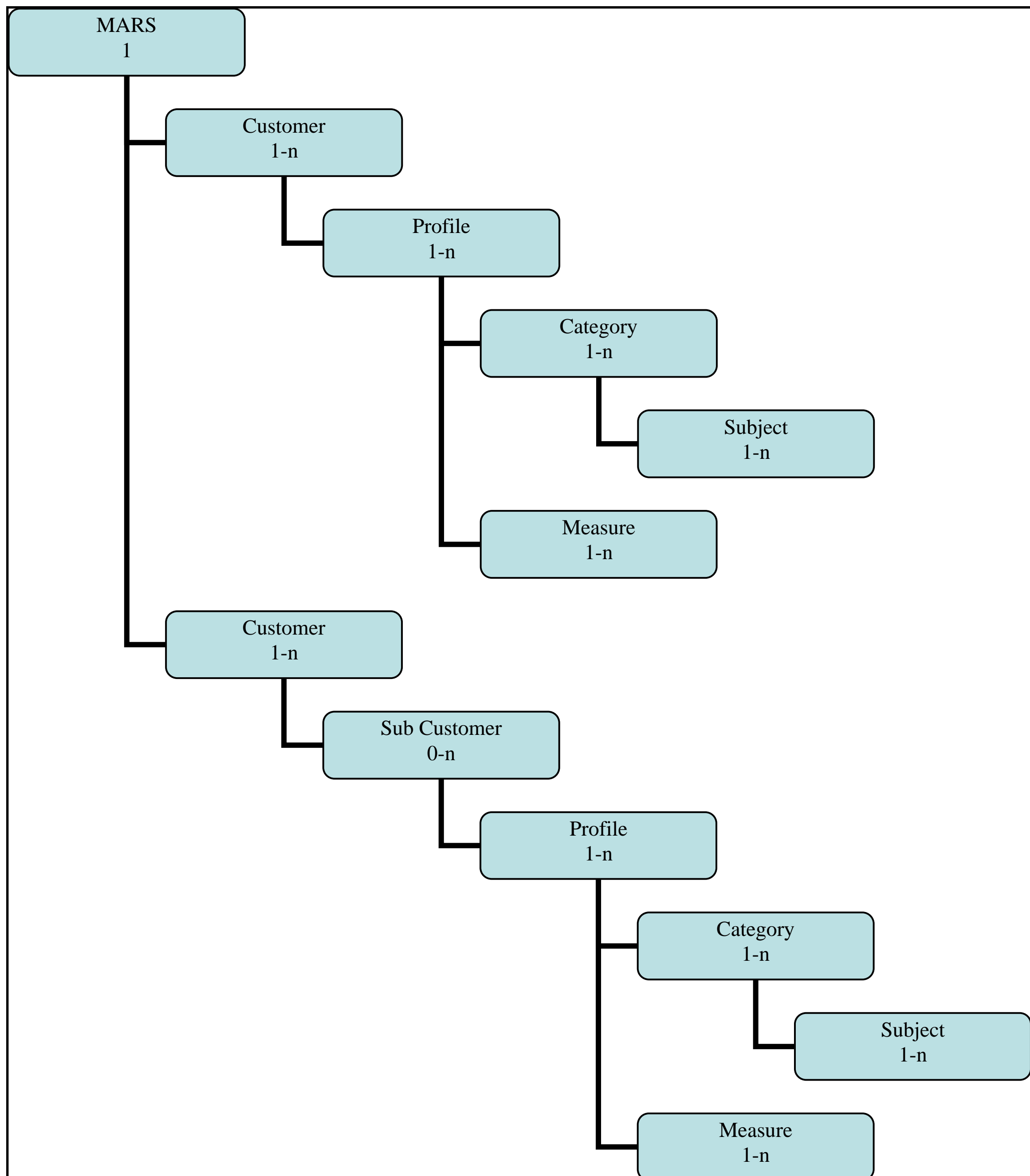
## 9.2 Projektplan

| **Uger / Punkter** | **Uge 1** | **Uge 2** | **Uge 3** | **Uge 4** | **Uge 5** | **Uge 6** | **Uge 7** | **Uge 8** | **Uge 9** | **Uge 10** |
|---|---|---|---|---|---|---|---|---|---|---|
| **Design UML** | | | | | | | | | | |
| Generelt Design | | | | | | | | | | |
| Use Case Diagram | | | | | | | | | | |
| Use Cases | | | | | | | | | | |
| Domæne Diagram | | | | | | | | | | |
| Sekvens Diagram | | | | | | | | | | |
| Klasse Diagram | | | | | | | | | | |
| Deployment Diagram | | | | | | | | | | |
| **Implementering** | | | | | | | | | | |
| Database lag | | | | | | | | | | |
| WebService lag | | | | | | | | | | |
| GUI | | | | | | | | | | |
| **Test** | | | | | | | | | | |
| Løbende test af klasser | | | | | | | | | | |
| Hoved Test | | | | | | | | | | |
| **Rapportskrivning** | | | | | | | | | | |
| UML dokumentation | | | | | | | | | | |
| Analyse | | | | | | | | | | |
| Kommentarer til implementering | | | | | | | | | | |
| Samling og afslutning af rapport | | | | | | | | | | |

Mørke celler markeret med farven: indikere hovedpunkter.
Lyse celler markeret med farven: indikerer underpunkter.

Projektplanen er groft skitseret og er skal kun bruges som en vejledning til projektforløbet. Også selve punkterne er løst defineret, således enkelte punkter er udeladt eller erstattet af nye mere passende i projektforløbet.

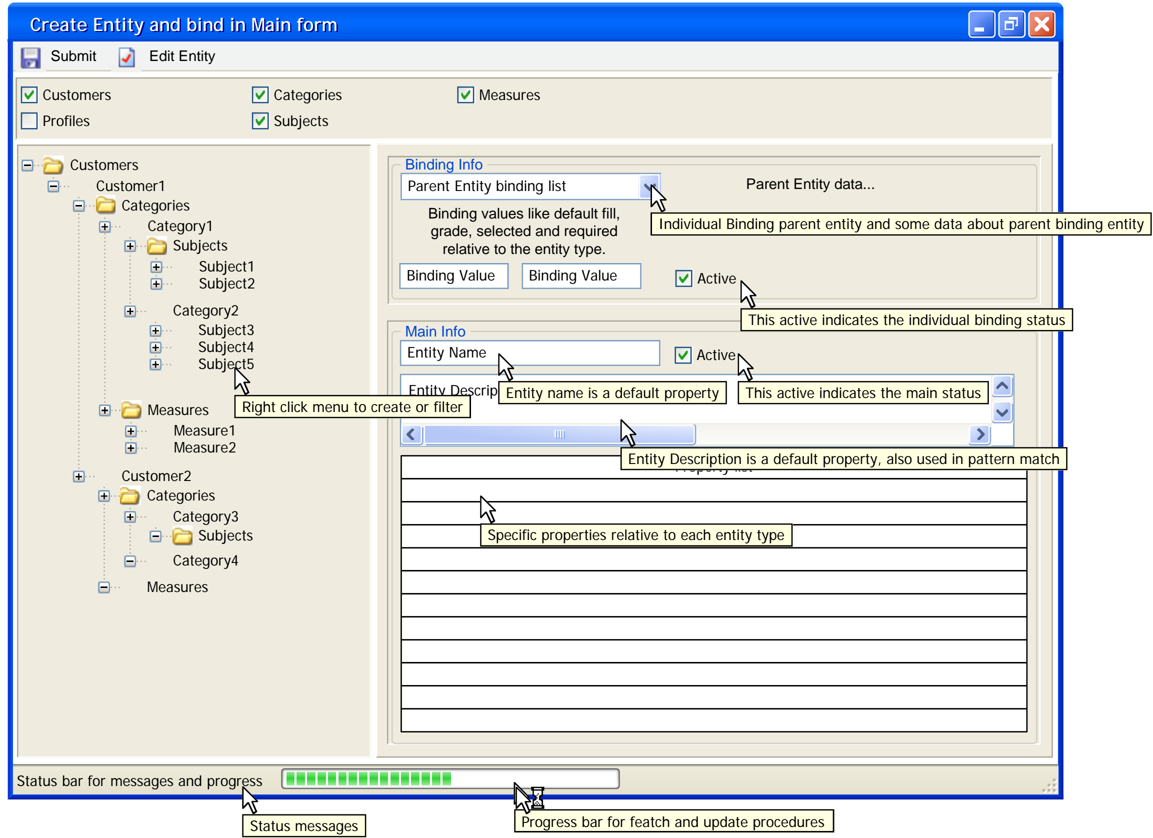
## *9.3 Element hierarki*



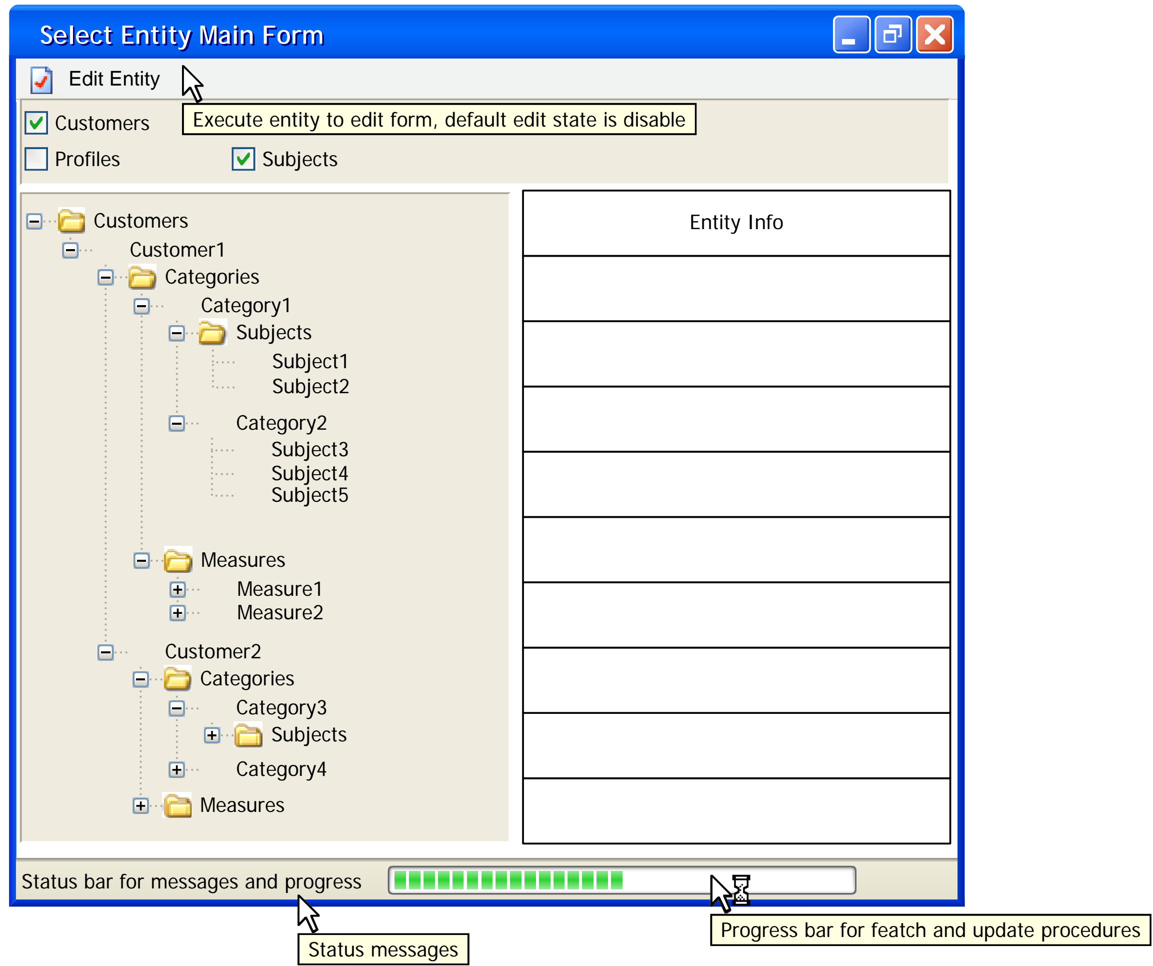

- Øverste element MARS er rodelementet.
- *Customer* er det eneste element der er barn af MARS Notation 1-n. *Sub Customer* er et selektivt barn til *Customer*. Notation 0-n
- Herunder er mindst ét element af hver obligatorisk. Notation 1-n, Profiler er barn af Customer typer, som nævnt har 1-n notation
- *Category* og *measure* har samme niveau med notation 1-0. *Category* har dog børn, *subjects* som også bruger notation 1-n

# *9.4 GUI Udkast*

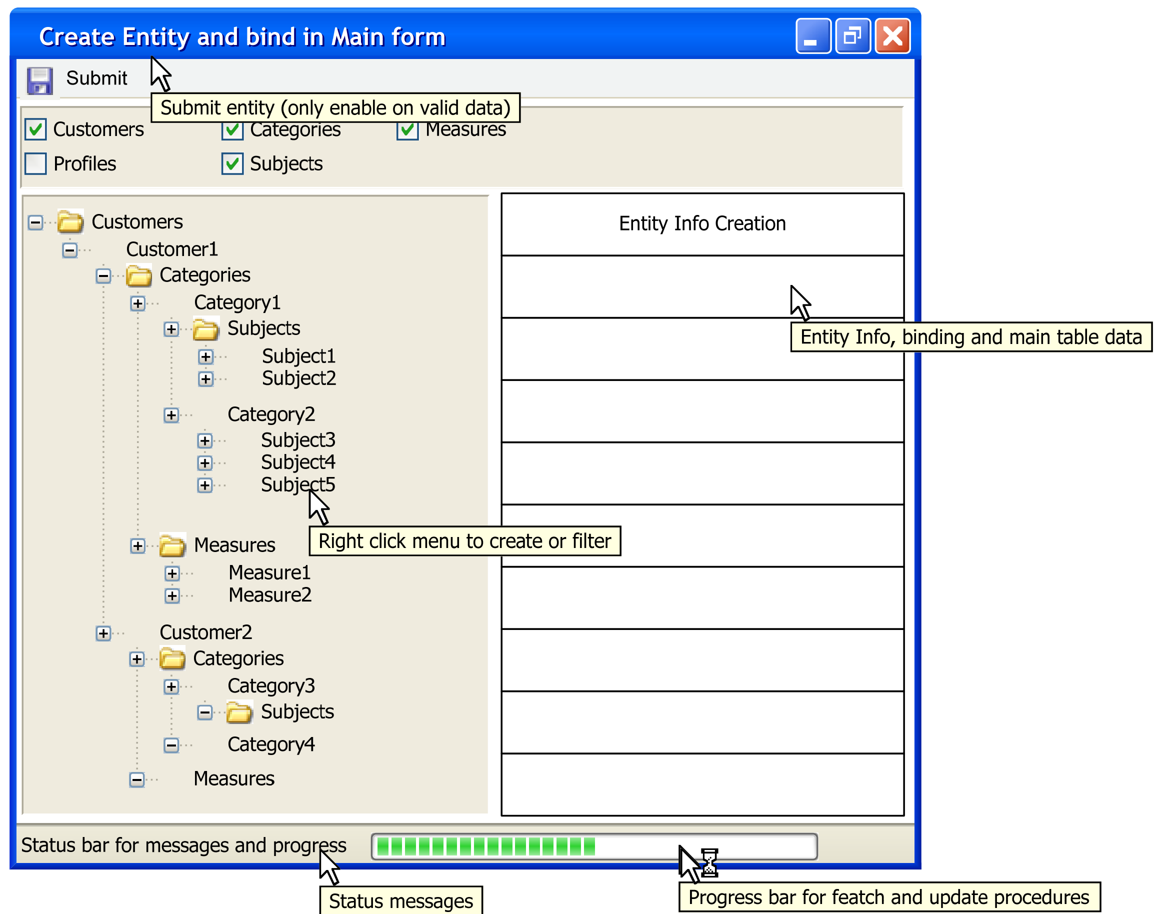
## 9.4.1 Hoved Form i abstrakt stadie



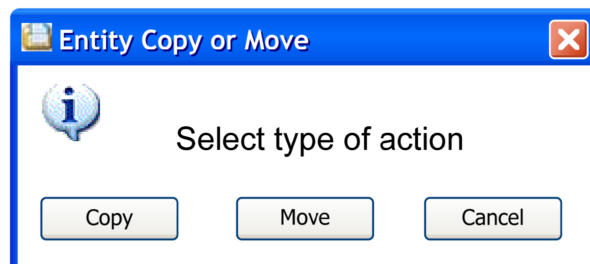
## 9.4.2 Valg af Entitet



## 9.4.3 Oprettelse af *element*




## 9.4.4 Flyt og Kopier Dialogboks

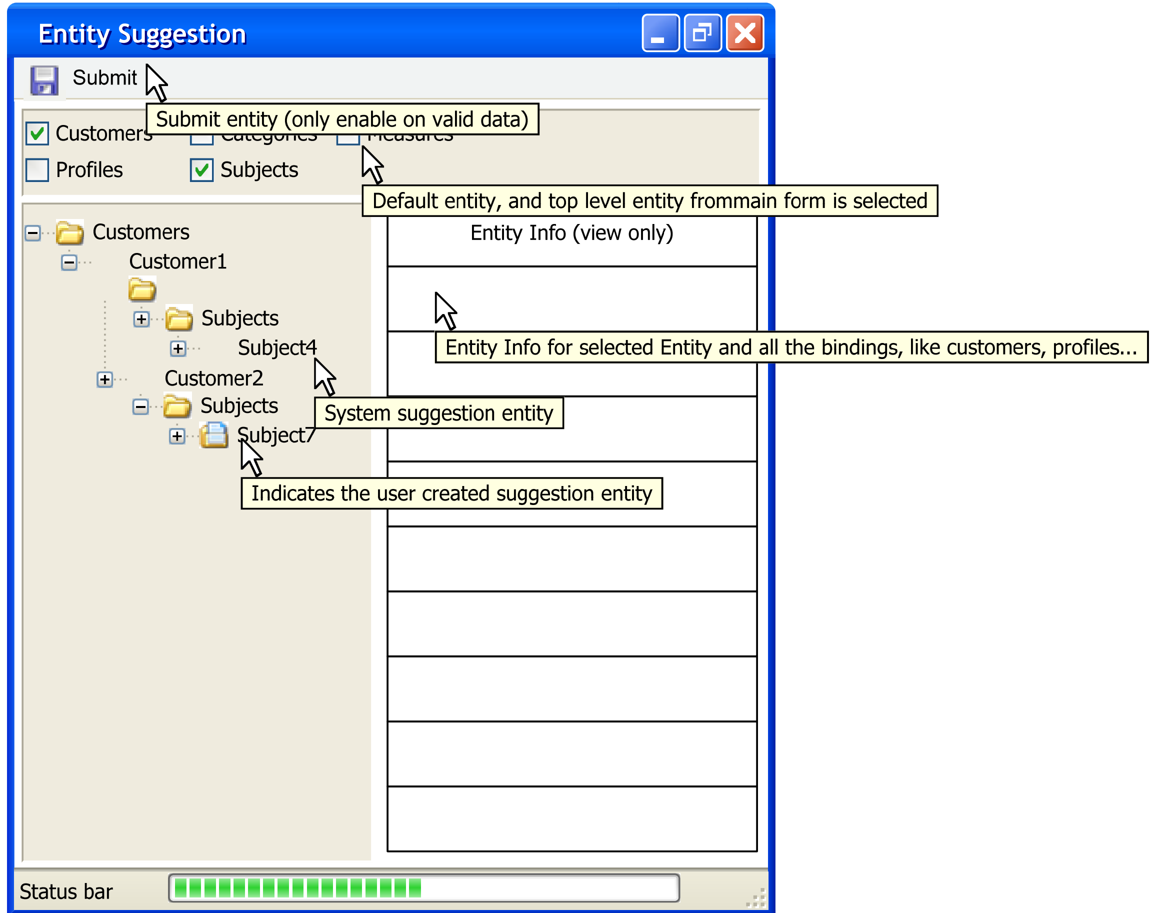


## 9.4.5 Visning af *element* detaljer

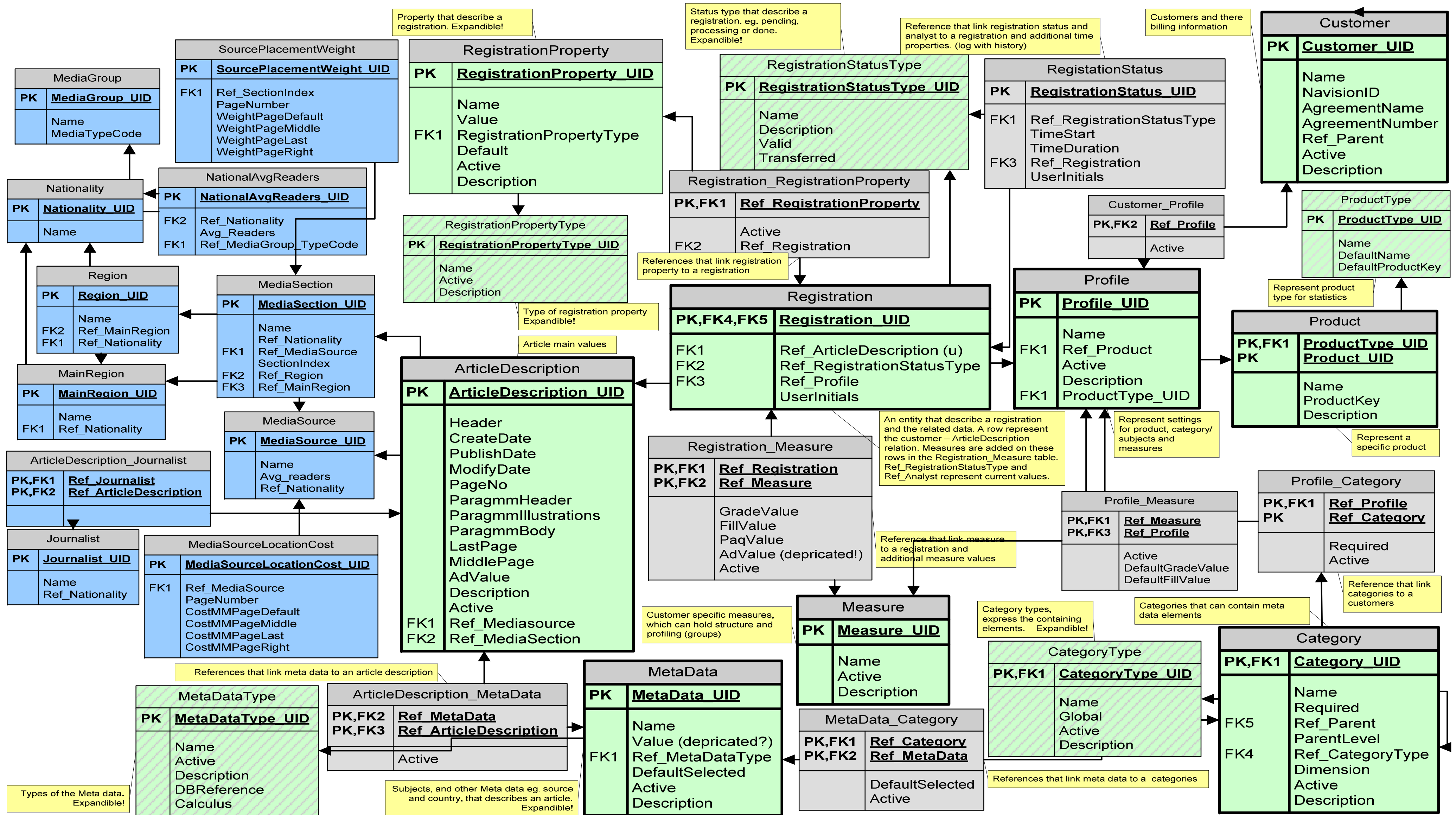
Binding Info
Parent Entity binding list
Parent Entity data...
Individual Binding parent entity and some data about parent binding entity
Binding values like default fill, grade, selected and required relative to the entity type.
Binding Value
Binding Value
Active
This active indicates the individual binding status

Main Info
Entity Name
Active
Entity Descrip
Entity name is a default property
This active indicates the main status
Entity Description is a default property, also used in pattern match
Specific properties relative to each entity type

## 9.4.6 *Element*-forslag Form

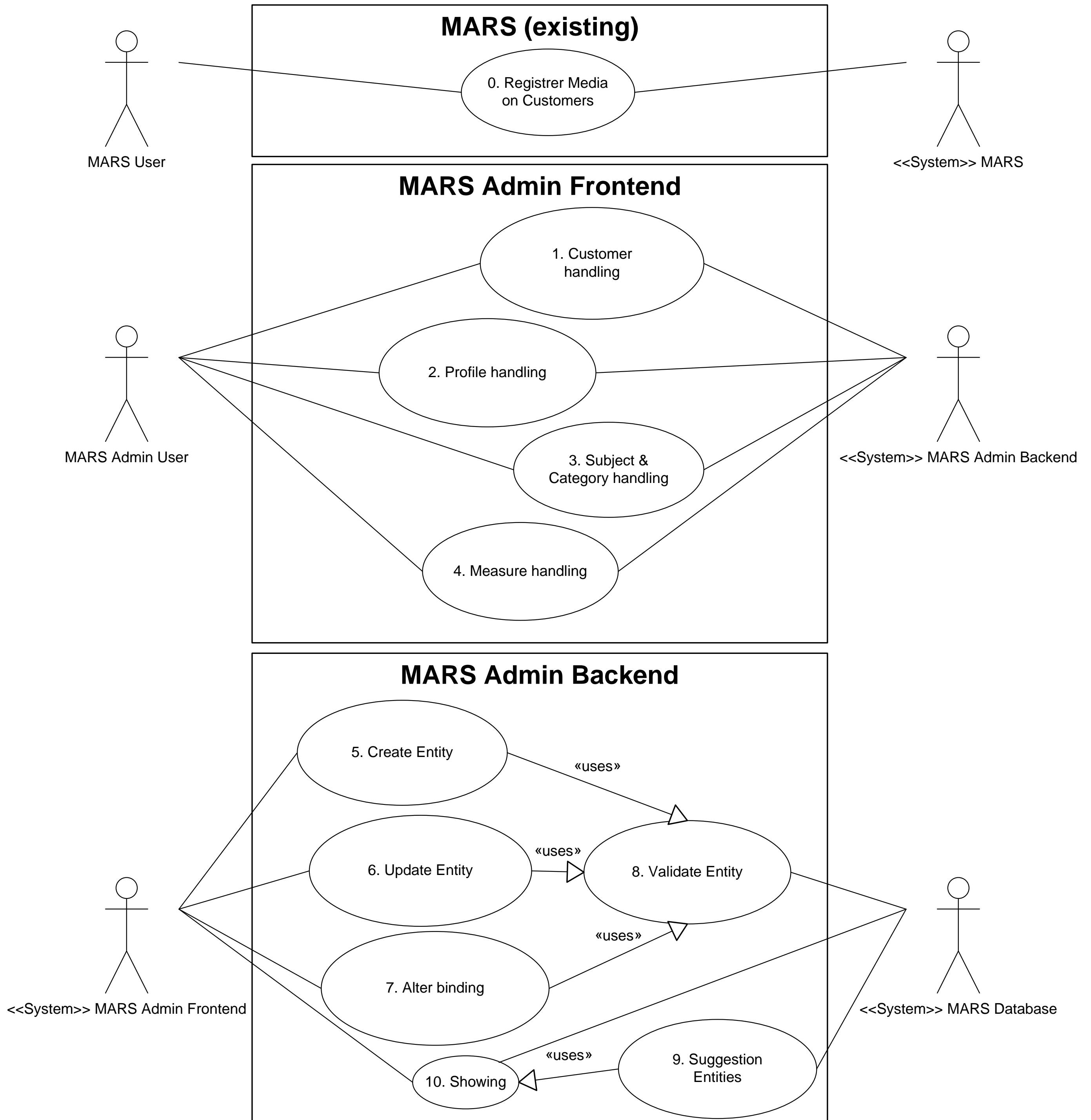


# 9.5 MARS Database model

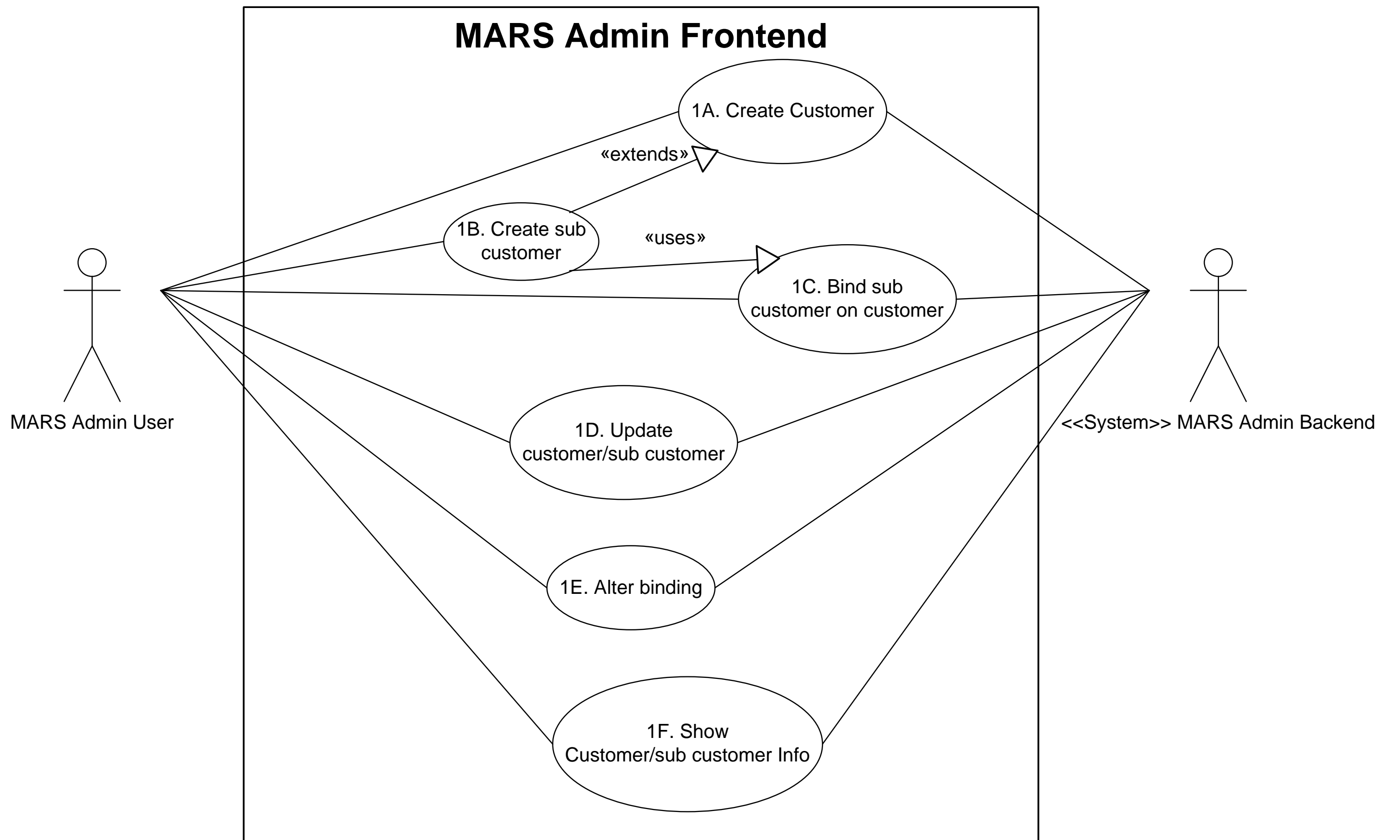


# 9.6 UML Use Case Diagrammer

## 9.6.1 Use Case Diagram Top Level

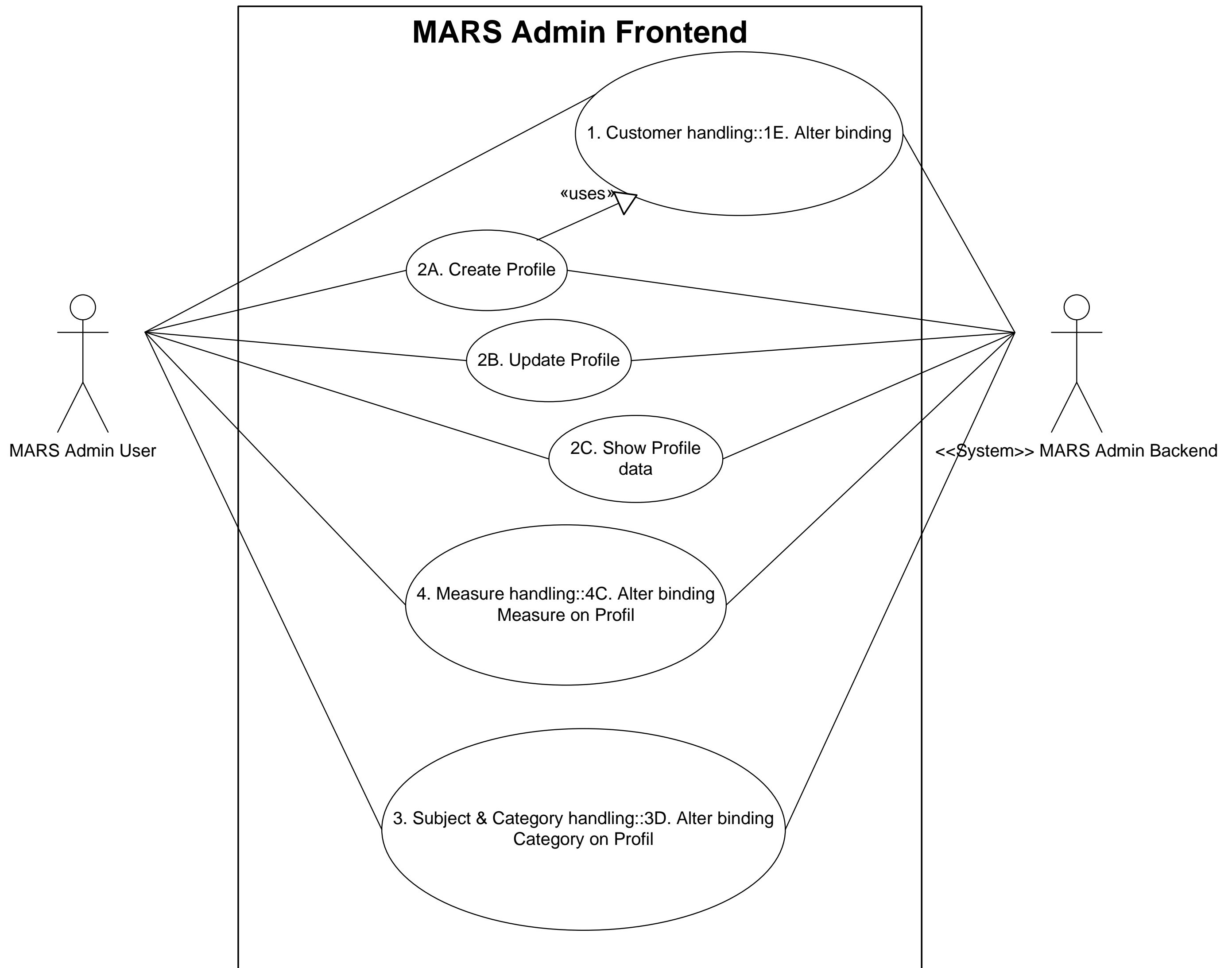


### 9.6.2 Use Case Diagram 1. Customer handling



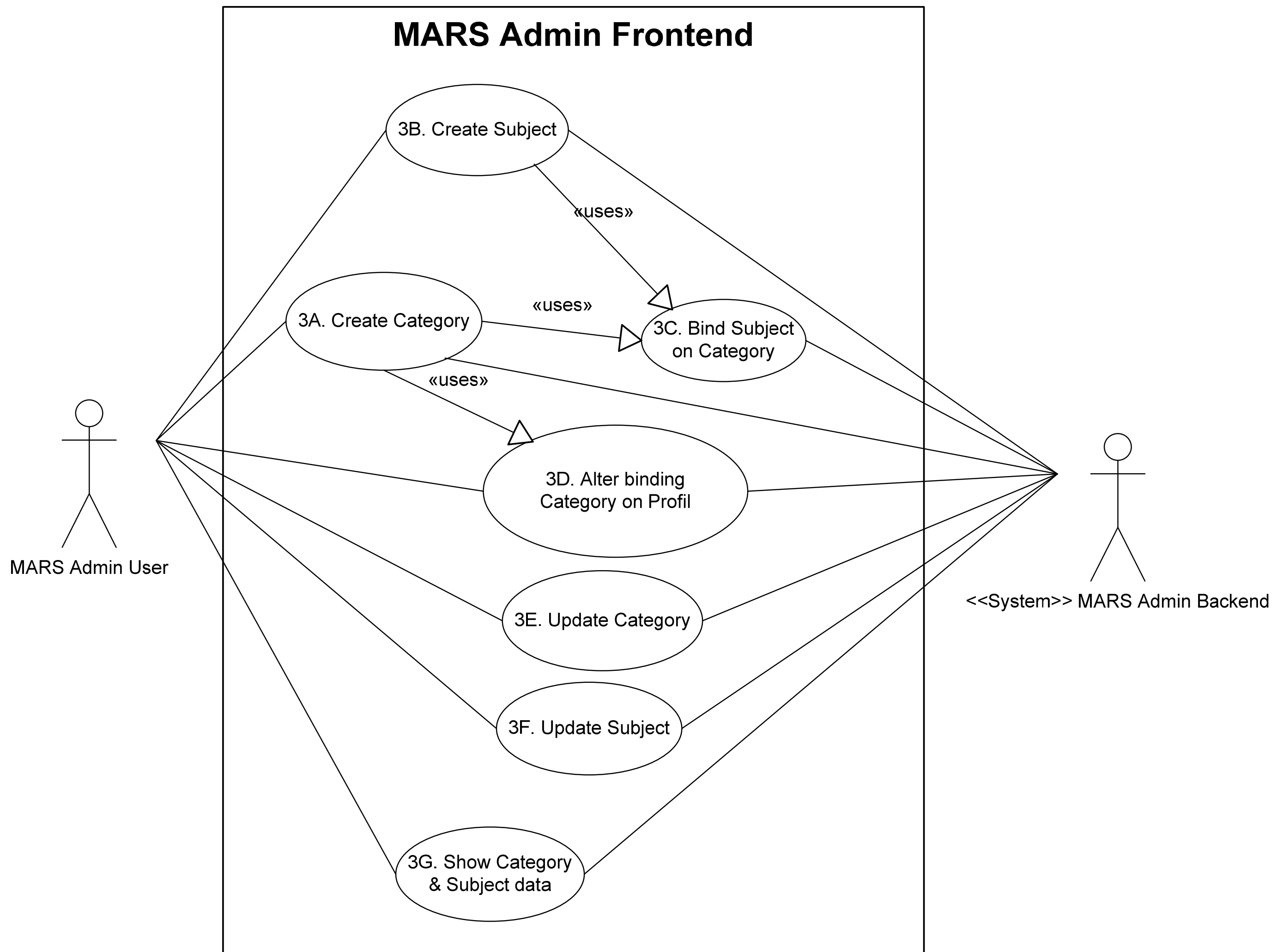

- 1A. Oprette en *customer* og systemet opretter den så videre ned i systemet.
- 1B. Udvidelse af 1A. som opretter *customer* som *sub customer*. Kan oprette *sub customer*. (samme som kunde bare med en forælder).
- 1C. Bliver altid brugt når en *sub customer* 1B. oprettes. kun 1 binding (ikke virtuel). Binder *sub customer* på en eksisterende *customer*.
- 1D. Opdatere *customer*. (kan inkludere 1C)
- 1E. Håndtering af bindinger fra en *customer* til en *profile*. en *profile* kan binde på flere *customers*, men bliver ikke brugt. (kan inkludere 1C).
- 1F. Visning af *customer* data, så *MARS Admin* kan håndtere disse.

## 9.6.3 Use Case Diagram 2. Profile handling



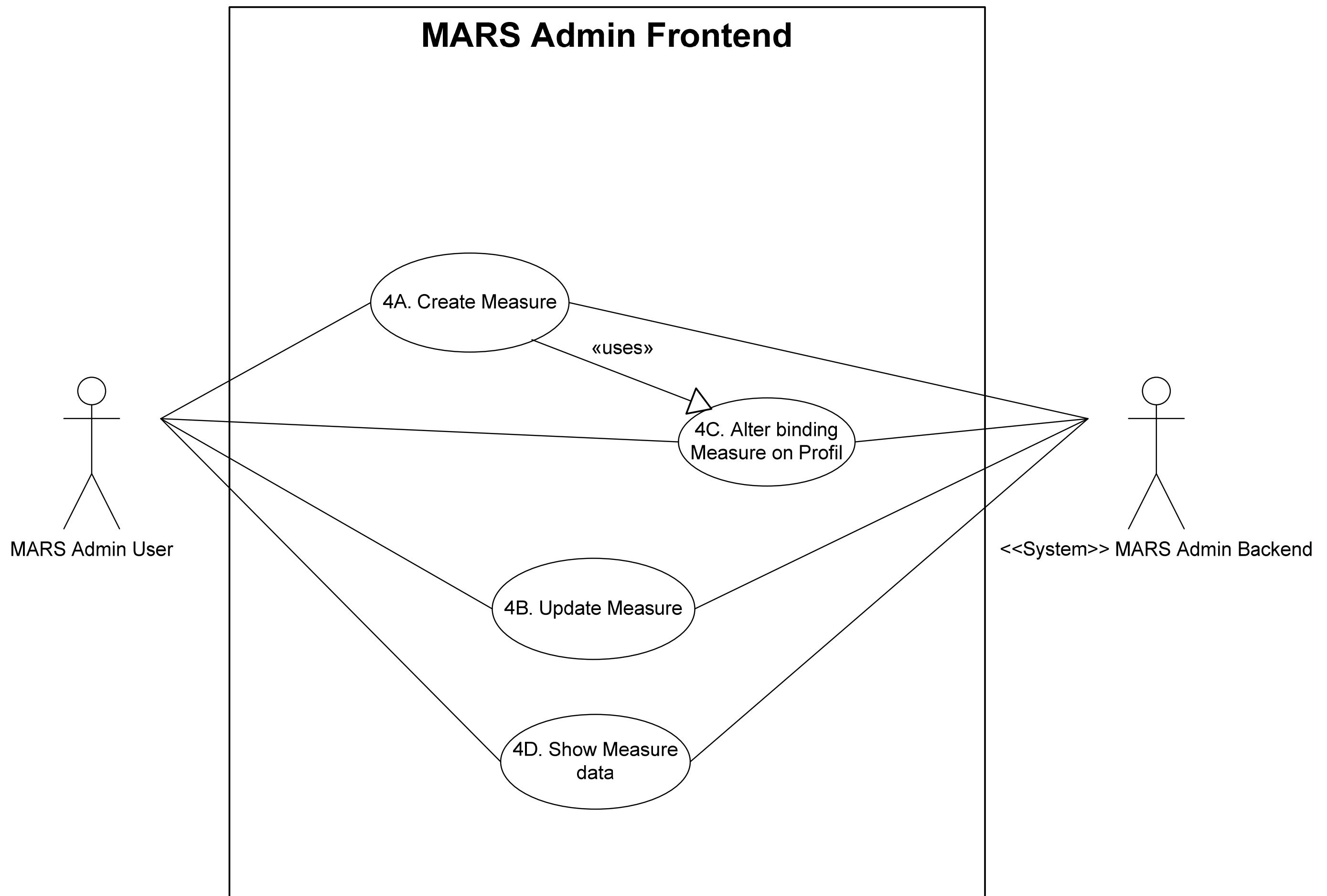

- 1E. beskrevet i bilag 9.6.2 Use Case Diagram 1. Customer handling, skal forekomme når der oprettes en ny *profile*, således en *profile* altid binder på en *customer*.
- 2A. Oprettelse af en *profile*.
- 2B. Opdatering af en *profile* (kan inkludere 1E).
- 2C. Visning af de forskellige *profile* data.
- 4C. beskrevet i bilag 9.6.5 Use Case Diagram 4. Measure handling.
- 3D. beskrevet i bilag 9.6.4 Use Case Diagram 3. Category and Subject handling.

## 9.6.4 Use Case Diagram 3. Category and Subject handling



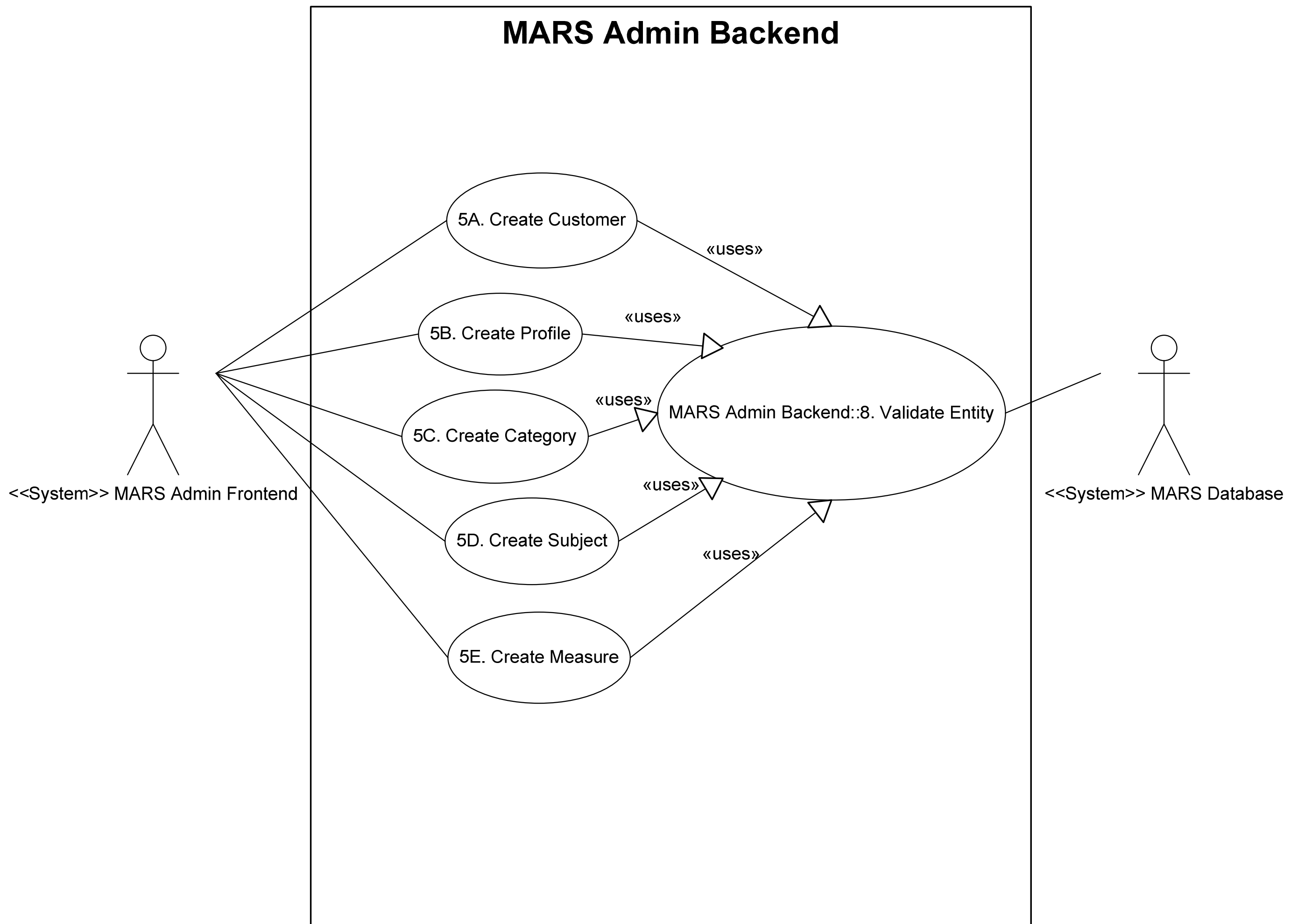

- 3A. Opret en *category* og systemet opretter den videre ned.
- 3B. Opret et *subject* og systemet opretter den videre ned.
- 3C. Ved oprettelse af både *category* og *subject* skal denne Use Case forekomme. Den binder et *subject* på en *category*. Der skal ikke kunne forekomme et *subject* uden *category*.
- 3D. Håndtering af bindinger mellem *profile* og *category*. Der skal ikke kunne forekommen en *category* der ikke er bundet til en *profile*.
- Flere *profiles* kan binde på den sammen *category* (virtuelt), men er i praksis bliver ikke brugt endnu.
- En *category* kan være bundet til flere *profiles*, men bliver ikke brugt.
- 3E. opdater *category* (kan inkludere 3C).
- 3F. opdater *subject* (kan inkludere 3C).
- 3G. Visning af de forskellige *category* og *subject* data, således de andre Use Cases kan udføres i dette diagram.

## 9.6.5 Use Case Diagram 4. Measure handling



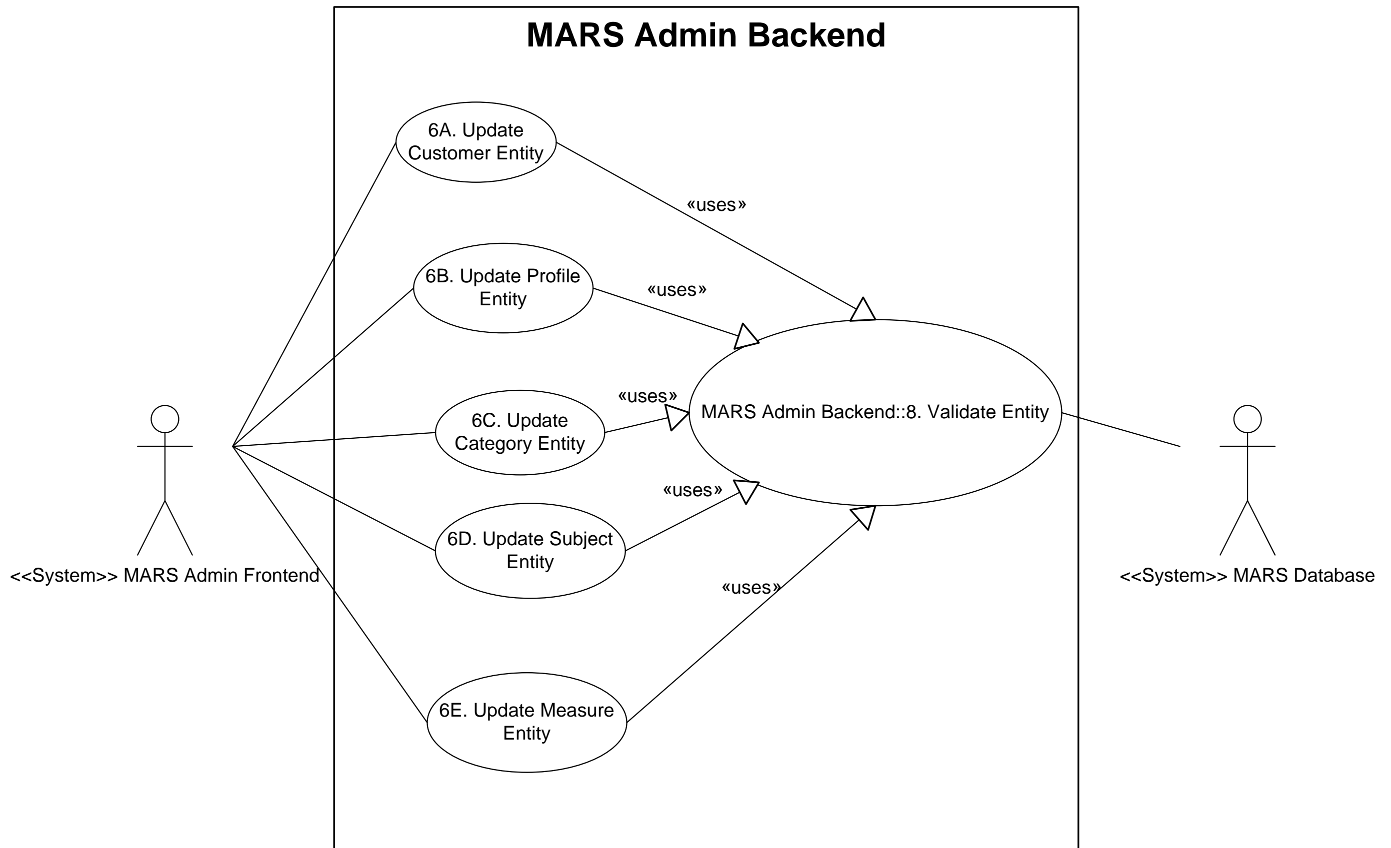

- 4A. Oprettelse af nyt *measure elementer*.
- 4B. Opdatering af *measure*, (kan inkludere 4C).
- 4C. Rediger binding mellem *profil* og *measure*. (flere profiler kan binde på samme *measure* (virtuelt), men bliver ikke brugt)
- 4D. Visning af de forskellige *measure* data.

## 9.6.6 Use Case Diagram 5. Create Entity



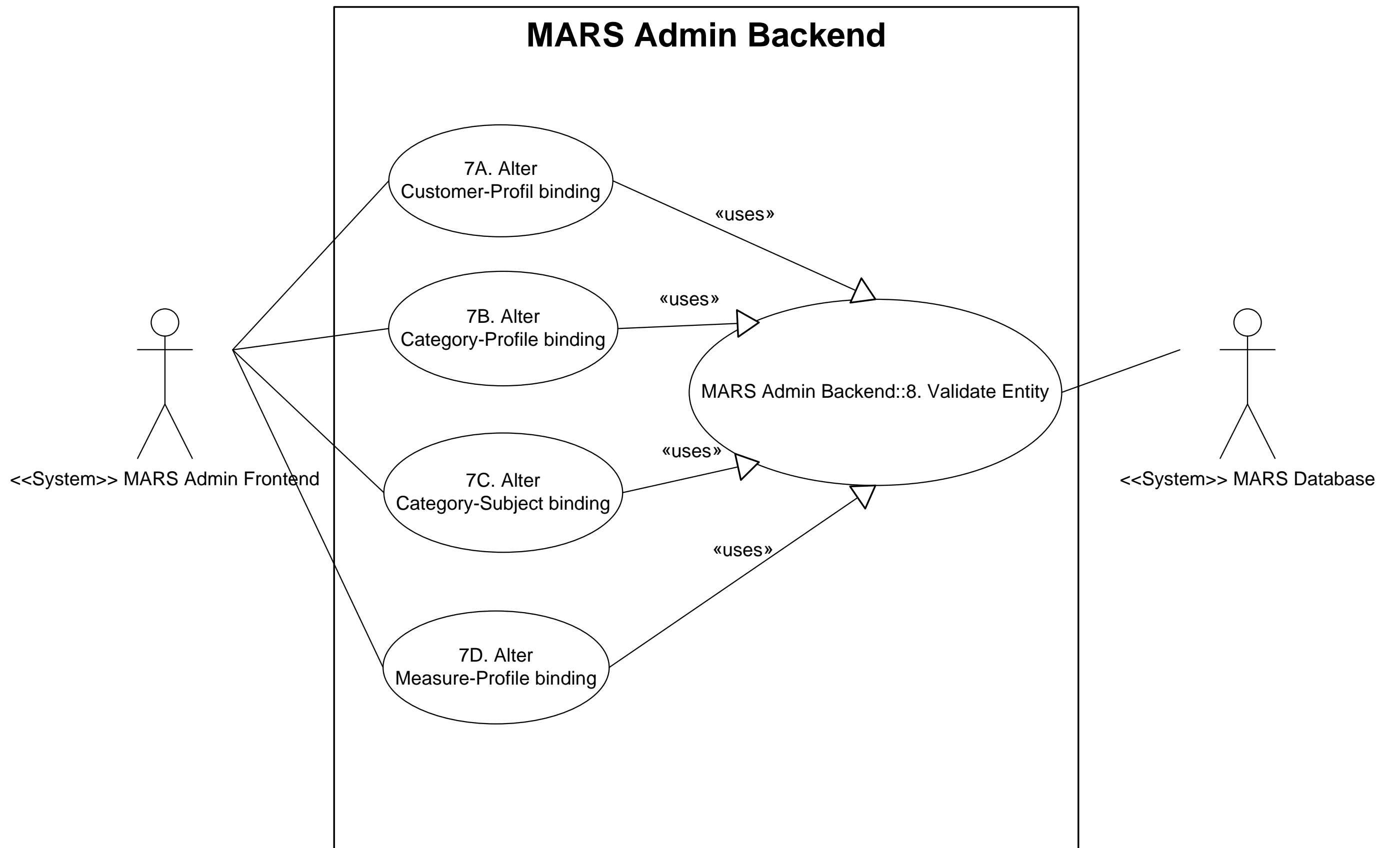

- 5A. Oprettelse af en *customer element*.
- 5B. Oprettelse af en *profile element*.
- 5C. Oprettelse af en *category element*.
- 5D. Oprettelse af *subject element*.
- 5E. Oprettelse af *measure element*.
- 8. Validering af *elementer* skal bruges af alle oprettelser, og er beskrevet i bilag 9.6.9 Use Case Diagram 8. Validate Entity.

## 9.6.7 Use Case Diagram 6. Update Entity



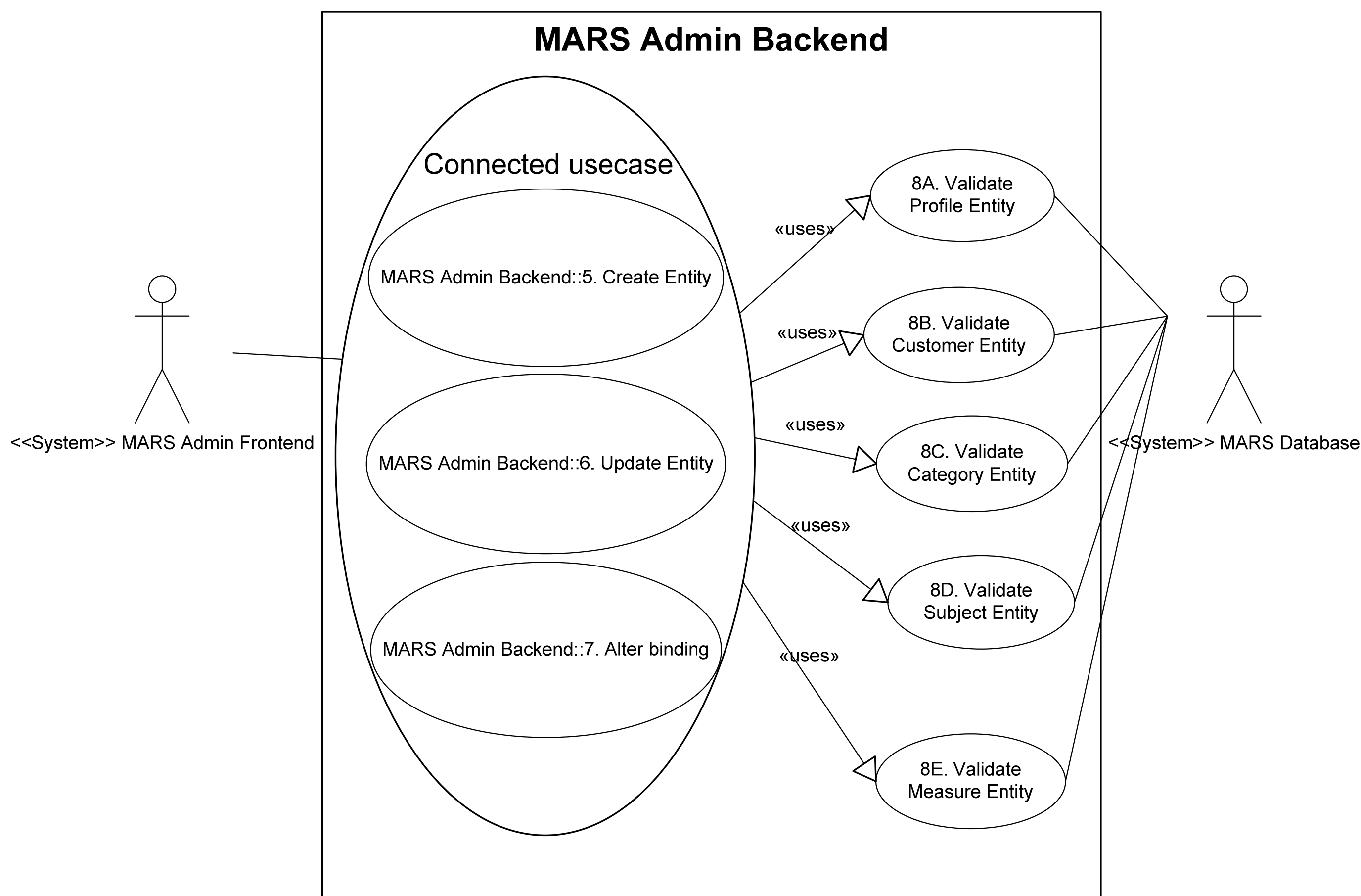

- 6A. Opdatering af *customer elementer*.
- 6B. Opdatering af *profile elementer*.
- 6C. Opdatering af *category elementer*.
- 6D. Opdatering af *subject elementer*.
- 6E. Opdatering af *subject elementer*.
- 8. Validering af *elementer* skal bruges af alle opdateringer, og er beskrevet i bilag 9.6.9 Use Case Diagram 8. Validate Entity.

## 9.6.8 Use Case Diagram 7. Alter Binding




- 7A. Rediger binding mellem *customer* og *profile*.
- 7B. Rediger binding mellem *category* og *profile*.
- 7C. Rediger binding mellem *category* og *subject*.
- 7D. Rediger binding mellem *measure* og *profile*.
- 8. Validering af *elementer* skal bruges af alle bindinger, og er beskrevet i 9.6.9 Use Case Diagram 8. Validate Entity.

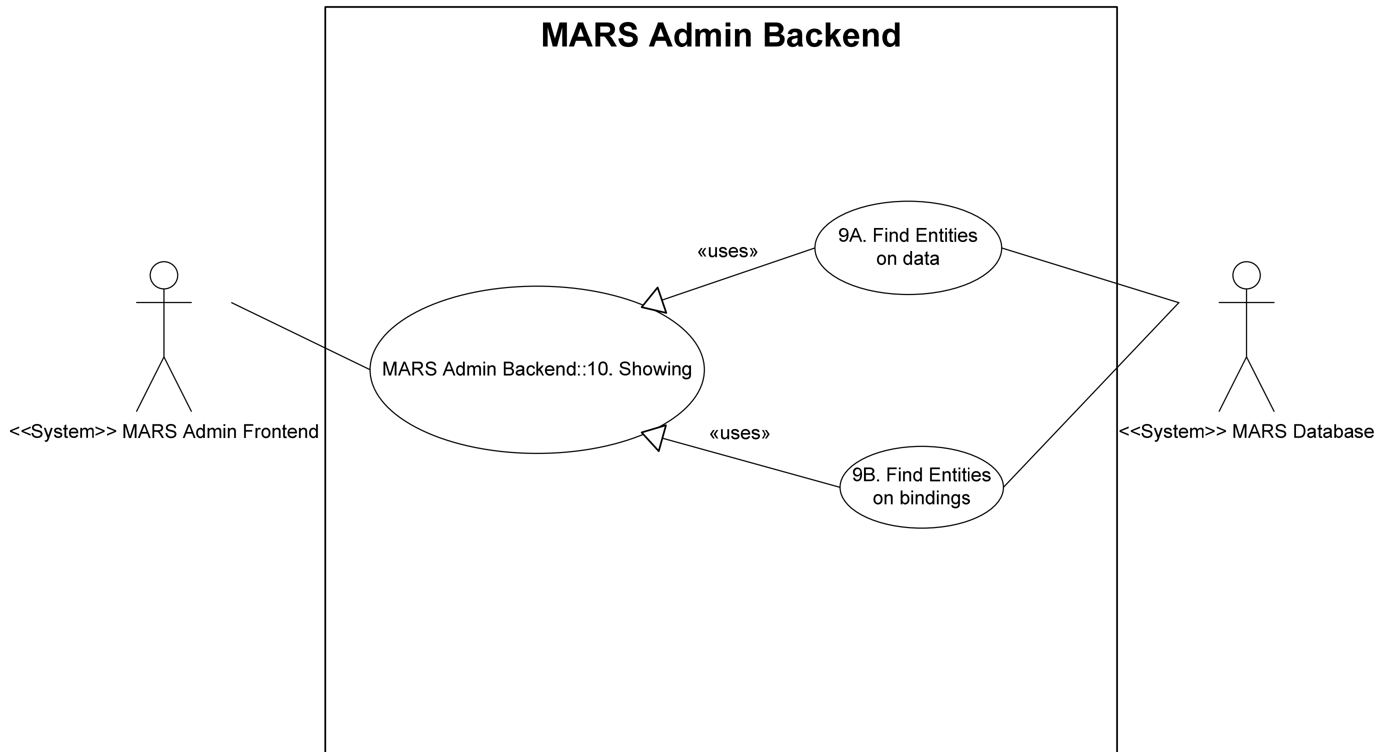
## 9.6.9 Use Case Diagram 8. Validate Entity




Diagrammet skal tolkes som hver Use Case 5, 6 og 7 i cirklen har <<Uses>> bindinger til hver af de højreliggende Use Cases 8A, 8B, 8C, 8D og 8E. Der findes en lignende notation i UML 2.0, men det brugte UML værktøj var ikke i stand til at generere en korrekt visualisering.
Det ville heller ikke give mening at opdele diagrammet yderligere, grundet overblikket over sammenhængen af top level Use Case Diagrammet ville forsvinde.

- 8A. Validering af *profile elementer*, dens logiske data og indholdet i forhold til eksisterende *profile elementer* og bindinger til *customers*.
- 8B. Validering af *customer elementer*, dens logiske data og indholdet i forhold til eksisterende *customers* og *profiles*.
- 8C. Validering af *category elementer*, dens logiske data og indhold i forhold til eksisterende *categories* på samme og andre profiler og kunder, samt validering af dens *subjects* i forhold til andre *categories*.
- 8D. Validering af *subject elementer*, dens logiske data og indhold i forhold til eksisterende subjects og deres *caterories*.
- 8E. Validering af Measures, det logiske data og indhold i forhold til eksisterende *measures* både i og uden for den *profile* og *customer*.

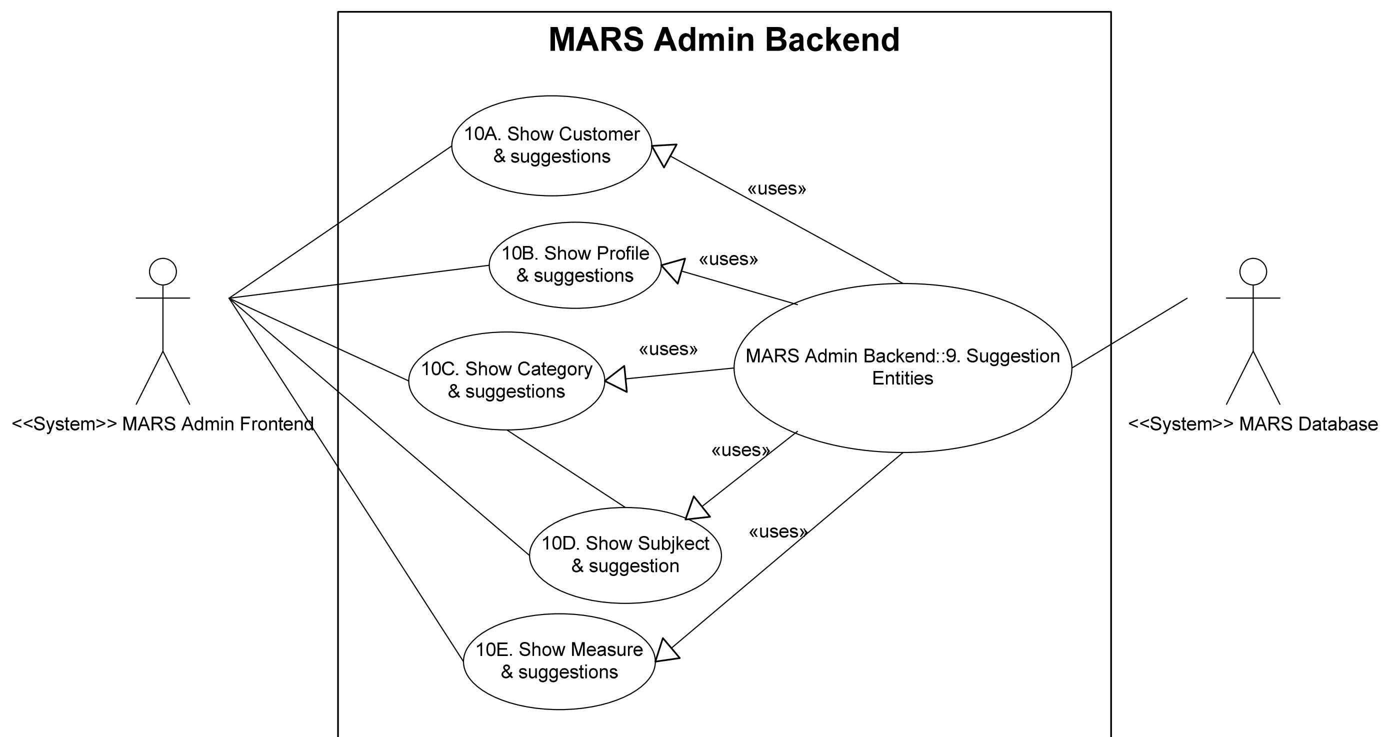
## 9.6.10 Use Case Diagram 9. Suggestion




Dette diagram viser måderne hvorpå systemet skal finde lignende entiteter der skal foreslås til visning videre op i systemet.

- 9A. Find *elementer* der har lignende dataværdier.
- 9B. Find *elementer* der har lignende bindinger, som fx. er i samme *category*, eller findes i samme *profile* eller *customer*
- Visning bruges altid af 9x. - og viser resultatet videre til *frontend*. Beskrevet i bilag 9.6.11 Use Case Diagram 10. Showing.

# 9.6.11 Use Case Diagram 10. Showing

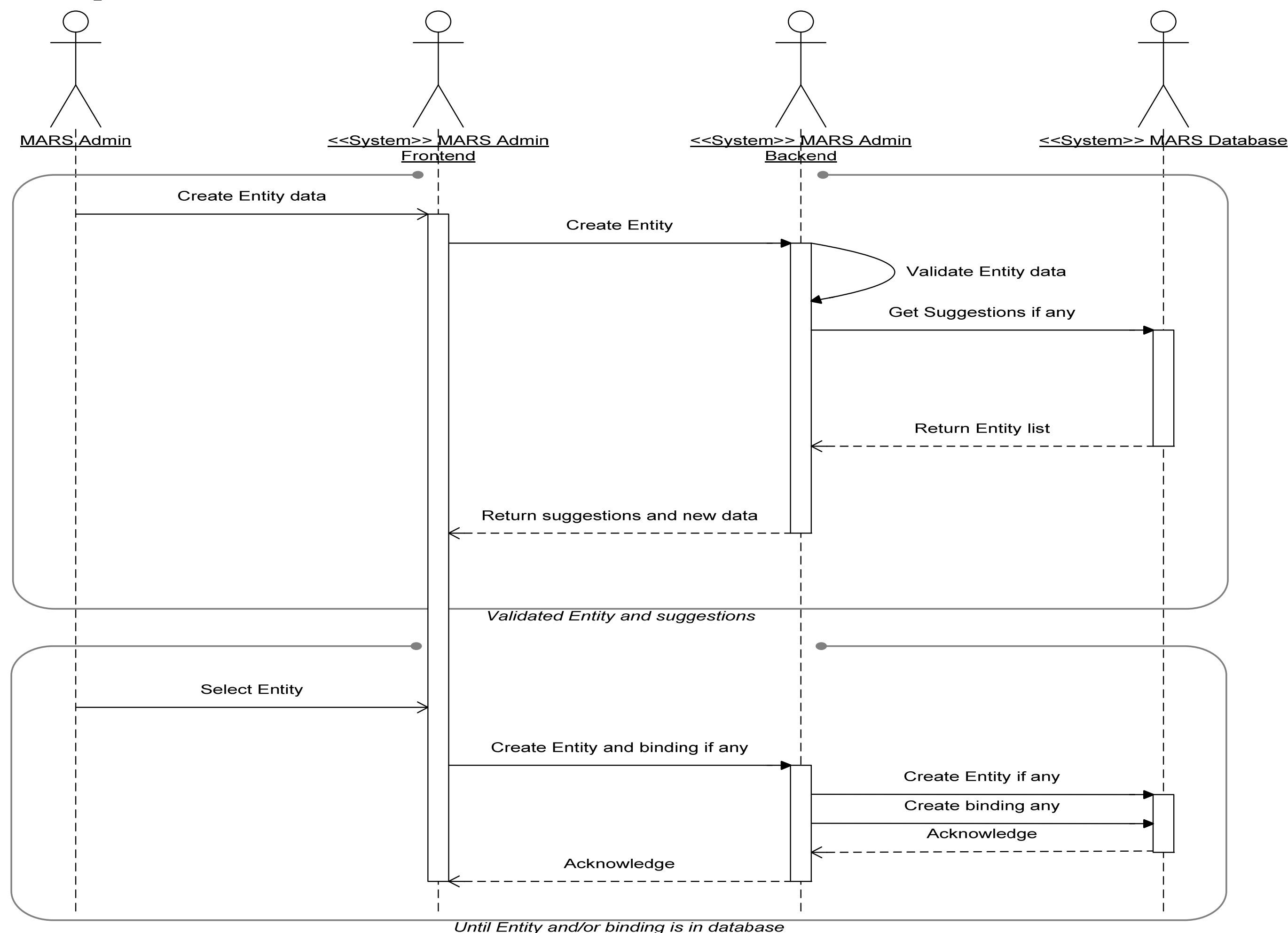



Dette diagram skal tolkes som et valg mellem de forskellige <<Uses>> bindinger, og ikke at alle <<Uses>> bindinger er aktive på samme tid.
Højre side er taget fra Use Case 9 som vist i bilag 9.6.10 Use Case Diagram 9. Suggestion, og skal ses som en samlet Use Case for at skabe et bedre overblik.

- 10A. Visning af *customer elementer* samt eventuelle forslag.
- 10B. Visning af *profile elementer* samt eventuelle forslag.
- 10C. Visning af *category elementer* samt eventuelle forslag.
- 10D. Visning af *subject elementer* samt eventuelle forslag.
- 10E. Visning af *measure elementer* samt eventuelle forslag.
- 9. beskrevet i bilag 9.6.10 Use Case Diagram 9. Suggestion.

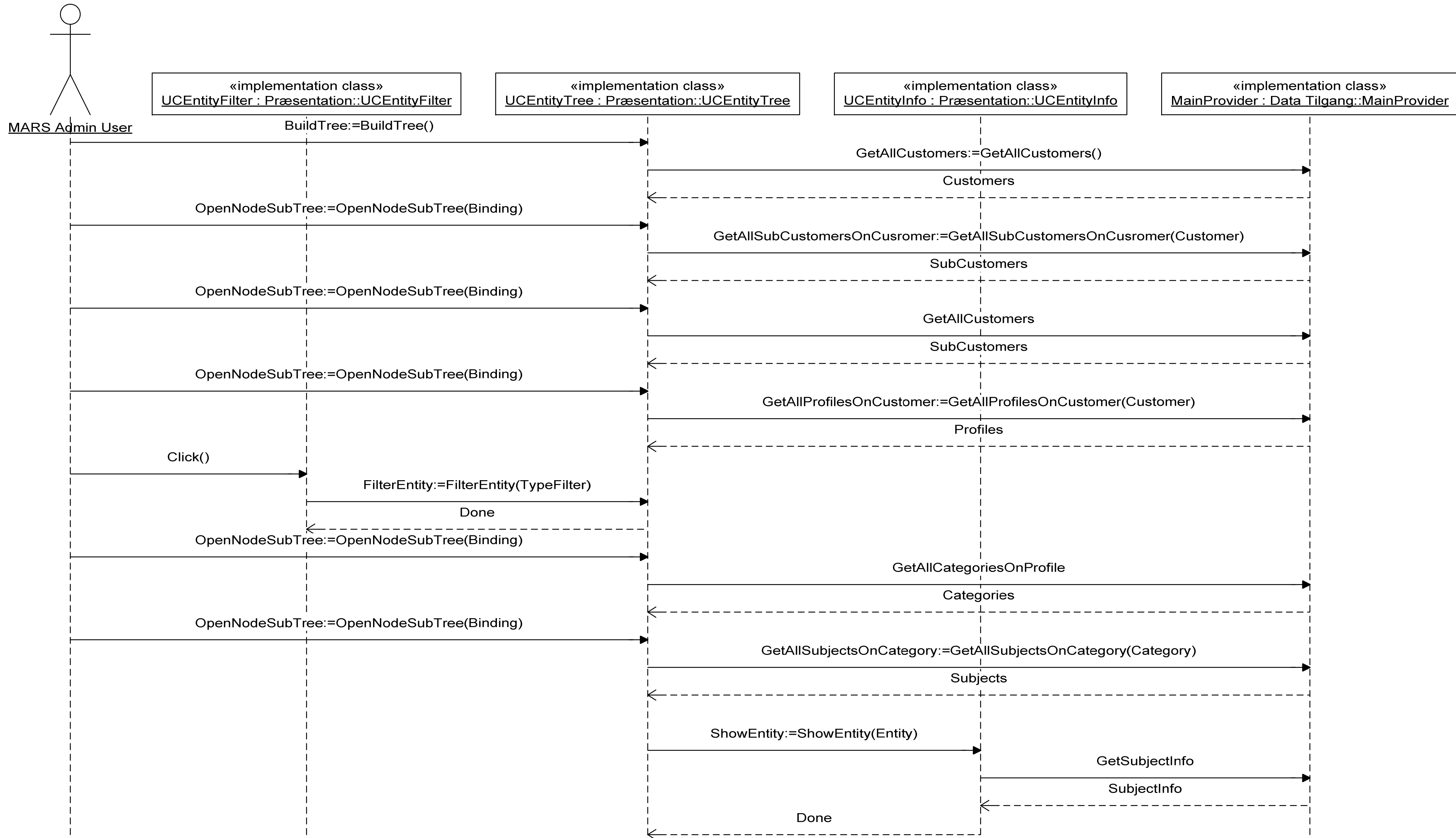
# 9.7 UML Sequence Diagrammer

## 9.7.1 Opret *elementer* abstrakt

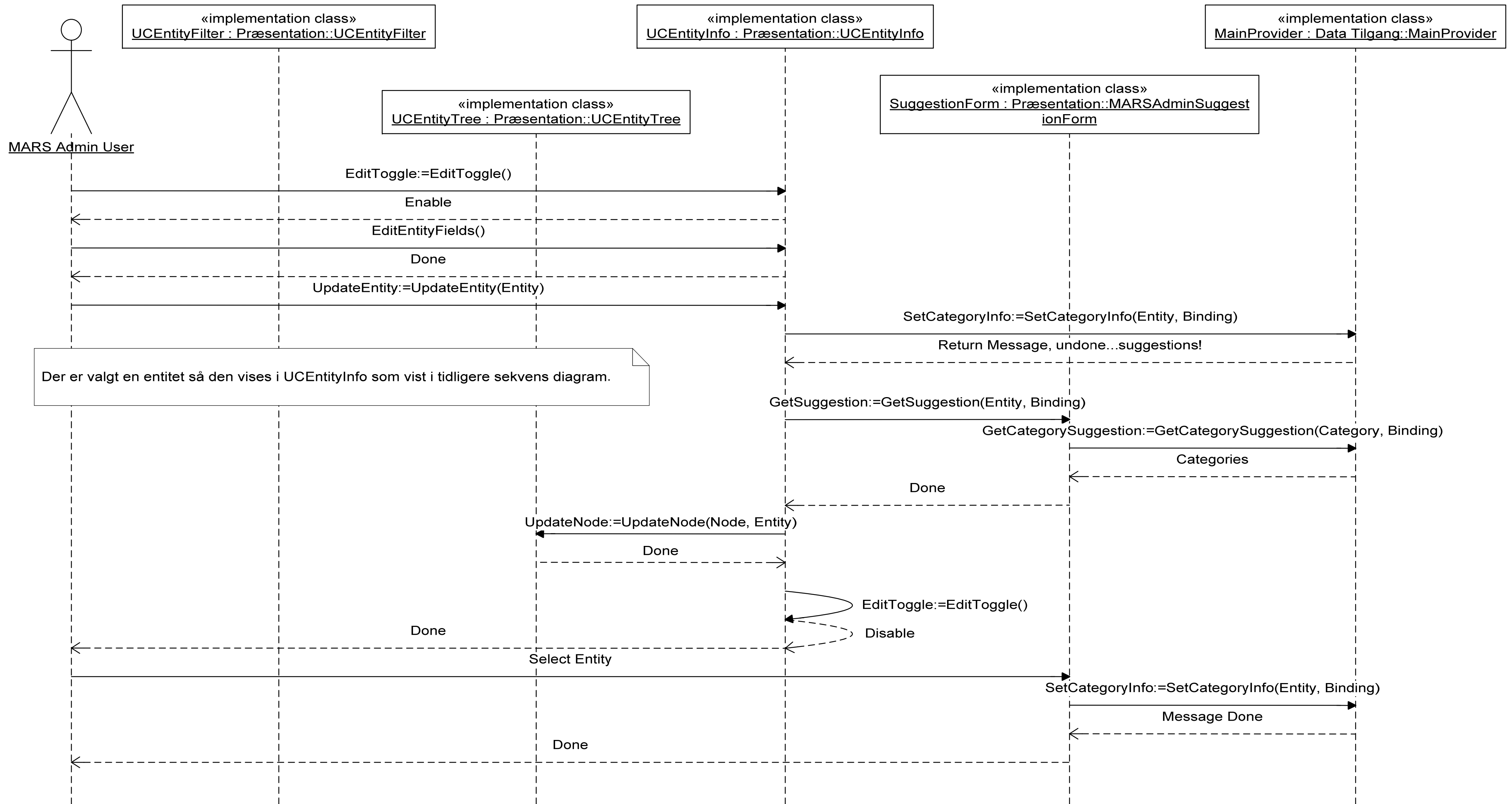



- Øverste sektion viser hvordan de rigtige *element*data vælges.
- Der bliver indtastet *element*data af *MARS Admin* bruger i *MARS Admin frontend*, hvorefter systemet forespørger *MARS Admin backend* om data er valide.
- *MARS Admin backend* spørger *MARS* databasen om *elementer* af lignende form, for ikke at oprette *elementer* der allerede findes i systemet.
- I *MARS Admin frontend* vil vise *elementer* og evt. data i forhold til deres indbyrdes bindinger, således det tydeliggøre *elementer* fra hinanden.
- Denne ovenstående sekvens er indkapslet som en færdig sekvens, men for at *elementet* bliver oprettet skal nedenstående sekvens også gennemløbes.
- Når *MARS Admin* bruger har valgt en foreslået eller det nye *element*, kan der oprettes et *element* i databasen alt efter *element*typen. Dette vil gøre forskellen om *elementet* bærer en binding eller om det kun er et ny virtuel *element* fra et allerede eksisterende *element*. I dette tilfælde vil det kun være bindingen der bliver oprettet.
- *MARS Admin frontend* vil oprette et *element* samt bindingen til det evt. overliggende *element*, hvis dette er nødvendigt.
- *MARS Admin backend* opretter *elementet* og binding via databasen, som bekræfter begge operationer, og ligeledes bekræfter *backend* til *frontend* således *MARS Admin* bruger ved at handlingen er sket korrekt.

## 9.7.2 Select Subject

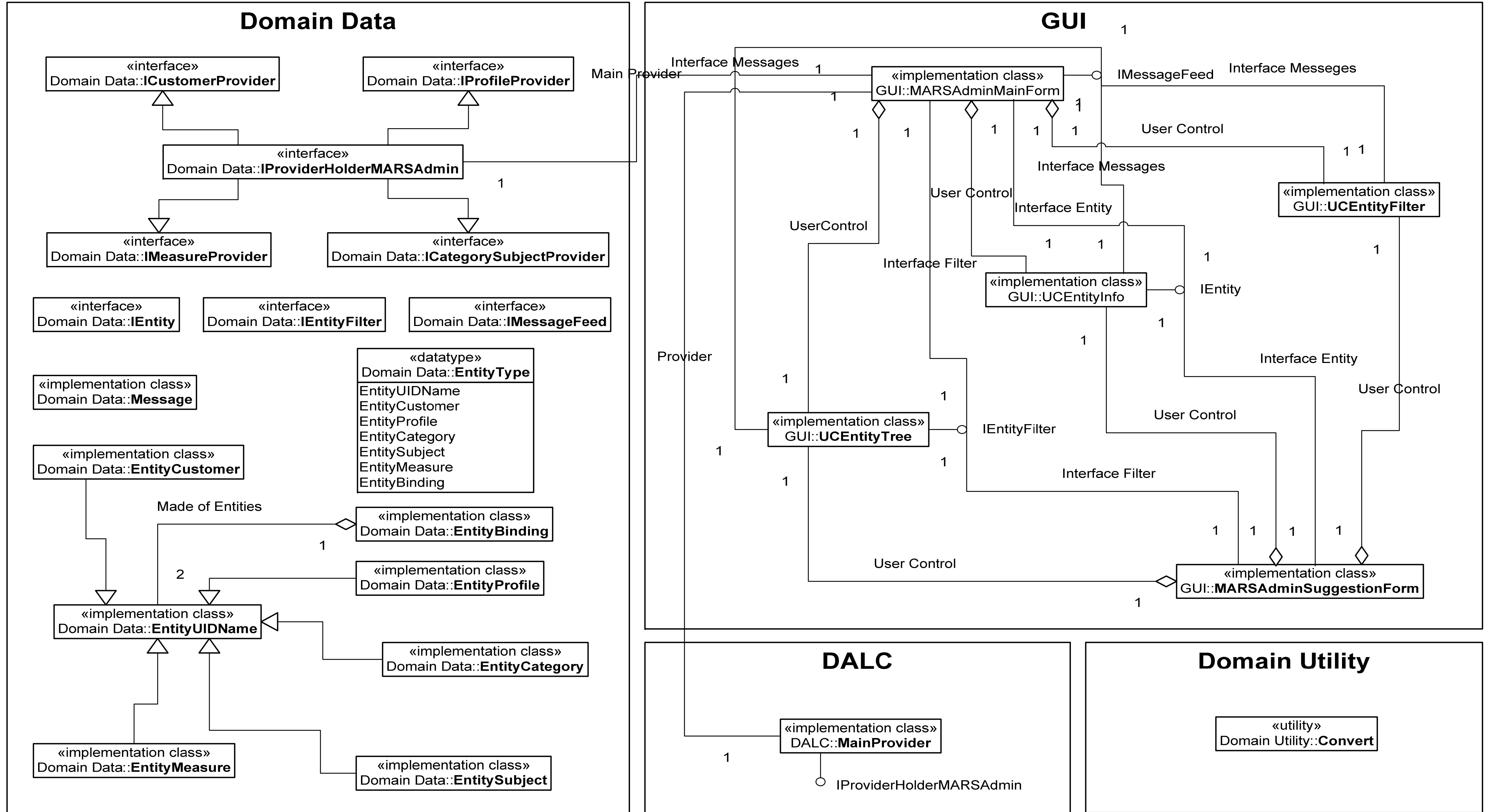


## 9.7.3 Edit Category

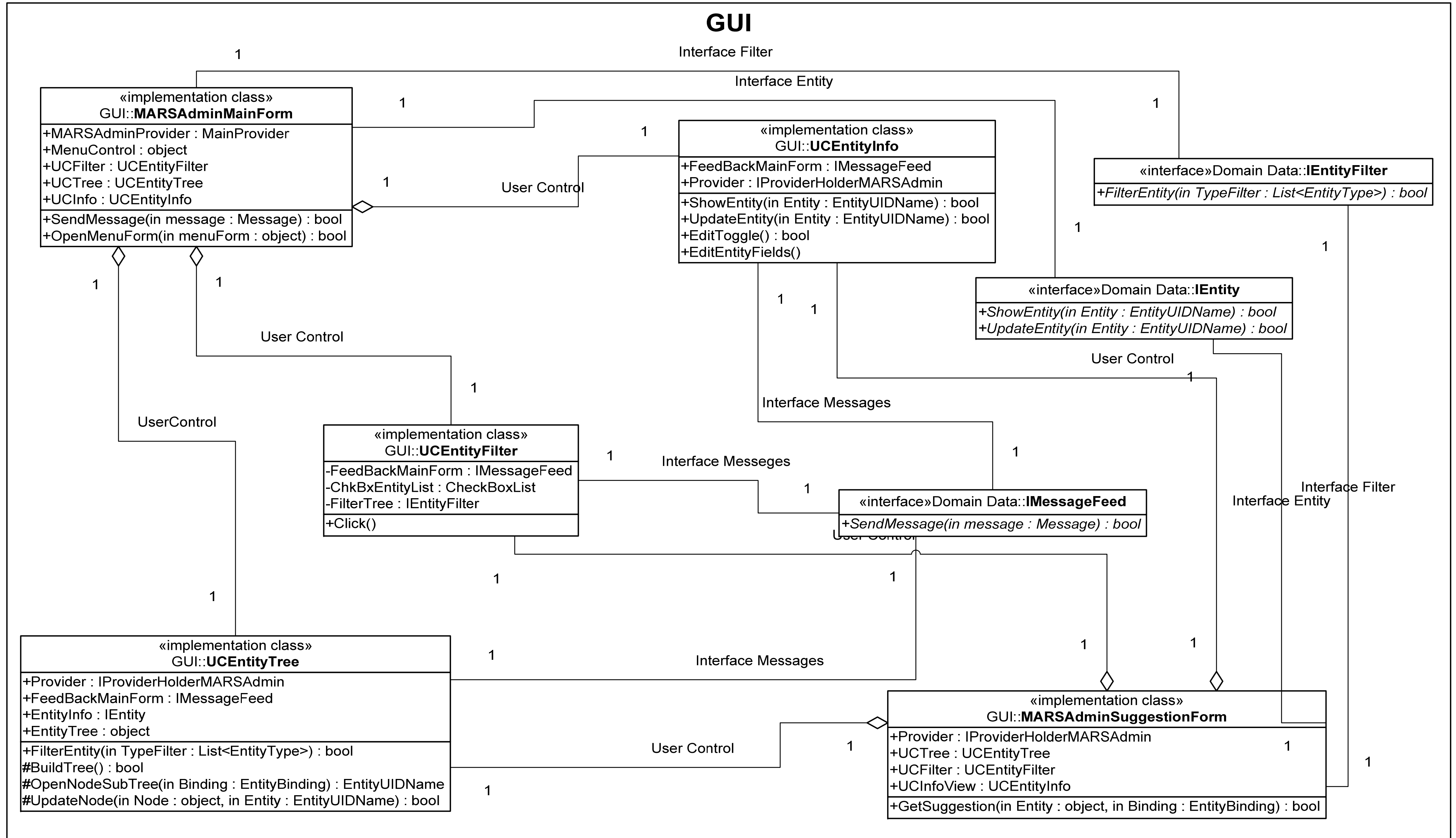


# 9.8 UML Class Diagram Concept

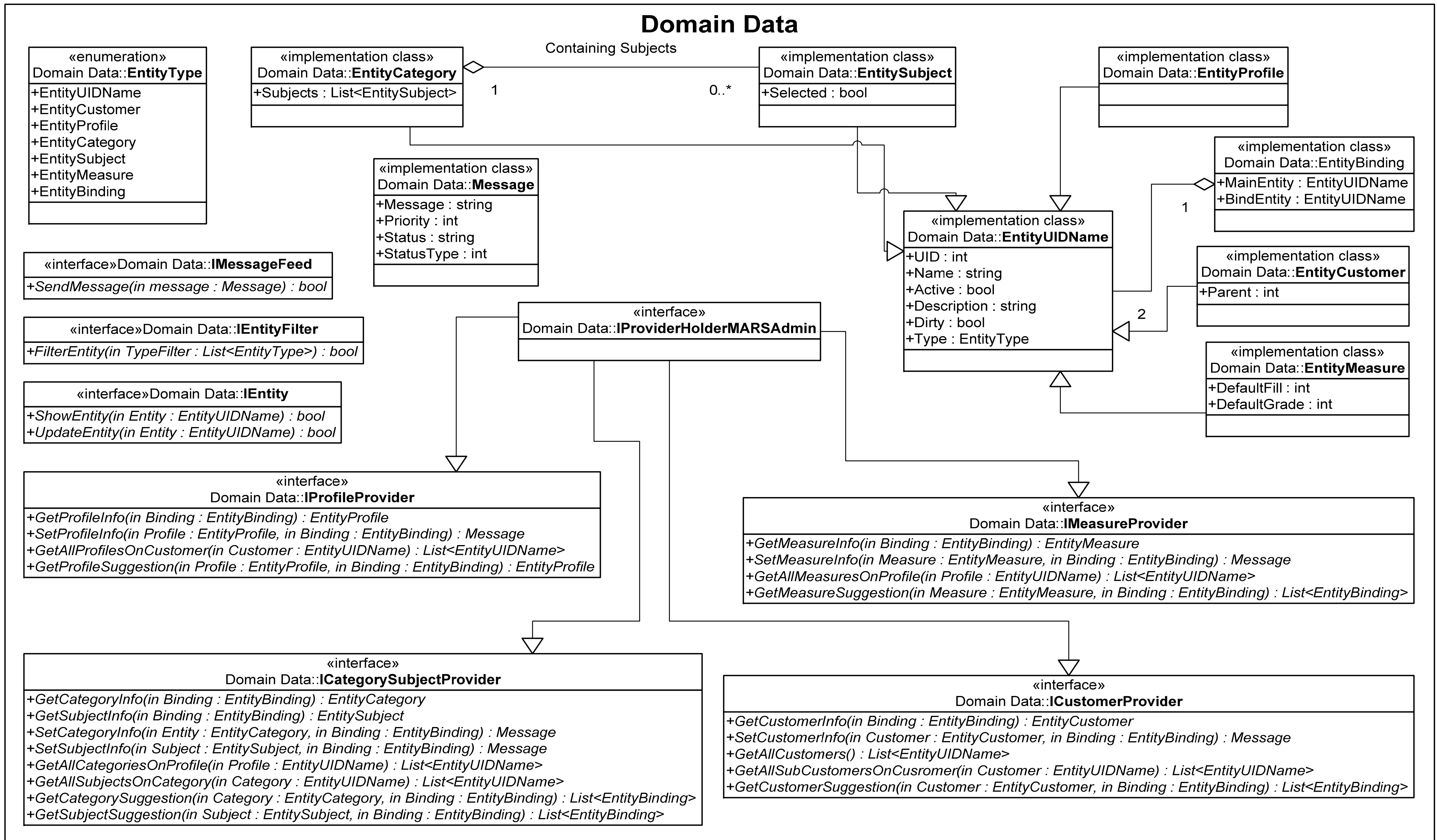
## 9.8.1 Frontend Overview Concept



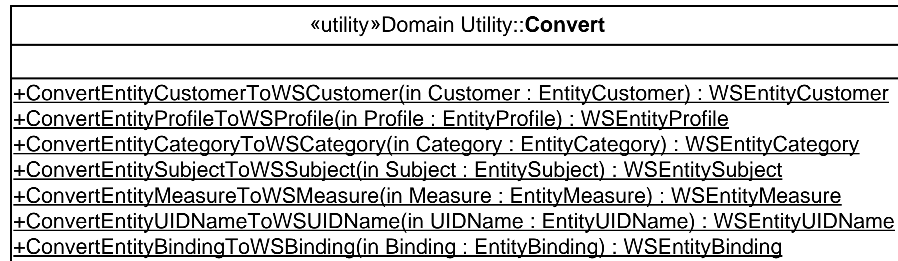
## 9.8.2 Frontend GUI Concept



## 9.8.3 Frontend Domain Data Concept

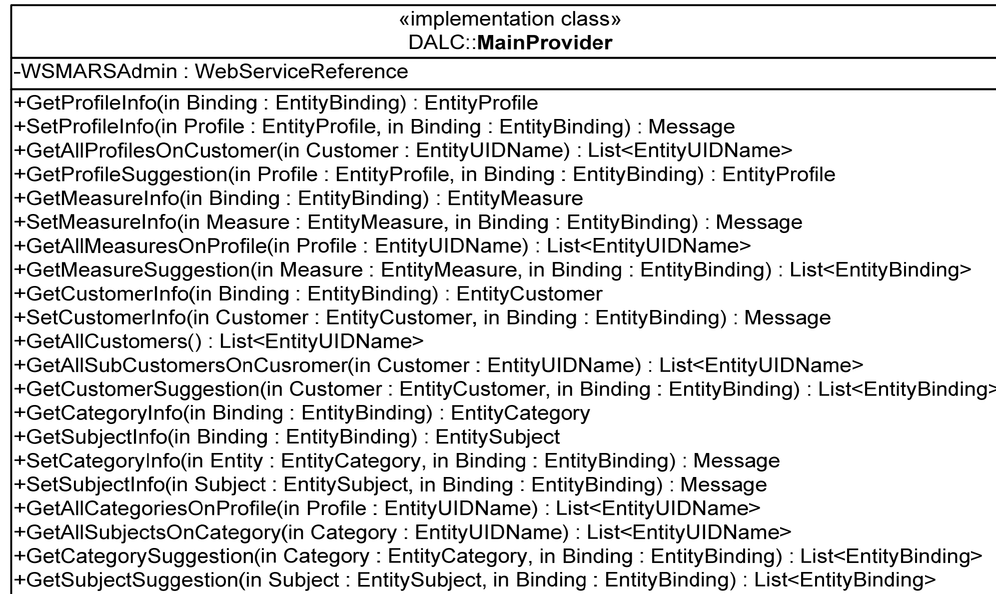


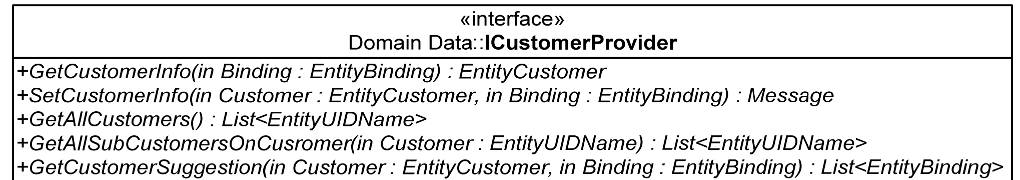
## 9.8.4 Frontend Domain Utility Concept

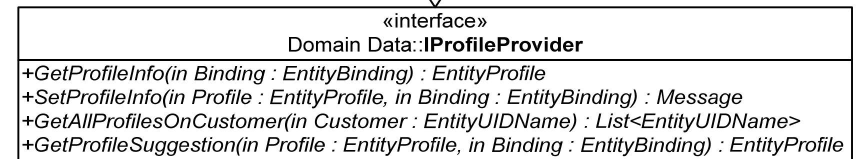


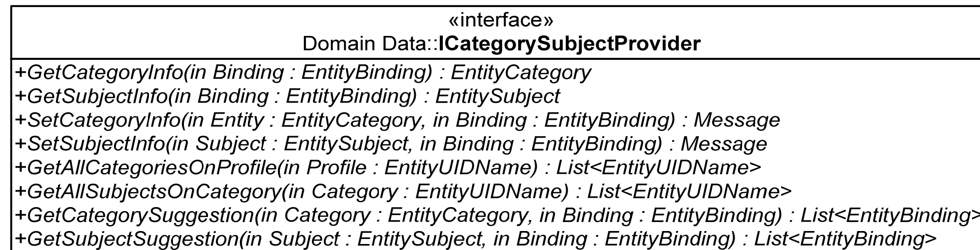
## 9.8.5 Frontend DALC Concept

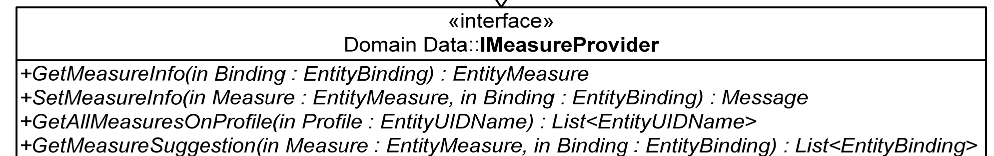
DALC



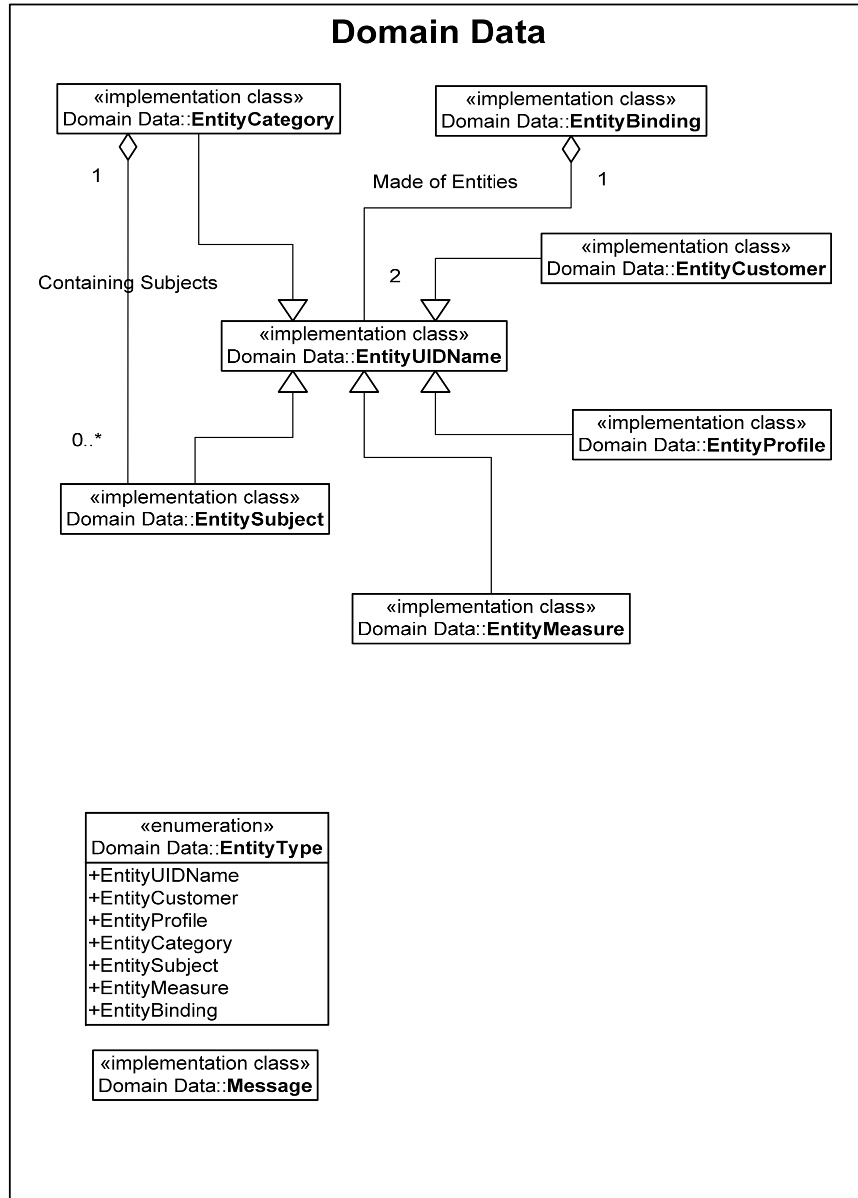





# 9.8.6.Backend Concept




Service

«implementation class»
Service::MARSAdminService

1

Business Logic

Domain Logic

«implementation class»
Domain Logic::ValidationLogic

1

1 Validation Logics

1

«implementation class»
Domain Logic::DomainLogic

1

1 Suggestion Logics 1

«implementation class»
Domain Logic::SuggestionLogic

Data Source

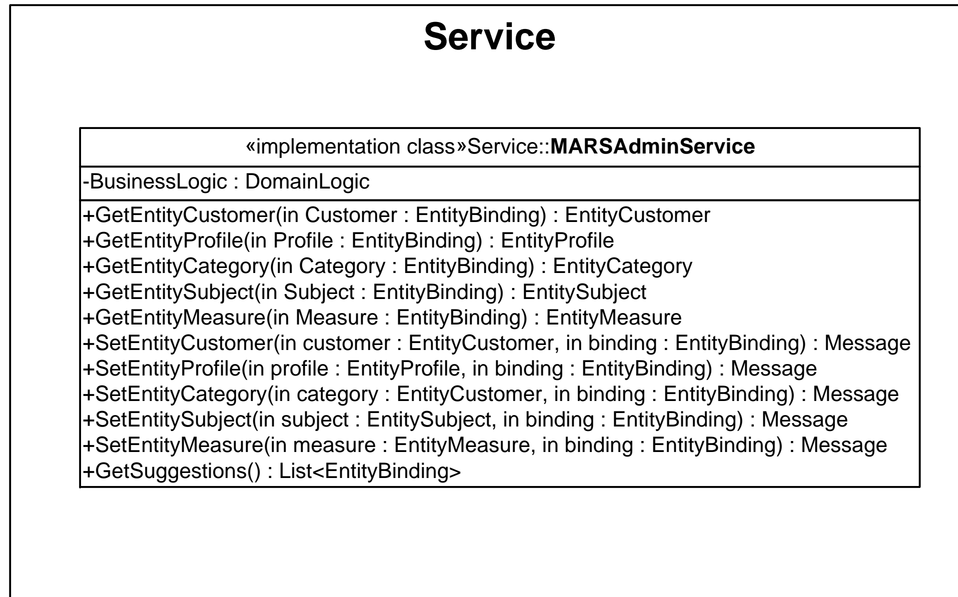
DALC

1

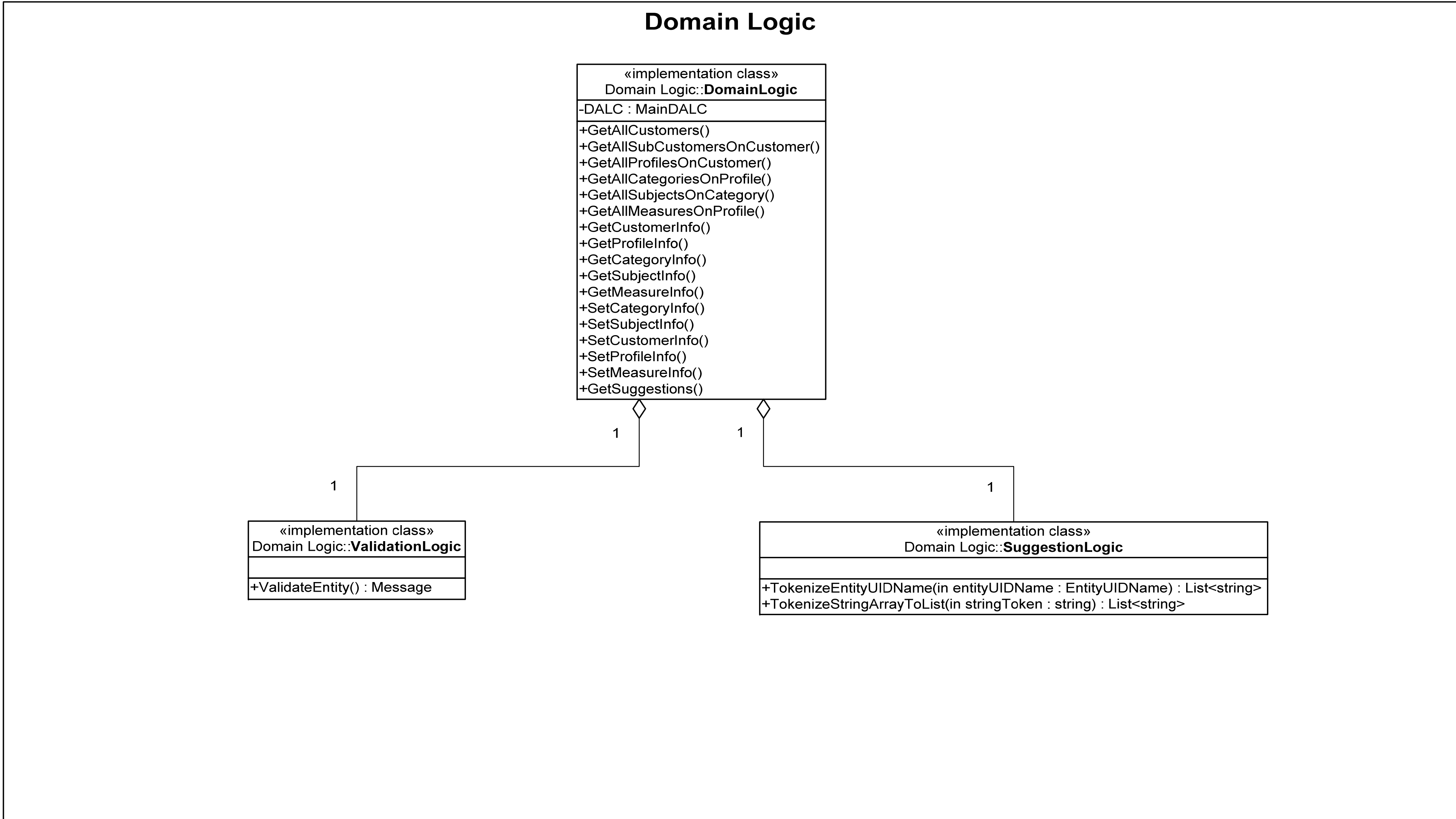
«implementation class»
DALC::MainDALC

«implementation class»
DALC::BaseDACL

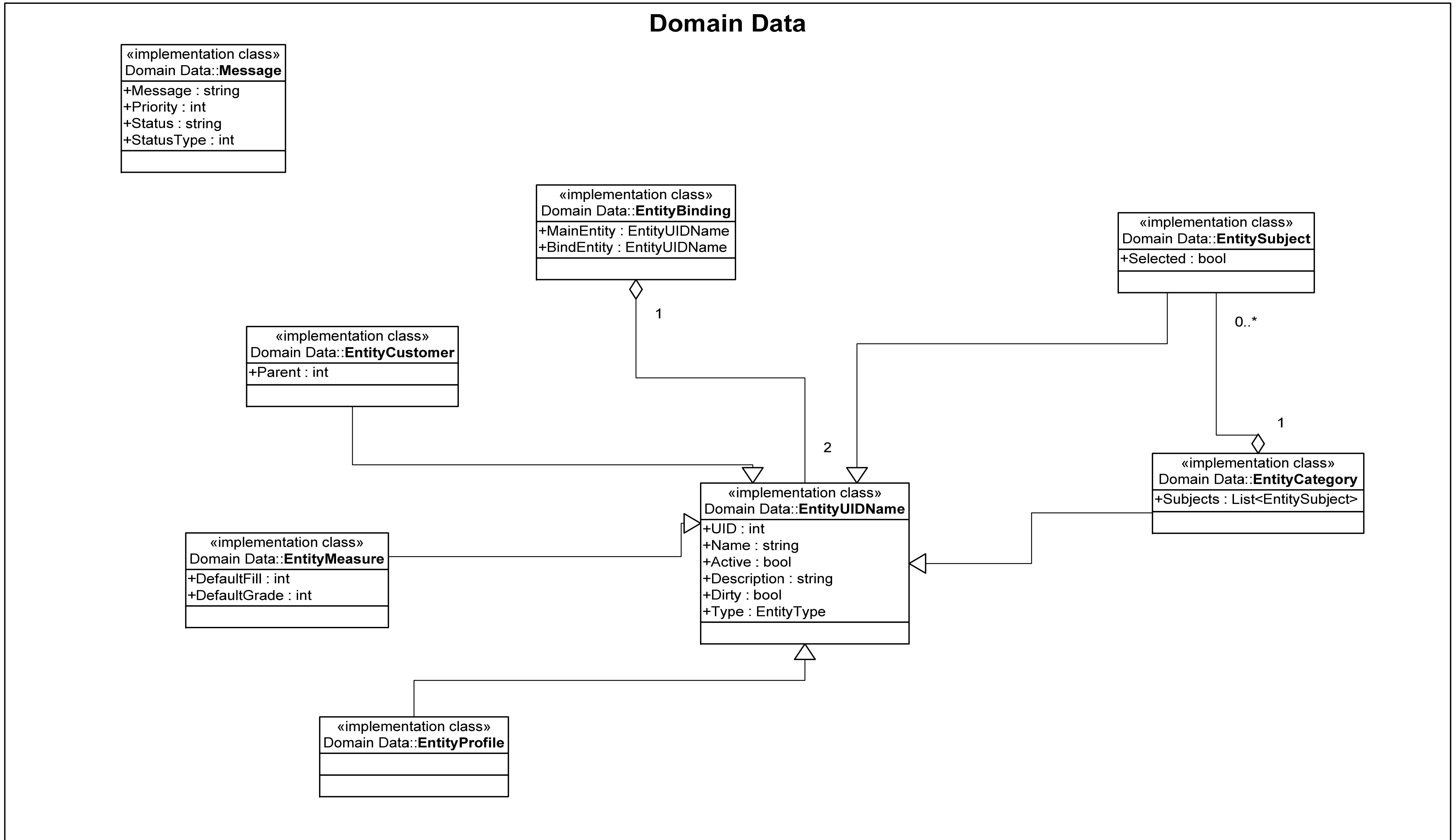
## 9.8.7 Backend Service Concept



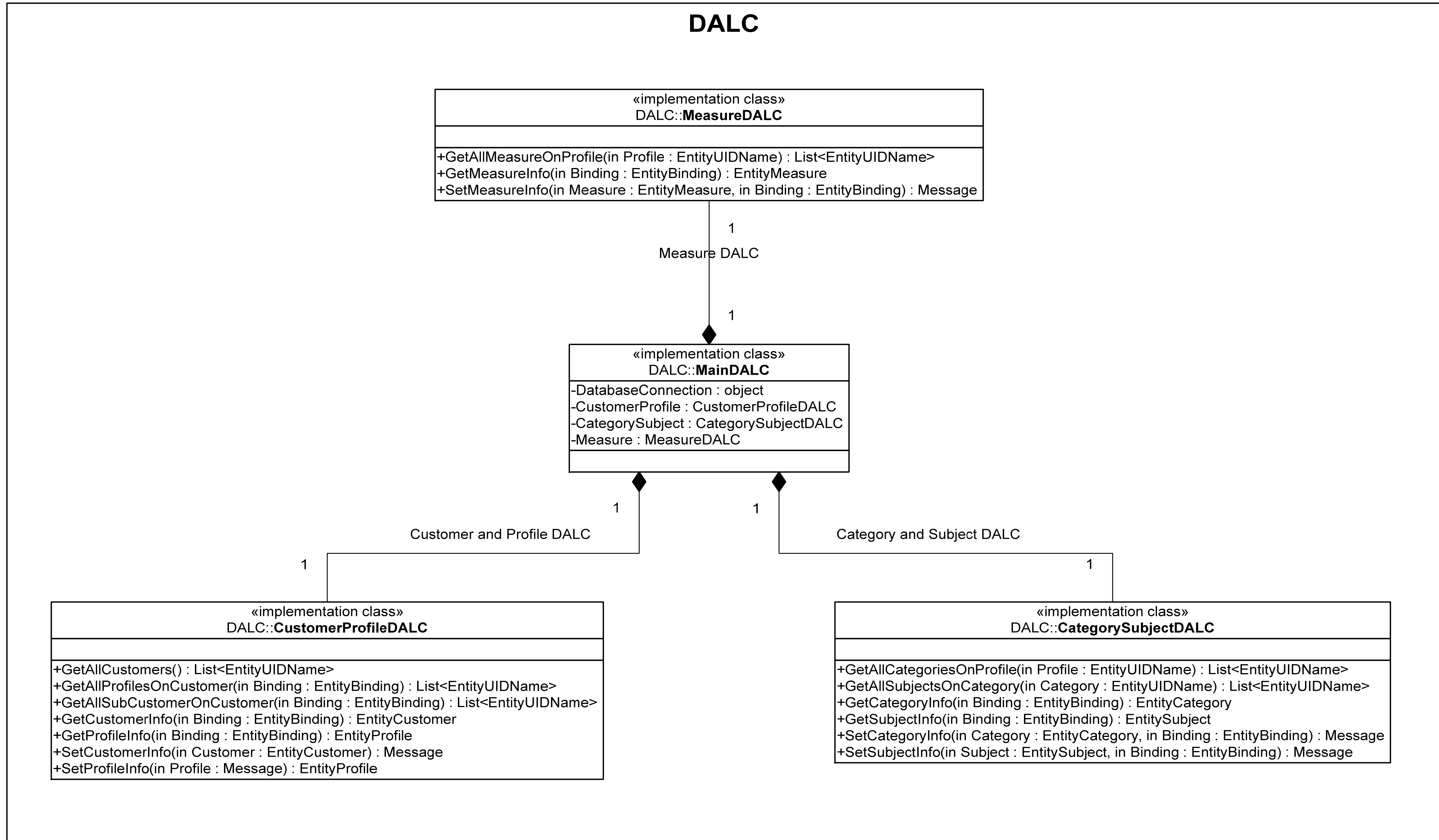
## 9.8.8 Backend Domain Logic Concept



## 9.8.9 Backend Domain Data Concept

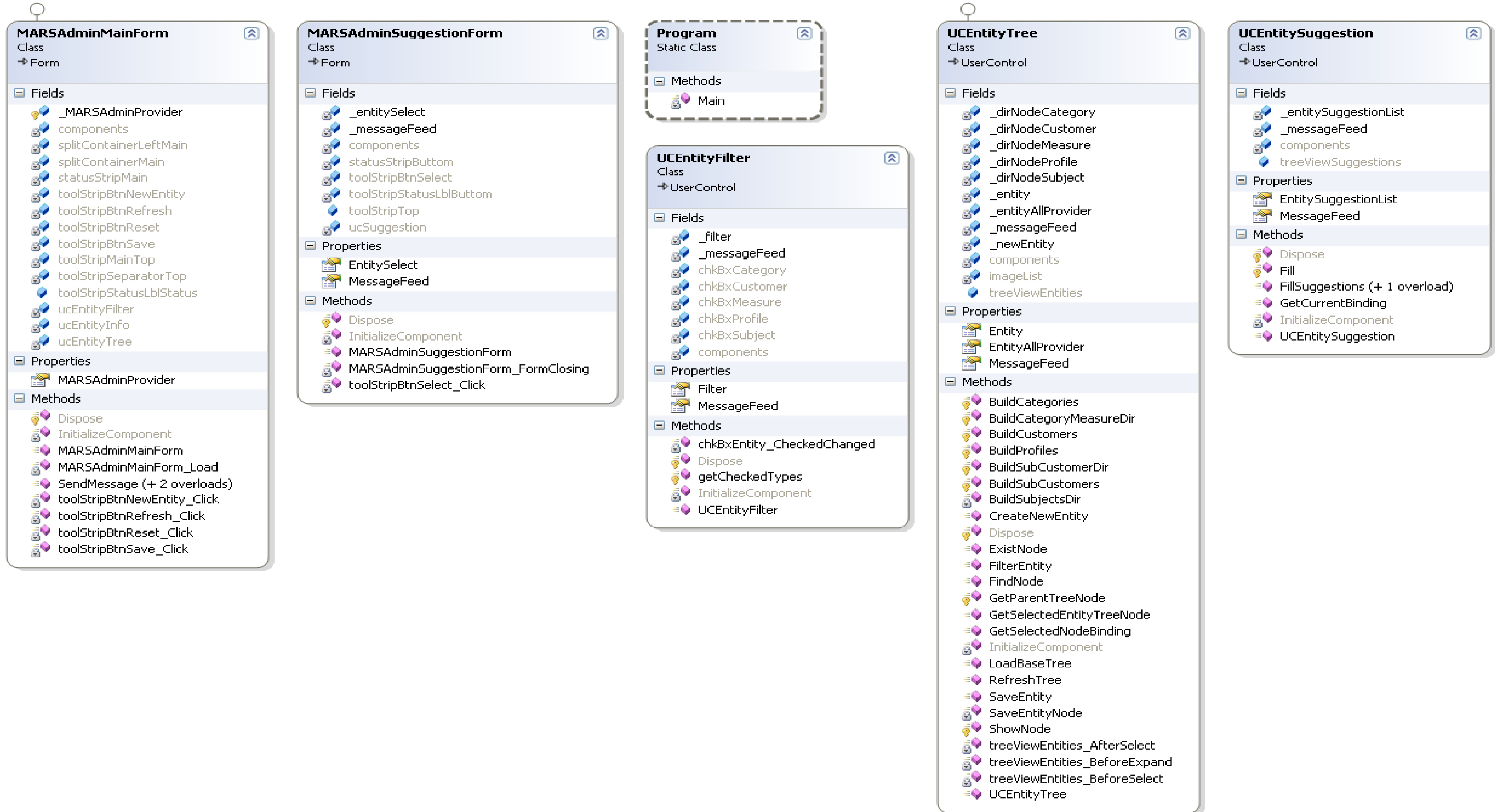


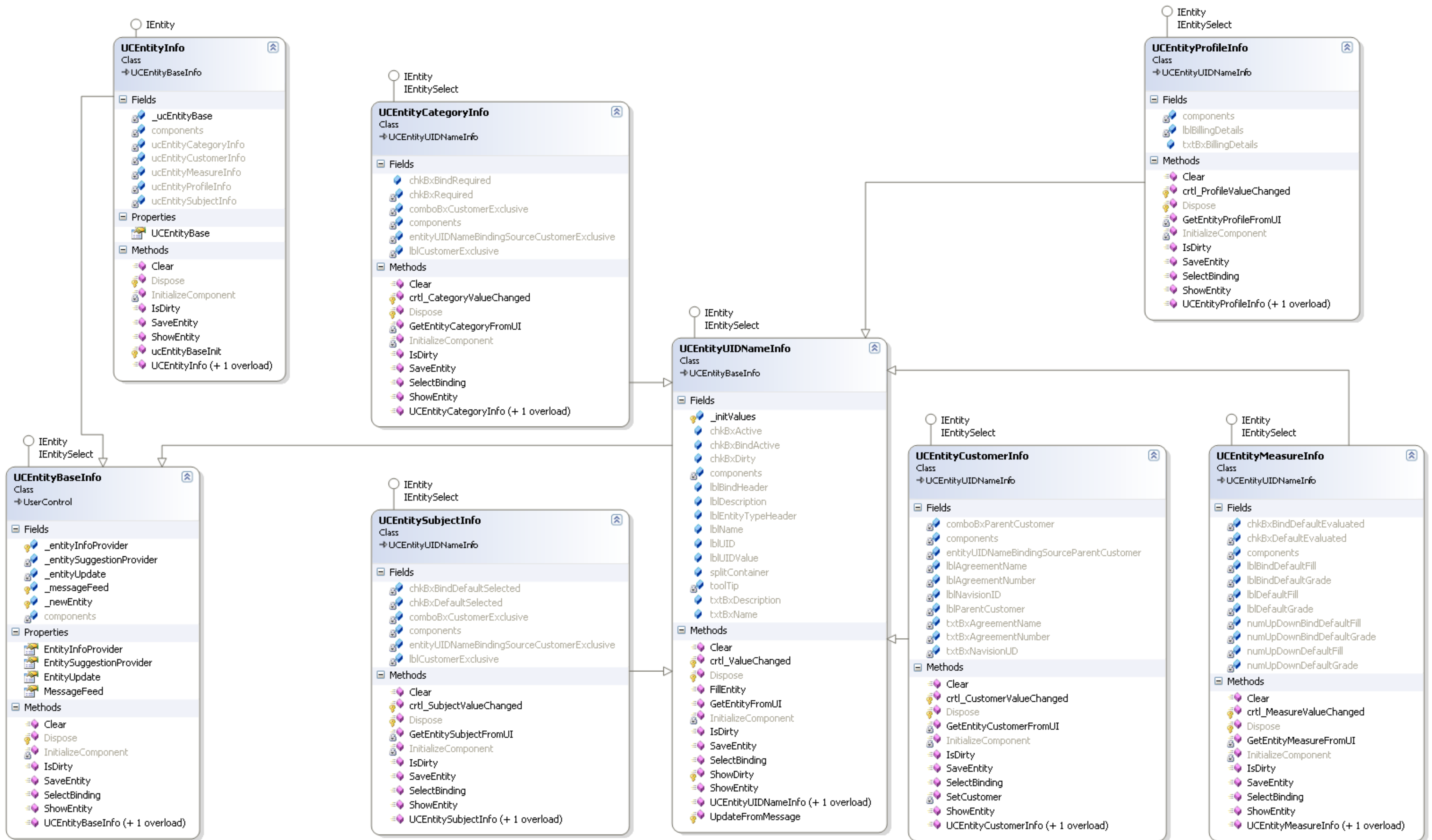
## 9.8.10 Backend DALC



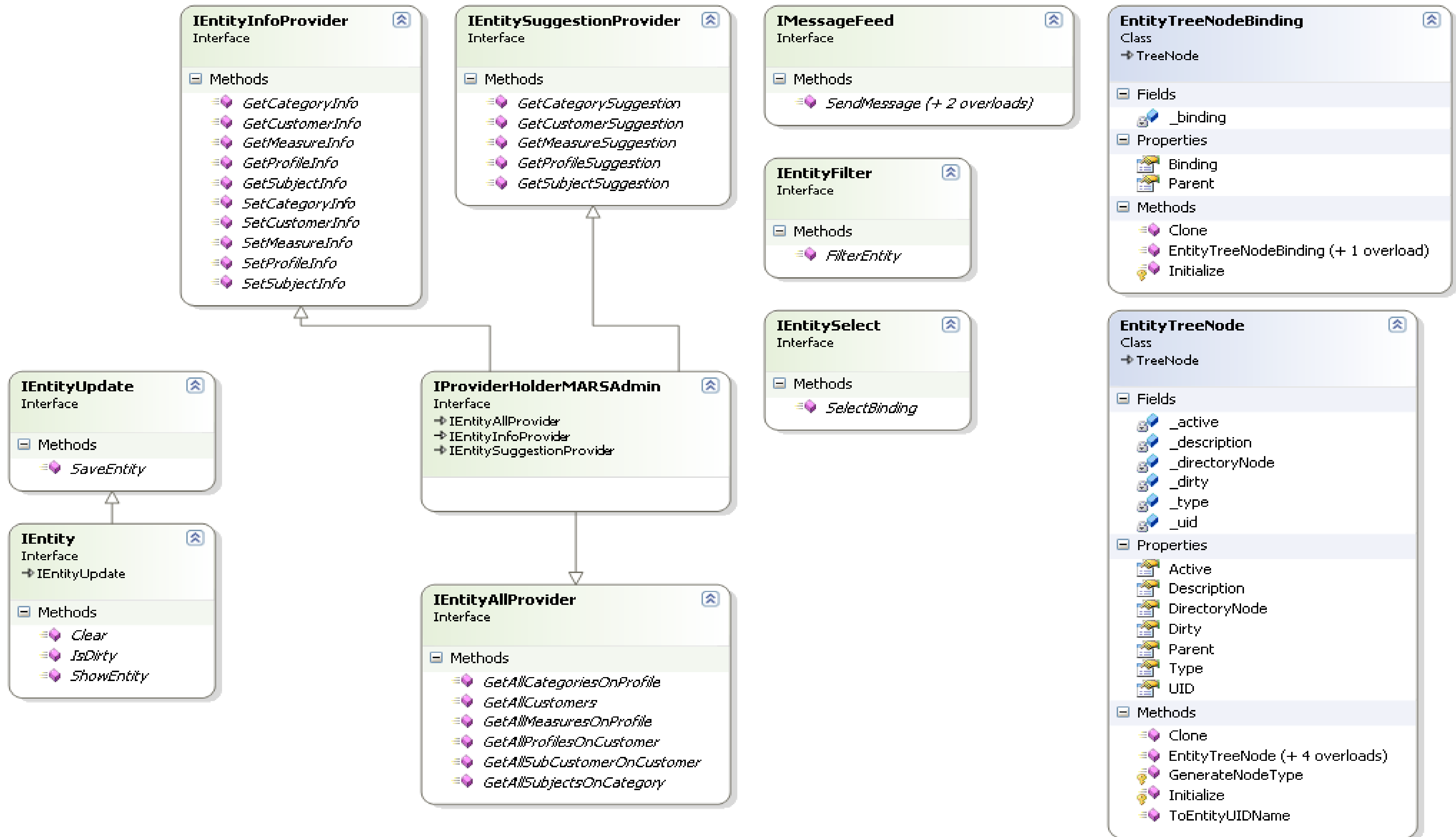
# 9.9 UML Class Diagrams

## 9.9.1 Frontend.GUI

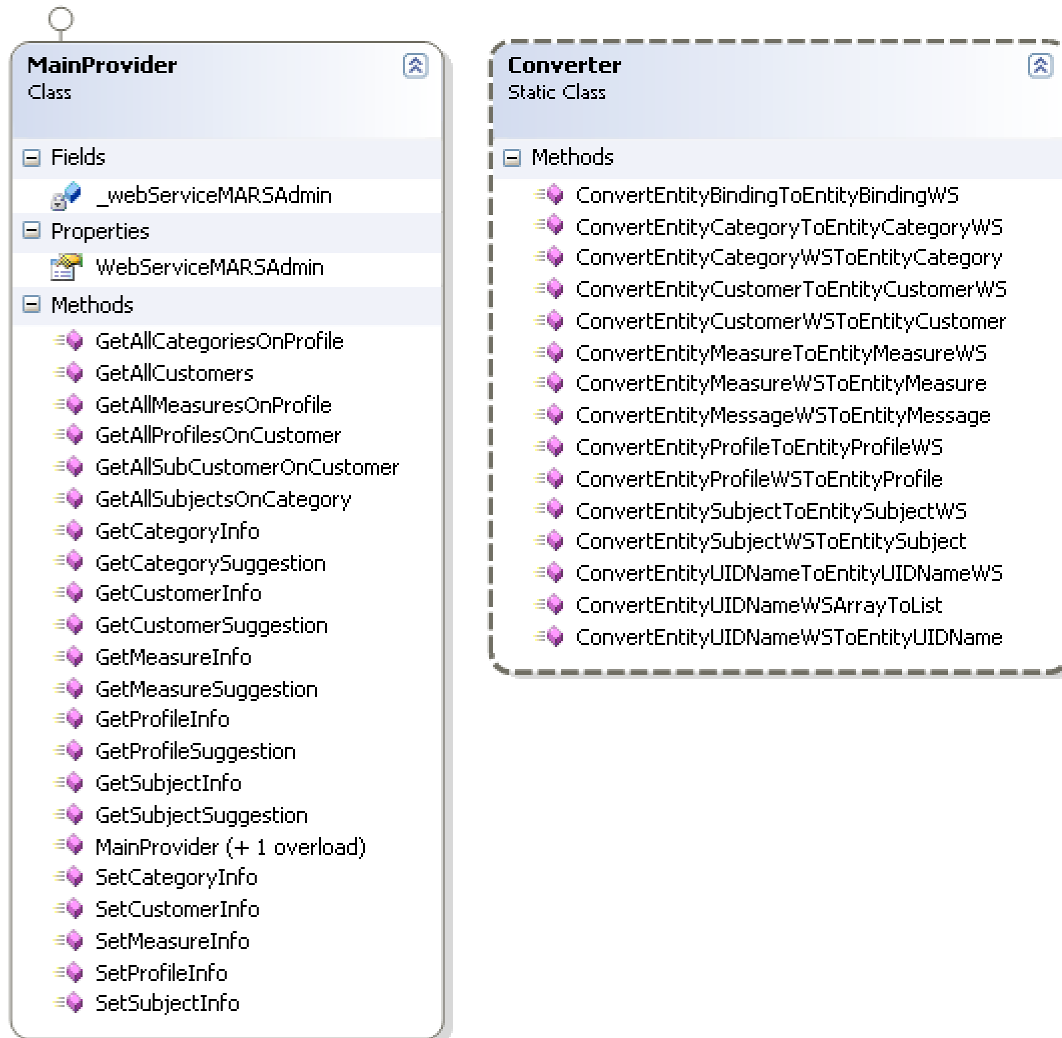



IEntity
UCEntityInfo
Class
UCEntityBaseInfo
Fields
_ucEntityBase
components
ucEntityCategoryInfo
ucEntityCustomerInfo
ucEntityMeasureInfo
ucEntityProfileInfo
ucEntitySubjectInfo
Properties
UCEntityBase
Methods
Clear
Dispose
InitializeComponent
IsDirty
SaveEntity
ShowEntity
ucEntityBaseInit
UCEntityInfo (+ 1 overload)
IEntity
IEntitySelect
UCEntityCategoryInfo
Class
UCEntityUIDNameInfo
Fields
chkBxBindRequired
chkBxRequired
comboBxCustomerExclusive
components
entityUIDNameBindingSourceCustomerExclusive
lblCustomerExclusive
Methods
Clear
crtl_CategoryValueChanged
Dispose
GetEntityCategoryFromUI
InitializeComponent
IsDirty
SaveEntity
SelectBinding
ShowEntity
UCEntityCategoryInfo (+ 1 overload)
IEntity
IEntitySelect
UCEntityProfileInfo
Class
UCEntityUIDNameInfo
Fields
components
lblBillingDetails
txtBxBillingDetails
Methods
Clear
crtl_ProfileValueChanged
Dispose
GetEntityProfileFromUI
InitializeComponent
IsDirty
SaveEntity
SelectBinding
ShowEntity
UCEntityProfileInfo (+ 1 overload)
IEntity
IEntitySelect
UCEntityUIDNameInfo
Class
UCEntityBaseInfo
Fields
_initValues
chkBxActive
chkBxBindActive
chkBxDirty
components
lblBindHeader
lblDescription
lblEntityTypeHeader
lblName
lblUID
lblUIDValue
splitContainer
toolTip
txtBxDescription
txtBxName
Methods
Clear
crtl_ValueChanged
Dispose
FillEntity
GetEntityFromUI
InitializeComponent
IsDirty
SaveEntity
SelectBinding
ShowDirty
ShowEntity
UCEntityUIDNameInfo (+ 1 overload)
UpdateFromMessage
IEntity
IEntitySelect
UCEntityBaseInfo
Class
UserControl
Fields
_entityInfoProvider
_entitySuggestionProvider
_entityUpdate
_messageFeed
_newEntity
components
Properties
EntityInfoProvider
EntitySuggestionProvider
EntityUpdate
MessageFeed
Methods
Clear
Dispose
InitializeComponent
IsDirty
SaveEntity
SelectBinding
ShowEntity
UCEntityBaseInfo (+ 1 overload)
IEntity
IEntitySelect
UCEntitySubjectInfo
Class
UCEntityUIDNameInfo
Fields
chkBxBindDefaultSelected
chkBxDefaultSelected
comboBxCustomerExclusive
components
entityUIDNameBindingSourceCustomerExclusive
lblCustomerExclusive
Methods
Clear
crtl_SubjectValueChanged
Dispose
GetEntitySubjectFromUI
InitializeComponent
IsDirty
SaveEntity
SelectBinding
ShowEntity
UCEntitySubjectInfo (+ 1 overload)
IEntity
IEntitySelect
UCEntityCustomerInfo
Class
UCEntityUIDNameInfo
Fields
comboBxParentCustomer
components
entityUIDNameBindingSourceParentCustomer
lblAgreementName
lblAgreementNumber
lblNavisionID
lblParentCustomer
txtBxAgreementName
txtBxAgreementNumber
txtBxNavisionUID
Methods
Clear
crtl_CustomerValueChanged
Dispose
GetEntityCustomerFromUI
InitializeComponent
IsDirty
SaveEntity
SelectBinding
SetCustomer
ShowEntity
UCEntityCustomerInfo (+ 1 overload)
IEntity
IEntitySelect
UCEntityMeasureInfo
Class
UCEntityUIDNameInfo
Fields
chkBxBindDefaultEvaluated
chkBxDefaultEvaluated
components
lblBindDefaultFill
lblBindDefaultGrade
lblDefaultFill
lblDefaultGrade
numUpDownBindDefaultFill
numUpDownBindDefaultGrade
numUpDownDefaultFill
numUpDownDefaultGrade
Methods
Clear
crtl_MeasureValueChanged
Dispose
GetEntityMeasureFromUI
InitializeComponent
IsDirty
SaveEntity
SelectBinding
ShowEntity
UCEntityMeasureInfo (+ 1 overload)

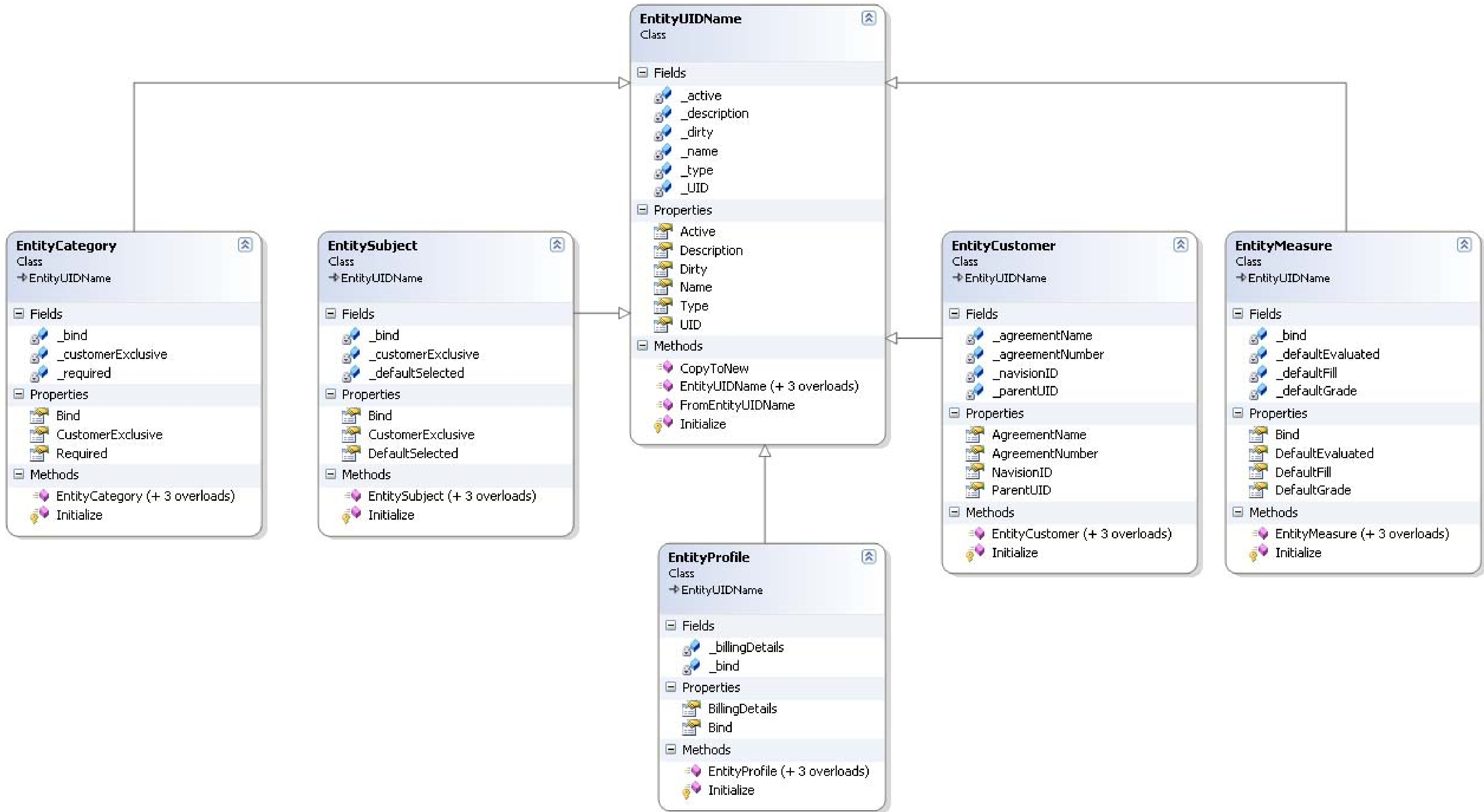
## 9.9.2 Frontend.Data

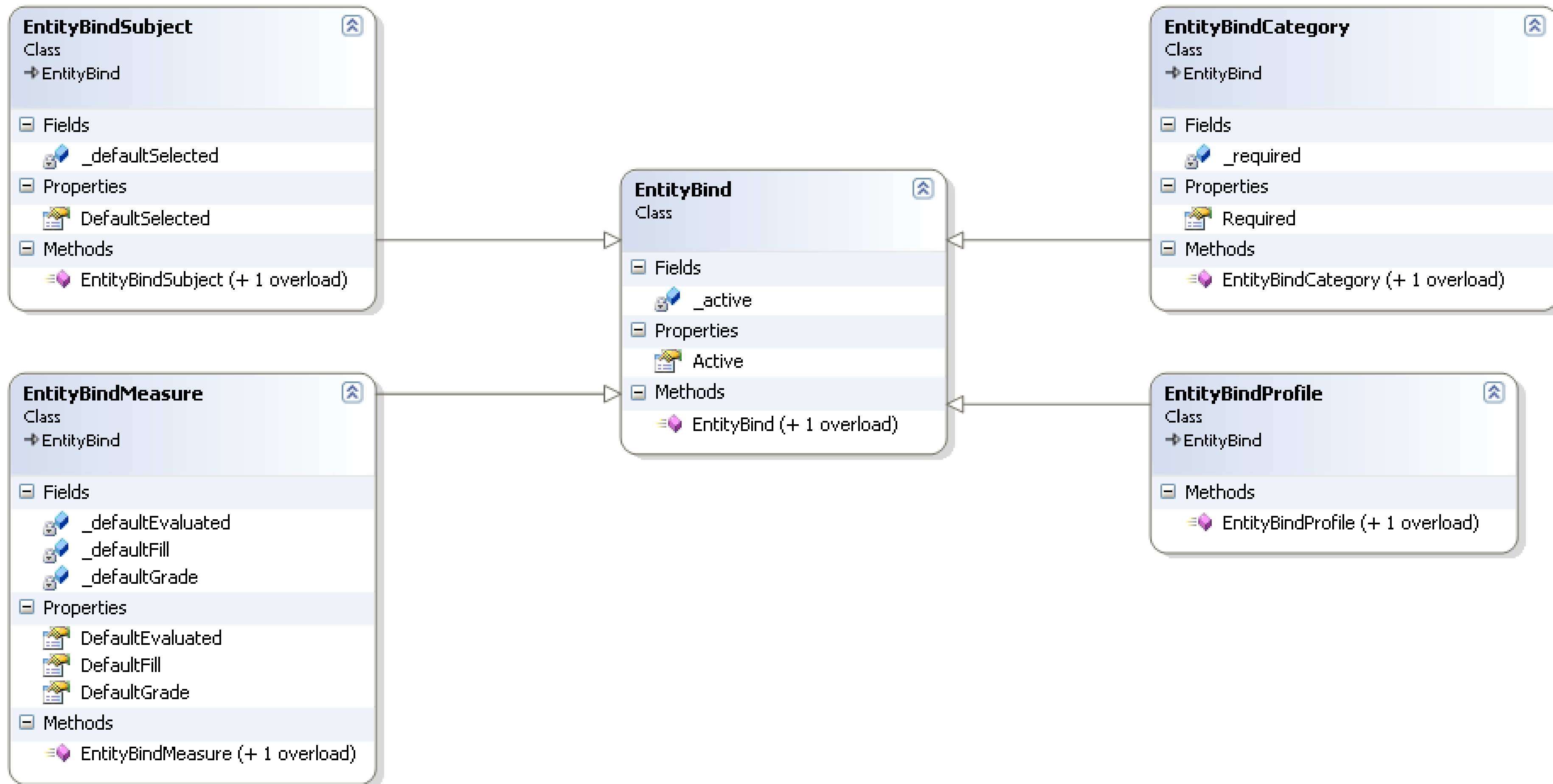


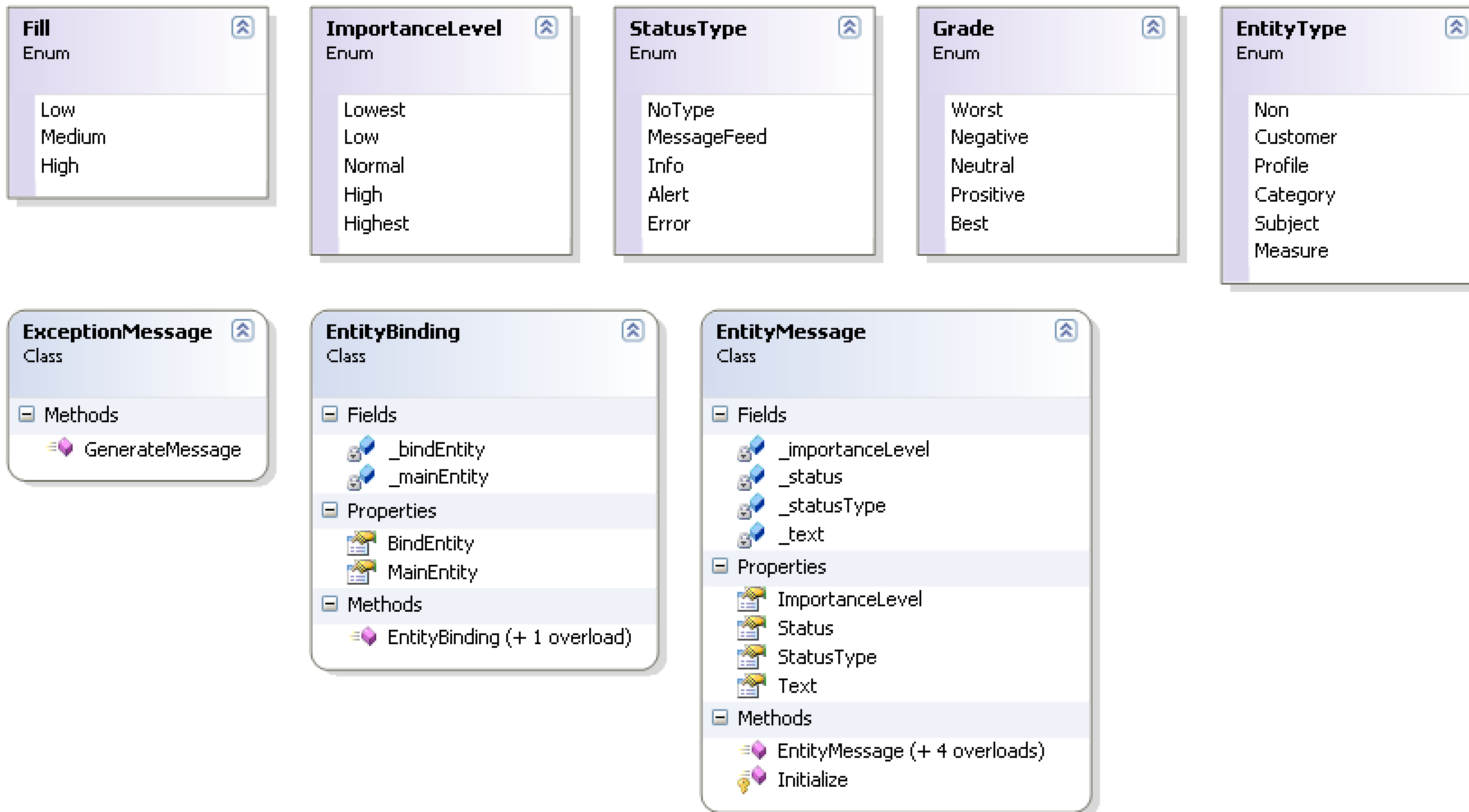
### 9.9.3 Frontend.DALC



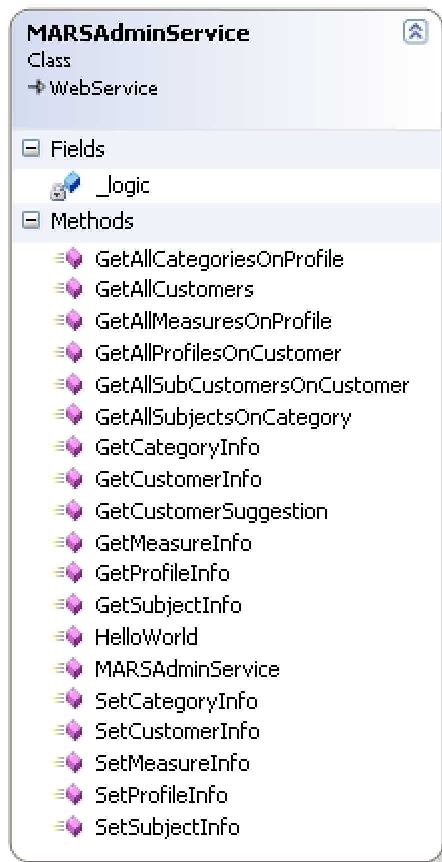
## 9.9.4 Domain.Data



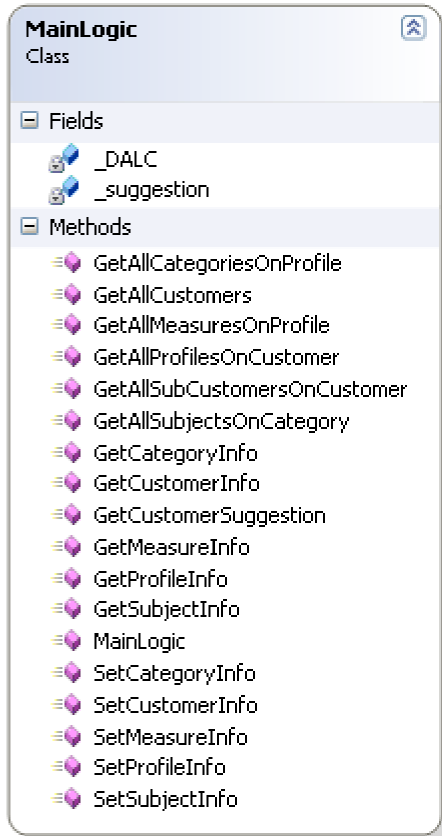

EntityBindSubject
Class
EntityBind
Fields
_defaultSelected
Properties
DefaultSelected
Methods
EntityBindSubject (+ 1 overload)
EntityBindCategory
Class
EntityBind
Fields
_required
Properties
Required
Methods
EntityBindCategory (+ 1 overload)
EntityBind
Class
Fields
_active
Properties
Active
Methods
EntityBind (+ 1 overload)
EntityBindMeasure
Class
EntityBind
Fields
_defaultEvaluated
_defaultFill
_defaultGrade
Properties
DefaultEvaluated
DefaultFill
DefaultGrade
Methods
EntityBindMeasure (+ 1 overload)
EntityBindProfile
Class
EntityBind
Methods
EntityBindProfile (+ 1 overload)

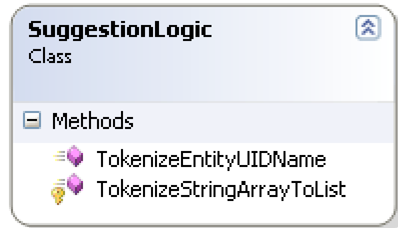

Fill
Enum
Low
Medium
High
ImportanceLevel
Enum
Lowest
Low
Normal
High
Highest
StatusType
Enum
NoType
MessageFeed
Info
Alert
Error
Grade
Enum
Worst
Negative
Neutral
Prositive
Best
EntityType
Enum
Non
Customer
Profile
Category
Subject
Measure
ExceptionMessage
Class
Methods
GenerateMessage
EntityBinding
Class
Fields
_bindEntity
_mainEntity
Properties
BindEntity
MainEntity
Methods
EntityBinding (+ 1 overload)
EntityMessage
Class
Fields
_importanceLevel
_status
_statusType
_text
Properties
ImportanceLevel
Status
StatusType
Text
Methods
EntityMessage (+ 4 overloads)
Initialize

## 9.9.5 Backend.Service

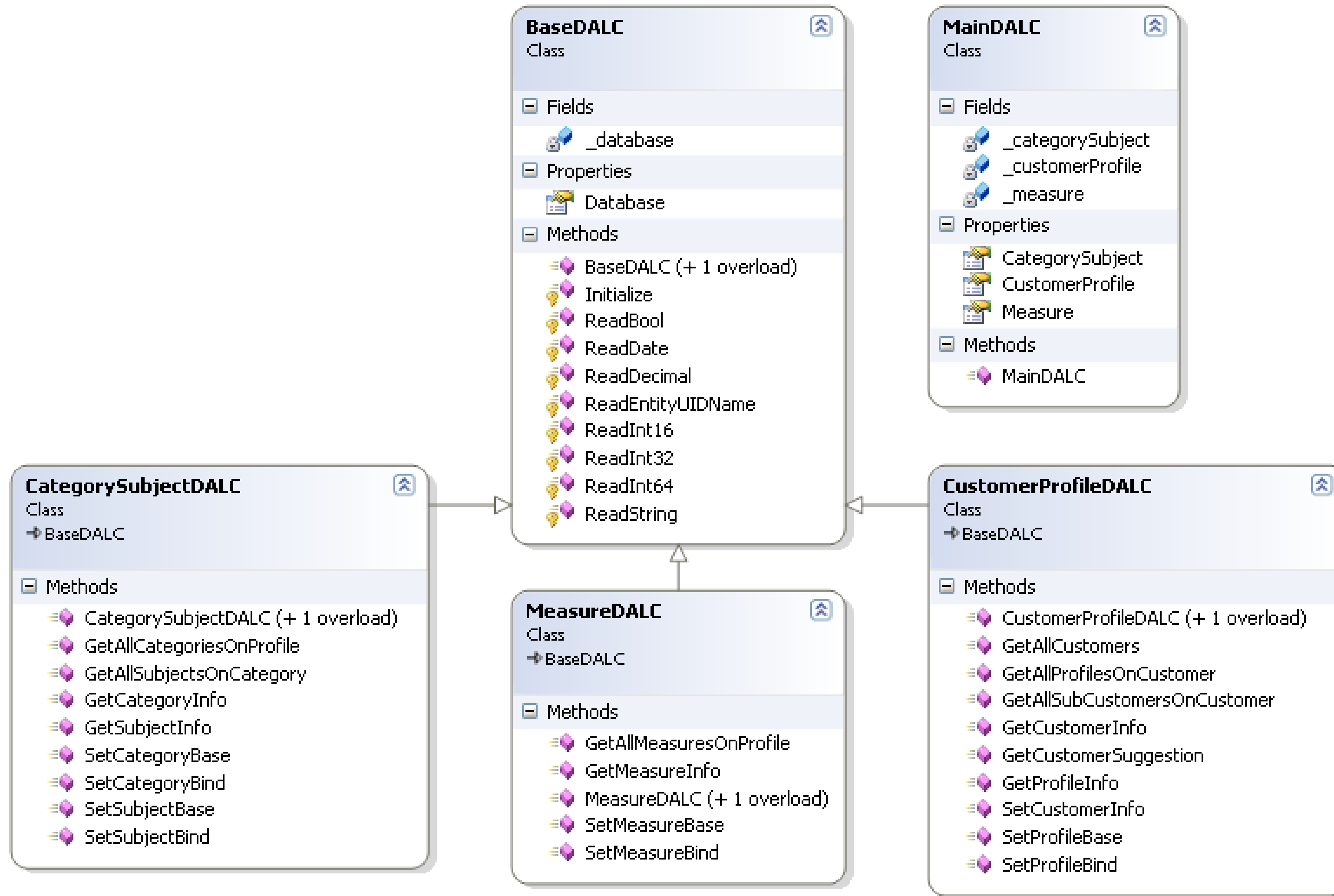


## 9.9.6 Backend.Logic



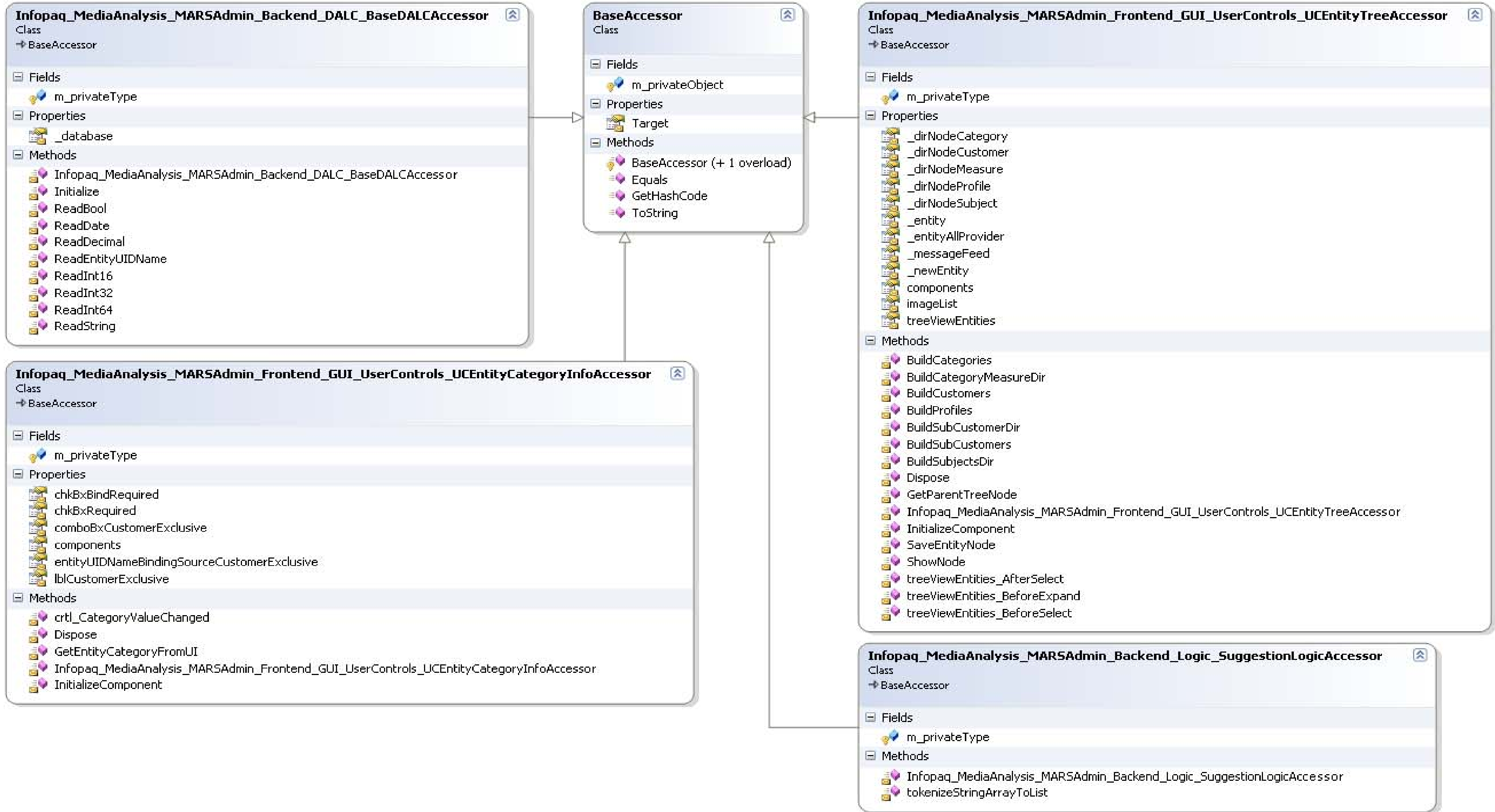



## 9.9.7 Backend.DALC



# 9.10 Test Class Diagrams

## 9.10.1 Accessors (UnitTestProject)



## 9.10.2 Test Classes (UnitTestProject)