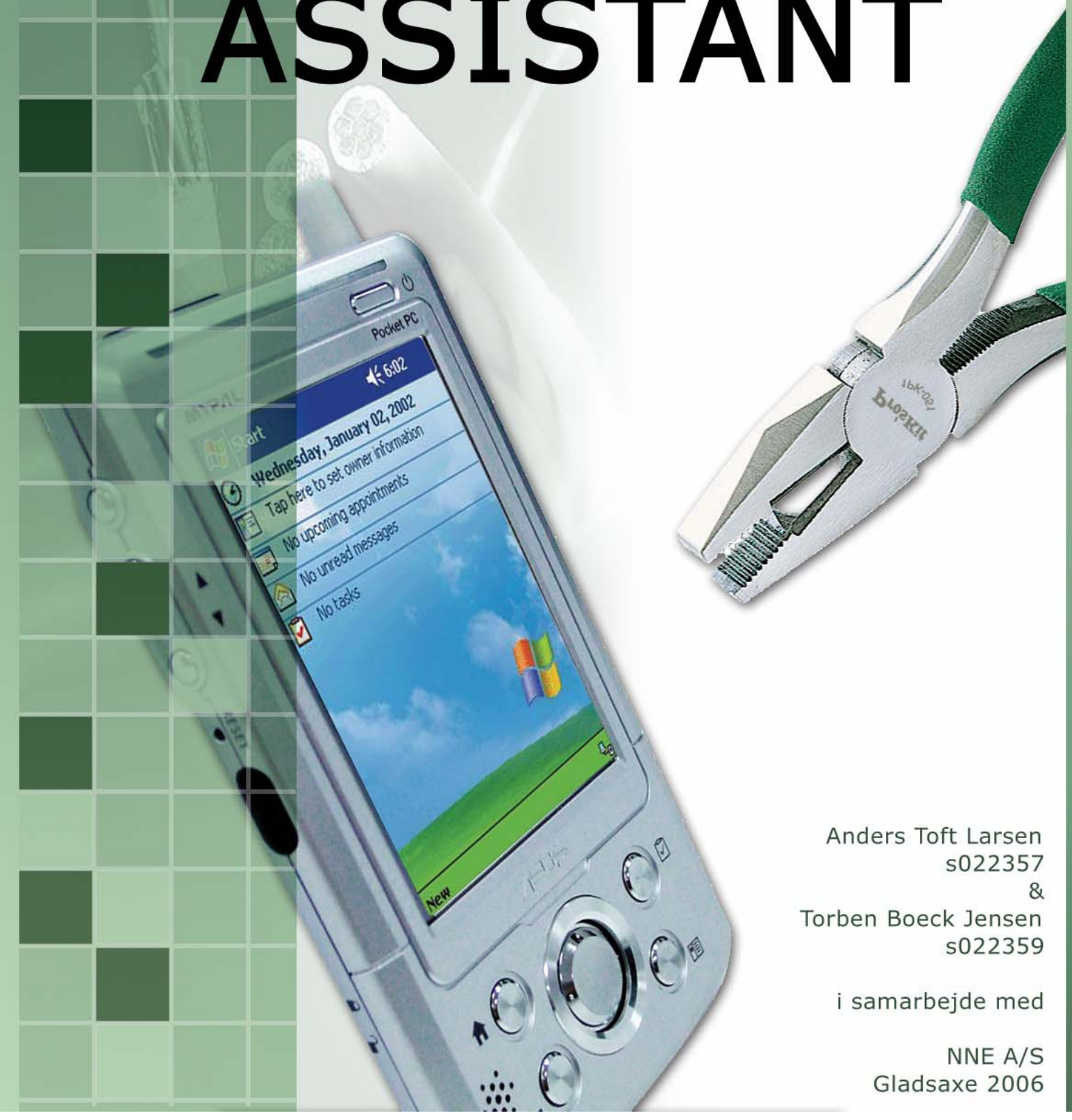
Eksamensprojekt udarbejdet ved IMM, DTU

PORTABLE OPERATOR ASSISTANT



Anders Toft Larsen
s022357
&
Torben Boeck Jensen
s022359

i samarbejde med

NNE A/S
Gladsaxe 2006

Abstract

Portable Operator Assistant

By Anders Toft Larsen (s022357) and Torben Boeck Jensen (2022359) in cooperation with NNE A/S

Background. The purpose of this project is to design a pilot-application, which will allow NNE A/S to determine whether or not it is desirable to integrate the use of handheld computers (PDA's) into their existing LMES+ product. The project will consist of an application running on a PDA, which can be used for performing simple equipment related queries. Besides the functionality another very important aspect is the possibility of changing and updating the graphical use interface (GUI) presented by the application without changing the application code.

Methods. To ensure that the development of the application is dealt with in a professional way it has been decided to use the *Object Oriented Analysis and Design*. More specifically the *Rational Unified Process* will be used along with *Unified Modelling Language* notation. To allow the GUI to be updated without changes to the application code, it must be designed as a dynamic data driven GUI. As a consequence the GUI just presents elements which have been specified elsewhere. In order to separate the specification of the GUI content completely from the application code, this information is to be stored in the existing LMES+ system and sent from there to the GUI.

Results. The final solution is a three-tier model consisting of a *Presentation Layer*, a *Business Logic Layer* and a *Data Storage Layer*. The *Presentation Layer* is run on the PDA and communicates with the *Business Logic Layer* through a WiFi connection. The *Business Logic Layer* consists of a *web service*, which communicates with the *Presentation Layer* and a *Data Access* part, which communicates with the *Data Storage Layer*. The *Data Storage Layer* consists of the existing LMES+ which uses an *Oracle* database, which has been augmented with a series of *stored procedures* which are used for servicing queries regarding equipment related information as well as queries required to build the GUI. Furthermore LMES+ has been augmented with a series of tables, which contain information about which elements the GUI must contain. The *Presentation Layer* receives information from the *Data Storage Layer* through the *Business Logic Layer* and uses it to present equipment related information and building the GUI. The *Business Logic Layer* is responsible for establishing the communication with LMES+. In its current version the *Business Logic Layer* is only fully equipped to communicate with an *Oracle* database, but it has been designed as generically as possible in order to ensure that it will be possible to replace the database with another one with as little effort as possible.

Conclusion. An application meeting the requirements set forth by NNE A/S as well the project authors has been successfully implemented. The project has been created in accordance with the methods from the *Rational Unified Process*, which has led to a method of working where there has been much focus on structured code as well as concise and thorough documentation. The final application is a complete product, which offers a dynamic GUI and a subset of the final functionality. Thus the application is useful for NNE A/S, when deciding whether or not this is a product they wish to develop further. The application has furthermore been documented and implemented in a way that aims at making it easy to continue working on and adding further functionality to it.

Key words: Management Execution System, MES, Portable Digital Assistant, PDA, .NET, C#, dynamic GUI.

Resumé

Portable Operator Assistant
Af Anders Toft Larsen (s022357) og Torben Boeck Jensen (2022359) i samarbejde med NNE A/S

Baggrund. NNE A/S ønsker, at det bliver undersøgt, om det er ønskværdigt at integrere brugen af håndholdte computere (PDA'er) i deres eksisterende LMES+ system. Til det formål ønskes udviklet et pilot-projekt, der består af en applikation, der kan afvikles på en PDA og kan bruges til at foretage simple udstyrsrelaterede forespørgsler. Udover selve funktionaliteten er der lagt megen vægt på, at den grafiske brugergrænseflade (GUI), applikationen præsenterer, skal kunne opdateres og ændres, uden at der er behov for at ændre på applikationskoden.
Metoder. For at sikre, at udviklingen af applikationen foretages på proffesionel vis, skal metoder fra *Object Oriented Analysis and Design* anvendes. Nærmere bestemt skal *Rational Unified Process* anvendes sammen med *Unified Modelling Language* notation. For at indfri ønsket om at GUI'en skulle kunne opdateres uden at ændre applikationen, skal denne designes som en dynamisk data-drevet GUI. Dette indebærer, at selve GUI'en blot præsenterer elementer, som er specificeret andetsteds. For at holde specifikationen af GUI'ens indhold helt ude af applikationens kode, skal disse oplysninger placeres i det eksisterende LMES+ system og derfra sendes til GUI'en.
Resultater. Den endelige løsning er en trelagsmodel bestående af *Presentation Layer*, *Business Logic Layer* og *Data Storage Layer*. *Presentation Layer* afvikles på PDA'en og kommunikerer med *Business Logic Layer* vha. en WiFi forbindelse. *Business Logic Layer* består af en *web service*, som kommunikerer med *Presentation Layer* og en *Data Access* del, som kommunikerer med *Data Storage Layer*. *Data Storage Layer* består af det eksisterende LMES+ system og den dertil hørende *Oracle* database, som er blevet udvidet med en række *stored procedures* til servicering af udstyrsrelaterede forespørgsler og forespørgsler vedrørende opbygning af GUI. Derudover er LMES+ blevet udvidet med en række tabeller, der indeholder informationer om hvilke elementer, GUI'en skal indeholde. *Presentation Layer* modtager informationer fra *Data Storage Layer* via *Business Logic Layer*, og på baggrund af disse præsenteres udstyrsrelaterede oplysninger og GUI'en opbygges. *Business Logic Layer* står for opsætning af kommunikationen til LMES+. I sin nuværende form er *Business Logic Layer* kun fuldt udbygget til at kunne kommunikere med en *Oracle* database, men det er designet så generisk som muligt for at sikre, at det med så lidt indsats som muligt vil kunne lade sig gøre at skifte den eksisterende database ud med en anden.
Konklusion. Det er lykkedes at implementere en applikation, der lever op til de krav, der er blevet specificeret af såvel NNE A/S som af projektforfatterne. Projektet er blevet udarbejdet i overenstemmelse med metoderne i *Rational Unified Process*, hvilket har betydet en arbejdsform med stort fokus på struktureret kode og på klar og grundig dokumentation. Den endelige applikation fremstår som et helstøbt produkt, der tilbyder en delmængde af den endeligt ønskede funktionalitet og en dynamisk GUI. Således er applikationen af en sådan karakter, at NNE A/S vil kunne anvende den til at vurdere, hvorvidt der er tale et produkt, der skal arbejdes videre med. Applikationen er endvidere dokumenteret og implementeret på en måde, der har til hensigt at gøre det nemt at arbejde videre med og tilføje ny funktionalitet til denne.
Nøgleord: Management Execution System, MES, Portable Digital Assistant, PDA, .NET, C#, dynamisk GUI.

Forord

Nærværende dokument udgør sammen med den i dokumentet omtalte applikation Diplom-IT eksamensprojekt for Torben Boeck Jensen (s022359) og Anders Toft Larsen (s022357). Projektet er blevet til i samarbejde mellem projektforfatterne, NNE A/S og DTU. Dokumentet indeholder en fuldstændig dokumentation af et add-on, der er blevet udviklet til et eksisterende NNE produkt.

Projektforfatterne vil gerne takke NNE A/S for at have stillet lokaler og computere til rådighed. Herudover skal Lars Grønnegaard Nielsen, Martin Hinge, Ole Wulff, Jesper Mandahl Hansen, Niels C. Andersen, Carl Høgstedt og Henrik Møller fra NNE A/S, som alle har bidraget med teknisk assistance og gode råd, have stor tak for at have afsat tid til os. Endvidere skal Peter Bertel Ottessen og Erik Sattler ligeledes fra NNE A/S, have stor tak for godt selskab og for at have båret over med os, mens vi har delt kontor under projektskrivningen. Til slut skal Mads Nyborg fra DTU have stor tak for vejledning i forbindelse med udarbejdelsen af opgaven og det samme skal Dortea Koldborg Jepsen, ligeledes fra DTU, for megen praktisk hjælp gennem hele forløbet.

Indholdsfortegnelse

Definitioner

BES: Batch Execution System
CLR: Common Language Runtime
EMS: Equipment Model System
ER-diagram: Entity-Relationship-diagram
GUI: Graphical User Interface (grafisk brugerflade)
HTTP: HyperText Transfer Protocol
ID: Interaktionsdiagram
IDE: Integrated Development Environment
IL: Intermediate Language
JIT: Just in Time
LMES+: Local Manufacturing Execution System+
MES: Manufacturing Execution System
OOAD: Object Oriented Analysis and Design
PDA: Personal Digital Assistant
POA: Portable Operator Assistant
RPC: Remote Procedure Call
SOAP: Simple Object Access Protocol
SQL: Structured Query Language
SSD: System Sequence Diagram
SSID: Service Set Identifier
UML: Unified Modelling Language
URL: Uniform Resource Locator
WSDL: *Web services* Description Language
XML: Extensible Markup Langauge
XSD: *XML Schema Definition*
UDDI: Universal Description, Discovery and Integration
ADO: ActiveX Data Objects
ERP: Enterprise Resource Planning
PCS: Process Control System
DBMS: Database Management System

Indledning

I det følgende gives først en kort gennemgang af de forskellige kapitler og herefter gennemgås den struktur, der er valgt for opgaven.

Kapitler

Kaptiel 1 indeholder en diskussion af de metoder, der anvendes i forbindelse med projektet. Herunder metoder til såvel applikations- som databasedesign.
Kapitel 2 giver en gennemgang af de teknologier og de værktøjer, der anvendes i forbindelse med projektet. Herunder gennemgås bl.a. .NET miljøer
Kapitel 3 omhandler projektstyring. I dette kapitel gennemgås den overordnede plan for afvikling af projektet samt hvilke tiltag, der er gjort, for at holde projektet inden for denne plan.
Kapitel 4 indeholder en gennemgang af NNE's LMES+ system.
Kapitel 5 indeholder kravspecifikationerne for projektet. Herunder afgrænses systemet, der præsenteres aktører og målsætninger, defineres *use-cases*, opstilles supplerende krav samt foretages risikovurdering.
Kapitel 6 indeholder designspecifikationerne for projektet, som er blevet til på baggrund af kravspecifikationerne. Herunder præsenteres klassediagram for applikationen, interaktionsdiagrammer for de forskellige *use-cases*, og de forskellige lag i applikationen gennemgås.
Kapitel 7 omhandler implementering og gennemgår den endelige løsning, som er blevet til på baggrund af designspecifikationerne.
Kapitel 8 gennemgår test af systemet. Herunder gennemgås modultest og acceptance-test.
Kapitel 9 gennemgår deployment af det endelige system. Herunder gennemgås applikationsinstallation samt hvilket udstyr er nødvendigt.
Kapitel 10 gennemgår mulige fremtidige udvidelser af den udviklede applikation.
Kapitel 11 indeholder projektets konklusion.
Kapitel 12 indeholder referencer til de kilder, der er anvendt i forbindelse med projektet.

Struktur

I forbindelse med projekter er der blevet vedtaget nogle retningslinier for layout og strukturering af rapporten.

Med hensyn til layout er der taget udgangspunkt i IMM's retningslinier for rapportskrivning [16]. Times New Roman i punktstørrelse 12 anvendes til brødtekst, bortset fra i tabeller og lister, hvor punktstørrelse 10 anvendes. Herudover er det vedtaget at bruge Courier New skrifttypen til kode og kursiv skrift til tekniske termer. Projektet er inddelt i en række kapitler med tilhørende underafsnit.

1 Metode

Dette kapitel diskuterer de metoder, der anvendes i forbindelse med projektet. Dette gælder såvel metoder til kravspecificering, design, projektstyring, implementering og andet.

1.1 Software Engineering

Software Engineering går ud på at udvikle og vedligeholde applikationer ved at anvende en række discipliner og praksiser fra forskellige felter, herunder computervidenskab og projektledelse. For at tilgå projektet på en professionel måde, er det vigtigt at vælge en metode til udarbejdelsen af det. Til det formål findes en lang række *Software Engineering* metoder, og i forbindelse med dette projekt vil en metode kaldet *Object Oriented Analysis and Design* (OOAD) blive anvendt. Der findes endvidere forskellige specifikke paradigmer indenfor OOAD, herunder *Rational Unified Process* (RUP), som vil blive anvendt til dette projekt. Et vigtigt værktøj for de fleste OOAD-discipliner, således også RUP, er *Unified Modelling Language* (UML), som er en standard for grafisk notation. OOAD, UML og RUP vil blive diskuteret i hhv. afsnit 1.1.1, 1.1.2 og 1.1.3.

Det er vigtigt, at den applikation, der bliver udviklet, er så robust som mulig. Dette indebærer en række overvejelser, af hvordan koden bedst struktureres. Til hjælp ved dette har forskellige softwareudviklere udviklet en række såkaldte *design patterns*. Disse repræsenterer "best practices" i forbindelse med design, og forskellige af disse vil blive anvendt i dette projekt til at sikre, en velfungerende applikation, der er så nem som mulig at vedligeholde og udvide med ny funktionalitet. *Design patterns* bliver diskuteret i afsnit 1.1.4.

Klasse-interfaces er et vigtigt instrument til at sikre kode, der er nem at opdatere. Disse gennemgås i afsnit 1.1.5.

1.1.1 OOAD

OOAD er en metode til at omsætte et problem til en objektorienteret løsning. Metoden indebærer at analysere problemet med henblik på at kunne opstille en kravspecifikation, der anvendes som basis for et objektorienteret design. Således beskæftiger OOAD sig med objektorienteret analyse og design. Disse vil blive gennemgået i de to følgende afsnit. For yderligere oplysninger om OOAD kan bl.a. henvises til Object Oriented Analysis and Design [14]. Sideløbende med analyse og designdisciplinen skal der genereres testdokumentation.

Som nævnt vil RUP blive anvendt som specifik *Software Engineering* metode, så de følgende afsnit har til formål at give en kort introduktion til de overordnede discipliner indenfor OOAD. De mere detaljerede aspekter diskuteres i forbindelse med gennemgangen af RUP i afsnit 1.1.3.

1.1.1.1 Objektorienteret analyse

Analysedelen af OOAD har at gøre med at opstille krav og specifikationer, der opstiller systemet som en samling objekter, hvor objekter er repræsentationer af ting fra den virkelige verden eller abstraktioner over disse. Det er her objekternes struktur og opførsel defineres, dvs. objektets navn, dets attributter og dets relationer til andre objekter. [15][1]

1.1.1.2 Objektorienteret design

Designdelen af OOAD har at gøre med udvikle en objektorienteret model på baggrund af de krav, der er blevet identificerede i forbindelse med den objektorienterede analyse. Dette foregår ved at de objekter, der blev defineret i

[1] http://www.sei.cmu.edu/str/descriptions/ooanalysis.html

forbindelse med analysen bliver repræsenteret som software-klasser. Det er også her at de forskellige klassers attributter får specificeret deres datatyper, samt hvilke metoder klasserne indeholder. [15][2]

1.1.2 UML

UML er et objekt modellering- og specificeringssprog, der anvender grafisk notation. Således anvendes UML til at visualisere et systems struktur og design på en måde, der lever op til de krav, der er specificerede for dette. [8]

UML bliver anvendt i forbindelse med dette projekt og specificeringen af den applikation, der skal udvikles, som det foreskrives i RUP. De forskellige diagrammer, der anvendes, bliver diskuteret i næste afsnit (1.1.3), der handler om RUP.

1.1.3 RUP

RUP indebærer en udviklingsproces, der inddeles i fire hovedfaser. Disse faser er *inception*, *elaboration*, *construction* og *transition*. Faserne gennemløbes flere gange, og hver af disse er inddelt i discipliner. De discipliner, der vil blive gennemført i forbindelse med denne applikation, er krav, design, implementering, test, deployment, og projektstyring. Det skal bemærkes, at adskillige af disciplinerne gentages og forfines i de forskellige faser. Et bærende princip for RUP er, en såkaldt *use-case* dreven udviklingsmodel, hvilket indebærer at kravspecifikationen for langt hoveddelens vedkommende består af *use-cases*. Disse er krav til systemet, der er udviklet i samarbejde mellem slutbrugeren og udvikleren, og vil blive diskuteret i afsnit 1.1.3. På baggrund af disse *use-cases* defineres en række såkaldte konceptuelle klasser, som repræsenterer de forskellige elementer i systemet. Disse kan præsenteres vha. forskellige UML-diagrammer, herunder *use-case* diagram, domænemodel, sekvensdiagrammer, klassediagrammer, interaktionsdiagrammer og andre. Disse bliver alle brugt i dette projekt og bliver gennemgået i de følgende underafsnit. For yderligere oplysninger om disse diagrammer henvises til *Applying UML and Patterns – An Introduction* [7].

Udviklingen af applikationen skal fokusere på nem vedligeholdelse, hvilket også inkluderer grundig dokumentation af de ovennævnte faser og discipliner. Det er endvidere vigtigt, at de beslutninger, der tages igennem hele projektforløbet afspejler de *use-cases*, der er blevet definerede.

1.1.3.1 Use-case diagram

Use-case diagrammet giver et overordnet overblik over de forskellige use-cases og hvorledes de er forbundne til ydre aktører samt til hinanden.

1.1.3.2 Sekvensdiagrammer

Sekvensdiagrammer bruges til at vise kommunikationen mellem de konceptuelle klasser, der er blevet defineret ud fra *use-casene*. Således kan de anvendes som et supplement til *use-casen* for at visualisere denne. Et sådant diagram står dog aldrig alene, og det er altid selve *use-casen*, der er den centrale del af kravspecifikationen.

[2] http://www.sei.cmu.edu/str/descriptions/oodesign.html

1.1.3.3 Domænemodel

Domænemodellen giver et overblik over de forskellige konceptuelle klasser og deres indbyrdes afhængigheder. Denne danner udgangspunkt for klassediagrammet, som gennemgås i afsnit 1.1.3.5.

1.1.3.4 Interaktionsdiagrammer

Interaktionsdiagrammer bruges til at visualisere kommunikationen imellem de software-klasser, der er blevet identificeret ud fra domænemodellens konceptuelle klasser. Med udgangspunkt i *use-casene*, bliver denne kommunikation mellem objekter vist på et interaktonsdiagram i form af at de forskellige objekter har ansvar for at udføre forskellige handlinger. Interaktionsdiagrammet er konstrueret vha. et sekvensdiagram, og beskriver horisontalt interaktionen mellem de forskellige klasser, hvor tiden aflæses vertikalt.

1.1.3.5 Klassediagram

Klassediagrammet giver et overblik over de forskellige software-klasser og deres indbyrdes afhængigheder. Her specificeres også eventuelle interface-klasser, som disse måtte implementere. Yderligere er det her, klassernes metoder og attributter bliver angivet.

1.1.4 Design patterns

Der findes en lang række forskellige *design patterns*. I de følgende afsnit bliver de *design patterns*, der bliver anvendt i forbindelse med dette projekt diskuteret, og der bliver her også gjort rede for, hvilke designmæssige aspekter de har med at gøre. De relevante *design patterns* bliver først introducerede herunder i hver deres afsnit, mens en mere anvendelsesspecifik diskussion af dem gives i kapitlet om design. Den diskussion af *design patterns*, der gives i forbindelse med dette projekt vil lægge vægt på elementer af de forskellige patterns, der har relevans for projektet. For en mere udtømmende omtale henvises til *Design Patterns* in C# [4] og *Applying UML and Patterns – An Introduction* [7], hvorfra en del af oplysningerne i den følgende diskussion ligeledes stammer.

1.1.4.1 Creator

Creator design pattern anvendes til metodisk at vurdere hvilke klasser, der skal være ansvarlige for at oprette instanser af andre klasser. En klasse B skal være ansvarlig for at oprette instanser af en anden klasse A, hvis den aggregerer A-objekter (fx collections), indeholder A-objekter, indeholder data, som skal anvendes til at skabe A-objekter eller på andre måder er tæt forbundet til A.

1.1.4.2 Information Expert

Information Expert bruges til at beslutte, hvilke klasser, der skal have ansvar for hvilke funktioner. Filosofien er, at de klasser, der har den fornødne information, skal være de klasser, der har ansvar for funktionalitet relateret til denne information. Dette betyder, at en klasse, der har en given information, ligeledes skal levere metoder til at hente og modificere denne information.

1.1.4.3 Observer

Der vil ofte forekomme situationer, hvor et givent objekt er interesseret i at blive informeret, når et andet objekt ændrer sig. *Observer design pattern* bruges til at

definere, hvorledes dette bør foregå. Dette indebærer, at der defineres en måde, hvorpå interesserede objekter kan lytte til det relevante objekt og derved blive gjort opmærksom på forandringer. Hermed opnås en en-til-mange relation mellem det interessante objekt og de interesserede objekter, således at når det interessante objekt ændrer sig, gøres alle de interesserede objekter opmærksom herpå og kan derefter reagere på denne forandring. [4][3]

C# giver mulighed for nem implementering af *Observer design pattern*. Dette er understøttet ved hjælp af såkaldte *delegates*. *Delegates* fungerer ved at den klasse, der ønsker at observere en anden klasse, bliver gjort opmærksom på en hændelse (*event*) i den observerede klasse. Dette sker i praksis ved, at der bliver specificeret en metode i den observerende klasse, som kaldes, når den relevante *event* i den observerede klasse forekommer. Adskillige klasser kan være sat til at lytte på den samme *event*. Et typisk tilfælde er et tryk på en knap i en GUI. Dette vil føre til at denne knaps *click-event* bliver genereret. Hvis en klasse er sat op til at reagere på denne *event*, bliver den metode, der er blevet specificeret kaldt. Der er således tale om, at håndteringen af, at der er blevet trykket på den relevante knap er blevet uddelegeret til en anden klasses metode (eller flere klassers metoder). Heraf navnet *delegate*, der kan oversættes til stedfortræder.

1.1.4.4 Low Coupling

Low Coupling bruges til at sikre, at klasser er så uafhængige af hinanden som muligt. Dette har bl.a. til formål at sikre, at ændringer i en klasse, så vidt det kan lade sig gøre, ikke påvirker andre klasser. I forbindelse med design af klasser med henblik på *Low Coupling* er der en række regler, man kan følge, for hvad en klasses metoder må kommunikere med. En samling af sådanne regler er den såkaldte *Law of Demeter*, der specificerer, at en klasses metoder, kun må kommunikere med følgende:

1. Klassen selv (this objektet) og dennes parametre
2. Parametre, der er sendt til klassens metoder
3. Objekter oprettet i klassens metoder
4. Objekter der er delkomponenter af objektet

Punkt 4 henviser til, at det i henhold til *Law of Demeter* er lovligt, at en klasse tilgår samlinger af objekter (såkaldte collections) og de til sådanne collections hørende objekter. I særdeleshed angiver *Law of Demeter*, at en klasses metoder bør undgå at kalde metoder i et objekt, der returneres af et andet objekt (dvs. konstruktioner af typen `objekt1.objekt2().metode();` bør undgås). For yderligere omtale af *Law of Demeter* henvises til Karl Lieberherrs artikel herom [6].

Udover at definere, hvad en klasse må kommunikere med, skal det også slås fast, at en klasse, der kommunikerer med adskillige forskellige klasser, ligeledes er i modstrid med princippet om *Low Coupling*, da klassen dermed kobles til adskillige klasser og en ændring i én af disse kan have konsekvenser for, hvordan klassen kan tilgå andre klasser. Ved at tilstræbe *Low Coupling* opnår man, at ændringer i en given klasse så vidt muligt kun har konsekvenser for de klasser, der direkte kommunikerer med dem og mindsker afledte virkninger af sådanne ændringer.

Det er vigtigt at bide mærke i, at høj indbyrdes afhængighed mellem objekter især er et problem, hvis et objekt bliver gjort afhængigt af et andet objekt, som er ustabilt, fx i form af at deres implementering ændres ofte, og en vis grad af indbyrdes afhængighed imellem objekter er nødvendig.

[3] Kapitel 9

1.1.4.5 High cohesion

High Cohesion bruges til metodisk at vurdere, hvilke opgaver en given klasse skal have ansvar for at løse. Formålet er at opnå klasser med metoder, der angriber problemområder, som er beslægtede. Hermed sikres klasser, som er nemme at gennemskue, genbruge og vedligeholde. Herudover sikres det, at klasserne er så resistente over for ændringer som muligt. Jo flere forskelligartede opgaver en klasse har, jo større er risikoen for at skulle ændre i den, når omgivelserne ændrer sig. Er alle klasser derimod designet i henhold til *High Cohesion* vil det være så få klasser som muligt, der skal ændres når en given del af miljøet ændrer sig – dvs. den eller de klasser, der har med det relevante område at gøre.

1.1.5 Klasse-interfaces

Et klasse-interface er en slags kontrakt, som de klasser, der implementerer klasse-interfacet, skriver under på. Det indeholder metoder, som kun er angivne ved navn og de parametre, de modtager, og altså ikke indhold. Klasser, der implementerer klasse-interfacet skal ligeledes implementere de i klasse-interfacet nævnte metoder.

Idet denne applikation udvikles i C#, vil der være en række designmæssige beslutninger, der motiveres af, hvorledes dette programmeringssprog er opbygget. C# er skabt på en måde, der lægger megen vægt på klasse-interfaces. I modsætning til andre objektorienterede programmeringssprog, har nedarvning ikke den helt store rolle i C#, idet det kun er muligt at nedarve fra en klasse. Den rolle, som nedarvning normalt har, er til dels overtaget af klasse-interfaces [3][4]. Selvom klasse-interfaces ikke leverer mulighed for at definere en base-klasse med fungerende metoder, er der mange af de samme ting, der kan opnås ved den rette brug af klasse-interfaces. Endvidere giver brug af klasse-interfaces en kode, der er nem at udvide. Hvis der specificeres klasse-interfaces til de forskellige klasser i applikationen, gøres det væsentligt nemmere at skrive nye klassevarianter samt i det hele taget at udvide applikationen på måder, som det ikke var tænkt under udviklingen af denne. Af samme årsag, skal der skrives klasse-interfaces til alle klasser, hvor det giver mening, i forbindelse med implementeringen af denne applikation. Dette kan i nogle tilfælde betyde, at der næsten er et 1:1 forhold mellem klasse-interfaces og klasser, men på længere sigt, kan det vise sig, at der opstår behov for yderligere varianter af klasserne til uforudsete anvendelser, og så vil det være en fordel, at der er specificeret klasse-interfaces.

1.2 Database design

Ved design af et større software system opstår ofte et behov for at anvende en database til håndtering af data. Ligesom det er tilfældet for resten af applikationen, er det også nødvendigt at analysere og designe en database ved brug af passende metoder [18][5]. Den metode, der her skal anvendes, opdeles normalt i seks trin, hvoraf dette projekt kun anvender de første, som er med til at udvikle selve databasegrundlaget. Efter dette er gjort er selve strukturen for databasen på plads, hvilket er fokus for denne opgave. Efterfølgende kan de tre sidst trin gennemføres. Disse beskæftiger sig med en række faktorer såsom normalisering, ydelse og sikkerhed, som kan overvejes for at gøre databasen mere robust og effektiv. De forskellige trin vil her kort blive introducerede:

[4] Side 87
[5] Se kapitel 2 for grundlæggende beskrivelse.

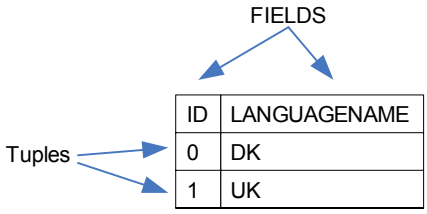
Requirements Analysis: Den ønskede applikations databehov skal findes, og det skal gøres klart, hvilken form for information det ønskes at gøre tilgængelig i databasen. Dette gøres bl.a. ved uformelle undersøgelser af eksisterende applikationer og datagrundlag, samt hvilke systemer den nye applikation skal erstatte og komplimentere.

Conceptual Database Design: Der udvikles en højniveaubeskrivelse af de data som skal lagres i databasen ud fra de opsamlede informationer fra *Requirements Analysis*-fasen. Her anvendes bl.a. ER-diagrammer til at beskrive en tidlig fase af datamodellen. Typisk er formålet med dette trin, at bruger og udvikler opnår en fælles forståelse for, hvad datagrundlaget består af, som efterfølgende skal udmunde i den relationelle model.

Logical Database Design: I dette trin skal der vælges et konkret DBMS, ved hjælp af hvilket det konceptuelle databasedesign skal implementeres. Det betyder at specifikke detaljer såsom *fields* (også kaldet *columns/attributes*) skal defineres. Herefter skal det konceptuelle ER-diagram konverteres til et *relational database schema* (også kaldet et *logical schema*).

Da der i forbindelse med dette projekt anvendes en relationel database, vil der her blive taget afsæt i den såkaldte *Relational Model*. Data repræsenteres i denne model som *relations*. Disse består af to hovedelementer: Et *relation schema* og en *relation instance*. *Relation instance* er en tabel i databasen, hvor *relation schema* beskriver kolonnerne som tabellen skal bestå af. Et *schema* beskriver således *fields*, samt et *domain* i hvert *field*. Et *domain* indeholder et *domain name* samt et antal *values*.

En instans af en *relation* består af et antal *tuples* (også kaldet *records/rows*), og i en typisk tabel på en database, er det en række som betegnes som en *tuple*, og hver række har det samme antal *fields*. Et simpelt eksempel på en instans af en *relation* kan ses på Tabel 1.



Tabel 1 - En instans af POA_LANGUAGE tabellen

Tabellen viser, to *fields* (ID og LANGUAGENAME), og ligeledes to *tuples*. For mere information om *relational model*, se [18][6]

Schema Refinement: For at optimerer databasen, og forebygge mod potentielle problemer i den oprindelige design, bør der foretages yderligere og mere grundige analyser af database *schema*. Dette er en mere teoretisk indgangsvinkel til konstruktionen af databasen. Problemer ved det oprindelige design kan være redundant data der gemmes i databasen, fejl (anormalitet) ved *update*, *insertion* eller *deletion* af data i databasen. For at optimere og forbedre et *relation schema*, kan *normal forms* anvendes. Der findes en række forskellige *normal forms*, herunder *Boyce-Codd Normal Form* og *Third Normal Form*. Yderligere information findes i [18][7]

Physical Database Design: I dette trin er designet nået et niveau, hvor kriterier vedrørende ydelsesevne samt tabelindeksering er i fokus. Dette er relevant når

[6] Se Kapitel 3 for mere om *Relatioanal Model*.
[7] Se kapitel 19 for mere om *Schema refinement*.

datapresset når et vist niveau, eller mængden af data er af betydelig mængde, og gennemløbningen af tabeller har en uacceptabel tidslængde. I dette trin designes et såkaldt *physical schema*. Her er det især af høj prioritet at definere arbejdsbyrden (*workload*) for databasen, og derefter opdatere de flaskehalse der fremgår af analysen. Det kan være hardwaremæssige optimeringer, men i højere grad selve designet af databasestrukturen. *Workload* er specificeret ved en række brugerkrav som skal være opfyldt, det kan både være maksimal tider på bestemte forespørgsler eller processerede transaktioner pr. sekund. Her kan *indexing*, *search key* og *Clusters* implementeres som et led i optimeringen af databasedesignet. Yderligere information findes i [18][8]

Application and Security Design: Udvikling af applikationer, hvor der indgår databasebrug, indeholder ofte aspekter som går udover selve databasen. Dette betyder at metoder der kan beskrive softwareprojektet som en helhed også er nødvendige for at opnå et fuldendt system. Hvis man ikke er i stand til at beskrive hele systemet, kan den rette sikkerhedspolitik ikke implementeres, og sårbare data kan være tilgængelige for uautoriserede personer. Der skal bl.a. tages højde for brugere, brugergrupper, afdelinger og processer der er involverede i applikationen, som derigennem interagere med databasen. Her skal der tages stilling til områder af databasen der skal være tilgængelig for specifikke brugere, mens andre ikke må tilgå samme områder. Det er her der kan drages stor fordel af udbyggede DBMS-systemer. Yderligere information findes i [18][9]

[8] Se kapitel 20 for mere om *Physical database design and tuning*.
[9] Se kapitel 21 for mere om *Physical Security and Authorization*.

2 Teknologi

Dette kapitel indeholder en diskussion af teknologier i bred forstand, der blevet anvendt ved udarbejdelsen af dette projekt. Herunder diskuteres udviklingsmiljø, andet programmel og software teknologier.

Applikationen skal udvikles i .NET-miljøet. Nærmere bestemt skal *.NET Framework* 2.0 og *.NET Compact Framework* 2.0 anvendes. Herunder skal C# 2.0 anvendes som udviklingssprog. Til dette formål anvendes *Microsoft Visual Studio* 2005 (herefter omtalt som *Visual Studio*). Dette vil blive diskuteret herunder.

Herudover diskuteres bl.a. *web services*, *Extensible Markup Language* (XML), LMES+ og ADO.NET 2.0.

2.1 .NET miljøet

I forbindelse med dette projekt anvendes .NET miljøet som udviklingsplatform. I og med, at der skal udvikles både til en Pc-platform og en håndholdt platform skal såvel *.NET Framework* som *.NET Compact Framework* anvendes. *Compact Framework* udgør en begrænset delmængde af *.NET Framework*, og er optimeret til at køre på indlejrede enheder med begrænsede ressourcer. En udtømmende diskussion af forskellene på *.NET Framework* og *.NET Compact Framework* er uden for rammerne af denne opgave. Der henvises til *.NET Compact Framework* Programming with C# [5].

Der er en række ting, der er relevante i forbindelse med .NET miljøet, og disse vil blive diskuteret i det følgende. I afsnit 2.1.1 gennemgås *Intermediate Language* kompilering, i afsnit 2.1.2 gennemgås *Common Language Runtime* (CLR), i afsnit 2.1.3 gennemgås *assemblies* og i afsnit 2.1.4 gennemgås *namespaces*.

2.1.1 Intermediate Language

Når en .NET kodefil kompileres med en kompiler, der er kompatibel med CLR, bliver denne ikke kompileret til maskinspecifik kode, men derimod til et såkaldt *Intermediate Language* (IL), som er et assemblerlignende sprog. Filer som er kompileret til IL gemmes som enten eksekverbare filer (.exe) eller library-filer (.dll). Det skal dog bemærkes, at der ikke er tale om almindelige eksekverbare filer, da de er afhængige af at *.NET Framework Runtime* kører. Ved afvikling af filer sørger *Runtime* systemet for at kompilere IL-koden til maskinspecifik kode ved hjælp af den såkaldte *Just in Time* (JIT) kompiler, som findes i CLR.

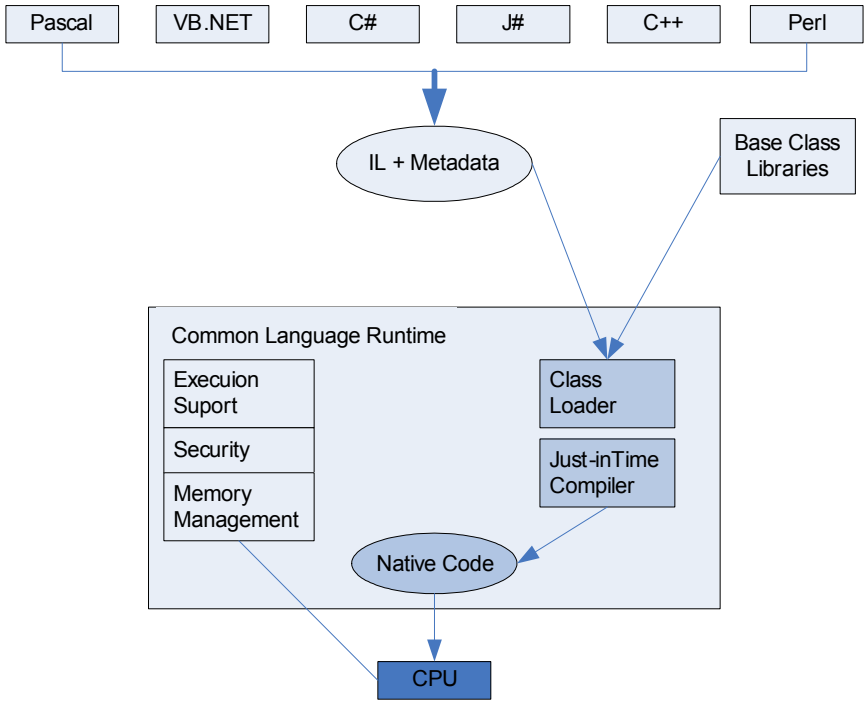
Denne kompilering foretages ”on the fly” ved at Runtime systemet tager ukompileret kode ind og kompilerer efterhånden som applikationsafviklingen kræver det. På den måde er det muligt at optimere kompilering dels i form af at kode kun kompileres færdigt, hvis det skal anvendes og dels i form af at det er muligt at bruge CPU-specifikke JIT-kompilere, der generer kode, som er optimeret til CPU’en på det system, hvor applikationen afvikles. Kode som på den måde håndteres af CLR, kaldes *Managed Code*. Til IL er knyttet *metadata*, som bl.a. bruges af JIT kompileren og til *Garbage Collection*.

2.1.2 Common Language Runtime

CLR er navnet på den *virtual machine* og det *runtime library*, der ligger under .NET. CLR er på den måde helt central for .NET og det er igennem denne, at det gøres muligt, at programmer skrevet i mange forskellige programmeringssprog kan køre under .NET.

Figur 1 er taget fra Core C# and .NET [3][10] og giver et overblik over CLR. Heraf kan ses, hvorledes kode fra en række forskellige programmeringssprog kompileres til IL og senere via JIT kompileren til maskinspecifik kode.

[10] Side 9, figur 1-3



Figur 1 - Diagram over Common Language Runtime

2.1.3 Assemblies

I .NET indeholdes al *managed Code* i *assemblies*. Et *assembly* består af delvist kompileret kode, og genereres når .NET kode kompileres. Et *assembly* kan være enten en eksekverbar fil eller en dll-fil, og kan være genereret ud fra flere forskellige klasser og kodefiler.

Assemblies er et vigtigt led i at skabe sikker kode og sikre versionsstyring, hvilket blandt andet gøres ved at gøre *assemblies* til såkaldte *strongly named assemblies*. Såvel *strongly named assemblies* som de sikkerhedsmæssige aspekter ved *assemblies* giver anledning til meget omfattende omtale, hvilket ligger uden for rammerne af denne opgave. Der vil i afsnit 2.1.3.1 og afsnit 2.1.3.2 blive præsenteret nogle overordnede aspekter ved disse emner. For en mere udtømmende omtale af disse emner henvises til Core C# and .NET [3][11].

2.1.3.1 Sikkerhedsaspekter ved assemblies

Ofte indeholder applikationer kode, der er skrevet af tredjepartsleverandører. For at sikre at utroværdig kode ikke får privilegeret adgang til applikationer, har *Microsoft* defineret *.NET Code Access Security*. Denne anvender *assemblies* og såkaldt *evidence* til at vurdere, hvilken kodegruppe et *assembly* tilhører og hvilke dertil knyttede rettigheder den har.

Evidence kan bestå af flere forskellige ting. Et meget vigtigt værktøj til unikt at identificere *assemblies* er ved at signere dem og derved gøre dem til *strongly named assemblies*. Dette diskuteres i det følgende afsnit (2.1.3.2).

[11] I denne forbindelse har kapitel 15 stor interesse.

2.1.3.2 Strongly named assemblies

Et *strong name* gør det muligt at identificere et *assembly* unikt. Et *strongly named assembly* har fire attributter, der tilsammen udgør den unikke identifikation af denne: *Assembliets* filnavn, *assembliets* versionsnummer, *assembliets culture* identitet og *assembliets public key token*. Tilsammen udgør disse fire attributter *assembliets strong name*. *Culture* kan bruges til at forbinde en *assembly* til en specifik kultur eller et specifik sprog (fx "*en*" for engelsk). Dette kan bl.a. bruges til at lave sprogspecifikke versioner af en applikation. *Public Key Token* er en nøgle, der sammen med et *Private Key Token* bruges til at identificere *assemblien* unikt. [3]

Det giver en række fordele, at et *assembly* er *strongly named*. Dels, som nævnt i afsnit 2.1.3.1, er der nogle sikkerhedsmæssige aspekter, og ikke mindst knyttes der herved et versionsnummer til *assembliet*. Versionsstyring diskuteres i næste afsnit (2.1.3.3).

2.1.3.3 Versionsstyring

Et *assembly*, der er *strongly named,* har som nævnt knyttet et versionsnummer til sig. Dette bruges til at sikre, at forskellige *assemblies* der er afhængige af hinanden har samme versionsnummer. Er dette ikke tilfældet, vil der blive generet en *exception*. [3]

2.1.4 Namespaces

I .NET inddeles koden i *Namespaces*. Disse svarer til fx *packages* i Java og bruges til opdeling og indkapsling af programmer. *Namespaces* bruges til at inddele koden logisk, ligesom alle .NET klasse biblioteker er organiseret i *Namespaces*. For at tilkendegive, at en klasse skal kunne tilgå klasser fra et givent *Namespace*, skal `using` kommandoen bruges efterfulgt af navnet på det relevante *Namespace*. Når en klasse med et givent navn er indkapslet i et *namespace*, er det et krav, at den er den eneste klasse i *namespacet* med dette navn, hvorimod flere klasser med samme navn kan eksistere på tværs af *namespaces*.

2.2 C#

C# skal anvendes som programmeringssprog i dette projekt. C# er et objektorienteret programmeringssprog, der kan anvendes til at udvikle applikationer af mange forskellige slags, det være sig konsol applikationer, Windows Forms applikationer eller Web applikationer.

For at sikre gennemskuelig og ensartet kode, er det vigtig at anvende en standard for kodeskrivning. Den standard, der er anvendt i projektet gennemgås i afsnit 2.2.1. Til sidst gennemgås *generic collections*.

2.2.1 Kodestandarder

For at have en metodisk tilgang til strukturering af koden, anvendes den standard for kode, der er beskrevet i C# Coding Standard fra IDesign Inc. [10]. I tillæg hertil anvendes Hungarian Notation – se *.NET Compact Framework* Programming with C# [5][12] – som hjælp til at sikre en ensartet og overskuelig kode.

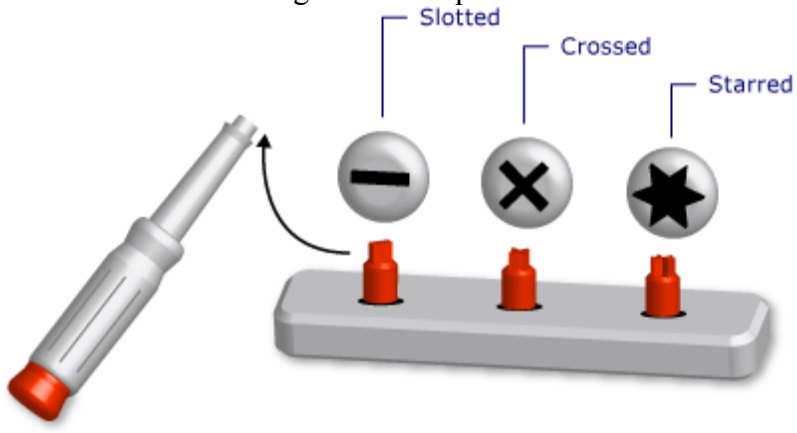
Herudover har udviklerne af .NET defineret en række regler for kodeskrivning. I den forbindelse har *Microsoft* udviklet et program, der hedder FxCop[13]. Det bruges til at gennemlæse kode og vurdere, hvorvidt den lever op til dette regelsæt. Nogle af

[12] Appendiks A, side 1247.
[13] http://www.gotdotnet.com/team/fxcop/

disse regler kan vurderes at være for restriktive alt efter hvilket projekt, der arbejdes på, men det kan bruges som en huskeliste over ting, der bør overvejes.

2.2.2 Generic Collections

I forbindelse med *.NET Framework 2.0* og *.NET Compact Framework 2.0* blev såkaldte *generic* `Collections` introducerede. En `Collection` er en samling af relaterede objekter, og at den er *generic* vil sige, at det tilpasser sig til at yde samme funktionalitet for en række forskellige data typer [17]. Figur 2 er taget fra [17][14] og viser et eksempel med en skruetrækker som et *generic* værktøj, idet det tilbyder samme funktionalitet med forskellige skruebits påmonteret.



Figur 2 - Skruetrækker som generic værktøj

Fordelen ved at anvende *generic* `Collections` er, at man på forhånd angiver hvilken type objekter, der skal opbevares i `Collectionen` og derved opnår at denne er *strongly typed*. Ved dette opnås det, at det ikke er nødvendigt at foretage typetjek i koden, idet dette overlades til kompileren.

I forbindelse med dette projekt skal anvendes to forskellige typer *generic* `Collections`, `List<T>` og `Dictionary<T>`, hvor `T` angiver den type objekter, der opbevares i `Collectionen`.

2.2.2.1 List

En `List` er en *generic* implementering af `ArrayList`-klassen. Udover den typesikkerhed, som en `List` giver gennem at være *strongly typed*, har denne yderligere den fordel fremfor en `ArrayList`, at når man indekserer et objekt i `Listen`, får man adgang til alle de metoder, der normalt ville være knyttet til objektet.

2.2.2.2 Dictionary

Et `Dictionary` er en *generic* implementering af `Hashtable`-klassen. `Dictionariet` består af en samling af `Keys` og `Values`. De forskellige værdier vil typisk være objekter, som har hver deres `Key` som indeks. Udover den typesikkerhed, som et `Dictionary` giver gennem at være *strongly typed*, har dette yderligere den fordel fremfor en `HashTable`, at når man indekserer et objekt i `Dictionariet`, får man, ligesom det var tilfældet med `List`-klassen, adgang til alle de metoder, der normalt ville være knyttet til objektet.

[14] http://msdn2.microsoft.com/en-us/library/w256ka79(VS.80,d=ide).aspx

2.3 Visual Studio

Visual Studio er et såkaldt *Integrated Development Environment* (IDE). I forbindelse med udviklingen af applikationen til dette projekt, skal *Visual Studio* anvendes. Dette program er i 2005 versionen skrevet specifikt til at understøtte *.NET Framework* 2.0 såvel som *.NET Compact Framework* 2.0. Anvendelse af *Visual Studio* giver en række muligheder, som vil blive nævnt forskellige steder i rapporten. Specifik skal her nævnes, at *Visual Studio* giver mulighed for at tilføje XML-kommentarer til koden. Disse kan senere hen fortolkes og på baggrund heraf kan der blive genereret kode-dokumentation.

2.4 *Microsoft Project*

I forbindelse med projektstyringen til dette projekt skal *Microsoft Project* anvendes. Dette program giver mulighed for at oprette en projektplan, hvor der angives varighed og afsættes timer og ressourcer til dette. Programmet giver bl.a. mulighed for at vise projektplanen som et Gantt diagram.

2.5 TestDirector

Til testformål skal programmet *TestDirector 8.0* anvendes. Dette program kan standardisere og digitalisere måden at teste på. Udover at sikre elektronisk dokumentation af testforløbet, giver anvendelse af *TestDirector* mulighed for styring med hvilke personer, der er knyttet til de forskellige tests, og hvornår og under hvilke forhold, de skal udføres. Dvs. både personer som designer, udfører, reviewer og godkender testen er beskrevet gennem hele forløbet, da det er et inkorporeret element i *TestDirector*. I NNE anvendes *TestDirector* som standard værktøj til softwaretest.

Programmet består af flere elementer. Først og fremmest skal der defineres en række *Requirements*. Disse *Requirements* repræsenterer krav, de forskellige tests skal indfri. Det kan fx være forudsætninger, som skal være opfyldte, inden testen udføres, og det kan være henvisninger til steder i dokumentationen, hvor funktionelle krav til applikationen er specificerede. Efter at have defineret *Requirements*, skal selve testen defineres. Dette gøres ved at der beskrives en *Test Plan*, hvor testens forløb angives trin for trin. Efter dette er gjort, anvendes et *Test Lab*, hvor selve testen eksekveres. Det skal her bemærkes at der ikke er tale om en autogenereret test, men derimod et automatiseret testforløb. Dvs. testen skal udføres på vanlig vis, blot dokumenteres og rapporteres der elektronisk. Det kan dog lade sig gøre at automatisere selve testeksekveringen vha. tilføjelse af *test scripts*, men sådanne anvendes ikke i forbindelse med dette projekt. Testtrinene, der er beskrevet i Test Planen, skal afvikles, og det noteres om testtrinnet er succesfuldt udført. Under testafviklingen i *Test Lab* er der plads til at skrive kommentarer om testudførelsen, og ligeledes tilføje *screen dumps* hvis det er nødvendigt. Når testudførslen er udført i *Test Lab*, sendes testen til review, og godkendes herefter af en tredje part. På baggrund af den eksekverede test kan der herefter genereres en elektronisk rapport, som kan anvendes som dokumentation for testen. *Test Plan* og *Test Lab* gennemgås i de følgende afsnit.

2.5.1 Test Plan

En *Test Plan* består af fem elementer, der tilgås ved hjælp af hver deres faneblad:

Faneblad	Beskrivelse
Details	Her gives en beskrivelse af den aktuelle test. Herunder gives en beskrivelse af testen, samt angives personer, der er ansvarlige for forskellige led af denne såsom at godkende (*Approve*) og gennemgå (*Reviewe*) testen, inden denne udføres. Det er ligeledes her status på *test planen*, *reviews* m.m. beskrives.
Design Steps	Her angives de trin, som testen skal bestå af. Disse er listet i en tabel, hvori der indgår kolonner med trinnavn, beskrivelse og det forventede udfald. I forbindelse med det enkelte trin er det muligt at vedhæfte filer; det kan fx være programstubbe og screen dumps.
Test Script	Her kan der tilføjes *test scripts*, hvis der er valgt at automatisere dele af testen. Da typen af test er sat til Manual, kan dette element ikke anvendes i denne brug af *TestDirector*.
Attachments	Her kan vedhæftes filer, der er generelt anvendelige i forbindelse med testen. Det være sig både screen dumps, dokumenter eller URL'er.
Reqs Coverage	Her skal de relevante krav for at testen er opfyldt defineres. Kravene tilføjes ved at hive dem ind fra listen over allerede definerede *Requirements*. Kravene er, lige som design steps, listet på tabelformat, og indeholder kolonner til kravnavn, *reviewer* (ansvarlige person) samt kravbeskrivelse.

2.5.2 Test Lab

Det er i *Test Lab* selve eksekveringen af testen foregår. Først og fremmest skal et *Test Set* defineres. Under normale omstændigheder, er det ikke tilladt at tilføje tests til et *Test Set*, med mindre de alle er *Reviewede* og *Approvede*. Dette krav kan dog omgås ved at efterfølge navnet på *Test Settet* med _draft (fx test_draft). Efter at *Test Settet* er oprettet, tilføjes de forskellige tests ved at trække dem over fra listen over allerede definerede tests. Herefter er det muligt at vælge den enkelte test og eksekvere denne.

2.5.3 Testudførsel

Udførsel af testen startes fra *Test Lab*. Når testen er i gang, bliver testpersonen præsenteret for de enkelte testtrin et for et med dertilhørende beskrivelse og forventede resultat. Herudover er der er et vindue til indtastning af det faktiske resultat, hvor det dokumenteres, hvad der skete under udførelsen af det relevante testpunkt. Her kan også vedhæftes filer som et led i dokumentationen af de enkelte testtrins udførelse. Ønskes det at vedhæfte screen dumps, indeholder *TestDirector* en kamerafunktion, der nemt tager snapshots af indholdet af vinduer. Efter at have dokumenteret et testtrin, skal testpersonen angive, om trinet er afviklet succesfuldt eller ej og præsenteres derefter for næste trin. Efter alle trin er gennemgået afsluttes testen, og hvis alle trin er afviklet succesfuldt, fremgår testen som Passed af testplanen i *Test Lab* vinduet.

2.5.4 Automatisk rapport generering

TestDirector giver en lang række muligheder for automatisk at generere dokumentation af de udførte tests. Dels er det muligt at få genereret en standardrapport, der resulterer i en web-side og dels er det muligt at selv opsætte en rapport, som efterfølgende kan genereres.

2.6 *Web services*

En *web service* er en software teknologi, der tillader kommunikation mellem flere maskiner over et netværk.

Web services kan beskrives som en web-applikation, der fungerer som en separat enhed og afvikles på en applikationsserver. Dette skal ikke forstås sådan, at en *web service* ikke kan være en del af en større applikation. Enhver applikation kan have en *web service* komponent [11]. Dette fungerer i praksis ved, at *web servicen* gør en eller flere metoder[15] tilgængelige for omverdenen. For at en ekstern computer skal kunne tilgå *web services* på applikationsserveren, er det nødvendigt at udveksle beskeder mellem de to computere. I forbindelse med *web services* er der defineret en kommunikationsprotokol kaldet *Simple Object Access Protocol* (SOAP), der anvendes til dette formål. For at eksterne computere kan se, hvilke metoder en *web service* offentliggør, er det nødvendigt at den giver nogle oplysninger om sig selv i form af metadata. Til dette formål er defineret *Web services Description Language* (WSDL), der bruges til at beskrive *web services*. Begge disse standarder er XML-baserede. Den eksterne computer skal således blot understøtte WSDL og SOAP, samt kende den URL, hvorpå den ønskede *web service* findes, og herefter har den adgang til de metoder, som denne gør tilgængelig. *Web services* kan tilgås ude fra på forskellige måder. Det kan fx være en hjemmeside, der gør brug af Googles *GoogleSearchService*, der kan bruges til forskellige web-søgningsrelaterede handlinger. Den samme *web service* kan imidlertid også anvendes af andre applikationer end hjemmesider. Fx kan Googles søge *web service* integreres in en .NET Windows Forms applikation. Denne fleksibilitet er en af de store fordele ved *web services*, der forstærkes yderligere af, at *web services* kan oprettes i et hvilket som helst programmeringssprog og den applikation, der tilgår *web servicen* kan ligeledes være skrevet i et hvilket som helst sprog uafhængigt af, hvad *web servicen* er skrevet i. Denne store fleksibilitet skyldes, at SOAP-protokollen og WSDL-sproget begge er definerede i XML. *Web services* såsom Googles er gjort tilgængelige ved hjælp af en central opslagsfacilitet kaldet *Universal Description, Discovery and Integration* (UDDI).

2.6.1 WSDL

WSDL er et metadata sprog, der bruges til at beskrive *web services* overfor omverdenen. Til enhver *web service* er der knyttet et WSDL-dokument, der beskriver denne. Beskrivelsen består af, hvor *web servicen* findes, og hvilke metoder den offentliggør. WSDL-dokumentet er skrevet i XML og indeholder en række definitioner, der beskriver *web servicen*. De overordnede elementer er følgende:

- `<portType>`
- `<message>`
- `<types>`

[15] I *web service* terminologi kaldes metoder *operations*, men undtagen, når der specifikt henvises til kode, hvor ordet *operation* indgår, vil metode blive anvendt.

- <binding>

<portType> kan sammenlignes med en klasse i et objektorienteret programmeringssprog. Den bruges til at definere de metoder, en *web service* offentliggør, herunder de beskeder, der anvendes i den forbindelse, samt hvor vidt disse er input eller output beskeder. Dette foregår ved, at man under <portType> elementet definerer en eller flere <operation> elementer, der repræsenterer metoderne. Hvert <operation> element har et navn, som svarer til metodenavnet. Under hver operation er så defineret de forskellige input og output messages, svarende til en metodes input og output parametre. Definitioner under <portType> udgør til sammen det, der i forbindelse med *web services* kaldes en port.
<message> bruges til at definere en metodes dataelementer. Det er disse dataelementer, der blev defineret under <operation> som værende input eller output messages. Under <message> bliver de definerede første gang og deres datatype bliver angivet.
<types> definerer de datatyper en *web service* anvender.
<binding> definerer beskedformat og protokol for hver port. Den definerede protokol er ofte SOAP-protokollen, der gennemgås i afsnit 2.6.2.
Følgende er et eksempel på et udpluk af et WSDL-dokument. Det er udarbejdet med inspiration fra et eksempel fra W3Schools Online Web Tutorials [11].

```
<message name="InputMessage">
   <part name="InputParameter" type="xs:string"/>
</message>

<message name="OutputMessage">
   <part name="OutputParameter" type="xs:string"/>
</message>
<portType name="InputOutput">
  <operation name="InputOutputMethod">
      <input message=" InputMessage "/>
      <output message=" OutputMessage "/>
  </operation>
</portType>

<binding type="InputOutput" name="b1">
<soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"
/>
  <operation>
    <soap:operation
     soapAction="http://example.com/getTerm"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
```

Liste 1 - WSDL eksempel

Eksemplet i Liste 1 definerer en port ved navn InputOuput. Denne port har en enkelt tilhørende metode, InputOutputMethod, som er sat op til at modtage en string (InputParameter) som intputparameter og returnere en anden string

(`OutputParameter`), som er dennes outputparameter. Dette kan sammenlignes med en C#-klasse ved navn `InputOuput` som har en metode – `InputOutputMethod` – der tager en `string` som input og returnerer en anden `string`. Til sidst angives i eksemplet en binding til en protokol. Der er i dette tilfælde tale om en binding, hvor `InputOutput`-porten bindes til SOAP-protokollen.

Der indgår andre elementer i en WSDL-beskrivelse. For at få en komplet oversigt over WSDL-syntaks, kan henvises til W3Schools [11][16].

2.6.2 SOAP

SOAP er en protokol, der tillader kommunikation mellem applikationer over *HyperText Transfer Protoco*l (HTTP). SOAP-beskeder er ligesom WSDL-dokumenter skrevet i XML. SOAP bruges til at sende beskeder mellem en *web service* og den applikation, der kalder denne. SOAP-protokollen består af tre dele. *Envelope*-delen definerer hvad en SOAP-besked indeholder, *encoding*-delen definerer datatyper til brug ved *serialization* og *Remote Procedure Call* (RPC)-delen anvendes til at definere, hvilke metoder, der kan kaldes og deres *return*-værdier. SOAP-protokollen er ganske omfangsrig, og en dybdegående diskussion af den ligger uden for rammerne af denne opgave. For yderligere information vedrørende SOAP henvises til World Wide Web Consortium [12][17], der specificerer SOAP-envelopens XML-schema og World Wide Web Consortium [12][18], der gennemgår SOAP-protokollen.

Envelope-delen definerer som sagt beskeden som værende en SOAP-besked. Denne del fungerer som rodelementet i XML-dokumentet og omkranser således al den anden tekst. Under dette element ligger tre *child*-elementer: *Header*, *body* og *fault*.

- *Header*-elementet er et ikke obligatorisk element. Hvis det findes, skal det ligge lige under rodelementet, og indeholder applikationsspecifik information.
- *Body*-elementet er obligatorisk, og det er dette element, der indeholder selve beskeden. Her fremgår metoderne, som *web servicen* offentliggør samt hvilke input- og output-parametre, den indeholder.
- *Fault*-elementet er ikke obligatorisk og indeholder eventuelle fejlmeddelelser.

2.6.3 UDDI

UDDI fungerer som et bibliotek, hvor virksomheder kan registrere *web services* og hvor det ligeledes er muligt at lede efter *web services*. UDDI er ligesom WSDL og SOAP baseret på XML og platformuafhængig.

2.6.4 Udvikling af *web services*

Udviklingen af selve *web servicen* har mange fællestræk med udvikling af en almindelig softwareklasse, der er blot en række ting, der skal være specificerede. I forbindelse med dette projekt anvendes, som nævnt, *Visual Studio*, og dette automatiserer en del af arbejdet ved at udvikle *web services*. Som udvikler er man blot nødt til at specificere i koden til den relevante klasse, at der er tale om en *web service* og ud for de metoder, man ønsker at offentliggøre over netværket, skal man specificere, at disse er såkaldte *web mehtods*. Når *web servicen* herefter kompileres,

[16] http://www.w3schools.com/wsdl/wsdl_syntax.asp
[17] http://www.w3.org/2001/12/soap-envelope
[18] http://www.w3.org/TR/2000/NOTE-SOAP20000508

genererer *Visual Studio* den nødvendige WSDL og SOAP XML-kode. I C# angiver man, at en klasse skal være en *web service* ved at anvende `[WebService]` og `[WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]` atributterne før navnet på den klasse, man ønsker at oprette som en *web service*. Herudover skal klassen nedarve fra `System.Web.Services.WebService`-klassen. Inde i selve klassens kode anvender man `[WebMethod]` attributten før de metoder, man ønsker skal være tilgængelige over netværket. Følgende liste giver et eksempel, hvor der oprettes en *web service* med en enkelt `WebMethod`.

```
using System;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;

[WebService(Namespace = "http://tempuri.org/")]
[WebServiceBinding(ConformsTo =
WsiProfiles.BasicProfile1_1)]
public class Service :
System.Web.Services.WebService
{
    public Service () {

        //Uncomment the following line if using
        //designed components
        //InitializeComponent();
    }

    [WebMethod]
    public string HelloWorld() {
        return "Hello World";
    }

}
```

Liste 2 - *Web service* eksempel

Som det fremgår, af Liste 2, er der tale om en klasse ved navn `Service`, som inkluderer `System.Web`, `System.Web.Services` og `System.Web.Services.protocols` *namespaces*. Dette gøres for at kunne erklære klassen som en *web service* og erklære en tilhørende `webmethod`. Programmet er det standardprogram, *Visual Studio* opretter, når man vælger at oprette en ny *web service*. Når man afvikler `HelloWorld()`-metoden, får man følgende output:

```
<?xml version="1.0" encoding="utf-8" ?>
<string xmlns="http://tempuri.org/">Hello World</string>
```

Liste 3 - Resultat af HelloWorld()-kald

Som det fremgår af Liste 3, returnerer kaldet en XML-formateret tekst. Denne kan modtages af det system, der kalder metoden.

2.6.5 Kald af *web services*

En udvikler, der anvender en *web service* vil kunne betragte de metoder, *web servicen* offentliggør, som metoder, der stammer fra et hvilket som helst klassebibliotek. Der skal blot defineres en reference til den URL, hvorpå *web servicen* findes, hvorefter metoderne er tilgængelige via denne reference.

Data mellem en *web service* og den kaldende klasse overføres vha. XML, da det er standarden for kommunikation med *web services*. Håndteringen af dette foregår automatisk, da *Visual Studio* sørger for at genere den nødvendige kode til dette. Dette gør sig gældende både for *web servicens* vedkommende samt for den klasse, der tilgår *web servicen*s vedkommende.

2.7 XML

XML er ligesom HTML et såkaldt *Markup Language*. Det bruges til at beskrive data med og kan også indeholde data (ligesom en database). Dets primære anvendelsesområde er deling af data over netværk. Dette gøres ved, at der defineres *tags*, der beskriver data, hvorefter fx *HTML* kan anvendes til at vise data.

Konvertering af data fra fx et databaseformat til XML giver den store fordel, at det herefter kan sendes til et andet system, der anvender et andet databaseformat. På modtagersiden kan den modtagne XML-data herefter konverteres til det lokale databaseformat. Det giver mulighed for overførsel af data mellem systemer, der ikke er kompatible. For en mere uddybende omtale af XML henvises til W3Shools [11][19].

2.8 ADO .NET

ADO .NET er *Microsoft*s relationelle database model til .NET-baserede applikationer. ADO står for ActiveX Data Objects og bruges til at tilgå datakilder, der har defineret en såkaldt .NET *Provider* eller en .NET *Bridge Provider*.

2.8.1 Indledning

I *.NET 2.0 Framework*, findes ADO.NET 2.0 som muliggør tilgang til relationelle databaser, XML filer m.m. ADO.NET opererer under .NET's *managed enviroment*, hvilket medfører et udvalg af forskellige modeller kan anvendes til databasehåndtering. Disse beskrives i afsnit 2.8.2 og 2.8.3.

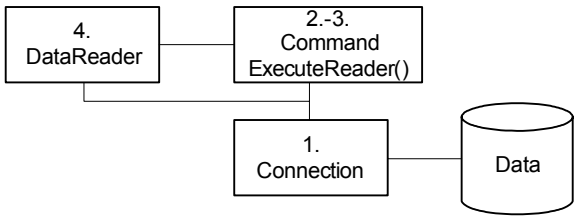
ADO.NET understøtter datatilgang vha. to modeller; *Connected* og *Disconnected*. En beskrivelse af de to modeller, samt forskellene, vil her blive præsenteret, hvor der tages udgangspunkt i .NET-terminologi, når der henvises til datatyper og lignende.

2.8.2 Connected

Den traditionelle fremgangsmåde at tilgå databaser på, er vha. den såkaldte *Connected*-model, hvor der er en åben forbindelse mellem applikationens `DataReader` og databasen under hele forløbet. De læste data bliver gemt i et `ResultSet` én række af gangen, og kan efterfølgende aflæses i applikationen. En `Connection` indeholder de informationer som er nødvendige for, at oprette en forbindelse til en konkret database. Herefter skal der oprettes et såkaldt `Command`-objekt, der kan udføre en operation på den aktuelle database. Dette foregår ved at anvende nogle underliggende metoder fra `Command`-objektet, samt sætte en række parametre. Derefter kan en `DataReader` benyttes til at læse resultatet fra en

[19] http://www.w3schools.com/xml/xml_whatis.asp

forespørgsel. Figur 3, der er lavet med inspiration fra Core C# and .NET [3][20], viser den sammenhæng der eksisterer mellem de forskellige komponenter som en kommunikation, mellem applikationen og databasen, indeholder.



Figur 3 - DataReader i ADO.NET's connected model

Som Figur 3 illustrerer, indebærer det typisk 4 trin at arbejde med en `DataReader`:

1. Opret et `ConnectionString`-objekt
2. Opret en SQL-streng med den relevante kommando.
3. Opret et `Command`-objekt med de dertil hørende `ConnectionString` og SQL-Query.
4. Opret en `DataReader`, og eksekver `Command`-objektet's `ExecuteReader` metode på denne.
    a. Her tilgås den konkrete database, og SQL-forespørgslen eksekveres.

Herefter kan `DataReaderen` gennemløbes, og de resulterende data kan gemmes i lokalvariabler, udskrives i systemet eller bindes til forskellige databeholdere (`Controls`). Det er vigtigt at bemærke, at det ikke er muligt at opdaterer eller tilføje data til en database vha. en `DataReader`, udelukkende aflæsning. Hvis en ændring af databasen ønskes (fx en opdatering vha. et SQL `Update` statement), skal en af `Command`-objektets andre eksekveringsmetoder benyttes. Fx ved afvikling af *stored procedures* er det `ExecuteNonQuery` der skal bruges, idet SQL-koden i dette tilfælde ligger på databasen, og altså ikke håndteres som en normal `DataReader`.

Det er forbindelse med ovenstående gennemgang vigtigt at bemærke, at der under hele afviklingen af databasebehandlingen er en åben forbindelse til databasen, og denne først afsluttes når forbindelsen aktivt lukkes af applikationen.

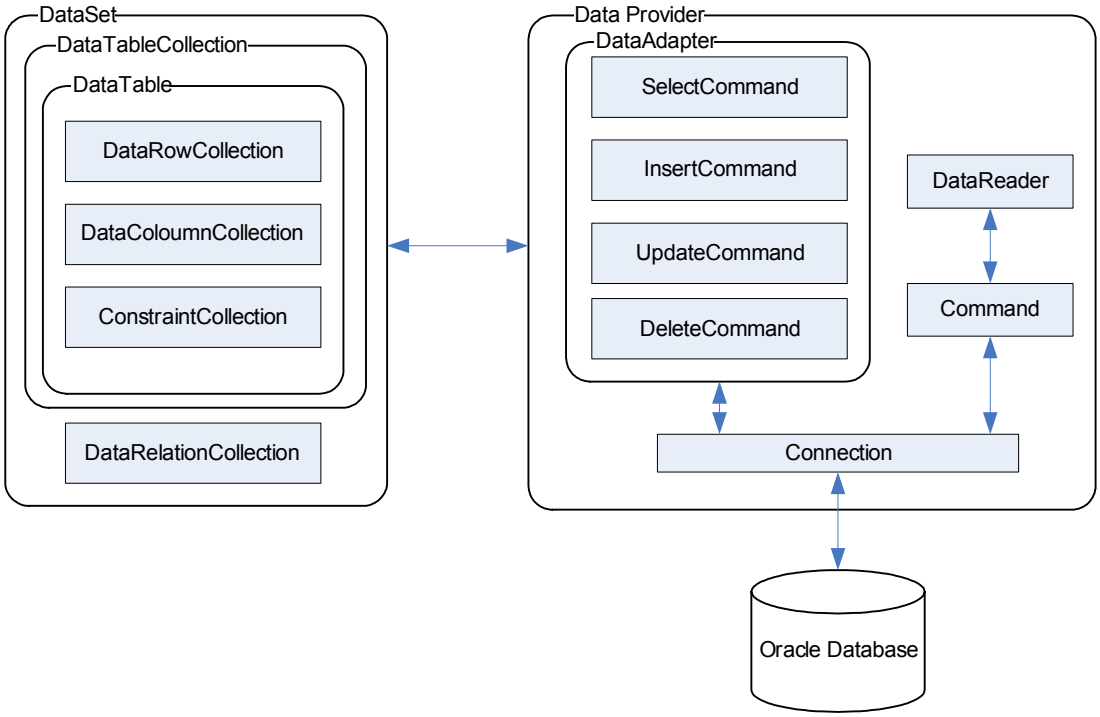
Fordelen ved *Connected*-modellen er, at den er hukommelsesbesparende, modsat *Disconnected*-modellen (se afsnit 2.8.3). Dog er `DataReaderen` begrænset til *Read-Only* databehandling, og hvis der er behov for dataredigering skal `Command`-objektet anvendes på anden måde.

2.8.3 Disconnected

En betydelig udvikling fra den oprindelige ADO model til ADO.NET, er begrebet *Disconnected*-databasetilgang. Denne model indebærer kort sagt, at den forespurgte data bliver overført fra den eksisterende database til en hukommelsesbaseret relationel database på den lokale computer, og at der kun er en åben forbindelse til databasen i det tidsrum, hvor dataet bliver overført fra databasen til hukommelsen. Når dataet er indlæst i hukommelsen bliver databaseforbindelse lukket, og selve håndteringen af resultatet sker herefter i den lokale maskines hukommelse. Dette er grunden til at der er tale om en afbrudt (*Disconnected*) model. Det at der anvendes en lokal kopi af databasen, betyder at der er mindre kommunikation med den faktiske database.

[20] Side 503 figur 11-4

Der eksisterer to enheder i arkitekturen for *ADO.NET*, et `DataSet` og en `Data Provider`, hvor `DataSet`-delen skal opfattes som en *Disconnected* del, og `Data Provideren` som en *Connected*. Figur 4, som er lavet med udgangspunkt i ADO.NET 2.0 Applications – Advanced Topics [13][21], illustrerer de grundlæggende opdelinger mellem ADO.NET-klasserne, og deres indbyrdes sammenhæng. Der er med *Disconnected* ADO.NET altså tale om et to niveaus hierarki hvori såvel klasser af typen *Disconnected som Connected* indgår.



Figur 4 - ADO.NET

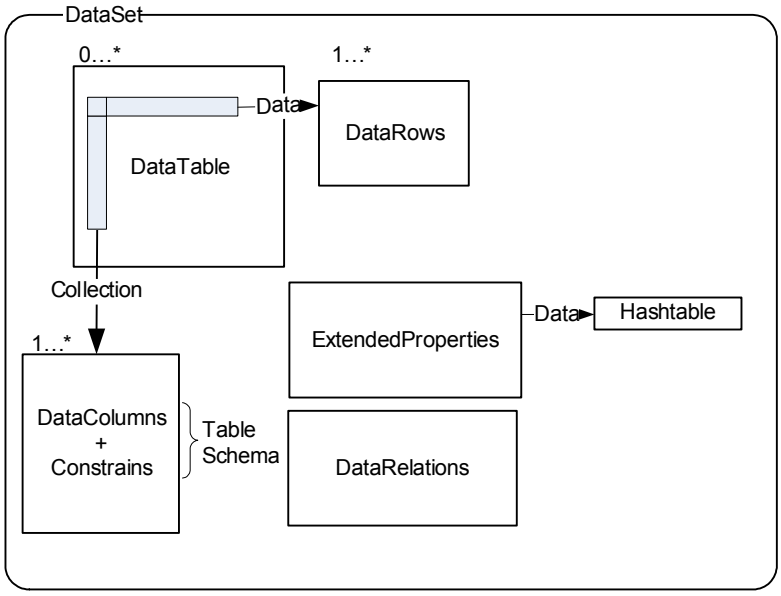
Et `DataSet` er en lokal instans af databasen; eller nærmere betegnet den del af databasen, der er relevant for applikationen. Dvs. de informationer der indgår i den relationelle database, forekommer ligeledes i den lokale instans. Som Figur 4 viser, indeholder et `Dataset` både tabeller, relationer osv. akkurat som en database.

Som standard oprettes `DataSet` som såkaldte *untyped* `DataSet`. For disse er der ikke taget stilling til, hvilket indhold de har i form af `DataTables` og dertilhørende `DataRows` og `DataColumns`, hvilket vil sige, at der ikke er oprettet et *schema* for dem. Dette betyder, at eventuelle fejl i forbindelse med tilskrivning af og læsning fra `DataSettet` først bliver opdaget ved *run time*. ADO.NET 2.0 giver imedlertid mulighed for at oprette *typed* `DataSet`. Dette gøres ved, at der til `DataSettet` knyttes en *XML Schema Definition* (XSD) fil, som specificerer dettes struktur. Ved anvendelse af *untyped* `DataSet` er det for at tilgå `DataTablen XXX` nødvendigt at skrive `DataSet.Tables["XXX"]`, hvorimod det ved et *typed* `DataSet` er muligt at tilgå denne som en property. Dette betyder, at der i stedet kan skrives `DataSet.XXX`. Herudover har *typed* `DataSet` den store fordel, at eventuelle fejl ved anvendelse af disse, bliver opdaget i forbindelse med kompilering. Endvidere er det muligt at foretage tjek på *typed* `DataSet`, inden deres værdier anvendes, således

[21] Side 1 figur 1-1

at man ikke risikerer at anvende en *null*-værdi. Dette gøres ved at foretage følgende boolske tjek: `DataSet.IsXXXNull()`.

`Data Provideren` er således den *Connected*-del, som foretager den egentlige kommunikation med databasen. Her forekommer nogle af de samme operationer som omtales i afsnit 2.8.2. `Data Providerens` væsentligste funktion er, at den er ansvarlig for at levere data til `DataSettet` fra databasen. Til det formål anvendes en `DataAdapter`, der kan opfattes som en bro, over hvilken der importeres såvel data som de skematiske oplysninger der kræves for at konstruerer det lokale `DataSet`.



Figur 5 - DataSet in ADO.NET

På Figur 5 ses den overordnede struktur af et `DataSet`. Det kan indeholde flere `DataTables`, som repræsenterer en tabel fra databasen. Hver `DataTable` indeholder `DataColumns`, der både består af selve kolonnen fra en tabel, og eventuelle tabelbetingelser som tilsammen udgør tabellens schema (`TableSchema`) der beskriver datastrukturen. `DataColumn` og `DataRows` udgør derved tilsammen lokaludgaven af tabellen. Ligeledes indeholder et `DataSet` egenskaber vedrørende informationer om `DataSettet` der er specifikke for applikationen (`ExtendedProperties`). Det kan fx være tidsstempler eller valideringskrav for tabellerne, der evt. kan lagres i en `Hashtable`. Relationerne i et `DataSet` muliggør associationer mellem `DataRows` i en `DataTable` og `DataRows` i en anden `DataTable` (såkaldte `DataRelations`). `DataRelations` definerer altså hvorledes de forskellige `DataTables` interagerer i `DataSettet`. Et `DataSet` skal således opfattes som en relationel database, der både repræsenterer data og tabelinformation, som applikationen kan bruge. Således kan en `DataTable` indeholde *Primary* og *Foreign Keys*, ligesom det er tilfældet for en database. Definitionen af sådanne *Keys* er et eksempel på en `DataRelation`.

Førend et `DataSet` kan have en reel anvendelse, er det nødvendigt at have kontrol over to informationer; versionen og tilstanden af den data, der findes i `DataSettet`.

En DataRow kan gennemgå en række tilstande i løbet af dens levetid i DataSettet. For overhoved at kunne sammenligne disse forskellige tilstande af dataet, må der nødvendigvis altid være en tilstand knyttet til det data som er indeholdt i det lokale DataSet. Dette er indeholdt i *enumeratoren* DataRowState, og kan tilgås gennem egenskaben RowState på en given række i en DataTable. Der findes fem tilstande i DataRowState: Added, Deleted, Detached, Modified og Unchanged. En DataRow har altid én af disse tilstande knyttet til sig.[13][22]

Det er vigtigt at have styr på om versionen af den DataRow der arbejdes med er tidssvarende, eller om der er kommet en nyere version i databasen. Det modsatte forhold kan også opstå, hvor det er et lokalt DataSet som har en nyere version af en DataRow, end den som forekommer i databasen. Til det formål er et DataSet konstrueret til at indeholde flere versioner af hver enkelt DataRow. Der kan eksistere optil tre versioner: Original, Current og Proposed.

I DataProvideren findes som tidligere nævnt en DataAdapter, der anvendes som en bro mellem den lokale kopi og databasen. For at hente data mellem databasen og DataSettet, kan DataAdapterens SelectCommand anvendes. SelectCommand skal indeholde et gyldig Command-object, samt en ConnectionString. Herefter afvikles SelectCommand's property ExecuteReader, der bruges til at fylde et DataTable-objekt med den relevante data. Dette er helt analogt med fremgangsmåden der er beskrevet under *Connected* datatilgang.

En udpræget fordel ved at arbejde *disconnected* og dermed anvende DataSet er udover muligheden for *strong typing*, at disse er yderst velegnede i forbindelse med *data binding* til forskellige Controls. For ydderlige diskussion af *data binding* se [3][23].

2.9 Rational Rose

Rational Rose er et software modelleringsværktøj. I forbindelse med dette projekt anvendes Rose til at producere use-case diagram, domænemodel, klassediagrammer, sekvensdiagrammer og interaktionsdiagrammer.

2.10 PL/SQL sproget

PL/SQL er proceduralt sprog udviklet af Oracle til brug ved afvikling af *stored procedures* på Oracle databaser. I forbindelse med dette projekt skal der afvikles en række *stored procedures*. Disse skal udvikles i det tilhørende PL/SQL developer program.

2.11 PL/SQL Developer

PL/SQL Developer er et *Database Management System* (DBMS) program udviklet af Oracle. Det anvendes til at håndtere Oracle databaser. Herunder kan det bruges til at oprette tabeller med dertilhørende kolonner, rækker og relationer tabellerne imellem samt til at oprette *stored procedures*. Når en *stored procedure* er oprettet, er det muligt at kompilere denne, og det kan derved opdages, om der er opstået fejl i definitionen af denne. Herudover er det muligt at debugge *stored procedures*.

[22] Side 1-10.
[23] Kapitel 12.

Det anvendes endvidere til at organisere tabeller og *stored procedures* logisk. I forbindelse med etablering af nye tabeller og tilføjelse af kolonner og rækker til eksisterende, sørger PL/SQL Developer for at generere den nødvendige SQL hertil, således at brugeren ikke behøver at tage stilling til dette.

3 Projektstyring

For at sikre, at projektet ikke løber af sporet, vil der løbende blive foretaget projektstyring. Dette vil foregå i to dimensioner. Den ene dimension består af, at der oprettes et projekt i *Microsoft Project*, hvor hele projektforløbet er opridset med de forskellige discipliner, der indgår. Den anden dimension består af, at der hver uge oprettes en ugeplan, hvor det for hver arbejdsdag indføres, hvad der konkret skal arbejdes med den pågældende dag samt en liste over hvilke mål, der skal være indfriet ved ugens afslutning.

3.1 *Microsoft Project*

Ved starten af projektforløbet, skal en projektplan oprettet i *Microsoft Project*. Gennem hele forløbet skal denne løbende opdateres, således, at den overordnet viser, hvilke faser og dertilhørende discipliner, der skal påbegyndes hvornår. Denne projektplan skal bruges som en grovskitse, hvor der ikke foretages detailstyring af, hvornår enkeltdiscipliner i en fase skal afvikles i forhold til hinanden. På den måde bruges planen til at danne et overblik over, hvilke større projektdele, der skal afvikles og til at angive en overordnet sekvens i hvilke ting, der skal gøres før end andre kan påbegyndes. Til styring af den detaljerede planlægning, skal der anvendes ugeplaner, hvor det dag for dag planlægges, hvilke ting, der skal gennemføres. Den endelige projektplan ses af Figur 6 på næste side. Som det kan ses, er projektplanen vist som et Gantt diagram. Det bør bemærkes, at der ikke er specificeret ressourcer og hvor langt de enkelte discipliner er fra at være gennemførte, da diagrammet som nævnt kun har været brugt til at danne et overblik og se hvornår opgaver skal begyndes og afsluttes.

3.2 Ugeplaner

Som nævnt anvendes ugeplaner til detailstyringen af opgaverne i projektafviklingen. Disse ugeplaner, består af felter til brug ved dag- til dagplanlægning samt en overordnet agenda for hele ugen. Derudover kan også angives punkter, der følger af planen for den enkelte uge, men først skal ses på ugen efter. En noget nedfotograferet version af en sådan ugeplan ses af Tabel 2.

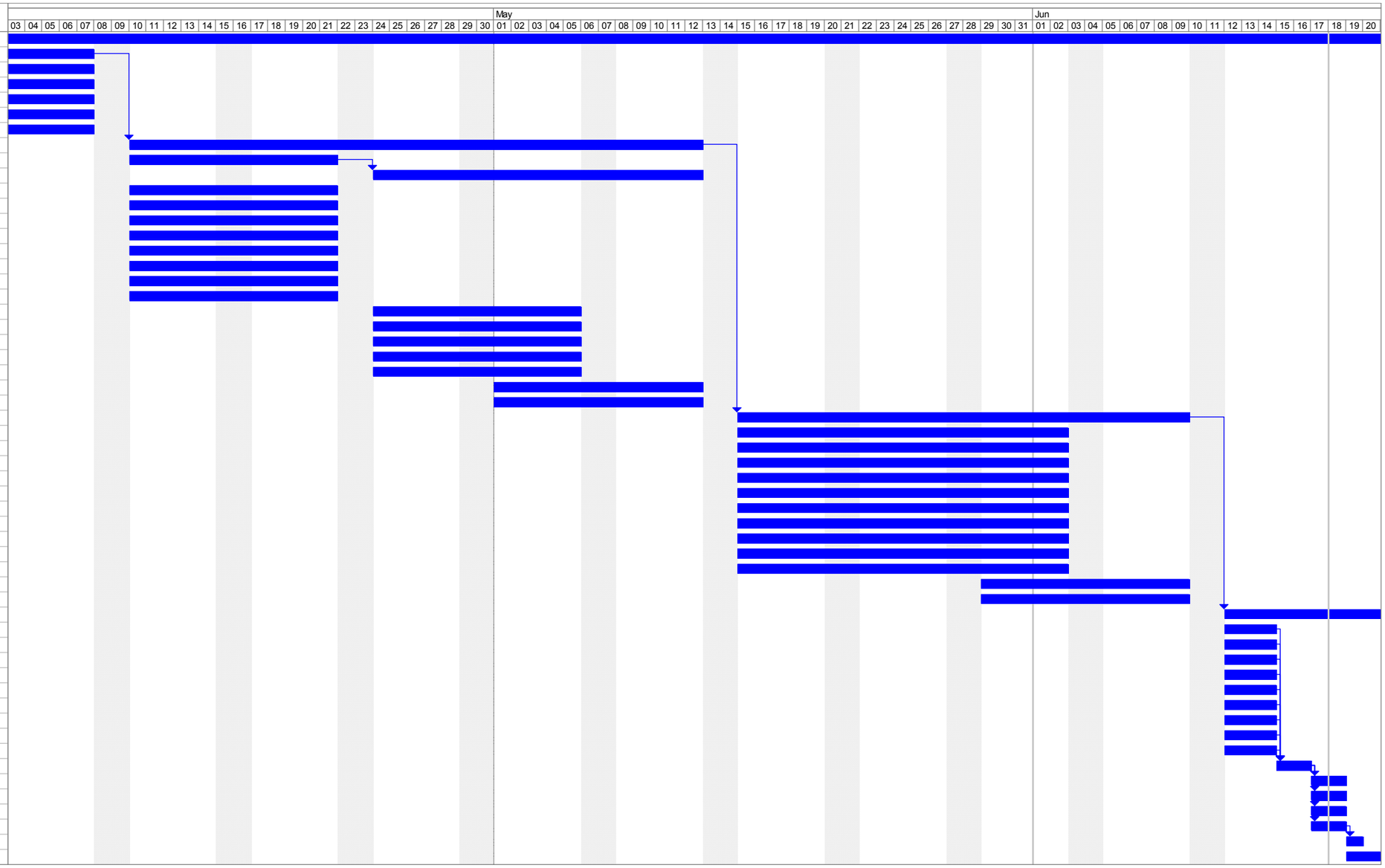
Uge:							
Tid	**Mandag**	**Tirsdag**	**Onsdag**	**Torsdag**	**Fredag**	**Weekend**	**Agenda**
8-12							
13-17							**Næste uge**

Tabel 2 – Ugeplan

3.3 Dokumentstyring

Til dokumentstyring er *Campusnet* anvendt til håndtering af de tekstdokumenter og figurer af forskellig art, der er blevet oprettet i forbindelse dokumentation af projektet. Til håndtering af kildekoden er *Visual Source Safe* blevet anvendt.

ID	Task Name
1	Eksamensprojekt
2	Inception
3	Identifikation af stakeholders
4	Udviklingsplan
5	Risikoanalyse
6	Business case
7	Modellering af use-cases
8	Elaboration
9	1. iteration
10	2. iteration
11	Domænemodel
12	Use-cases
13	Designmodel
14	SW arkitektur dokument
15	Datamodel
16	Testmodel
17	Implementeringsmodel
18	UI skitse
19	Implementér prototype til forespørgsel
20	Implementér simpel GUI
21	Implementér skal til WSDataService
22	Opret GUI-tabeller i LMES+
23	Implementér simpel DataAcces m. SELECT statement
24	Opdatér kravspecifikation
25	Opdatér design
26	Construction
27	Implementér POARepresentation
28	Implementér POAGUI
29	Implementér MainMenuForm
30	Implementér Show InfoForm
31	Implementér POADataBusinessLogic
32	Implementér WSDataService
33	Implementér POADataAccess
34	Implementér DataStorage
35	Færdiggør GUI-tabeller med alle controls
36	Implementér stored procedures
37	Opdatér kravspecifikation
38	Opdatér design
39	Transition
40	Deploy test version på PDA
41	Test trigger
42	Test stored procedures
43	Test database kommunikation
44	Test generiske database komponenter
45	Test eksekvering af ServiceRequest
46	Test eksekvering af ServiceGUIRequest
47	Test MainMenuForm
48	Test dynamisk GUI
49	Foretag eventuelle rettelser og gentag test
50	Acceptance test af use-case #1
51	Acceptance test af use-case #2
52	Acceptance test af use-case #4
53	Foretag eventuelle rettelser og gentag test
54	Deploy endelig version på PDA
55	Afslut dokumentation



Figur 6 - Projektplan

4 LMES+

I dette kapitel vil det eksisterende LMES+ system blive gennemgået. Ligeledes vil en kortere beskrivelse af principperne bag LMES+ fremgå. Den del af dette, der udgør datagrundlaget for POA-applikationen vil også blive beskrevet i dette afsnit, herunder de relevante elementer i LMES+, samt databasebeskrivelse og tilhørende ER-diagrammer.

4.1 Indledning

LMES+ er en NNE specifik realisering af et *Management Exectuion System* (MES). Et MES bruges til at automatisere produktionen på en fabrik. MES kan betragtes som et bindeled mellem et overordnet forretningssystem (ERP) og processkontrolsystemer (PCS) i fabrikken. Her kan bl.a. ordrer og angivelser af hvilke materialer, som skal anvendes, sendes ned i hierarkiet fra ERP-systemet. Disse modtages på MES-laget, hvor de enkelte ordrer opdeles i underordrer og håndteres. MES uddelegerer således opgaver til PCS-systemerne, som styrer og regulerer de valgte procesudstyr. MES monitorerer procesudstyret, og indeholder produktskedulering, produktafvikling, datahistorik m.m.

En fabriks produktion inddeles i batches, hvor en batch er resultatet af en batch-proces. En batch-proces består af at bruge udstyr, mandetimer og ingredienser til at skabe et slutprodukt – slutproduktet er det, der kaldes batchen. Ved batchproduktion er et produceret emne knyttet til en konkret batch, hvilket medfører en høj grad af sporbarhed.

Formålet med et MES er at automatisere batch-processen. Dette gøres ved anvendelse af recepter, som specificerer hvilket udstyr og hvilke ingredienser, der skal anvendes til denne. Til dette kan S88-standarden anvendes, som definerer modeller og terminologier til afvikling af batch produktion. S88 arbejder med to hierarkiske modeller; en fysisk og en procedural model. Den fysiske model beskriver det samlede udstyrshierarki i fabrikken, og den proceduralle model indeholder beskrivelser af procedurer, operationer og faser. Ved anvendelse af S88-koncepter til design af et MES system, opnås et meget modulært system, da alt udstyr således er bygget op i enheder med tilknyttet kontrol. Dette har flere fordele, da gennemtestede moduler kan genbruges og konstruktionen af en fabrik bliver hurtigere, og med større sandsynlighed har fungerende enheder der kan kommunikerer indbyrdes. Disse to modeller samt tilstandskontrol og recepter er essensen i et MES system; for yderligere beskrivelse henvises til S88 [19]. I de følgende afsnit gennemgås den del af LMES+'s datagrundlag, der har relevans for denne opgave og derefter præsenteres et ER-diagram for denne.

4.1.1 Datagrundlag

LMES+ består af adskillige undermoduler. De dele af LMES+, som er centrale for dette projekt er *Batch Execution System* (BES), EMS og den underliggende Oracle database. I det følgende gives en kort introduktion til disse moduler.

BES er kernen i LMES+ systemet og står for at kontrollere og koordinere udførslen af en batch-produktion. BES bruges til at oprette batche, oprette kontrol recepter for planlagte batches, indsamle statusinformation fra en igangværende batch, allokere ressourcer, udføre manuelle procedurer, linieoprydning osv. [1]. BES er baseret på S88 koncepterne, hvilket betyder, at en kontrolrecept er opbygget af et hierarki af operationer, som igen består af mindre operationer.

EMS bruges til at definere udstyr og til at håndtere udstyrsstatus (rensning, kalibrering og vedligeholdelse) [2]. Det er herfra, at oplysninger om såvel udstyrsstatus og udstyrsoperationer hentes (se afsnit 4.1.1.1 og 0)

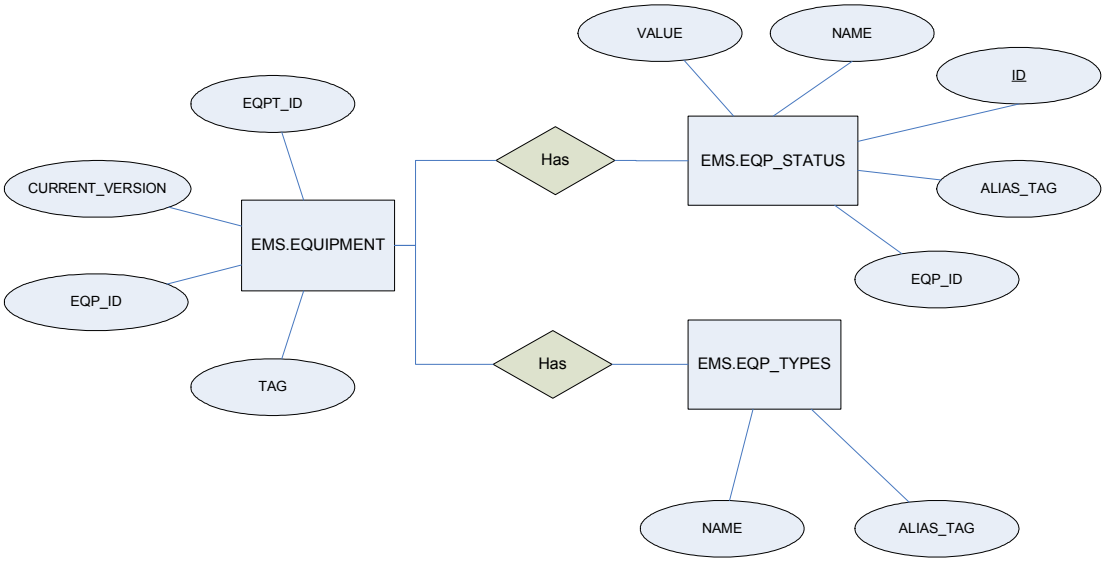
Alle oplysninger i LMES+ er lagret i en Oracle database, som applikationen skal kunne tilgå. Oracle databasen indeholder data for hele LMES+, herunder både BES og EMS. Denne information er tilgængelig gennem LMES+'s grafiske brugerflade og dele af denne information kan tilgås af fabriksoperatørerne. LMES+ opbevarer en række oplysninger om bl.a. fabriksudstyr.

Til udviklings- og testformål opretholder NNE en kopi af LMES+ databasen fra en af Novo Nordisk fabrikker. Det er denne NNE-kopi, der vil blive anvendt som datagrundlag under udvikling og test af applikationen.

4.1.1.1 Udstyrsstatus

Udstyrsstatus dækker over følgende Oplysninger: Varenummer, batchnummer, rengøringsstatus, interlock-status, udstyrstype og local/remote-status. Varenummer og batchnummer er unikke identificeringsnumre. Er der ikke en batch under afvikling i det pågældende udstyr, vil batchnummer værdien være NULL. Rengøringsstatus angiver, om udstyret er blevet vasket, og kan antage følgende værdier. Interlock angiver, hvorvidt en batch har låst udstyret eller ej og kan antage værdierne ”Yes” og ”No”. Dette bruges til at sikre, at udstyr kun har en enkelt batch under afvikling ad gangen. Local/remote angiver, om styringen af udstyret foretages lokalt eller ude fra og kan antage værdierne ”Local” og ”Remote”. Den normale indstilling er Remote, hvor udstyret styres af LMES+. I nogle tilfælde kan det dog være nødvendigt at fx en operatør har mulighed for at styre udstyret direkte, der hvor det står.

De informationer der skal bruges til at repræsentere udstyrsstatus, eksisterer i tre tabeller, som illustreres på ER-diagrammet på Figur 7.



Figur 7 - ER-diagram for udstyrsstatus

Som det fremgår af Figur 7, er det i *packagen* EMS, de tre tabeller eksisterer. De forskellige kolonner i tabellerne skal anvendes til *stored procedures*, og nogle af disse gennemgås i det følgende afsnit.

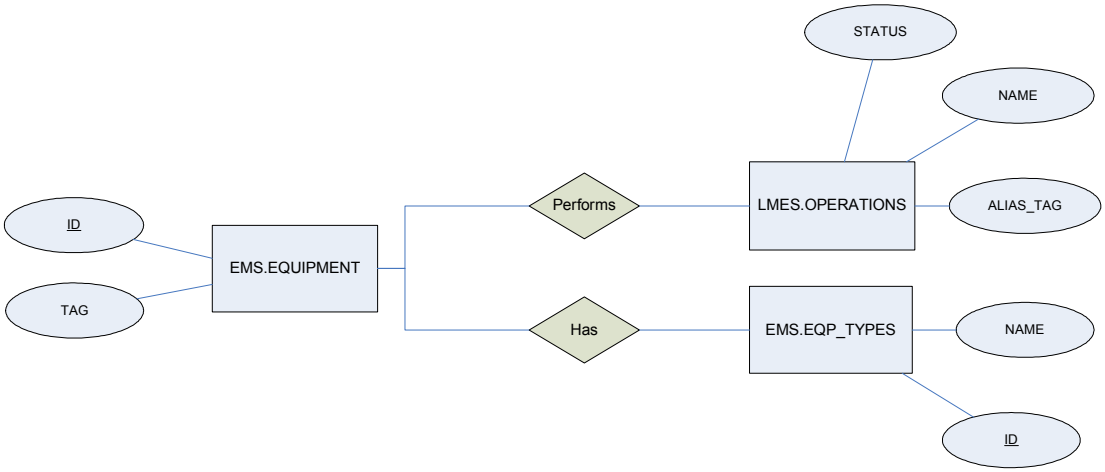
Et hvilken som helst udstyr, der findes i `EQUIPMENT`-tabellen, har således både en udstyrstype og en udstyrsstatus knyttet til sig. Udstyrstyper er indeholdt i `EQP_TYPES`-tabellen, hvor `NAME` er en læsevenlig udgave af navnet på udstyrets type (fx Buffer tank) og `ALIAS_TAG` er selve udstyrstypen (fx `BUFFER_TANK`). Udstyrsstatus findes i `EQP_STATUS`. Her er navnene på de forskellige oplysninger lagret i *attributten* `NAME`, hvor en given status på de enkelte oplysninger er lagret i `VALUE`. I `EQP_STATUS` er `ALIAS_TAG` selve udstyrsstatusen (fx `BATCH_NUMBER`). `EQP_STATUS.EQP_ID` anvendes i sammenhæng med `EQUIPMENT.TAG` til unikt at udvælge udstyr. `EQUIPMENT.CURRENT_VERSION`

bruges til at sikre at udstyret er aktivt. Det skal bemærkes at de *attributes* som er knyttet til tabellerne på ER-diagrammet kun er et udsnit af den totale data model, da der kun er fokus på de dele af databasen som anvendes i dette projekt.

4.1.1.2 Udstyrsoperationer

Udstyrsoperationer dækker over oplysninger om hvilke operationer, der er under afvikling i udstyret. En operation er en del af den batch, der er under afvikling. Er der fx tale om en rengøringsbatch på en tank, kan en operation fx være, at tanken påfyldes syre. En efterfølgende operation kan være, at tanken påfyldes vand for at skylle syren ud. Det er muligt at foretage flere samtidige operationer på udstyr, men der kan kun være en enkelt batch under afvikling ad gangen, jf. Interlock, der som nævnt angiver, om udstyret er låst af en batch eller ej.

De informationer der skal bruges til at repræsentere udstyrsoperationer, eksisterer i tre tabeller, som illustreres på ER-diagrammet på Figur 8.



Figur 8 - ER-diagram for udstyrsoperationer

Udstyrsoperationer, nærmere betegnet `OPERATIONS`-tabellen, findes i BES, der af historiske ligger i LMES *packagen* og ikke i en BES *package*. `OPERATIONS.NAME` er et læsevenligt navn på udstyrsoperationen, hvor `OPERATIONS.ALIAS_TAG` indeholder selve udstyrsoperationen. `OPERATIONS.STATUS` er *Running* for aktive operationer. `EQUIPMENT.TAG` og `EQP_TYPES`-tabellen anvendes ligesom beskrevet i afsnit 4.1.1.1.

5 Krav

Dette kapitel udgør projektets kravspecifikation. Herunder gennemgås visionen for projektet, hvor det motiveres, hvorfor projektet udarbejdes. Derudover bliver en række nøgleelementer såsom *use-cases*, specifikation af brugergrænseflade, målsætninger, systemafgrænsning, supplerende krav og risikohåndtering gennemgået.

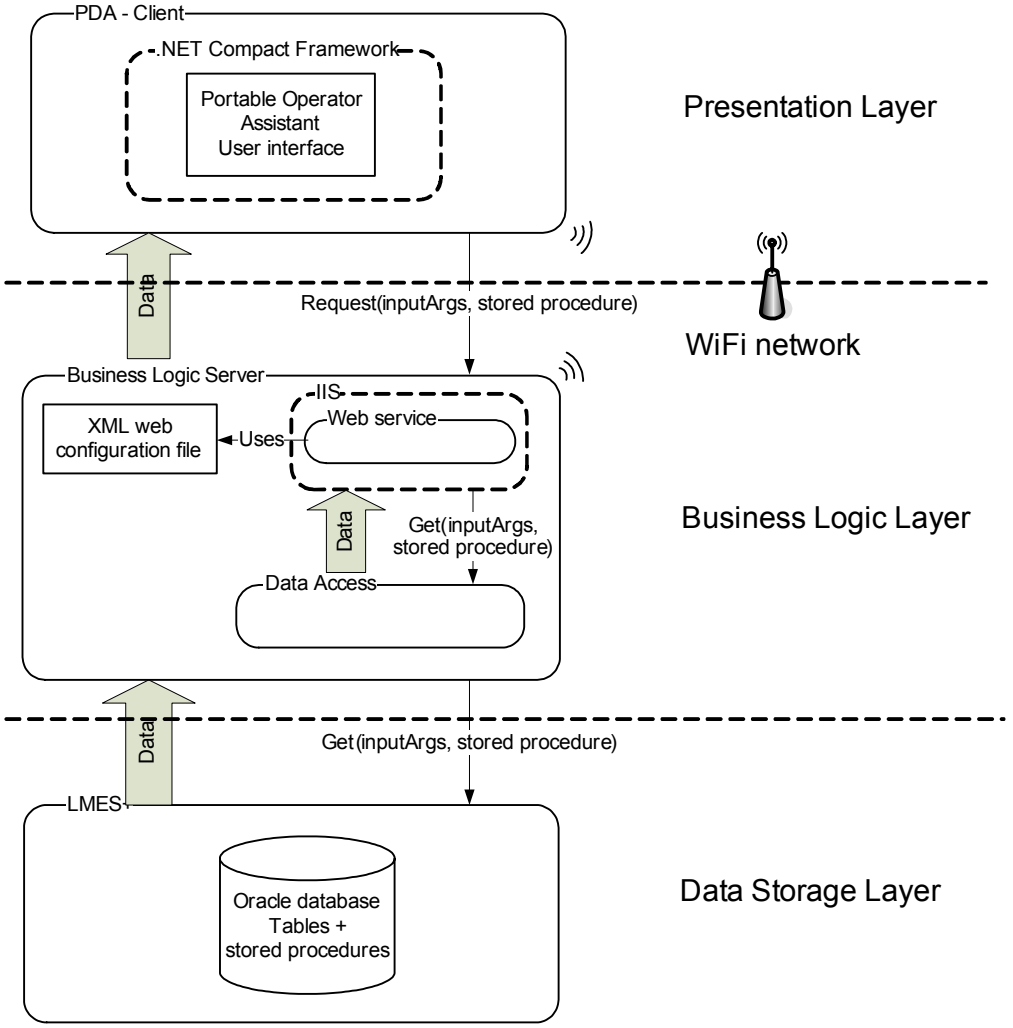
5.1 Vision

Dette projekt dækker over et ønske fra NNE's side om at undersøge, hvorvidt der er et behov for at udvide funktionaliteten af det eksisterende LMES+ produkt med en håndholdt applikation. Til det formål ønskes der udviklet et pilotprojekt på en sådan applikation for at finde ud af, hvorvidt NNE's slutbruger (Novo Nordisk) er interesseret i det nye produkt, samtidig med at NNE får kendskab til håndholdt teknologi, og hvor godt denne teknologi virker sammen med det eksisterende produkt. Det bør bemærkes, at projektet specificeres i samarbejde mellem projektforfatterne og NNE, og at Novo Nordisk først bliver inddraget efter at pilotprojektet er udviklet og derefter kan stille krav og komme med kommentarer. Derfor er NNE den umiddelbare slutbruger af den udviklede applikation, og når der fremover henvises til en slutbruger, er der tale om NNE. Applikationen får navnet *Portable Operator Assistant* (POA). Fra nu af når der bliver talt om applikationen eller systemet, er det POA, der er tale om.

I den nuværende implementering af LMES+ er der meget af det data, som fabriksoperatørerne anvender, der kun kan tilgås via stationære computere gennem en besværlig grafisk brugergrænseflade (GUI). De data, som operatørerne typisk har brug for har at gøre med status på fabrikkens udstyr samt status på afvikling af batche. I dette projekt arbejdes med data vedrørende udstyr. Det bør her bemærkes, at der for fabriksudstyr kan findes oplysninger vedrørende hvilke batche, der har været afviklet og er under afvikling. Der er derfor et vist overlap mellem oplysninger vedrørende udstyr og batche.

For at gøre datatilgangen mere brugervenlig overvejes det, at fabriksoperatørerne kan anvende en PDA, som de kan have på sig og bruge til at tilgå data med. PDA'en skal være udstyret med en stregkodeskanner, som skal bruges til at identificere fabriksudstyr og hente oplysninger om dette. Den hentede information kan omhandle adskillige aspekter ved udstyret, fx udstyrsstatus. Applikationen skal kunne kommunikere trådløst med LMES+ og skal indeholde en opdateret GUI, som er mere anvendelig og intuitiv end den nuværende.

Figur 9 giver et indblik i den overordnede arkitektur for applikationen. Denne figur diskuteres yderligere under afsnit 5.2 om systemafgrænsning.



Figur 9 - Applikationsarkitektur

For at gøre applikationen brugbar i forbindelse med slutbrugerens evaluering, skal vægten lægges på at få opbygget en applikation, som har den ønskede funktionalitet i form af at kunne hente data og præsentere dette frem for andre aspekter såsom sikkerhed, automatisk opdatering af applikationen og andet. Endvidere er der fra NNE's side lagt meget stor vægt på, at slutbrugeren skal have mulighed for at ændre på brugergrænsefladen uden at skulle ændre i selve applikationens kode.

Interessenterne i dette projekt udgøres af leverandøren af applikationen (NNE – specifikt Automation afdelingen), slutbrugeren (Novo Nordisk/NNE) såvel som projektforfatterne. For NNE's vedkommende er man, som det er blevet nævnt, interesseret i at kunne demonstrere et muligt fremtidigt produkt for Novo Nordisk samt at få kendskab til ny teknologi. Novo Nordisks interesse i applikationen er at vurdere, om den giver mulighed for at forbedre arbejdsgangene på deres fabrikker. For projektforfatternes vedkommende udgør projektet det afsluttende projekt på Diplom-IT studiet.

5.2 Systemafgrænsning

I dette projekt udgøres systemet af selve den applikation, der skal udvikles. Således defineres systemet som værende den software, der skal udvikles, hvorimod såvel den

underliggende hardware, eksisterende software samt fabriksoperatører defineres som værende udenfor systemets grænser.

Som det fremgår af Figur 9 skal applikationen bestå af en række lag. Det øverste lag kaldes *Presentation Layer* og skal afvikles på en PDA. *Presentation Layer* skal implementeres i *.NET Compact Framework 2.0*. Midterlaget hedder *Business Logic Layer* og skal afvikles på en *application server*. *Business Logic Layer* skal implementeres i *.NET Framework 2.0*. Kommunikationen mellem disse to lag består af en WiFi-forbindelse. Det nederste lag i applikationen hedder *Data Storage Layer*. Dette lag består af LMES+ og herunder en række tabeller og *stored procedures*, der anvendes af de øvre lag. Disse ligger alle på en *Database Server*, hvorpå der findes Oracle database.

5.3 Aktører og målsætninger

Til projektet er der knyttet en række aktører. Disse aktører er enten personer eller systemenheder. Hver aktør har forskellige interesser i forhold til projektet. Disse kaldes målsætninger. Tabel 3 og Tabel 4 viser en oversigt over, hvilke aktører, der er blevet identificeret, og hvilke målsætninger de har i forhold til udviklingen af systemet. Tabel 3 viser en oversigt over de primære aktører, mens Tabel 4 viser de sekundære aktører.

Aktør	Målsætning
Fabriksoperatør	Forespørg informationer Start systemet op Luk systemet ned
Vedligeholdelses system	Automatisk opdatering af software
System administrator	Sørg for at nyeste version af systemet er tilgængelig
Ledelsen	Tilgå historik
Slutbruger	Forbedrede arbejdsgange Forbedret fabriksinformation

Tabel 3 - Primære aktører

Aktør	Målsætning
LMES+	Levere informationer til systemet

Tabel 4 - Sekundære aktører

5.4 Use-cases

I dette afsnit defineres de funktionelle krav. De bliver præsenteret i form af *use-cases* og supplerende krav. Hver *use-case* repræsenterer et mål, og tilhørende aktør(er). Det betyder at de konkrete mål defineres i hver deres *use-case*, og supplerende krav, der er gældende for flere *use-cases*, for systemet generelt eller givne på forhånd, defineres overordnet i afsnit 5.7.

Der anvendes en punktopstillet *use-case* skabelon – baseret på Alistair Cockburns Basic Use Case Template [9] – hvor den samme struktur er gennemgående for alle *use-cases*. Betydningen af de enkelte punkter bliver beskrevet i Tabel 5.

Use-case #[nummer]	[Use-case navn]	
Formål	Formålet med den konkrete *use-case*. Dette skal repræsentere ét mål.	
Forudsætninger	Nødvendige betingelser for at succeskriteriet kan opfyldes.	
Succeskriterium	Systemets tilstand efter denne *use-case* er blevet indfriet.	
Fejlkriterium	Systemets tilstand hvis denne *use-case* ikke er blevet indfriet.	
Primær aktør	Den primære aktør skal indfri formålet i *use-casen*.	
Sekundær(e) aktør(er)	Den sekundære aktør leverer en service (evt. information) til systemet. Dette kan både være et computersystem eller en person.	
Udløser	Den begivenhed som udløser *use-casen.*	
Beskrivelse	**Trin**	**Hændelse**
	1	Trinvis beskrivelse af systemet i aktion
Udvidelse – alternativt flow	**Trin**	**Hændelse**
	1a	Beskriver yderligere/alternative udfald set i forhold til den direkte vej.
Forgrening	**Trin**	**Hændelse**
	1.1	Et operatørvalg, hvor flere muligheder bliver præsenteret.
Relaterede informationer		
Prioritet	Implementeringsprioritet – se Tabel 6.	
Ydelse	Et kvantitativt krav, som dikterer minimumsydelsen for den beskrevne funktion i *use-casen*.	
Hyppighed	Et kvantitativt krav, som estimerer hvor ofte funktionen, der beskrives i *use-casen*, vil blive anvendt.	
Udestående(r)	Emner der bevidst ikke er taget stilling til endnu, men som bør inkluderes i *use-casen*.	
Forfaldsdato	En deadline – enten konkret dato, eller iterationsnummer.	
Overordnede use-case(s)	Andre *use-cases* der bruger den aktuelle *use-case*.	
Underordnede use-case(s)	Andre *use-cases* der bliver brugt af den aktuelle *use-case*.	
Eventuelt	Yderligere information som ikke forefindes i de førnævnte punkter.	

Tabel 5 - Use-case skabelon

5.4.1 Identifikation af use-cases

De *use-cases*, som er blevet identificerede fremgår af Tabel 6. Som antydet under prioritet i Tabel 5, er alle *use-cases* opdelt efter prioritet. Dette er en afbildning af vigtigheden af de enkelte funktioner, der bliver præsenteret i de forskellige *use-cases*. Prioriteringen skal også afspejle, hvorvidt nogle *use-cases* er nødvendige at have implementeret før end andre kan implementeres, og på den måde bruges den ligeledes til, at angive rækkefølgen de forskellige *use-cases* skal designes og implementeres i. Prioriteten for realiseringen af de forskellige *use-cases* ses i Tabel 6.

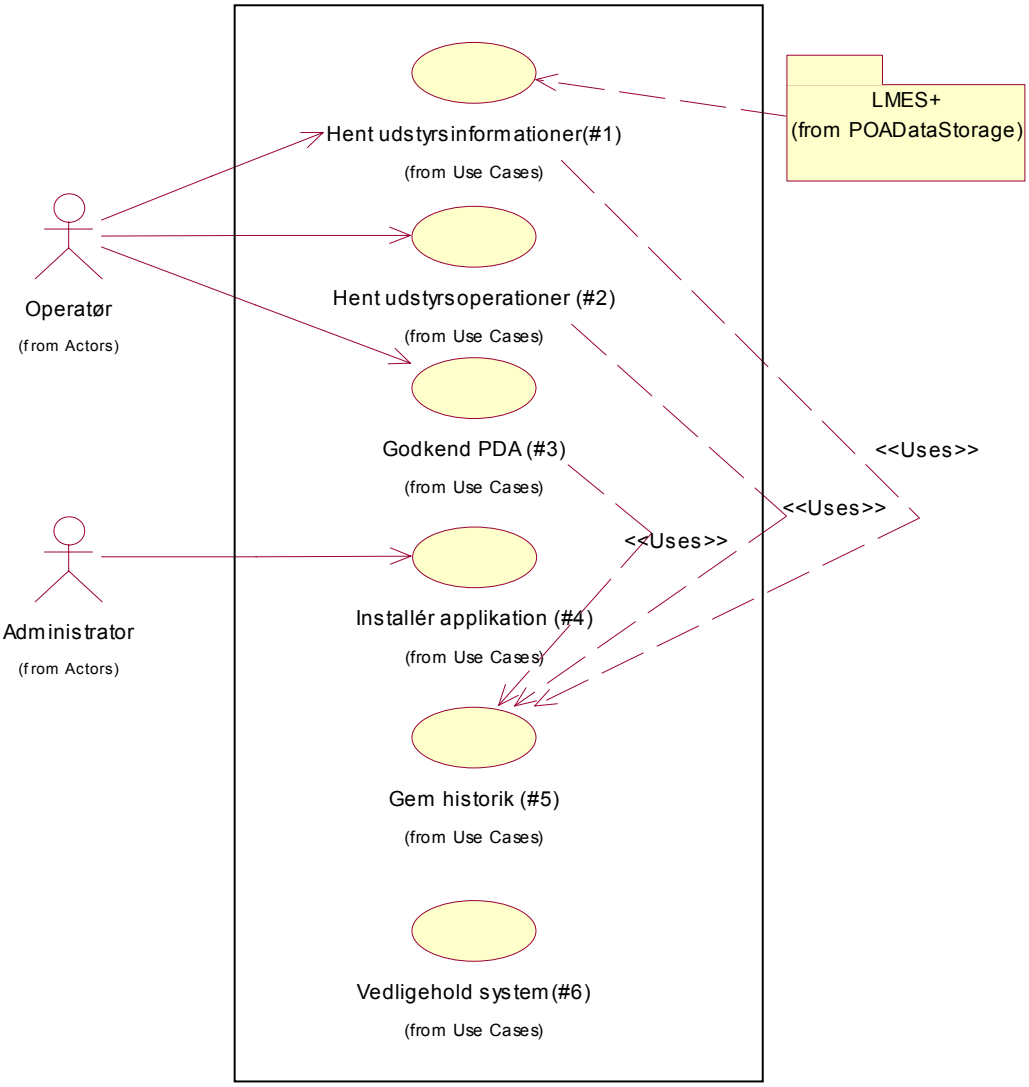
Prioritet	Krav Use-cases	Kommentarer
Høj	Hent udstyrsstatus	Denne *use-case* er helt central for applikationen. For at denne skal kunne realiseres, skal stort set alle grundlæggende aspekter af applikationen være implementerede.
Mellem	Hent udstyrsoperationer	Har store ligheder med "Hent udstyrsstatus", men har andre karakteristika for af hyppighed, ydelse og prioritet. Behandles derfor separat. Skal delvist implementeres i forbindelse med dette projekt.
Mellem	Installér applikation	Er ikke en nødvendighed for en tynd implementering af systemet. Skal dog være delvist funktionel i forbindelse med dette projekt.
Lav	Godkend PDA	Skal ikke implementeres i forbindelse med dette projekt, men skal betragtes som en nødvendig senere tilføjelse.
Lav	Vedligehold system	Er afledt af Installér applikation. Skal ikke implementeres i forbindelse med dette projekt, men skal betragtes som en nødvendig senere tilføjelse.
Lav	Gem historik	Skal ikke implementeres i forbindelse med dette projekt, men skal betragtes som en mulig senere tilføjelse.

Tabel 6 - Use-case prioritet

Som det fremgår af Tabel 6, er det ikke alle use-*cases*, der skal implementeres i forbindelse med dette projekt. Det skyldes, at fokus her er på funktionalitet. Endvidere har der fra NNE været nogle meget specifikke krav til opbyggelse af en dynamisk brugergrænseflade, hvilket vil blive diskuteret yderligere i afsnit 5.5. Selvom de *use-cases*, der ikke bliver implementeret vurderes at være mindre betydende for dette projekt, medtages de alligevel her, da de vurderes at være vigtige på længere sigt. De vil dog kun blive præsenteret som use-cases, og de vil ikke blive gennemgået i design- og implementeringsafsnittene.

5.4.2 Use-case-diagram

De *use-cases*, som fremgår af Tabel 6 repræsenterer systemet i sin helhed. Figur 10 viser en oversigt over de forskellige *use-cases*, aktørerne og disses indbyrdes relationer i form af et *use-case*-diagram.



Figur 10 - Use-case diagram

5.4.3 Gennemgang af use-cases

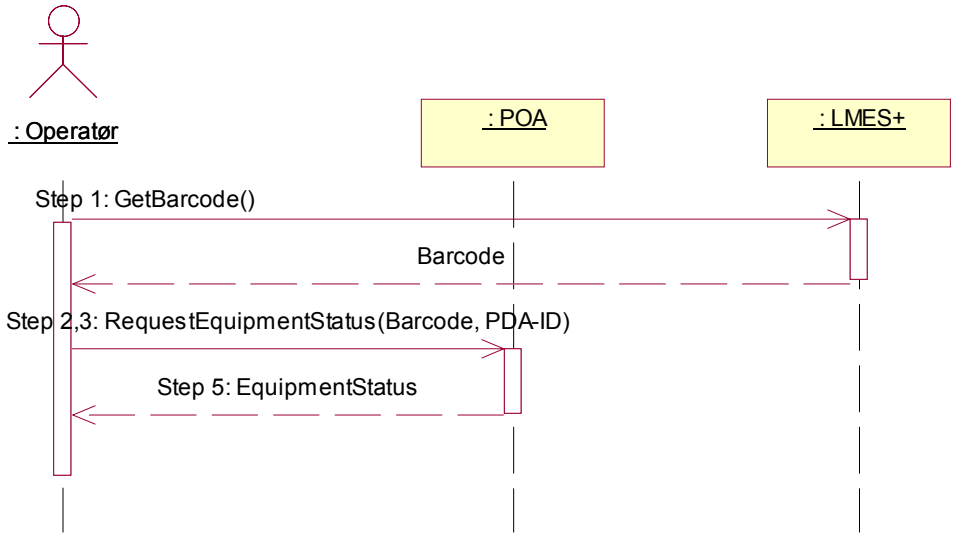
I dette afsnit gennemgås de *use-cases*, der fremgår af Figur 10. Først præsenteres de i det skema, der blev vist i Tabel 5, og hvor det er relevant vises et systemsekvensdiagram, der viser kommunikationen i forbindelse med *use-casen*.

5.4.3.1 Hent udstyrsstatus

Use-case #1	Hent udstyrsstatus	
Formål	Operatøren skal have præsenteret udstyrsstatus for det ønskede fabriksudstyr.	
Forudsætninger	1. Systemet er startet op.	
Succeskriterium	Operatøren får præsenteret udstyrsstatus korrekt.	
Fejlkriterium	Operatøren får slet ikke præsenteret den efterspurgte information, eller denne præsenteres forkert.	
Primær aktør	Primær aktør er operatøren.	
Sekundær(e) aktør(er)	Sekundær aktør er LMES+	
Udløser	Når operatøren har behov for at få udstyrsstatus for fabriksudstyr, udløser han en forespørgsel.	
Beskrivelse	**Trin**	**Hændelse**
	1	Operatør: Vælger udstyrsstatus som datavalg
	2	Operatør: Indlæser stregkoden på fabriksudstyr.
	3	Operatør: Sender forespørgslen videre i systemet.
	4	System: Verificerer, at den PDA operatøren anvender, er registreret (*use-case* #3). Implementeres ikke i forbindelse med dette projekt.
	5	System: Præsenterer udstyrsstatus for det ønskede fabriksudstyr.
Udvidelse – alternativt flow	**Trin**	**Hændelse**
	1a	System: De relevante datavalg kan ikke foretages. 1. Fejlhåndtering foretages. 2. Gem historik for fejl (*use-case* #6). Implementeres ikke i forbindelse med dette projekt.
	1b	System: Forespørgslen bliver ikke sendt videre i systemet. 1. Fejlhåndtering foretages. 2. Gem historik for fejl (*use-case* #6). Implementeres ikke i forbindelse med dette projekt.
	2a	System: Stregkoden bliver indlæst forkert. 1. Fejlhåndtering foretages. 2. Gem historik for fejl (*use-case* #6). Implementeres ikke i forbindelse med dette projekt.
	3a	System: Kan ikke vise den ønskede data. 1. Fejlhåndtering foretages. 2. Gem historik for fejl (*use-case* #6). Implementeres ikke i forbindelse med dette projekt.
	4a	System: PDA'en er ikke registreret. 1. Forespørgselen afvises og punkt 5 udføres ikke. Implementeres ikke i forbindelse med dette projekt.
Forgrening	**Trin**	**Hændelse**
	2.1	Datavalg: 1. Udstyrsstatus. 2. Udstyrsoperationer (*use-case* #2).
	3.1	Herunder videresendes ligeledes automatisk oplysninger, der identificerer den PDA, operatøren anvender. Implementeres ikke i dette projekt
Relaterede informationer		
Prioritet	Høj. Denne *use-case* indeholder interaktionen mellem operatøren og	

	systemet, dette er en vital funktion i systemet, og bør derfor prioriteres højt.
Ydelse	En forespørgsel skal tage < 6 sek.
Hyppighed	Mellem. En forespørgsel fra operatøren til det underliggende system vedrørende udstyrsstatus kan forekomme hyppigt, og systemet skal altid være i stand til at modtage denne forespørgsel. Systemet skal være i stand til at præsentere information fra den relevante forespørgsel, hurtigt nok til at informationen stadig er relevant i et realtidssystem. Det antages at der maksimalt er 10 operatører der kan lave forespørgsler samtidigt, og at disse højst laver 1 forespørgsel i minuttet. Dette giver en frekvens på 6 sekunder pr. forespørgsel.
Udestående(r)	
Forfaldsdato	Skal være funktionel efter *Construction.*
Overordnede use-case(s)	Godkend PDA (*use-case* #3).
Underordnede use-case(s)	Gem historik (*use-case* #6), Hent udstyrsoperationer (*use-case* #2).
Eventuelt	

Figur 11 viser et *System Sequence Diagram* (SSD) for Hent Udstyrsstatus. Eftersom trin 4 fra *use-casen* ikke skal implementeres i forbindelse med dette projekt, fremgår det ikke af tegningen. Figuren har til formål at visualisere trinene i denne *use-case*. I forbindelse med trinnet GetBarcode() bør det bemærkes, at der ikke som sådan er tale om at operatøren skal tilgå LMES+ systemet for at skaffe stregkoden, men snarere at denne skal aflæses fra noget fabriksudstyr. Dette betragtes dog som en handling, der udføres på LMES+, idet stregkoderegistrering af udstyr er nødvendigt element i at få LMES+ til at virke, da alle informationer vedrørende udstyr lagres på baggrund af udstyrets respektive stregkoder.



Figur 11 - SSD Hent Udstyrsstatus

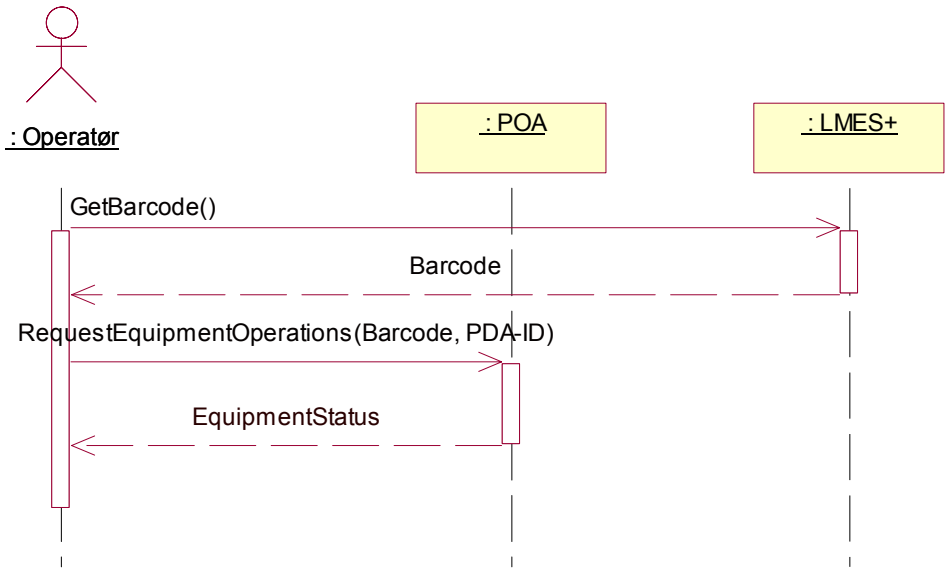
5.4.3.2 Hent udstyrsoperationer

Skal kun implementeres delvist. Det skal kun være muligt at vise en enkelt operation og automatisk skal ikke foretages.

Use-case #2	Hent udstyrsoperationer	
Formål	Operatøren skal have præsenteret informationer vedrørende de operationer, der er under afvikling på det relevante fabriksudstyr. Denne skal løbende opdateres, så der gives et realtidsbillede.	
Forudsætninger	1. Systemet er startet op.	
Succeskriterium	Operatøren får præsenteret udstyrsstatus korrekt.	
Fejlkriterium	Operatøren får slet ikke præsenteret den efterspurgte information, eller denne præsenteres forkert.	
Primær aktør	Primær aktør er operatøren.	
Sekundær(e) aktør(er)	Sekundær aktør er applikationen. Sekundær aktør er LMES+	
Udløser	Når operatøren har behov for at få udstyrsstatus for fabriksudstyr, udløser han en forespørgsel.	
Beskrivelse	**Trin**	**Hændelse**
	1	Operatør: Vælger udstyrsoperationer som datavalg
	2	Operatør: Indlæser stregkoden på fabriksudstyr.
	3	Operatør: Sender forespørgslen videre i systemet.
	4	System: Verificerer, at den PDA operatøren anvender, er registreret (*use-case* #3). Implementeres ikke i forbindelse med dette projekt.
	5	System: Præsenterer information om udstyrsoperationer på det ønskede fabriksudstyr.
Udvidelse – alternativt flow	**Trin**	**Hændelse**
	1a	System: De relevante datavalg kan ikke foretages. 1. Fejlhåndtering foretages. 2. Gem historik for fejl (*use-case* #6). Implementeres ikke i forbindelse med dette projekt.
	1b	System: Forespørgslen bliver ikke sendt videre i systemet. 1. Fejlhåndtering foretages. 2. Gem historik for fejl (*use-case* #6). Implementeres ikke i forbindelse med dette projekt.
	2a	System: Stregkoden bliver indlæst forkert. 1. Fejlhåndtering foretages. 2. Gem historik for fejl (*use-case* #6). Implementeres ikke i forbindelse med dette projekt.
	3a	System: Kan ikke vise den ønskede data. 1. Fejlhåndtering foretages. 2. Gem historik for fejl (*use-case* #6). Implementeres ikke i forbindelse med dette projekt.
	4a	System: PDA'en er ikke registreret 2. Forespørgselen afvises og punkt 5 udføres ikke. Implementeres ikke i forbindelse med dette projekt.
Forgrening	**Trin**	**Hændelse**
	2.1	Datavalg: 1. Udstyrsstatus (*use-case* #1). 2. Udstyrsoperationer.
	3.1	Herunder videresendes ligeledes automatisk oplysninger, der identificerer den PDA, operatøren anvender.
Relaterede informationer		
Prioritet	Mellem. Denne *use-case* indeholder interaktionen mellem operatøren og systemet, dette er en vital funktion i systemet, og bør derfor prioriteres højt.	
Ydelse	En forespørgsel skal tage < 6 sek.	

Hyppighed	Høj. En forespørgsel fra operatøren til det underliggende system vedrørende udstyrsoperationer kan forekomme meget hyppigt, og systemet skal altid være i stand til at modtage denne forespørgsel. Systemet skal være i stand til at præsentere information fra den relevante forespørgsel, hurtigt nok til at informationen stadig at relevant i et realtidssystem. Det antages, som angivet under formål, at de operatører, der tilgår oplysninger om udstyrsoperationer løbende skal have opdateret disse, så der gives et realtidsbillede af, hvilken operation, der er under afvikling på udstyret.
Udestående(r)	
Forfaldsdato	Skal være delvist funktionel efter construction.
Overordnede use-case(s)	Godkend PDA (*use-case* #3).
Underordnede use-case(s)	Gem historik (*use-case* #6), Hent udstyrsstatus (*use-case* #1).
Eventuelt	

Figur 12 viser SSD for *use-case #2*. Denne er grundlæggende den samme som for *use-case #1*, blot er der tale om et *RequestEquipmentOperations*-kald til applikationen.



Figur 12 - SSD for Hent Udstyrsoperation

5.4.3.3 Godkend PDA

Skal ikke implementeres.

Use-case #3	Godkend PDA	
Formål	Den PDA, operatøren anvender, skal godkendes.	
Forudsætninger	Systemet er startet op, der er foretaget en forespørgsel, hvor PDA-informationer er blevet videresendt (*Use-case* #1 eller *Use-case* #2).	
Succeskriterium	Operatørens PDA er blevet korrekt godkendt.	
Fejlkriterium	Godkendelse af PDA'en er håndteret fejlagtigt.	
Primær aktør	Primær aktør er operatøren.	
Sekundær(e) aktør(er)	Sekundær aktør er applikationen.	
Udløser	Der forekommer en forespørgsel med tilhørende PDA-informationer.	
Beskrivelse	**Trin**	**Hændelse**
	1	Operatør: Foretager forespørgsel
	2	System: Modtager PDA-informationer
	3	System: Godkender PDA'en. For at PDA'en kan godkendes, skal den være oprettet i systemet (*Use-case* #4)
	4	System: Den aktuelle forespørgsel kan fortsætte (*Use-case* #1 eller *Use-case* #2).
Udvidelse – alternativt flow	**Trin**	**Hændelse**
	2a	System: PDA-informationerne modtages ikke korrekt 1. Fejlhåndtering foretages. 2. Gem historik for fejl (*use-case* #6). Implementeres ikke i forbindelse med dette projekt.
	3a	System: PDA'en godkendes ikke 1. Den aktuelle forespørgsel afbrydes. 2. Fejlmeddelelse returneres.
	3b	System: PDA'en findes i systemet, men godkendes ikke 1. Fejlhåndtering foretages. 2. Gem historik for fejl (*use-case* #6). Implementeres ikke i forbindelse med dette projekt.
	4a	System: Forespørgslen får OK til at fortsætte, selvom PDA'en ikke er oprettet i systemet. 1. Fejlhåndtering foretages. 2. Gem historik for fejl (*use-case* #6). Implementeres ikke i forbindelse med dette projekt.
Forgrening	**Trin**	**Hændelse**
Relaterede informationer		
Prioritet	Lav. Denne *use-case* indeholder valideringen af PDA'en i forhold til systemet. Dette er en forudsætning for at systemet kan bruges af operatøren. Det er mere et sikkerhed/valideringsaspekt end et funktionelt aspekt, og har derfor lav prioritet i forbindelse med dette projekt.	
Ydelse	Ventetiden skal være < 10 sek.	
Hyppighed	Høj Dette tjek foretages hver gang, en operatør foretager en forespørgsel.	
Udestående(r)		
Forfaldsdato	Skal ikke implementeres i forbindelse med dette projekt.	
Overordnede use-case(s)		
Underordnede use-case(s)	Gem historik (*use-case* #6)	
Eventuelt		

5.4.3.4 Installér applikation

Skal kun implementeres delvist. Det skal blot være muligt at tilgå et centralt installationsmodul og foretage installation.

Use-case #4	Installér applikation	
Formål	Brugerdelen af systemet skal installeres med minimal indsats.	
Forudsætninger	1. Det underliggende system er startet op. 2. Brugerdelen af systemet skal forefindes på et eksternt system, som administratoren kan tilgå.	
Succeskriterium	Systemet bliver korrekt installeret, og kan efterfølgende tilgås af operatøren.	
Fejlkriterium	Systemet er ikke blevet installeret, og det kan ikke betjenes.	
Primær aktør	Primær aktør er administrator.	
Sekundær(e) aktør(er)		
Udløser	Administratoren tilgår installationsmodulet.	
Beskrivelse	**Trin**	**Hændelse**
	1	Administrator: Opretter PDA i systemet
	2	Administrator: Tilgår installationsmodulet.
	3	System: Installerer de relevante data.
	4	System: Efter systemet er installeret, opstartes brugerdelen af systemet.
Udvidelse – alternativt flow	**Trin**	**Hændelse**
	1a	System: PDA'en er allerede oprettet 1. Fortsæt til trin 2
	2a	System: Installationsmodulet kan ikke tilgås 1. Der foretages den fornødne fejlhåndtering, og processen genstartes.
	3a	System: Installationsmodulet konstaterer at systemet allerede er installeret. • Installationen afbrydes. Systemet er klar til brug.
	4a	System: Brugerdelen kan ikke startes 1. Der skal være mulighed for at geninstallere systemet.
Forgrening	**Trin**	**Hændelse**
	1	Administratoren kan foretage installation via: 1. Web browser 2. Deployment i *Visual Studio*
Relaterede informationer		
Prioritet	Mellem. Forudsætningen for at afvikle systemet, er en korrekt installation. Fra et funktionelt synspunkt, har det dog ikke højest prioritet, at et fuldt udbygget installationsmodul forefindes.	
Ydelse	Systemet skal køre uden afbrydelser	
Hyppighed	Sjældent. Der er behov for installation ved førstegangsbrug af systemet, hvilket er sjældent. Det bevirker at installationsforløbet gerne må tager længere tid.	
Udestående(r)		
Forfaldsdato	Skal være delvist funktionel efter *Construction*.	
Overordnede use-case(s)		
Underordnede use-case(s)	Vedligehold system (*use-case* #7)	
Eventuelt		

5.4.3.5 Gem historik

Skal ikke implementeres.

Use-case #5	Gem historik	
Formål	Når der forekommer en fejl, skal informationer vedrørende denne gemmes	
Forudsætninger	1. Systemet er startet op	
Succeskriterium	Informationer vedrørende den aktuelle fejl er blevet gemt	
Fejlkriterium	Informationer vedrørende den aktuelle fejl er ikke blevet gemt	
Primær aktør	Primær aktør er operatøren	
Sekundær(e) aktør(er)		
Udløser	Operatøren foretager en forespørgsel, der resulterer i en fejl	
Beskrivelse	**Trin**	**Hændelse**
	1	Operatør: Foretager en forespørgsel (*use-case* #1 eller *use-case* #2)
	2	System: Lagrer relevante oplysninger vedrørende fejlen
Udvidelse – alternativt flow	**Trin**	**Hændelse**
	1a	System: Forespørgslen slår fejl 1. Fejlhåndtering foretages
	2c	System: Lagringen slår fejl 1. Fejlhåndtering foretages
Forgrening	**Trin**	**Hændelse**
	2.1	Relevante oplysninger: • PDA • Dato/tid • Fejltekst
Relaterede informationer		
Prioritet	Lav	
Ydelse	Skal kunne håndteres indenfor rammerne af den tid, der er afsat til en operatørforespørgsel	
Hyppighed	Lav. Udføres hver gang en forespørgsel udført af operatøren fejler	
Udestående(r)		
Forfaldsdato	Skal ikke implementeres i forbindelse med dette projekt.	
Overordnede use-case(s)	Godkend PDA (*use-case* #3), Hent udstyrsstatus (*use-case* #1), Hent udstyrsoperationer (*use-case* #2)	
Underordnede use-case(s)		
Eventuelt		

5.4.3.6 Vedligehold system

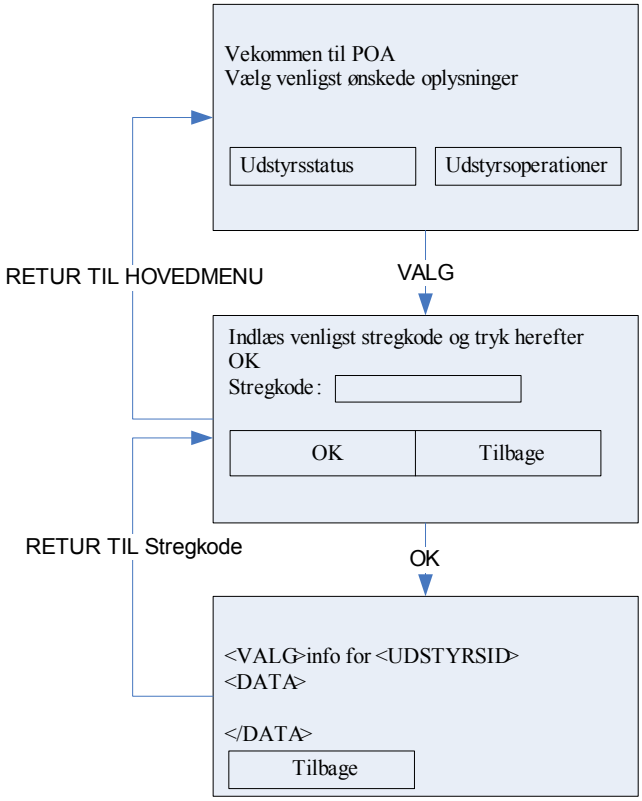
Skal ikke implementeres.

Use-case #6	Vedligehold system	
Formål	Brugerdelen af systemet skal opdateres automatisk. I tilfælde af at en nyere version findes, skal systemet selv sørge for opdateringen.	
Forudsætninger	1. Det underliggende system er startet op. 2. Systemet er installeret (*use-case* #4)	
Succeskriterium	Operatøren har nyeste version af systemet installeret	
Fejlkriterium	Operatøren anvender en version, der er ældre end den nyeste, der er tilgængelig	
Primær aktør	Primær aktør er systemet.	
Sekundær(e) aktør(er)		
Udløser	Der forefindes en ny version af systemet	
Beskrivelse	**Trin**	**Hændelse**
	1	Operatør: Brugerdelen af systemet startes
	2	System: Det tjekkes, om der findes en ny version
	3	System: Det konstateres, at der findes en ny version
	4	System: Der tages backup af den gamle version
	5	System: Vedligeholdelsessystemet igangsætter installation
	6	System: Efter endt installation genstartes brugerdelen af systemet
Udvidelse – alternativt flow	**Trin**	**Hændelse**
	2a	System: Tjekket slår fejl 1. Fejlhåndtering foretages
	3a	System: Det konstateres, at der ikke findes en ny version 1. Der skal ikke foretages nogen vedligeholdelse, og systemet startes normalt
	4a	System: Backup fejler 1. Fejlhåndtering foretages 2. Vedligeholdelse afbrydes og systemet genstartes
	5a	System: Installationen fejler 1. Fejlhåndtering foretages 2. Systemet bringes tilbage til sin tidligere version 3. Brugerdelen af systemet genstartes
	6a	System: Genstart fejler 1. Fejlhåndtering foretages
Forgrening	**Trin**	**Hændelse**
Relaterede informationer		
Prioritet	Lav	
Ydelse	Hurtig <2 min	
Hyppighed	Afhængig af opdateringsfrekvensen, men sandsynligvis sjældent forekommende	
Udestående(r)		
Forfaldsdato	Skal ikke implementeres i forbindelse med dette projekt.	
Overordnede use-case(s)	Installér applikation (*use-case* #4)	
Underordnede use-case(s)		
Eventuelt		

5.5 Brugergrænseflade

Brugergrænsefladen er det applikationslag, som fabriksoperatørerne skal interagere med. Det er herfra denne foretager valg og data præsenteres. Kravene for brugergrænsefladen er definerede med udgangspunkt i Hent udstyrsstatus (*use-case* #1) og Hent udstyrsoperationer (*use-case* #2).

Figur 13 giver et overblik over, hvorledes brugergrænsefladen ønskes konstrueret, dvs. hvilke valgmuligheder skal der være og hvilket hierarki af menuer og undermenuer, den skal indeholde. Dette skal bruges som grundlag for designbeslutninger vedrørende udarbejdelsen af en GUI til PDA-applikationen. Teksten på forbindelsesstregerne, der går mellem de forskellige kasser, repræsenterer de valg, en operatør kan foretage, og hvilke skærmbilleder der skal vises på baggrund af disse.



Figur 13 – Brugergrænseflade visualisering

Brugergrænsefladen skal startes, når operatøren starter sin PDA op. Herefter skal operatøren præsenteres for et skærmbillede, hvor det er muligt foretage forskellige datavalg. Efter at have gjort dette, skal operatøren præsenteres for et skærmbillede, hvor vedkommende bliver bedt om at indlæse en udstyrsstregkode eller gå tilbage til hovedmenuen. Efter at have indlæst stregkoden, skal operatøren præsenteres for et nyt skærmbillede, der viser den ønskede information. Her skal operatøren kunne vende tilbage til indlæsning af stregkode.

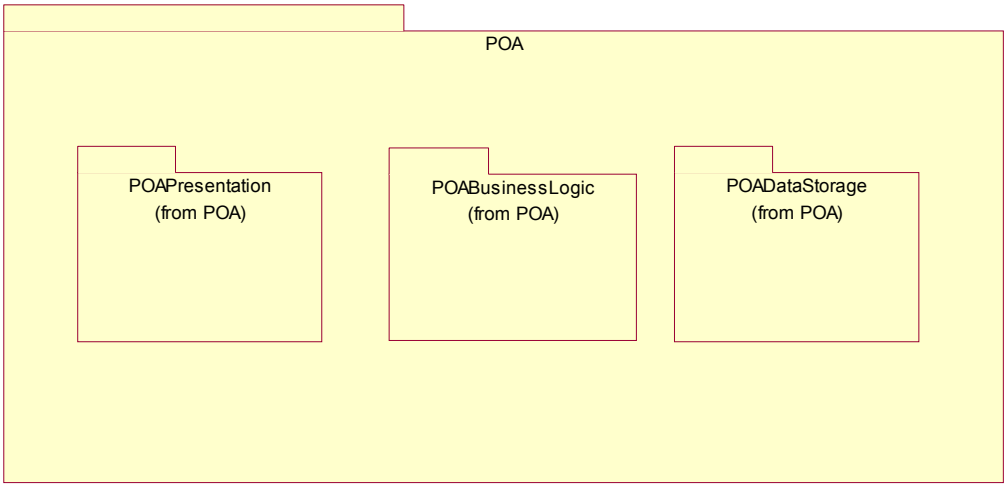
Brugergrænsefladen skal fungere som et visualiseringslag. Det vil sige, at der ikke foretages databearbejdning i dette lag. Indholdet af den brugergrænseflade, der udvikles, skal ikke defineres i selve *Presentation Layer*, men derimod skal det underliggende LMES+ system udvides, således at det kan lagre disse oplysninger. *Presentation Layer* skal blot fortolke de oplysninger, der hentes herfra og oprette brugergrænsefladen på baggrund af dette. Det vil sikre, at det vil være muligt at

udskifte den definerede brugergrænseflade med en anden uden at ændre i applikationskoden. En sådan data dreven dynamisk opbygning af brugergrænsefladen er i overensstemmelse med NNE's ønsker herom. Udover at sikre, at brugergrænsefladen nemt kan udskiftes, skal denne løsning også sikre, at det er nemt at opdatere og videreudvikle det eksisterende layout. Dette skal gøres ved, at det skal være muligt for en administrator at oprette nye skærmbilleder med tilhørende grafiske elementer og nye datavalg ved blot at tilføje og ændre i brugergrænsefladens definition i LMES+. Ved applikationsopstart skal det første skærmbillede således genereres på baggrund af de skærmbilleder, der på forhånd er blevet defineret. Ligeledes skal alle efterfølgende skærmbilleder, der fremkommer efterhånden som brugeren bevæger sig rundt i applikationen, genereres dynamisk, når de bliver forespurgt. Idet alle disse oplysninger skal findes i LMES+, skal POA forespørge disse på samme måde, som når oplysninger om fx udstyrsstatus forespørges. På denne måde sikres det, at slutbrugeren kan opsætte sin brugergrænseflade, som denne måtte ønske. Kravet om en dynamisk brugerflade er i løbet af projektarbejdet yderligere blevet indskærpet fra NNE's side, og bliver af disse betragtet som en helt central feature ved applikationen.

5.6 Domænemodel

Med udgangspunkt i de definerede *use-cases* er der blevet udarbejdet en række konceptuelle klasser. En konceptuel klasse repræsenterer fx fysiske objekter og begreber, der indgår i systemet.

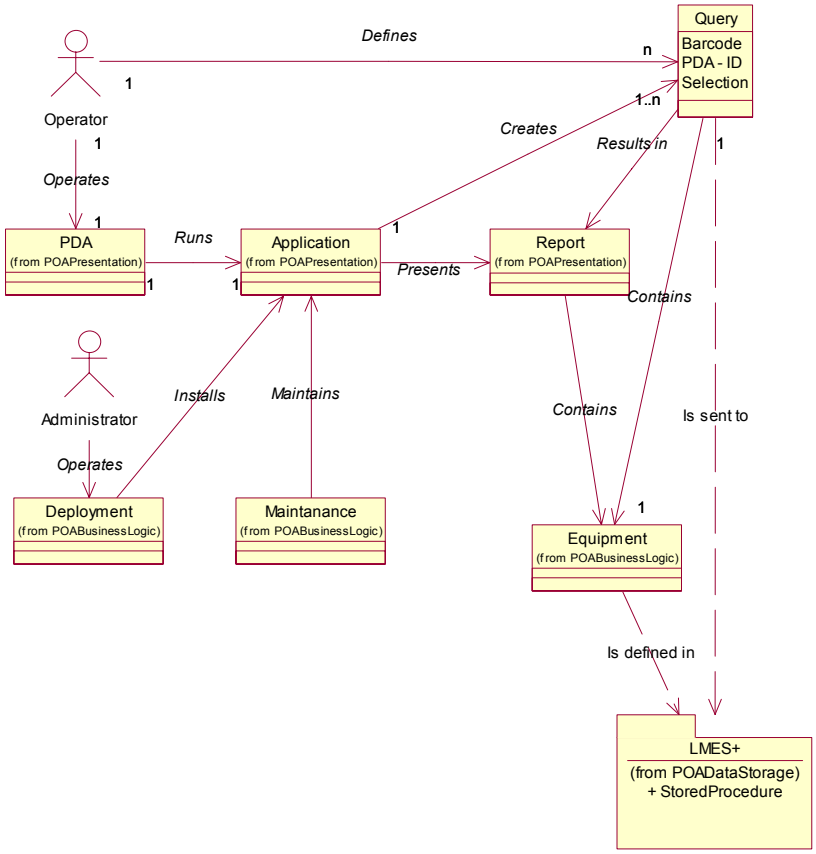
Som det fremgår af Figur 9, skal applikationen bestå af tre lag. Dette betyder, at de konceptuelle klasser logisk kan inddeles i hver deres enhed – såkaldte *packages*. Disse *packages* skal i forbindelse med design af systemet ligeledes anvendes som udgangspunkt for en inddeling i *software packages*. Den overordnede *package*-struktur kan ses af Figur 14. Her ses også, at POA betragtes som den overordnede referenceramme for hele systemet. Denne figur er udarbejdet med inspiration fra [7][24]



Figur 14 – Konceptuel package struktur

Figur 15 viser en oversigt over de konceptuelle klasser, som er blevet udarbejdede. Dette gøres i form af en domænemodel. Her fremgår i parentes fra hvilken *package* de enkelte konceptuelle klasser stammer.

[24] Side 426, figur 27.21.



Figur 15 – Domænemodel

Det kan ses af Figur 15, at Query-klassen er helt central den konceptuelle model. Det er denne klasse, der indeholder data vedrørende de forespørgsler, der skal føre til at operatøren får præsenteret de oplysninger, denne forespørger. Operatøren tilkendegiver sin sine ønsker overfor systemet ved at definere en forespørgsel.

5.7 Supplerende krav

I dette afsnit gennemgås krav, der ikke på en passende måde kan præsenteres i form af en *use-case*. Det kan være tekniske krav, som er givne på forhånd eller krav, som går på tværs af flere *use-cases* eller er gældende for hele systemet.

5.7.1 Produktperspektiv

Applikationen, der skal designes er et add-on til et eksisterende system. Den skal anvendes på fabrikker og skal tilgå informationer fra det eksisterende LMES+ system. Som det er blevet nævnt i *use-case* #1 og *use-case* #2, er der tale om at give fabriksoperatører mulighed for visuelt at blive præsenteret for udstyrsstatus og udstyrsoperationer for fabriksudstyr.

5.7.2 Funktionalitet

Jf. afsnit 5.7.1, skal applikationen anvendes til at vise oplysninger om fabriksudstyr. Applikationen skal dels bestå af en brugerdel, der afvikles på en PDA og dels af noget underliggende logik, der sørger for kommunikation og dataoverførsel mellem applikationen og det eksisterende LMES+ system. Den underliggende logik skal

ligeledes sørge for deployment (se afsnit 5.7.9) og vedligeholdelse (se afsnit 5.7.10) af brugerapplikationen.

5.7.3 PDA

Den PDA, applikationen skal afvikles på, skal leve op til en række krav. Den skal som minimum kunne køre *Compact Framework 2.0*, hvilket er understøttet fra og med Windows CE 5.0. Microsoft anvendte før i tiden terminologien *Pocket PC*, der dækker over en håndholdt computer, der afvikler en version af *Windows CE*. Disse *Pocket PCer* findes i en række versioner. *Pocket PC* navnet er imidlertid nu blevet erstattet af *Windows Mobile*, der i sin nuværende udgave er specificeret i en 5.0 version. Denne version definerer en håndholdt computer, der kører *Windows CE 5.0*, understøtter *Active Sync 4.0*, har *Bluetooth* understøttelse og en lang række andre krav til såvel hardware som software. Den PDA, der skal anvendes til dette projekt skal være en *Windows Mobile 5.0* PDA.

Derudover skal den være udstyret med en stregkodescanner og være beregnet til industriel brug med de dertilhørende krav om at kunne tåle slag, ekstreme temperaturer, væske og andet. Til testformål er det imidlertid ikke nødvendigt at anvende en PDA, der lever op til kravene om holdbarhed, den skal blot kunne køre *Compact Framework 2.0*. Stregkodescanneren er heller ikke absolut nødvendig for test, da dennes funktionalitet kan erstattes af manuel indtastning af stregkoden. På længere sigt er det dog essentielt, at applikationen testes på den type PDA, der fremover skal anvendes i Novo Nordisk fabrikkerne.

5.7.4 XML-kommentarer

I forbindelse med kodningen af de forskellige klasser, skal der angives XML-kommentarer. Ved kompileringen af koden skal der på baggrund af disse genereres en *XML Documentation File*. Dette er en setting, der kan vælges i *Visual Studio*. Denne skal herefter behandles i programmet NDoc, som kan anvendes til at generere en MSDN-dokumentationsfil.

5.7.5 Brugerkarakteristika

Brugerne udgøres primært af fabriksoperatører. Disse skal kunne lære at bruge applikationen med et minimum af træning. Det forudsættes at operatørne kender til den normale forretningsgang i firmaet.

5.7.6 Applikationsafvikling

Brugerdelen af applikationen skal kunne afvikles på den valgte PDA (se afsnit 5.7.3). Den del af applikationen, der kommunikerer med LMES+ og står for deployment og vedligeholdelse skal afvikles på en server.

5.7.7 Formodninger og afhængigheder

Applikationen er afhængig af og skal virke i samspil med en eksisterende applikation. Derfor er der en række ting, der er givet på forhånd:

- De data, applikationen skal tilgå, forefindes, som nævnt, i en Oracle database.
- Det eksisterende system kører under .NET, hvorfor add-on applikationen ligeledes skal gøre dette.

5.7.8 Datagrundlag

Applikationens datagrundlag udgøres af LMES+. For mere information herom se kapitel 4. LMES+ opbevarer en række oplysninger om bl.a. fabriksudstyr. Alle oplysninger lagres i en Oracle database, som applikationen skal kunne tilgå. Til udviklings- og testformål opretholder NNE en kopi af LMES+ databasen fra en af Novo Nordisk fabrikker. Det er denne NNE-kopi, der vil blive anvendt som datagrundlag under udvikling og test af applikationen.

Som nævnt, skal applikationen kunne præsentere operatøren for data vedrørende fabriksudstyrs' status og operationer, ligeledes skal applikationen være i stand til at tilgå databasen med henblik på at hente GUI informationer.
Database informationerne er følgende:

Tag	Value
Data Source	devora03_ibpkopi
Persist Security Info	True
User ID	ems
Password	ems

Det er således nødvendigt at være i stand til at lave forbindelse til denne database på et hvert tidspunkt applikationen skal anvendes.

5.7.9 Deployment

Installation af applikationen skal foretages af en administrator. Først skal *Business Logic Layer* deployes på en server, dernæst skal de forskellige *stored procedures* og tabeller oprettes i LMES+ og herefter skal PDA'erne have installeret PDA-applikationen (*use-case #4*). Derudover skal PDA'erne registreres, så det er muligt at sørge for at kun godkendte PDA'er kan tilgå oplysninger i LMES+ (*use-case #3* – skal ikke implementeres). Endvidere skal der være en WiFi forbindelse oppe at køre, således at PDA'en kan få adgang til *Business Logic Layer* (afsnit 6.7).

5.7.10 WiFi

For at PDA'en kan kommunikere med *Business Logic Layer* skal der være sat et WiFi *Access Point* op. Herefter skal de enkelte PDA'er konfigureres med følgende oplysninger:

Indstillinger for det trådløse netkort:
IP-nummer: 10.15.9.14X
Netværksmaske: 255.255.254.0
Standard gateway: 10.15.9.254

Indstillinger for forbindelsen til *access pointet*:
Service Set Identifier (SSID – dvs. navnet på WiFi netværket): TEST2NNE
Authentication: Open
Data Encryption: WEP
Network Key: B328iB328i123
Key Index: 1IEEE 802.1x: Disabled

5.7.11 Vedligeholdelse

Applikationen skal være nem at vedligeholde. Hver gang operatøren starter PDA-applikationen op, skal denne undersøge, om der findes en opdateret software version, og hvis dette er tilfældet, skal der automatisk foretages en opdatering. Den automatiske vedligeholdelse skal jf. *use-case #6* ikke implementeres.

5.8 Risici

I forbindelse med ethvert projekt forekommer der risikofaktorer. Disse skal overvejes, og det skal vurderes, hvad der kan gøres for at imødegå dem. Der arbejdes i dette projekt med to typer risici. Der er såkaldte *functional risks*, som udspringer af de krav og forudsætninger, som de enkelte *use-cases* opstiller, og så er der mere overordnede såkaldte *non functional risks*. Disse er relaterede til gennemførslen af projektet og hvor vidt de supplerende krav er indfriede.

5.8.1 Functional risks

Tabel 7 viser en oversigt over *functional risks*. Der er kun medtaget risici for de *use-cases*, der vil blive implementeret.

Påvirkede use-cases	**Risici**
Use-case #1: Hent udstyrsstatus *Use-case* #2: Hent udstyrsoperationer	o Databasesystemet kan ikke levere den fornødne ydeevne. o Netværket kan ikke levere den fornødne ydeevne. o WiFi *access pointet* har utilstrækkelig dækning. o WiFi har for dårlig sikkerhed. o PDA'en har for ringe ydeevne. o PDA'en lever ikke op til sikkerhedskravene på Novo Nordisks fabrikker. o PDA'ens stregkodelæser er for ringe.
Use-case #1: Hent udstyrsstatus *Use-case* #2: Hent udstyrsoperationer *Use-case* #4: Installér applikation	o Webserveren har for megen nedetid. o Webserveren er for langsom.

Tabel 7 - Functional risks

5.8.2 Håndtering af functional risks

For at undgå at de ovennævnte *functional risks* bliver problematiske, er det nødvendigt at tage stilling til, hvorledes disse ønskes håndteret. Det vil her blive gennemgået for de forskellige typer af risici.

5.8.2.1 Databasesystem

Hvis ydelsen på den anvendte database viser sig ikke at være tilstrækkeligt, påvirker dette hele systemet, hvilket kan udvikle sig til en uacceptabel situation. En sådan problemstilling kan opstå på længere sigt i takt med at antallet af forespørgsler genereret af applikationen stiger og databasen vokser i størrelse. Sandsynligheden for at dette sker, er svær at vurdere, idet mange faktorer spiller ind. Herunder bl.a. hvorledes andre dele af LMES+ udvikler sig. Bliver problemet aktuelt, kan det dog være yderst kritisk, da den applikation, der udvikles i dette projekt, vil være fuldstændig afhængig af tilfredsstillende ydelse fra databasen.

For at løse ovennævnte problemer vil det være nødvendigt at investere i en forbedret databaseløsning. Dette kan fx være i form af en kraftigere *database-server*.

5.8.2.2 Netværk

I takt med at antallet af forespørgsler måtte stige, kommer såvel WiFi- som LAN-netværket under pres, ligesom det kunne være tilfældet for databasen. Sandsynligheden for at netværkets ydeevne skulle være utilstrækkeligt vurderes ikke at være stor. Skulle det imidlertid blive aktuelt, er der tale om et yderst kritisk

problem, da den applikation, der skal udvikles, er fuldstændigt afhængig af netværkskommunikation over såvel LAN som WiFi. For at løse dette, vil det være nødvendigt at forbedre netværksinfrastrukturen ved at indkøbe yderligere hardware.

5.8.2.3 WiFi

Tilgængeligheden af WiFi forbindelse er yderst kritisk for denne applikation. Da applikationen skal anvendes i et fabriksmiljø, vil der være forskellige elementer, der kan påvirke denne forbindelse. Dette kan medføre, at dækningen ikke er tilstrækkelig. Sandsynligheden for, at dette forekommer, kan være ganske stor, hvis der ikke tages fornødne forholdsregler, og hvis fabriksområdet ikke udstyres med et tilstrækkeligt antal *access points*. Løsningen på dette problem er at sikre, at der investeres i et passende antal *acces points*, således at der er fuld WiFi dækning i fabriksområdet.

Sikkerheden i WiFi forbindelsen har i dette projekt ikke høj prioritet, men i produktionsøjemed vil det være kritisk, hvis sårbare fabriksinformationer lækkes gennem en usikker forbindelse. For at sikre imod dette, skal al WiFi trafik krypteres.

5.8.2.4 PDA

Det kan vise sig, at de indkøbte PDA'er ikke har tilstrækkelig ydeevne. Dette vil give udslag i uacceptable ventetider, når operatørerne foretager forespørgsler og i ekstreme tilfælde vil det betyde, at applikationen slet ikke kan afvikles. Ventetider er ikke et kritisk problem, men for at applikationen skal være brugbar er det dog nødvendigt, at denne har responstider, der opleves som tilfredstillende. Dette problem vurderes at have en vis sandsynlighed på længere sigt i takt med at applikationen udvides. Problemet med slet ikke at kunne afvikle applikationen vurderes at være usandsynligt og ville, i tilfælde af at det blev aktuelt, skyldes kraftige udvidelser af applikationen. I så fald er der tale om et yderst kritisk problem, og det vil være nødvendigt at udskifte de indkøbte PDA'er med kraftigere enheder.

På Novo Nordisks fabrikker opereres med en såkaldt positivliste, der angiver, hvilket hardware, der må anvendes på disse. Dette har til formål at sikre, at der kun anvendes hardware, der lever op til Novo Nordisks sikkerhedskrav. Det er et yderst kritisk problem, hvis de PDA'er der indkøbes til at afvikle applikationen ikke kan sikkerhedsgodkendes af Novo Nordisk, hvilket bør kontrolleres inden sådanne indkøbes.

Hvis det viser sig, at de indkøbte PDA'er er udstyret med en for ringe stregkodeskanner, udgør dette et yderst kritisk problem, da denne funktionalitet er helt central for den applikation, der skal udvikles. Bliver dette et problem, er det nødvendigt at skifte PDA'erne ud med enheder, der har bedre stregkodeskannere.

5.8.2.5 Websever

Det kan på længere sig vise, at det bliver et problem, at *webserveren* ikke leverer tilfredstillende ydelse. Dette kan dels gøre sig gældende ved, at den er langsom eller ustabil og derved har megen nedetid. Disse problemer vurderes ikke at være sandsynlige, men kunne dog forårsages af massiv netværkstrafik. I så fald vil dette enten skyldes, at den anvendte *webserver*-software er for ringe, eller hardwaren er utilstrækkelig. Har problemerne form af at *webserveren* svarer langsomt, er dette ikke et kritisk problem, men ligesom det var tilfældet ved for langsomme PDA'er, er det dog ikke tilfredsstillende. Har problemerne med *webserveren* derimod karakter af stabilitetsproblemer, er dette væsentligt mere kritisk, da det kan medføre sporadiske applikations-crashes og, at applikationen ikke kan tilgå den nødvendige information

fra databasen. Problemet kan løses ved at udskifte såvel *webserver* hardware som software.

5.8.3 Non functional risks

Tabel 8 viser en oversigt over en række af de *non functional risks*, der vurderes at være indeholdt i projektet.

Risiko	Beskrivelse
Urealistisk kravspecifikation	Der kan forekomme krav, der senere hen viser sig at være utilstrækkelige, overflødige eller på en anden måde uhensigtsmæssige.
Urealistisk tidsplan	Forskellige aspekter af projektet kan vise sig at tage længere tid end ventet
Systemet lever ikke op til slutbrugerens forventning	Dette kan forekomme, hvis den aktuelle implementering ikke indfrier kravspecifikationen
Koden bliver uigennemskuelig	Dette kan betyde, at det bliver svært at vedligeholde og udbygge koden.
Manglende gennemtest af systemet	Det er vigtigt at alle aspekter af systemet er blevet testet, da systemet ellers kan blive ustabilt
Sikkerhed	Der skal sikres ordentlig sikkerhed i systemet, således at det ikke er muligt at tilgå systemet med udstyr, der ikke er blevet godkendt (*use-case* #3)
Hastighed	Det er vigtigt, at de kriterier for ydelse, der er specificerede i de enkelte *use-cases* efterleves.
Udvikling	De udviklingsværktøjer, der anvendes viser sig at være uegnede. Dette kan dels skyldes, evt. autogenereret kode er utilfredsstillende og dels at de forskellige værktøjer ikke kan arbejde sammen.

Tabel 8 – Non functional risks

5.8.4 Håndtering af non functional risks

For at undgå at de ovennævnte non functional risks bliver problematiske, er det nødvendigt at tage stilling til, hvorledes disse ønskes håndteret. Det vil her blive gennemgået for de forskellige tilfælde.

5.8.4.1 Urealistisk kravspecifikation

Det er næsten utænkeligt, at der til at begynde med udvikles en fuldstændigt perfekt kravspecifikation. Det er derfor nødvendigt at have en strategi til at holde denne ajour. Det skal i forbindelse med dette projekt gøres ved, at der ved afslutning af hver iteration tages stilling til, hvilke aspekter af kravspecifikationen, der skal redigeres. På denne måde kan det sikres, at kravspecifikationen ikke er ude af trit med virkeligheden, og at der ikke er længere mellem den aktuelle implementering og kravspecifikationen, end at det er muligt at gå tilbage, hvis en evt. ændring viser sig at være uhensigtsmæssig.

I og med at dette projekt har en deadline, der ligger i den overskuelige fremtid, kan det vise sig, at den oprindelige kravspecifikation har elementer, der ikke kan implementeres indenfor den mulige tidshorisont. Dette skal som udgangspunkt helst undgås ved fornuftig projektstyring, men det kan være nødvendigt at revidere kravspecifikationen, hvis det viser sig, at tidsplanen skrider pga. at der ikke er afsat tid nok til de forskellige opgaver.

5.8.4.2 Urealistisk tidsplan

En række dele af projektet kan vise sig at tage længere tid end ventet. Et vigtigt redskab til at undgå at dette bliver et problem er at have løbende projektstyring. På den måde kan det sikres, at tidsplanen hele tiden tilpasses realiteterne, og det ligeledes hele tiden er muligt at få overblik over, hvor vidt der skal indsættes ekstra kræfter på nogle områder. I yderste konsekvens kan det, som det er nævnt i afsnit 5.8.4.1, være nødvendigt at foretage en prioritering af projektets delelementer for at sikre at kritiske dele gennemføres – muligvis på bekostning af mindre essentielle dele.

For at undgå at tidsplanen skrider, skal der tages en række instrumenter i brug. Først og fremmest skal der defineres en overordnet projektplan. Til dette formål skal *Microsoft Project* anvendes, og der skal her udarbejdes en plan for hele projektforløbet. Denne skal løbende justeres, efterhånden som delopgaver bliver færdige, og der skal korrigeres, når delopgaver tager kortere eller længere tid end ventet. Overordnet skal denne plan inddeles i de forskellige RUP iterationer. Der skal endvidere udarbejdes ugeplaner, der beskriver de opgaver, der skal afvikles i løbet af ugen.

5.8.4.3 Systemet lever ikke op til slutbrugerens forventning

Det er et alvorligt problem, hvis det produkt, der leveres ikke lever op til NNE's opfattelse af, hvad der skulle laves. Der skal derfor arbejdes tæt sammen med dem om at specificere systemet. Det betyder, at NNE løbende skal involveres, herunder både i forbindelse med udarbejdelse af *use-cases* og efterhånden som applikationen udvikles. Det er derfor også vigtigt at alle designvalg afspejler det system, der er blevet opridset vha. *use-cases*. Hvis der dukker designmæssige problemer op, der skyldes uhensigtsmæssigheder i *use-cases*, må den relevante use-case tages op og ændres, hvor det er nødvendigt, således at NNE hele tiden er involverede i, hvad der foretages af funktionelle ændringer. På den måde undgås det også, at der i forbindelse med design laves uigennemskuelige lappeløsninger, som eventuelt senere hen viser sig at forringe eller ændre applikationens funktionalitet i forhold til det, som NNE forventer.

5.8.4.4 Koden bliver uigennemskuelig

Uigennemskuelig kode vil gøre vedligeholdelse yderst besværlig. Det er derfor vigtigt, at det overvejes, hvorledes koden kan gøres nemt forståelig. Der er forskellige måder at gøre dette på. En måde er at man undgår at lave "smarte løsninger", hvor det er muligt at lave en traditionel løsning. Ulempen ved sådanne smarte løsninger er, at de ofte er meget specifikke for det enkelte problem og oftest siger det ikke sig selv, hvad der sker i koden. Til tider kan det dog være nødvendigt at lave noget kode, der ikke er selvforklarende, hvorfor det er vigtig at kommentere koden tilstrækkeligt. Et andet vigtigt instrument til at sikre mere gennemskuelig og struktureret kode er at anvende *design patterns* til at sikre en systematisk opbygget kode.

5.8.4.5 Manglende gennemtest af systemet

Det er vigtigt at alle aspekter af systemet er blevet testet, da systemet ellers kan blive ustabilt.

5.8.4.6 Sikkerhed

Det skal sikres, at det ikke er muligt at tilgå oplysninger gennem systemet, hvis man ikke har rettigheder dertil. For brugerdelen af applikationens vedkommende skal det, som nævnt i *use-case* #3, sikres ved, at der opretholdes et register over, hvilke PDA'er, der har lov til at tilgå systemet. Dette skal gøres ved at en administrator

vedligeholder en tabel i LMES+ databasen, der indeholder lovlige Mac-adresser. Skal ikke implementeres i forbindelse med dette projekt.

5.8.4.7 Hastighed

Det er vigtigt, at de kriterier for ydelse, der er specificerede i de enkelte *use-cases* efterleves. Dette punkt har høj prioritet, og der skal optimeres såfremt dette ikke er tilfældet.

5.8.4.8 Udvikling

I forbindelse med udvikling af applikationen vil en række IDE-produkter blive anvendt, og dette kan afføde nogle problemer i det omfang, disse anvendes til automatisk kodegenerering. Fx indeholder *Visual Studio* en *Designer*, der kan anvendes til at oprette databaseforbindelser. Det er en risiko, at anvendelse af denne gør, at udvikleren mister overblikket over, hvorledes der oprettes forbindelse til databasen, samt at der muligvis genereres store mængder kode, der er overflødig eller unødigt kompliceret. Derudover er det også et problem, hvis denne kode ikke er tilstrækkelig og skal redigeres, da autogeneret kode ofte kan være svær at gennemskue og i nogle tilfælde bliver den autogenerede del af koden hele tiden opdateret, således at eventuelle ændringer forsvinder. Denne problemstilling skal holdes for øje, og der skal kun anvendes autogenereret kode, hvor dette er en fordel.

Derudover kan det udgøre et stort problem, hvis forskellige produkter ikke kan arbejde sammen. Det skal derfor sikres på forhånd, at de produkter, der anvendes er indbyrdes kompatible, således at der ikke vælges et udviklinsværktøj, der ikke kan anvendes sammen med de applikationer, der i forvejen anvendes.

6 Design

I dette kapitel gennemgås designkravene til systemet. Der tages udgangspunkt i kravspecifikationen, og på baggrund af denne designes klasser, og den indbyrdes kommunikation klarlægges. Dette kapitel beskriver således, hvilke software klasser, der skal udvikles i projektet, idet der tages udgangspunkt i de konceptuelle klasser, der er blevet identificerede ud fra domænemodellen (se Figur 15).

Til at begynde med præsenteres et klassediagram, der giver et overblik over alle de software-klasser, der skal indgå i systemet. Herefter laves der interaktionsdiagrammer for de forskellige *use-cases*, der illustrerer kommunikationsflowet i forbindelse med realisering af disse. Først diskuteres de enkelte *use-cases* overordnet, men da applikationen er bygget op omkring tre lag, som nogle *use-cases* går på tværs af, vil den mere detaljerede gennemgang af hvorledes de skal realiseres foregå i forbindelse med diskussionen af disse lag. Således er dette kapitel inddelt i afsnit, der vedrører design klasse-diagrammet, interaktionsdiagrammer samt gennemgang af de tre lag, applikationen består af, dvs. *Presentation Layer*, *Business Logic Layer* og *Data Storage Layer*.

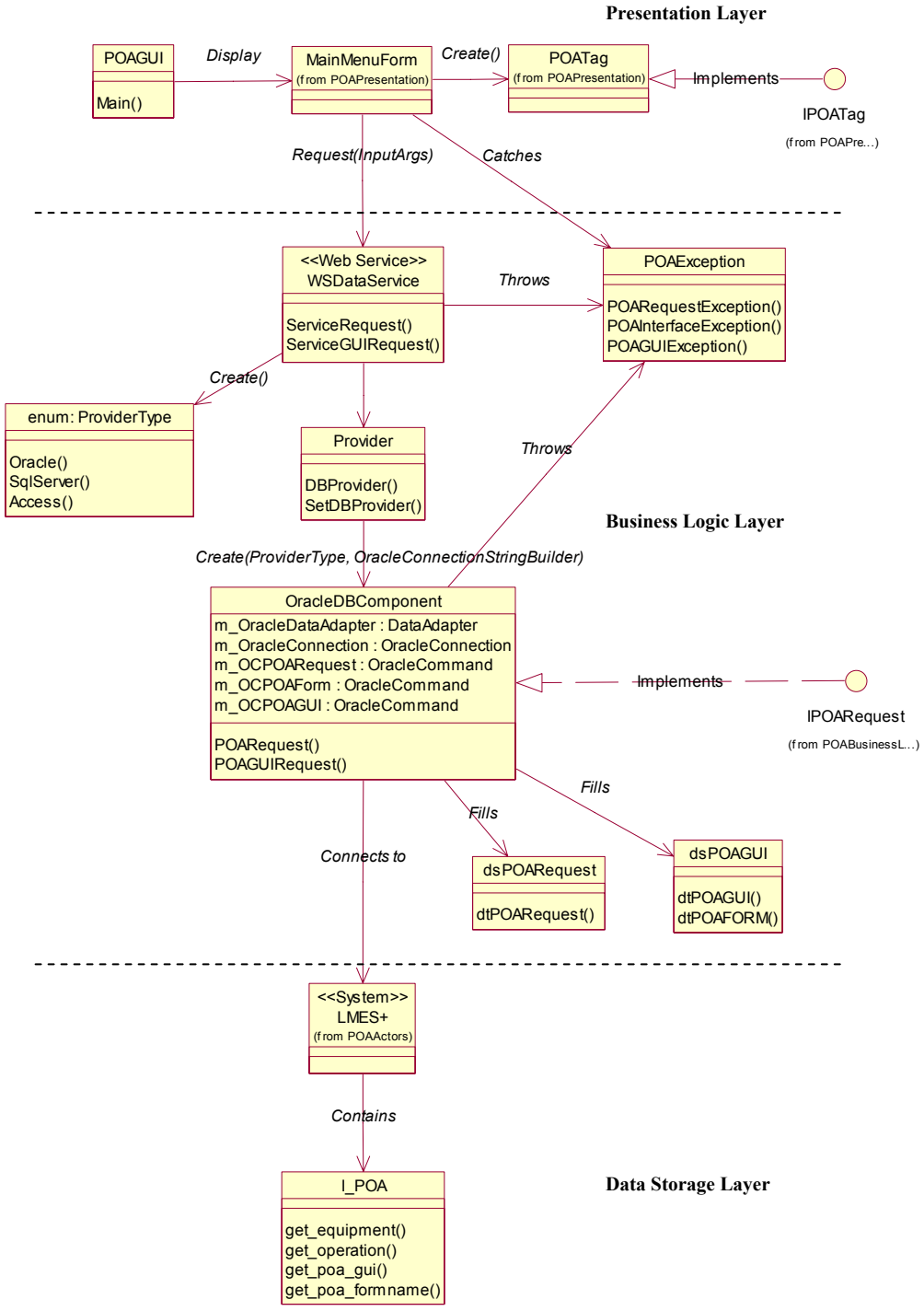
Når der henvises til datatyper i det følgende, er der tale om enten *C#*-datatyper eller Oracle-datatyper.

6.1 Overordnet system design

Overordnet skal systemet opbygges som en såkaldt *Three-Tier Architecture* eller på dansk en trelagsmodel. Dette har til formål at logisk adskille forskellige funktionelle enheder for at sikre modularitet og fleksibilitet. De tre lag, der opereres med i dette projekt er, som nævnt, *Presentation Layer*, *Business Logic Layer* og *Data Storage Layer*. Alle disse skal kompileres med *Strong Name*. Til dette formål skal defineres en key-file ved navn NNEPOAKEY.snk, der har passwordet ”Drumz1!”.

6.2 Klasse diagram

Figur 16 viser en oversigt over de software-klasser, der indgår i systemet og deres indbyrdes relationer. Diagrammet er på den måde en videreudvikling af domænemodellen, hvor de konceptuelle klasser er blevet oversat til software-klasser. Dette betyder, at enkelte konceptuelle klasser bliver til en eller flere software-klasser, nogle forsvinder, mens nogle software-klasser ikke er at finde i domænemodellen, men derimod er opstået på baggrund af rent implementeringsmæssige behov. Som det kan ses, er den konceptuelle Query-klasse forsvundet. Den funktionalitet, denne repræsenterede varetages i stedet af `InputArgs`, der sendes som input til `WSDataService`.



Figur 16 - Klassediagram

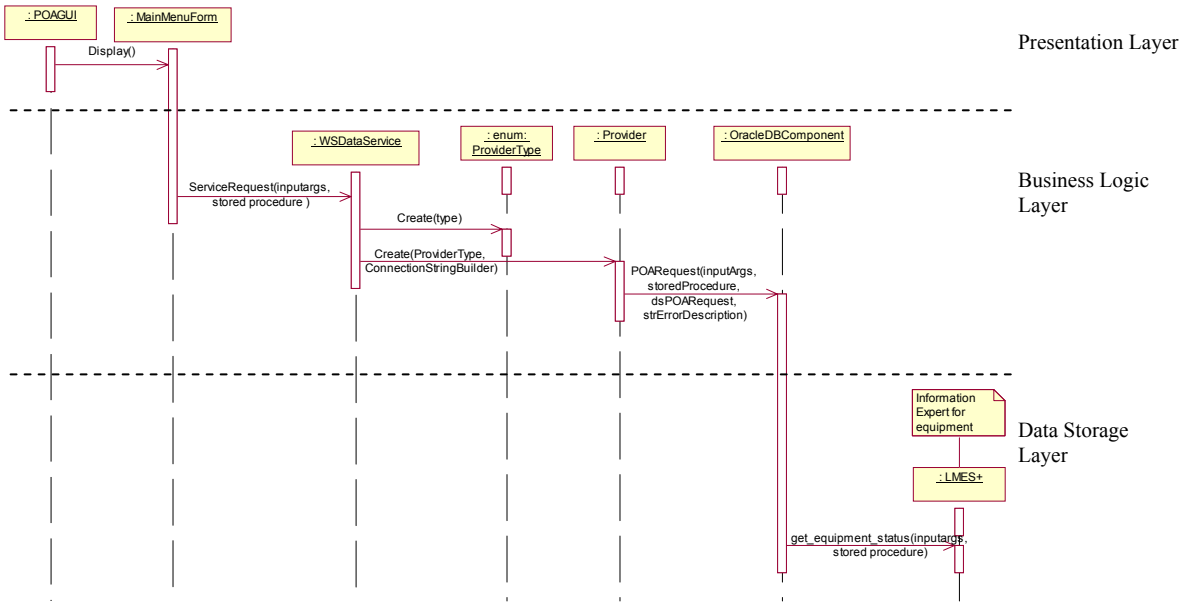
I *Visual Studio* skal applikationens tre lag oprettes som separate *projects*, der lægges ind en fælles *solution*, der skal kaldes POA. Idet *Business Logic Layer* udgør to funktionelt adskilte enheder, skal der defineres et *project* til hver af dem. Dette skyldes, at `WSDataService` skal implementeres som en *web service*, der i *Visual Studio* oprettes som et *Web Site*. Den samlede applikation skal ligge i en *Visual Studio*

Solution, der hedder *POA* og de fire projekter skal hedde hhv. `POAPresentation`, `POAWebServices`, `POADataAccess` og `POADataStorage`.

6.3 Use-case realisering

De opstillede Use-cases beskrevet i kravspecifikationen (kapitel 5) skal videreudvikles, så de bliver beskrevet mere fra en softwareudviklingssynsvinkel og yderligere bliver konkretiseret. Her skal opsættes interaktionsdiagrammer, hvor flowet på tværs af lagene bliver illustreret med udgangspunkt i de nødvendige software-klasser. En beskrivelse, og en argumentation for valget af de designede software-klasser, vil ligeledes følge de forskellige interaktionsdiagrammer. I de følgende underafsnit gennemgås realiseringen af de tre *use-cases*, der skal implementeres i forbindelse med dette projekt (se Tabel 6). Hent Udstyrsstatus og Hent Udstyrsoperation stiller grundlæggende samme krav til det underliggende system, og de gennemgås derfor under et. Installér Applikation gennemgås særskilt.

6.3.1 Hent Udstyrsstatus og Hent Udstyrsoperation



Figur 17 - ID for udstyrsstatus

Figur 17 viser hvilke trin, der skal gennemføres for at en operatør kan få adgang til udstyrsstatus for fabriksudstyr. Figuren tager direkte udgangspunkt i Hent Udstyrsstatus, *use-case #1*. Denne figur kan ligeledes anvendes for *use-case #2* blot skal `get_equipment` *stored procedure* erstattes af `get_operation`.

Som det kan ses, indgår der fem software-klasser, en `enum` samt undersystemet LMES+ i diagrammet. Den første klasse er `MainMenuForm`, indeholdende GUI'en, der skal implementeres i overensstemmelse med kravene i afsnit 5.5, og findes på *Prensentation Layer*. På *Business Logic Layer* skal der være en *web service* ved navn `WSDataService`, der skal servicere disse forespørgsler, en `enum`, der indeholder de mulige data providere og en `Provider`-klasse, der anvender den valgte `ProviderType` til at tilgå en `Component` ved navn `OracleDBComponent`,

som sørger for adgangen til LMES+. På *Data Storage Layer* er LMES+ i denne forbindelse en black box, der returnerer et output.

Hver gang operatøren vælger at sende en forespørgsel af sted via brugergrænsefladen, genereres en *event*. De *events*, der genereres af brugergrænsefladen, opstår, når operatøren trykker på de knapper, der indgår i denne. Disse *events* skal håndteres ved at anvende C#'s indbyggede *event* handlers til brug ved håndtering af tryk på knapper og lignende.

Designet af den omtalte trelagsmodel, der realiserer Hent Udstyrsstatus og Hent Udstyrsoperation, gennemgås i afsnit 6.6, 6.7 og 6.8. Det skal bemærkes, at der i forbindelse med Hent Udstyrsoperation ikke laves et fuldstændigt design og dermed heller ikke en fuldstændig implementering (jf. Tabel 6). Denne *use-case* implementeres kun i det omfang, hvor dens krav til *Presentation Layer* og *Business Logic Layer* ikke adskiller sig fra Hent Udstyrsstatus. På *Data Storage Layer* vil det kræve, at der implementeres en *stored procedure* til servicering af udstyrsoperationsforespørgsler, men dette er det eneste punkt, hvor en forsimplet implementering af denne *use-case* kræver yderligere arbejde, end hvad der i forvejen er nødvendigt for at implementere Hent Udstyrsstatus. Dette betyder, at der kun skal vises en enkelt udstyrsoperation og ikke alle, der måtte være aktive på udstyret.

Endvidere skal det bemærkes, at der i forbindelse med dette projekt ikke anvendes de reelle stregkoder for udstyr, men derimod en såkaldt Tag-værdi, der er indeholdt i LMES+ og er unik for hvert udstyr. Dette gøres for at gøre implementeringen enklere, da det ellers ville være nødvendigt at dechifrere de forskellige stregkoder, hvilket er en del mere komplekst end at udnytte, at Tag-værdien kan bruges direkte. På længere sigt vil det dog være nødvendigt at implementere stregkodefunktionalitet, men det vurderes ikke at være kritisk for dette projekt.

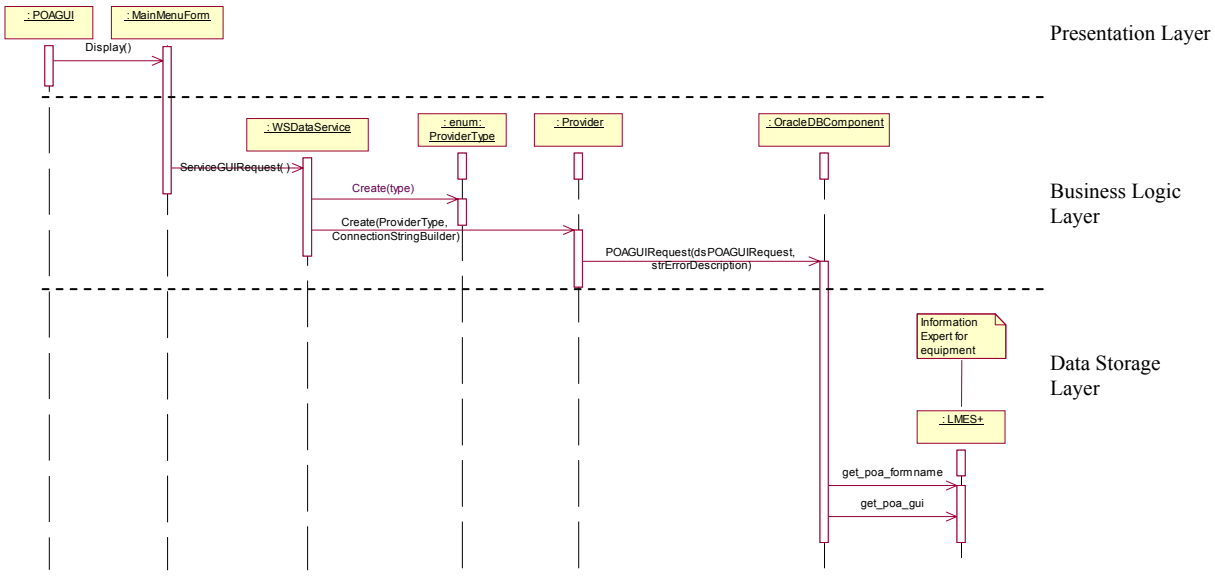
6.3.2 Installér Applikation

Installation af applikationen på PDA'en kræver, at denne er tilsluttet netværket ved hjælp af WiFi forbindelsen. Installationsprocessen skal igangsættes ved, at administratoren ved hjælp af en web browser på PDA'en tilgår en URL der peger på en installationsfil.

Designet af Installér Applikation gennemgås i afsnit 6.9.

6.4 GUI realisering

Det følgende interaktionsdiagram viser kommunikationflowet, når *Presentation Layer* indlæser oplysninger om hvilke `Forms` og tilhørende `Controls`, der skal oprettes.



Figur 18 - ID for GUI

Det kommunikationsflow, som ses af Figur 18, har til formål at realisere kravene fra afsnit 5.5, og illustrerer hvorledes GUI-oplysningerne hentes fra LMES+ systemet. De klasser, der anvendes er de samme, som dem, der fremgår af Figur 17, blot skal `WSDataService` udvides med en `ServiceGUIRequest()` −metode ligesom `OracleDBComponent` skal udsyres med en `POAGUIRequest()`-metode. Herudover skal der i LMES+ implementeres to *stored procedures* til at hente GUI-oplysningerne.

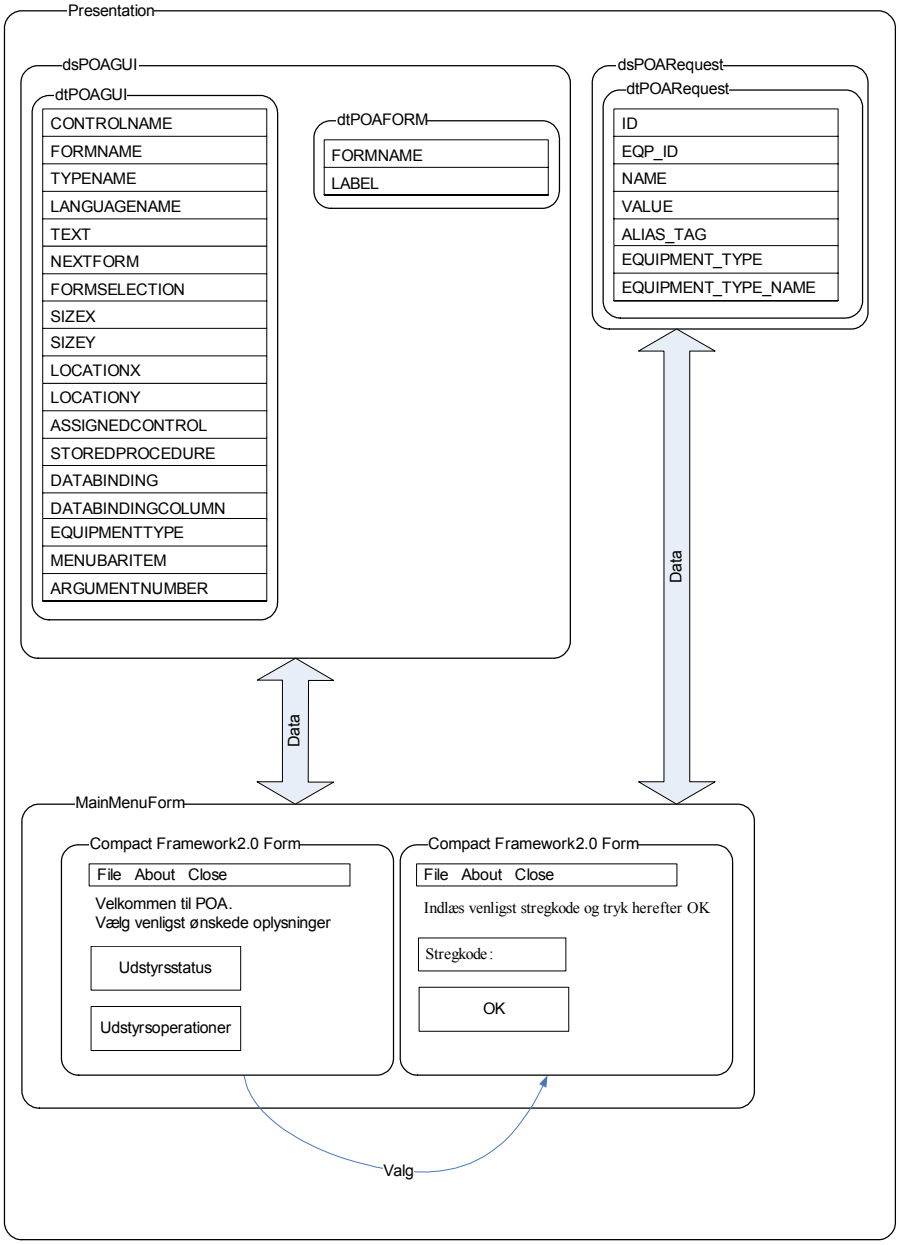
6.5 Exception handling

Exception handling skal håndteres på *Presentation Layer*. Dette skal fungere ved, at de underliggende lag `Catcher` og derefter `Thrower Exceptions` videre til *Presentation Layer*, når sådanne opstår.

Udover standard `Exceptions`, skal der i *Business Logic Layer* defineres en `POAException`-klasse, der bruges til at håndtere fejl i forbindelse med implementering af interfaces, fejl i forbindelse med GUI og fejl i forbindelse med requests. Denne klasse vil yderligere blive gennemgået under *Business Logic Layer* (afsnit 6.7). Der skal på *Presentation Layer* tilføjes en `Reference` til *Business Logic Layer*, således at denne klasse også understøttes her, og `Exceptions` af denne type hermed kan håndteres.

6.6 Presentation Layer

Figur 19 giver et overblik over de funktionelle enheder i *Presentation Layer*, som skal placeres i `POAPresentation` *projectet*. Der er tale om et hybriddiagram, der dels viser, hvilke datakilder, der skal være tilgængelige, dels viser layoutet. Formålet er at give et lidt mere specifikt billede af kommunikationsflowet i *Presentation Layer*, end klassediagrammet fra Figur 16 giver og tager udgangspunkt i denne figur samt Figur 13.



Figur 19 - Presentation Layer design diagram

Presentation Layer skal, som nævnt, udvikles i *Compact Framework 2.0*, afvikles under *Windows CE 5.0* og være bygget op omkring `Forms`. Applikationer der afvikles under *Windows CE* har nogle særlige karakteristika. Fx kører de altid i full screen. Endvidere er de normalt sat op til, at blive minimerede, når der trykkes på krydset i øverste højre hjørne. Dette skal her ændres, således at applikationen lukkes, når der trykkes i øverste højre hjørne. Dette vil medføre, at krydset erstattes af en *OK*-knap. Brugergrænsefladen skal bestå af en GUI, hvorfra operatøren, som nævnt i kravspecifikationen, foretager valg og bliver præsenteret for data. For at dette kan lade sig gøre, skal der, som det fremgår af Figur 19, eksistere et `DataSet – dsPOAGUIRequest` - indeholdende oplysninger om, hvorledes GUI'en skal opbygges samt et `DataSet – dsPOARequest` - der indeholder oplysninger om udstyr til brug ved servicering af operatørens forespørgsler. I de følgende afsnit gennemgås, hvorledes GUI'en skal implementeres. Herunder præsenenteres først anvendelsen af *Observer design pattern*, som er central i forbindelse med denne GUI.

Herefter gennemgås `dsPAOGUI DataSettet`, dernæst hvilket indhold GUI'en skal have i den aktuelle implementering, dernæst den dynamiske opbygning af GUI'en, og til sidst gennemgås `dsPOARequest`. Som det fremgår af de følgende afsnit, er *Presentation Layer* afhængig af information, der hentes fra *Business Logic Layer* gennem `WSDataService`. Adgangen til `wsDataService` sikres ved at der i *POAPresentation projectet* oprettes en *web reference*, der peger på den *URL*, hvorpå denne findes. I dette projekt skal denne hardcodes til at være http://10.15.14.6/wsdataservice.asmx. I en fremtidig udgave skal det være muligt at ændre denne igennem applikationsindstillinger, men dette vil ikke blive implementeret. `wsDataService` gennemgås i *Business Logic Layer* afsnittet (afsnit 6.7).

6.6.1 Observer design pattern

I forbindelse med design af GUI'en skal *Observer design pattern* anvendes (se afsnit 1.1.4.2). GUI-design er et godt eksempel på, hvor *Observer* med fordel kan anvendes. GUI'en skal generere *events*, hver gang operatøren foretager en handling, der involverer de lag, der ligger under GUI'en. Et eksempel på dette kunne være, at operatøren har foretaget et datavalg og derefter har valgt at sende dette af sted til systemet med henblik på at modtage data. Afsendelsen af data kan tænkes igangsat ved at operatøren trykker på en knap på GUI'en. Herefter skal det underliggende databehandlingssystem tage stilling til dataforespørgslen. Den *event*, der bliver genereret ved trykket på knappen skal, for at følge *Observer design pattern*, uddelegeres til en *delegate*. Dette gøres ved at tilføje en eller flere *delegates* til knappens *click*-metode. Ved tryk på knappen kaldes herefter de tilføjede *delegate*-funktioner, som tager sig af den videre håndtering af trykket på knappen. Den videre håndtering kan have mange former og forskellige *delegates* knyttet til samme *event* kan have vidt forskellig funktionalitet. Et eksempel på en *delegate* kunne være en funktion, der åbner et nyt vindue, når der trykkes på en knap.

6.6.2 dsPOAGUIRequest

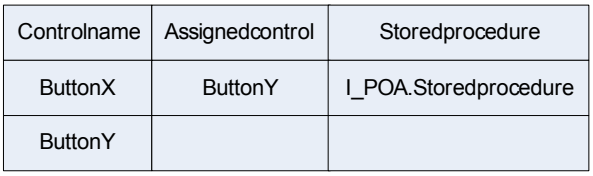
Som nævnt i kravspecifikationens afsnit om brugergrænsefladen (afsnit 5.5), skal denne være nem at ændre på, og den skal genereres dynamisk ved opstart af applikationen. Dette skal sikres ved, at *Presentation Layer* har adgang til et `DataSet`, som skal kaldes `dsPOAGUIRequest`, som det fremgår af Figur 19. Som det videre kan ses af figuren, skal dette `DataSet` indeholde to `DataTables`, der skal kaldes hhv. `dtPOAControl` og `dtPOAForm`. `dtPOAForm` skal indeholde en liste over alle de `Forms`, der skal oprettes og `dtPOAControl` skal indeholde alle oplysninger om hvilke elementer, der skal indgå i GUI'en. Dette indebærer, at den indeholder en liste over alle de `Controls` (dvs. knapper, lister o.a.), der skal indgå i de forskellige `Forms`. `dsPOAGUIRequest` får sit indhold ved at *Presentation Layer* kalder `ServiceGUIRequest()`-metoden i `WSDataService`

`dtPOAForm` skal have følgende felter:

Rækkenavn	Formål
ID	Fortløbende nummerering
FORMNAME	`Formens` navn
LABEL	Teksten på `Formens` menubjælke

Tabel 9 – dtPOAForm

dtPOAControl skal have følgende felter:

Rækkenavn	Formål
ID	Fortløbende nummerering
CONTROLNAME	Bruges til at angive Controllens navn og skal bruges til navngivning af denne i forbindelse med implementeringen.
TYPE	Controlens datatype. Det kan være Button, Label osv. I tilfældet, hvor der er tale om Buttons er det nødvendigt at definere en række standard-Buttons som kaldes CloseButton og NextButton. Dette skyldes, at der til disse skal defineres *event handler*, som det er nødvendigt at implementere på forhånd og så når GUI'en genereres skal de forskellige Buttons knyttes til den rette *event handler*. Følgende Controls skal eksistere: Label ButtonNext TextBox ButtonClose MenuItem LabelBind SubMenuClose SubMenuAbout
FORM	Hvilken Form tilhører Controllen. Bruges til at knytte de enkelte Controls til deres respektive Forms.
NEXTFORM	Hvilken form skal åbnes ved aktivering af Controllen
LANGUAGE	Hvilken sproggruppe tilhører Controllen
TEXT	Controllens tekstindhold. Det kan være den tekst, der vises på en Button eller en Labels tekstfelt
FORMSELECTION	Anvendes af Forms. Angiver hvilken Form, der skal åbnes, når Controllen aktiveres.
SIZEX	Controlens bredde
SIZEY	Controlens højde
LOCATIONX	Placering af Controlens øverste venstre hjørne i X-planen
LOCATIONY	Placering af Controlens øverste venstre hjørne i Y-planen
ASSIGNEDCONTROL	Anvendes af ButtonClose og ButtonNext. Denne Column definerer fx at ButtonX har en tilknytning til ButtonY (hvilket ses af at der i ButtonX's række er angivet ButtonY i ASSIGNEDCONTROL feltet – jf. nedenstående figur). I tilfælde af, at der bliver trykket på ButtonX sker der det, at den *stored procedure*, som denne har defineret skal gemmes i en variabel. Når der så efterfølgende trykkes på ButtonY, vil den *stored procedures*, der er defineret af ButtonX blive eksekveret. Trykkes fx på "Operationer" (ButtonX) i det første skærmbillede, er dette ensbetydende med, at der senere hen skal køres en *stored procedure*, der hører her til. Dette sker, når der trykkes på "OK" (ButtonY) i den næste Form, som så kan se, at der er blevet gemt en information i forbindelse med tryk på ButtonX. Denne sammehæng kan ses af nedenstående figur.  For at dette skal kunne virke kræves det, at der defineres en speciel RESULT_FORM, således at de Buttons, der fører til denne er af ButtonY typen fra ovennævnte eksempel. Der skal så altid gemmes en eventuel defineret *stored procedure*, når der trykkes på en Button

	af ButtonX typen (og blot blank, hvis ingen sådan er defineret for denne Button). Når så der herefter trykkes på en Button af ButtonY typen, skal den gemte *stored procedure* bringes til eksekvering og indholdet indlæses i den specielle RESULT_FORM.
STOREDPROCEDURE	Anvendes af ButtonClose og ButtonNext. Angiver hvilken *stored procedure*, der skal afvikles som følge af at denne Button er blevet trykket på.
DATABINDINGCOLUMN	Anvendes af LabelBind. Angiver den kolonne i den DataTable, som returneres af den kaldte *stored procedure*, hvis indhold skal læses ind i Labelelens Text felt. Den relevante række bestemmes af DATABINDING-feltet.
DATABINDING	Anvendes af LabelBind. Angiver hvilken udstyrsoplysning i den DataTable, som returneres af den kaldte *stored procedure*, der skal bruges til indlæsning i Label-objektets Text-felt. Hvilken række der skal bruges, afgøres ved at det indhold, som er specificeret i DATABINDING skal stemme overens med indholdet i ALIAS_TAG-feltet i en af rækkerne i den DataTable, som er blevet returneret af den pågældende *stored procedure*. Findes der således en række i udstyrs-DataTablen, der har den værdi som er angivet i DATABINDING i sit ALIAS_TAG-felt, er det denne række, der skal anvendes. ALL kan angives, hvis det er ligegyldigt hvad der står i ALIAS_TAG. Den relevante kolonne bestemmes af DATABINDINGCOLUMN–feltet.
EQUIPMENTTYPE	Anvendes af LabelBind. Bruges til at specificere udstyrstypen. Er relevant i forbindelse med nogle *stored procedures*, hvor der er forskelligt indhold fx alt efter hviken type udstyr, man har fat i.
MENUBARITEM	Anvendes af SubMenuAbout og SubMenuClose til at specificere hvilken overordnet menu de tilhører.
ARGUMENTNUMBER	Anvendes af TextBoxe. Bruges til at specificere rækkefølgen på *stored procedures.*

Tabel 10 - dtPOAControl

Indholdet til DataSettet og de dertilhørende DataTables skal hentes fra *Data Storage Layer*. Dette lag gennemgås i afsnit 6.8 om *Data Storage*, hvor de tekniske krav til den underliggende database præsenteres.

De nævnte DataTables skal bruges af MainMenuForm-klassen i *Presentation Layer* til dynamisk at oprette GUI'en ved at generere Forms og Controls på baggrund af deres indhold. Denne struktur skal medføre, at det, jf. afsnit 5.5, vil være muligt at lave GUI'en grundlæggende om uden at ændre en linje kode i selve applikationen, hvis det på længere sigt skulle være et ønske.

6.6.3 GUI-indhold

Som nævnt i *use-case* #1 og *use-case* #2, og præsenteret afsnit i 5.5, skal brugeren præsenteres for en række valgmuligheder, afhængigt af hvilke Forms og tilhørende Controls, der er blevet definerede. På længere sigt vil det være muligt at omstrukturere GUI'en fuldstændigt, men til at begynde med, skal den opbygges, som det bliver gennemgået i afsnit 5.5, som igen er udarbejdet på baggrund af de to *use-cases*.

I forbindelse med alle skærmbilleder skal der være adgang til en drop-down menu, hvor der findes et MenuItem objekt, der har hhv. et SubMenuClose objekt og SubMenuAbout objekt tilknyttet sig.

Det første skærmbillede operatøren skal se består af en Label med følgende velkomstbesked: "Velkommen til POA. Vælg venligst ønskede oplysninger.".

Herudover skal det indeholde en række `ButtonNexts` svarende til de mulige datavalg. Til dette formål skal der i `dtPOAControl` eksistere en `Row` med en `Label` til velkomstbeskeden samt `Rows` med `Buttons` svarende til hver valgmulighed. Disse `Rows` skal alle have MAIN_FORM stående i deres FORM-felt.

Efter at operatøren har foretaget et valg, skal brugeren præsenteres for det næste skærmbillede. Dette er for udstyrsstatus' vedkommende et skærmbillede, hvor vedkommende skal indskanne stregkoden på noget fabriksudstyr. Dette fører til, at der i `dtPOAForm` skal eksistere en `Row`, der indeholder oplysninger for SCAN_BARCODE_FORM. Tilsvarende skal dette skærmbillede have en række `Controls` defineret i `dtPOAControl`. De nødvendige `Controls` betyder for denne `Forms` vedkommende, at der i `dtPOAControl` eksisterer en `Row` med en `Label` med følgende velkomsttekst: "Indlæs venligst stregkode og tryk herefter OK". Herudover skal skærmbilledet have defineret en `Row` med en `TextBox`, som skal bruges til indlæsning af stregkode. Slutteligt skal der defineres `Rows` til to `Buttons` – en `ButtonNext` med teksten "OK" som skal bruges til at komme til næste skærmbillede og en `ButtoClose` med teksten "Tilbage", som skal bruges til at returnere til hovedmenuen. De skal alle have SCAN_BARCODE_FORM stående i deres FORM-felt.

Til slut præsenteres operatøren for et skærmbillede, der skal oprettes som RESULT_FORM. Denne `Form` har ikke noget selvstændigt indhold defineret; dette er derimod afhængig af hvilket data, der tidligere er ønsket. Er der fx tale om udstyrsstatus, vises det indhold, der er defineret i `Rows` hvis FORM-felt indeholder EQP_STATUS_FORM. For udstyrsstatus' vedkommende skal der oprettes en række `Rows` med `LabelBind` objekter til såvel databeskrivelse og selve data, som begge hentes fra databasen. Alle databeskrivelser og dataværdier skal oprettes som selvstændige `LabelBind` objekter. Derudover skal defineres en `Row` med en `ButtonClose`, som skal bruges til at komme tilbage til indskanningsskærmbilledet.

6.6.4 Dynamisk opbygning af GUI

`MainMenuForm` er central for den dynamiske opbygning af GUI'en. Det er denne klasse, der sørger for at indhente oplysninger om, hvorledes GUI'en skal opbygges for herefter at generere de relevante `Forms` og `Controls`. Dette skal, som nævnt, gøres ved at `WSDataServices ServiceGUIRequest()`-metode kaldes over netværket for at fylde `dsPOAGUIRequest`. `WSDataService` findes i *Business Logic Layer og* gennemgås under afsnittet herom (afsnit 6.7).

Efter at have opbygget GUI'en, skal `MainMenuForm` servicere de forespørgsler, operatøren generer. Disse forespørglser er beskrevet i hhv. Hent udstyrsstatus (*use-case* #1) og Hent udstyrsoperationer (*use-case* #2). Dette skal gøres ved at kalde `ServiceRequest()`.

GUI'en skal opbygges ved at `MainMenuForm` først indhenter oplysninger om hvilke `Forms` der findes fra `dtPOAForm`. For at dette skal kunne lade sig gøre, skal `dsPOAGUIRequest`, som nævnt, først være fyldt. Dette skal fungere ved, at kaldet til `ServiceGUIRequest()` returnerer et `DataSet`, som bruges til at fylde `dsPOAGUIRequest`, som herefter skal indeholde oplysninger om de `Forms` og `Controls`, der skal oprettes, og hvilket sprog deres tekstindhold er på. `dsPOAGUIRequest` skal indeholde `DataTables`, hvis `DataRows` har de nødvendige oplysninger.

Herefter skal `dsPOAGUIRequest` gennemlæses, og informationerne om de elementer, der skal oprettes skal hentes fra hhv. `dtPOAForm` og `dtPOAControl`, som det fremgår af Figur 19. `dtPOAControl` indeholder alle oplysninger vedrørende, hvilke `Controls`, der skal oprettes. De forskellige `Forms`, der er specificerede i `dtPOAForm` læses ind i et `Dictionary` objekt (se afsnit 2.2.2 om *generic* `Collections`).

Konceptet med anvendelse af `Dictionaries` er helt central for den måde, hvorpå den dynamiske GUI skal konstrueres. Som nævnt er det første, der skal ske efter at `DataSettet` indeholdende GUI elementer er læst fra *Business Logic Layer*, at alle Forms indlæses i et `Form Dictionary`. Dette sker ved, at en metode der skal kaldes `CreateForms` med parameteren "MAIN_FORM". MAIN_FORM defineres til altid at være det første skærmbillede, der vises.

`CreateForms` gennemløber `dtPOAForm` og opretter alle `Forms` i `Form Dictionariet`. Inden dette sker skal `Dictionariet` dog tømmes. Herefter skal funktionen kalde `CreateControls` med den `Form`, den selv modtog som inputparameter. Ved første kald vil der således her være tale om MAIN_FORM.

`CreateControls` gennemlæser `dtPOAControl` og herefter skal den gennemløbe et `switch` statement, hvor der `cases` på hvilken `Control` type, der angivet. Alt efter dette skal de forskellige rækker indlæses i hver deres `Control Dictionary`. Her skal der altså være tale om et `Dictionary` til alle de typer, der er defineret. Inden dette sker skal alle `Dictionaries` dog tømmes. Der er som nævnt en lang række forskellige typer `Controls`, der skal prædefineres. Disse skal alle være repræsenteret ved en `case` i `switch` statementet. I tilfældet med `LabelBind` objekter skal der yderligere tjekkes, om den konkrete `DataRow` med `LabelBind` indeholder et objekt, der hører til det aktive `valg` af data (fx hvis der tidligere er blevet valgt "Operationer", skal der kun vises `LabelBind` objekter, der er hører til dette valg). Under deres respektives `cases` oprettes de forskellige `Controls` som objekter i deres tilhørende `Dictionaries`. Herefter sættes deres forskellige properties. For alle `Controls` vedkommende skal der sættes `Size` og `Location` properties. For `Buttons` skal der tilknyttes `Delegates` og for `Buttons`, `Labels` og `TextBoxes` vedkommende skal der tilføjes et objekt til deres `Tag` property. `Tag` propertien bruges til at sætte et objekt, der indeholder informationer om objektet. Dette objekt kan være af alle slags typer. I forbindelse med dette projekt skal anvendes et særligt `POATag`-objekt, der indeholder properties til at sætte en alle de oplysninger, der er nødvendige på tværs af de forskellige objekter. `POATag` skal oprettes som et separat objekt, og der skal defineres et tilhørende klasse-interface der kaldes `IPOATag`. Alle `POATag` objekter skal udover dels tilføjes til deres respektive `Controls Tag`-felt og dels tilføjes til et `POATag Dictionary`, således at de er tilgængelige på tværs af objekterne. Det sidste skyldes, at en række af de forskellige objekter er knyttet funktionelt til andre objekter – her kan fx være tale om en `Button` hvis `Delegate` anvender indholdet fra en `TextBox`. I tilfælde af, at det senere viser sig at være ønskværdigt at tilføje nye objekttyper, gøres dette ved at oprette en `case` til dem, hvor de håndteres. `MainMenuForm` er *Creator* af `POATag`-objekter, og disse er *Information Expert* vedrørende alle oplysninger, der skal tilføjes til de forskellige `Controls' Tag`-*properties*. Dette kunne umiddelbart synes at være i modstrid med *High Cohesion*, da der her er tale om mange forskellige slags oplysninger. De har dog det til fælles, at de

skal indsættes i Tag-*properties* og anvendes til at knytte oplysninger til Controls. Endvidere udgør det, at der forekommer forskelligartede oplysninger ikke et problem, i forbindelse med tilføjelse af nye oplysninger. Dette skyldes, at der ingen *coupling* er mellem de *properties*, der anvendes til at tilskrive og aflæse oplysningerne.

MainMenuForm er endvidere ansvarlig for den overordnede *exception handling* for de tre lag. Som nævnt, modtager *Presentation Layer exceptions* fra de underliggende lag. Det drejer sig om ArgumentException, NullReferenceException, POAInterfaceException, POARequestException og POAGUIException. Herudover kan der også blive genereret *exceptions* i forbindelse med afvikling af *Presentation Layers* egen kode. Disse er WebExceptions i forbindelse med adgang til WSDataService (fx hvis der ikke er adgang til *web servicen*) og ArgumentExceptions ved indtastning i TextBoxe.

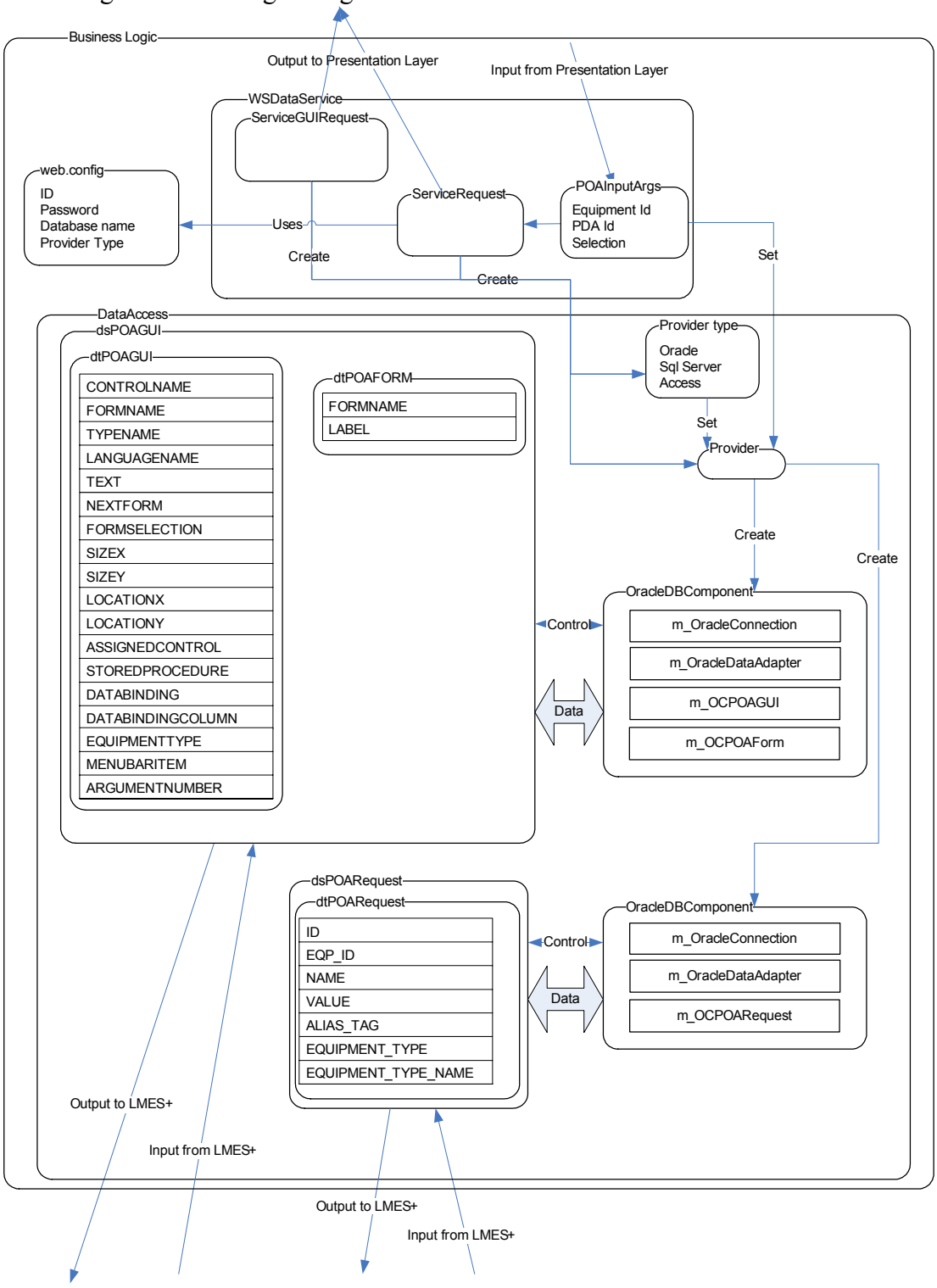
MainMenuForm skal håndtere Exceptions ved at der vises en MessageBox, der præsenterer indholdet af den opståede Exception.

6.6.5 dsPOARequest

dsPOARequest DataSettet skal indeholde udstyrsrelaterede oplysninger. På baggrund af hvilke valg, operatøren foretager, skal det fyldes op med relevant indhold. Af samme årsag er det ikke muligt på forhånd at definere, hvilke DataColumns det skal indeholde, da de DataColumns, som returneres, kan være forskellige både hvad angår antal og indholdet af disse. Dette kræver, at der i *Presentation Layer* eksisterer kode til at behandle disse forskelligartede oplysninger. Måden hvorpå det sikres, at de forskellige DataColumns-oplysninger bliver præsenteret de rigtige steder, er ved at benytte oplysningerne fra de forskellige Controls DATABINDING og DATABINDINGCOLUMN felter, som før beskrevet under dsPOAGUIRequest. DataSettet fyldes op ved at kalde WSDataServices ServiceRequest()-metode. Herefter løbes det igennem og de relevante oplysninger placeres i LabelBind-objekter på baggrund af hvad der er blevet specificeret i dsPOAGUIRequest og senere læst over i Dictionary-objekter. Som argument til ServiceRequest() sendes et string-array, der indeholder de argumenter, der er blevet angivet. Et eksempel på et sådant argument kunne være stregkoden, der indlæses i SCAN_BARCODE_FORM. I og med der kan være tale om et vilkårligt antal argumenter, der skal indlæses fra lige så mange TextBoxes, er det nødvendigt at anvende et List-objekt (se afsnit 2.2.2 om *Generic* Collections) til at opsamle disse, da et sådant objekt gør det nemt løbende at tilføje objekter. Dette List-objekt skal være af typen List<string> og skal kaldes POAInputArgs. Inden kaldet til ServiceRequest() skal indholdet af List-objektet lægges over i et array. Til at angive i hvilken rækkefølge argumenterne skal sendes til det underliggende system, og dermed i hvilken rækkefølge de skal ligge i arrayet, anvendes ARGUMENTNUMBER, som skal være knyttet til TextBoxene i tilfælde af flere argumenter.

6.7 Business Logic Layer

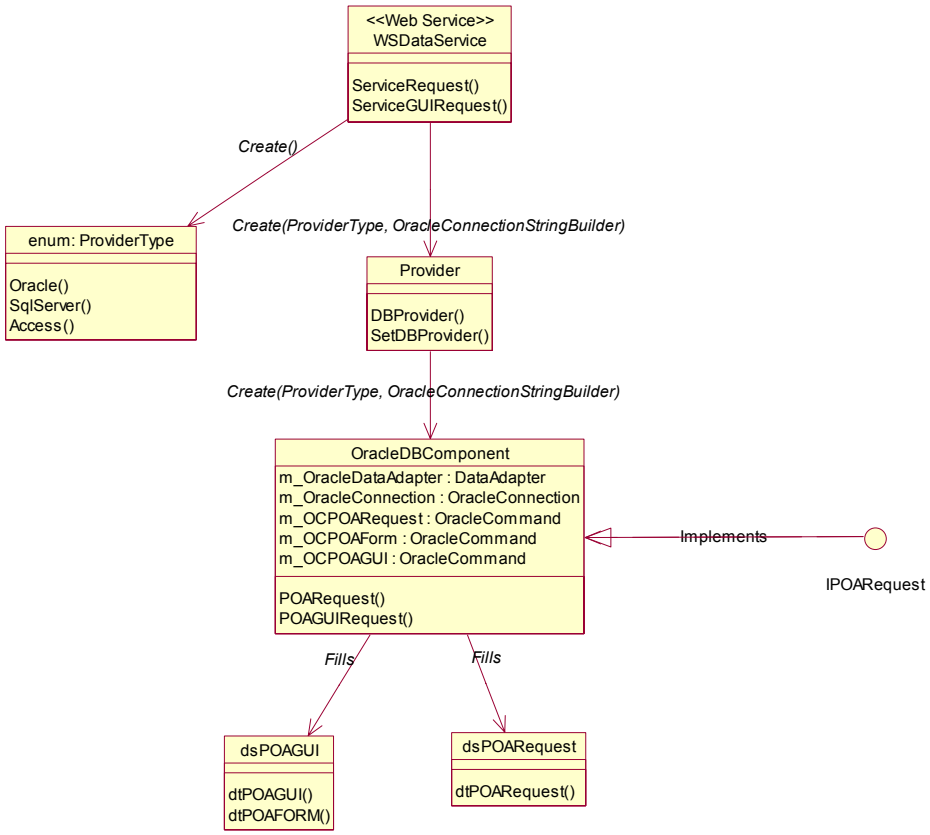
Figur 20 giver et overblik over de funktionelle enheder, der skal indgå i *Business Logic Layer*. Figuren tager udgangspunkt i Hent Udstyrsstatus (*use-case* #1). Formålet med figuren er, ligesom det var tilfældet med Figur 19, at give et lidt mere specifikt billede af kommunikationsflowet i *Business Logic Layer*, end klassediagrammet fra Figur 16 giver.



Figur 20 – Business Logic Layer design diagram

Som det fremgår af Figur 20, skal der være to centrale enheder i *Business Logic Layer*; `WSDataService` og `DataAccess`. Disse to enheder skal udgøre to selvstændige dellag i *Business Logic Layer*, og de skal indgå som separate *projects* i den samlede *solution*, hvor de skal udgøre to adskilte *assemblies*, men fysisk placeres og afvikles samme sted. `DataAccess` *assemblyet* skal eksistere som et *Class Library* i sin endelige udgave, hvilket betyder at projektet skal *buildes* som en `DLL`-fil, som skal hedde `POADataAccess.dll`. Dette betyder, at andre dele af applikationen på den måde kan anvende de klasser, som `DataAccess` indeholder, ved at inkludere `POADataAccess.dll` som en *Reference*, og derefter anvende denne som en tredjeparts komponent. Dette skal gøres i `WSDataService`, der således kan anvende de *public*-metoder som er tilgængelige i `DataAccess`.

For at give et bedre overblik, viser klassediagrammet på Figur 21 et udsnit af Figur 16, hvor kun klasser, der skal indgå i *Business Logic Layer* er medtaget.



Figur 21 - Klassediagram for Business Logic

En introducerende beskrivelse af de enheder, der fremgår af Figur 21, ses af nedenstående sammen med en beskrivelse af deres indbyrdes kommunikation. En mere detaljeret beskrivelse skal findes i afsnit 6.7.1 og 6.7.2.

`IPOARequest` er et *klasse-interface* som benyttes til at definere indholdet af `OracleDBComponent`. Dette *klasse-interface* bliver beskrevet yderligere i forbindelse med gennemgangen af `DataAccess` (afsnit 6.7.2). Der er ikke behov for et *klasse-interface* til `Provider`-klassen, da dennes primære funktion er at oprette objekter der alle har implementeret `IPOARequest`-interfacet.

Herudover skal der defineres tre komponenter: `OracleDBComponent`, `dsPOARequest` og `dsPOAGUI`, som adskiller sig fra almindelige klasser, da disse skal defineres som henholdsvis en `Component` og to `DataSets`. `OracleDBComponent` skal håndtere en række funktioner rettet mod databasetilgangen. Det skal her bemærkes, at `OracleDBComponent` udelukkende kan bruges til at tilgå en Oracle database, og hvis en anden type database skal anvendes, skal der implementeres en `Component` til dette formål. Det kan fx være *Microsoft SQL Server*, hvor det vil være nødvendigt at implementere en `SQLServerDBComponent`, som har de samme public-metoder som `OracleDBComponent`. Dette styres vha. `IPOARequest` *klasse-interfacet*, som skal implementeres af en given databasekomponent.

`dsPOARequest` og `dsPOAGUI DataSettene` skal indeholde en lokal udgave af de relevante informationer hentet fra databasen. For `dsPOARequests` vedkommende, skal data lagres i `DataTablen dtPOARequest`. `dsPOAGUI` skal bruges til at lagre GUI-informationer. Til dette formål skal `dsPOAGUI` indeholde to `DataTables`, nemlig `dtPOAGUI` og `dtPOAFORM`. Disse skal benyttes til at opbygge brugergrænsefladen på *Presentation Layer*. Både `OracleDBomponent`, samt de to `DataSets` vil blive yderligere beskrevet i afsnit 6.7.2 om `DataAccess`. `WSDataService` skal implementeres som en *web service* der har to `public`-metoder, `ServiceRequest` og `ServiceGUIRequest`, som kan tilgås fra *Presentation Layer*. Det er på via disse metoder, at komponenterne i `DataAccess` bliver benyttet. Det er `WSDataService` som skal være indgangen til *Business Logic Layer*, og denne vil blive beskrevet i afsnit 6.7.1.

6.7.1 WSDataService

`WSDataService` skal, som nævnt, fungere som indgangen fra *Presentation Layer* til *Business Logic Layer*. Idet disse er fysisk adskilte, skal kommunikationen herimellem foregå via et WiFi-netværk. Dette skal implementeres ved, at `WSDataService` oprettes som en *web service*, der skal servicere *Presentation Layer*. De informationer der skal være tilgængelige for dette lag, er både data til dynamisk opbygning af GUI'en (som beskrevet i afsnittene 6.6.2, 6.6.3 og 6.6.4), og resultatet af en given forespørgsel (som beskrevet i afsnit 6.6.5). For at opfylde de krav, *Presentation Layer* stiller, skal `WSDataService` have to metoder - `ServiceRequest()` og `ServiceGUIRequest()`. Disse skal implementeres som `WebMethods`, hvilket muliggør kald af dem fra *Presentation Layer*.

Det er ligeledes i *web servicens* konfigurationsfil, at databaseinformationerne skal forefindes. Konfigurationsfilen, `web.config`, er en tekst fil baseret på XML-tags. Der kan foretages ændringer i denne, mens applikationen kører, og disse vil straks have konsekvenser for applikationen uden at denne skal genstartes eller rekompileres. Der skal oprettes et `connectionStrings`-element, hvori `ProviderType` og `ConnectionType` er defineret. Det er `WSDataService` der skal sende disse databaseinformationer ned til `DataAccess`.

`ServiceRequest()`, skal tage et `List`-objekt samt en *stored procedure* som input-parametre, og returnere et `DataSet`. `List`-objektet skal indeholde et array af strenge, der skal bruges som et skalerbart antal af input argumenter til den *stored procedure* der skal kaldes. Når `ServiceRequest()` kaldes, skal `ConnectionString` indlæses fra web.config. Disse informationer skal derefter

tilføjes til et specifikt `ConnectionStringBuilder`-objekt, der på baggrund af databasetypen (`ProviderType`) videresendes til den respektive databasekomponent i `DataAccess`. Hvis der sker en fejl under tilskrivningen af `ConnectionStringBuilder`-objektet skal en `ArgumentException` `throwes`. Herefter oprettes et `Provider`-objekt, der tager `ProviderType` og `ConnectionStringBuilder` som input-parametre.

Da der ønskes en funktionel løsning baseret på en Oracle database, er det et `OracleConnectionStringBuilder`-objekt der skal oprettes af *web servicen*. Da der kan være tale om forskellige databasetyper, er det ligeledes forskellige databasekomponenter der bliver oprettet på baggrund af netop databasetypen. Derfor er det nødvendigt at tjekke, om den oprettede komponent understøtter det korrekte klasse-interface, inden de tilgængelige metoder anvendes. Hvis dette ikke er tilfældet, skal der `throwes` en `POAInterfaceException`. Se afsnit 6.5 for beskrivelse af *exception handling*.

Herefter kaldes den relevante *public*-metode som `ServiceRequest()` skal anvende; dvs. `POARequest()`, som findes i databasekomponenten. Denne metode skal have to input argumenter, de tidligere beskrevet `List`-objekt og *stored procedure*, samt to output-parametre (dvs. `Out`-parametre). Det første output skal være et generisk `DataSet` kaldet `dsPOARequest`, der kan indeholde et hvilken som helst resultat, der måtte være behov for at vise på *Presentation Layer*. Det andet output skal være en streng der bliver tilskrevet, hvis der sker en fejl længere nede i applikationen. Denne skal `throwes` som en `POARequestException`, hvis der opstår en fejl. Det generiske `DataSet` skal returneres til *Presentation Layer*, hvor det efterfølgende kan analyseres og anvendes til at vise oplysninger om henholdsvis udstyrsstatus (*use-case* #1) og udstyrsoperationer (*use-case* #2).

`ServiceGUIRequest()`-metoden bruges til at fylde `DataSettet` `dsPOAGUI`, som anvendes til at generere den dynamisk GUI. Når metoden er kaldt, og `DataSettet` er fyldt med data fra databasen, returneres disse til *Presentation Layer*, der herefter kan fortolke dataet og præsentere det korrekt. Strukturen for `ServiceGUIRequest()`-metoden skal være den samme som for `ServiceRequest()`. Dog skal `ServiceGUIRequest()` ikke have nogen inputargumenter. Derimod skal det være et `DataSet` af typen `dsPOAGUI`, som skal returneres. Dette skyldes at *Schema* for `dsPOAGUI` ikke skal ændres, da det altid er de samme `Columns` fra databasen der skal returneres. En fordel ved dette er, at `dsPOAGUI` kan defineres som et *strong typed* `DataSet`, og derved opnås en større typesikkerhed, når der arbejdes med `DataSettet`.

`ProviderType` og `Provider` benyttes af begge web metoder, defineres i `DataAccess`-projektet, og vil derfor blive diskuteret i afsnit 6.7.2.

6.7.2 DataAccess

Figur 21 viser den overordnede opbygning af *Business Logic Layer* og herunder `DataAccess`. *Disconnected* modellen, som er gennemgået i afsnit 2.8, skal anvendes som den overordnede designmodel for kommunikationen til *Data Storage Layer*. Som nævnt i beskrivelsen af *Business Logic Layer* indeholder delkomponenterne i `DataAccess` flere elementer, der kræver en uddybende forklaring. Overordnet består `DataAccess` af en `Enum` ved navn `ProviderType`, klassen `Provider`, `Componenten` `OracleDBComponent` og `DataSettene` `dsPOARequest` og `dsPOAGUI`.

For ikke på lang sigt at gøre sig afhængig af én bestemt dataudbyder, skal der laves en generisk databasetilgang, hvor der anvendes en klart defineret facade. Her skal *Facade*-pattern anvendes, så `DataAccess` *assembliet* designes, så det tilbyder et simpelt interface til et komplekst underliggende system. Se evt. Design Patterns in C# [4][25].

`DataAccess` har to primære ansvarsområder:

1. Database sikkerhed
2. Håndtering af SQL afvikling

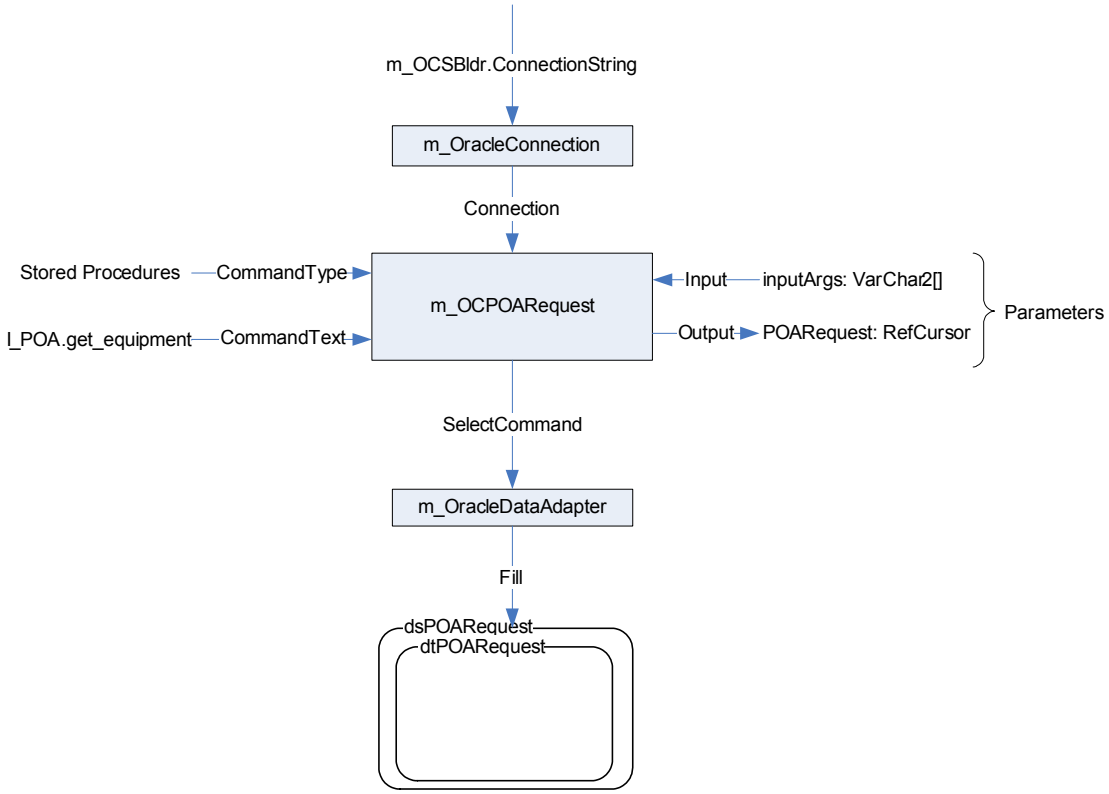
For at sikre kontrol med databasetilgangen skal *connection*-informationerne, der bruges til at skabe forbindelse til databasen, sendes som et `ConnectionStringBuilder`-objekt fra `WSDataService`. På den måde bliver sårbare informationer omkring databasen adskilt fra koden til `DataAccess`. I `WSDataService` benyttes `ProviderType`, som indeholder de tilgængelige databasetyper, til at angive hvilken type databasekomponent der skal laves en instans af. `ProviderType`-enumen skal eksistere `DataAccess`. På baggrund af det `ConnectionStringBuilder`-objekt og `ProviderTypen`, der sendes fra `WSDataService`, udvælger og opsætter `Provider`-klassen den relevante database-*connection*. Dette skal ske på en måde så udvikleren ikke skal tage stilling til andet end databasetype og `connectionString`, der som tidligere nævnt defineres i `web.config` i `WSDataService`. I forbindelse med dette projekt, er der valgt at give mulighed for at anvende en `Oracle`, `MS SQL Server` eller `Access` databasetype; dette gøres vha. `ProviderType`. Reelt er det dog kun `Oracle` der vil kunne anvendes, da LMES+ udelukkende anvender en `Oracle` database, og applikationen skal designes herefter. Der skal derfor oprettes et objekt af `OracleDBComponent`-klassen. På `WSDataService`, som skal have adgang til `DataAccess'` *public*-metoder, skal det kun være muligt at oprette et generisk `DBComponent`-objekt, hvor `DBProvider` er den eneste `public`-metode der kan tilgås. I kraft af dette design kan `WSDataService` ikke se, hvilken `DBComponent` der laves en instans af. Der er altså tale om en indkapsling af den konkrete databasebehandling.

`OracleDBComponent` er en relativt kompleks software-klasse, da den skal bestå af fem objekter af tre forskellige typer:

1. `m_OracleConnection` – af typen `OracleConnection()`
2. `m_OCPOARequest` – af typen `OracleCommand()`
3. `m_OCPOAForm` – af typen `OracleCommand()`
4. `m_OCPOAGUI` – af typen `OracleCommand()`
5. `m_OracleDataAdapter` – af typen `OracleDataAdapter()`

Disse fem objekter skal anvendes af to *public*-metoder – `POARequest()` og `POAGUIRequest()`. For at benytte objekterne til at hente GUI-information skal `POAGUIRequest()` kaldes, og `POARequest()` skal kaldes for at hente LMES+ relaterede informationer. Det er vigtigt at opfatte `OracleDBComponent` som en helhed, og ikke opdele komponenten i enkeltopgaver for de forskellige objekter, da objekterne bliver brugt i nær relation til hinanden. En illustration af anvendelsen af `OracleDBComponent` kan ses på Figur 22, som tager udgangspunkt i *Hent Udstyrsstatus* (*use-case* #1) – og derved `POARequest()`. Dette betyder at `m_OCPOAForm` og `m_OCPOAGUI` ikke ses på figuren.

[25] Side 35-37



Figur 22 - OracleDBComponent

For at lette beskrivelsen af `OracleDBComponent` vises ligeledes en kort kodeliste med tilskrivninger af de elementer der skal indgå i komponenten, som den efterfølgende beskrivelse kan sammenlignes med for overskuelighedens skyld. Dog vises *Exception handling* ikke. Den efterfølgende beskrivelse relaterer således både til Figur 22 og til Liste 4.

```
1  //Set the m_OracleConnection to the OracleConnectionStringBuilder.
2  // ConnectionString received from the web.config
3  m_OracleConnection.ConnectionString = OCSBldr.ConnectionString;
4
5  // Set the correct Connection-object to the m_OCPOARequest Command
6  m_OCPOARequest.Connection = m_OracleConnection;
7
8  // m_OCPOARequest info for the input and outputparameter
9  Direction = ParameterDirection.Input;
10 OracleDbType = OracleDbType.Varchar2;
11 ParameterName = "inputArgs";
12 Direction = ParameterDirection.Output;
13 OracleDbType = OracleDbType.RefCursor;
14 ParameterName = "POARequest";
15
16 m_OCPOARequest.CommandText = storedProcedure;
17 m_OCPOARequest.CommandType = CommandType.StoredProcedure;
18
19 //Fill the dtResult DataTable with the result.
20 m_OracleDataAdapter.Fill(dsPOARequest, "dtPOARequest");
```

Liste 4 - OracleDBComponent

`OracleDBComponent` skal oprette et `m_OracleConnection`-objekt, som bruges til at lave forbindelsen til databasen med de informationer der er modtaget fra `web.config` i `WSDataService`. Det er `ConnectionStringBuilderens ConnectionString`-property (der er modtaget som inputargument til *constructoren*), der skal sættes lig med `m_OracleConnections` egen `ConnectionString` i `OracleDBComponenten`. Hvis dette ikke går godt, skal der `throwes` en `ArgumentNullException`, der skal håndteres på et højere niveau. Herefter skal `m_OCPOARequest` sættes til at pege på `OracleConnection`, dette ses på linie 1-6 i Liste 4. I `OracleCommanden m_OCPOARequest` skal der defineres to parametre – et input og et output. Inputparameteren `inputArgs` er et array af `VarChar2` (strenge i PL/SQL), og skal tilskrives med `List`-objektet (der også hedder `inputArgs`) vha. dette objekts `ToArray()`-metode. Dette bruges til at sende et input med dynamisk størrelse til den relevante *stored procedure* på databasen. Resultatet fra den eksekverede *stored procedure* på databasen, skal lagres i en `RefCursor`, der efterfølgende kan tilskrives til `dtPOARequest DataTablen`. Både `CommandType` og `CommandText` i `m_OCPOARequest` skal tilskrive. `CommandType` sættes til `StoredProcedure`, som specificerer hvorledes `CommandText`-strengen fortolkes – altså som en *stored procedure*. `CommandText` tilskrives med inputparameteren `storedProcedure`, og definerer derved den konkrete *stored procedure* som skal afvikles på databasen. Er der fx tale om en udstyrsstatusforespørgsel, skal `OracleDBComponent` modtage `I_POA.get_equipment`. Det sidste led i komponenten er `OracleDataAdapter`, der bruges til at fylde `DataTablen dtPOARequest` i `DataSettet dsPOARequest` op med de relevante data.

`POAGUIRequest()` skal implementeres vha. samme designprincip som er beskrevet for `POARequest()`. Forskellen er, som beskrevet i afsnit 6.7.1, at `POAGUIRequest()` kun har outputparametre. For dennes vedkommende bliver resultatet returneret som et `DataSet` af typen `dsPOAGUI`. Dette fyldes vha. `OracleDataAdapteren`. I `dsPOAGUI` skal der fyldes to `DataTables` for at repræsentere GUI'en, nemlig `dtPOAGUI` og `dtPOAForm`. Dette skal ligeledes gøres, som beskrevet for `POARequest()`.

Til at hente de nødvendige informationer til *Presentation Layer*, skal der i Oracle databasen oprettes et antal *stored procedures* som kan tilgås ved kald igennem `OracleDBComponent`. Følgende *stored procedures* skal implementeres på databasen:

1. `get_equipment`
2. `get_operation`
3. `get_poa_gui`
4. `get_poa_formname`

Disse kan opdeles i forskellige grupper typer af *stored procedures*. `get_equipment` og `get_operation` har med databehandling at gøre, hvilket vil sige at disse returnerer resultatet fra en forespørgsel som en operatør har foretaget, hvilket gøres igennem `POARequest()`. Begge disse *stored procedures* skal have et inputargument af typen `VarChar2`, hvor antallet er af variabel længde (i form af et array). Ligeledes skal begge *stored procedures* have outputargumenter af typen `RefCursor`, som derved kan gennemløbes og skrives over i et `DataSet`. `get_poa_formname` og `get_poa_gui` anvendes til opbygning af GUI'en – ved brug af `POAGUIRequest()`. Da `POAGUIRequest()` ikke har inputargumenter,

er det kun nødvendigt at have et outputargument for de *stored procedures* som skal anvendes under denne metode. Dette skal være af typen `RefCursor`. Disse *stored procedures* har som nævnt til formål at servicere de krav, som *Presentation Layer* stiller til `WSDataService`. Dette kræver udover de ovenfor nævnte *stored procedures*, og de `Tables`, som i forvejen eksisterer i LMES+, at der oprettes en række `Tables` på databasen. De nødvendige `Tables` er:

1. `POA_FORM`
2. `POA_CONTROL`
3. `POA_TYPE`
4. `POA_LANGUAGE`

Ved udarbejdelsen af interaktionsdiagrammet for Hent Udstyrsstatus og Hent Udstyrsoperation, blev LMES+ identificeret som værende *Information Expert* vedrørende udstyrsoplysninger. LMES+ leverer data til `DataAccess` ved hjælp af *stored procedures*. Klasserne er udarbejdet under hensyntagen til forskrifterne i design pattern *Low Coupling* og *High Cohesion*. At klasserne har *High Cohesion*, ses af, at de er tildelt ansvarsområder, der er tæt relaterede til hinanden. Et eksempel på dette er `OracleDBComponent`, der som eneste ansvarsområde har at sørge for kommunikationen mellem applikationen og LMES+.

6.8 Data Storage

Data Storage udgøres, i denne applikation, af LMES+ Oracle databasen. I forbindelse med denne applikation, skal der dels anvendes oplysninger om forskelligt udstyr (*use-case #1 og use-case #2*) og oplysninger til brug ved dynamisk opbygning af GUI (afsnit 5.5). Begge typer information skal findes i LMES+ databasen. Alle ønskede oplysninger om udstyr findes i forvejen i form af `Tables`, hvorimod GUI oplysninger endnu ikke findes, hvorfor den nødvendige schema-struktur i form af `Tables` med tilhørende `Collumns` skal opbygges og indhold skal defineres for disse. Udover at de forskellige `Tables` skal forefindes, er det også nødvendigt at definere en række *stored procedures*, som afvikler SQL-forespørgsler. Disse *stored procedures* skal kaldes af `DataAccess`. Dette gør sig gældende for såvel udstyrsoplysninger som for GUI-oplysninger. Således er der tale om en vis grad af funktionsudbygning på den eksisterende database. Dette vil blive gennemgået i de følgende afsnit, hvor der tages udgangspunkt i teorien omkring opbygning af databaser, se afsnit 1.2 og Database Management Systems [18].

Opbygningen af en database består af flere trin. I og med en række ting er defineret på forhånd, idet der i forbindelse med denne applikation er tale om tilføjelser til en allerede eksisterende database, er der i forbindelse med *Data Storage Layer* kun lagt vægt på dele af databasedesignprocessen. I den forbindelse er det kun disciplinerne *Requirements Analysis*, *Conceptual Database Design* og *Logical Database Design* [18] der vil blive gennemført. På længere sigt, skal aspekter som raffinering af de tilføjede `Tables` og *stored procedures* samt sikkerhed overvejes, men det vil der ikke blive lagt vægt på i dette projekt. Det skal bemærkes, at *Requirements Analysis* allerede er gennemført i form af de krav, der er blevet specificeret i *Presentation Layer*.

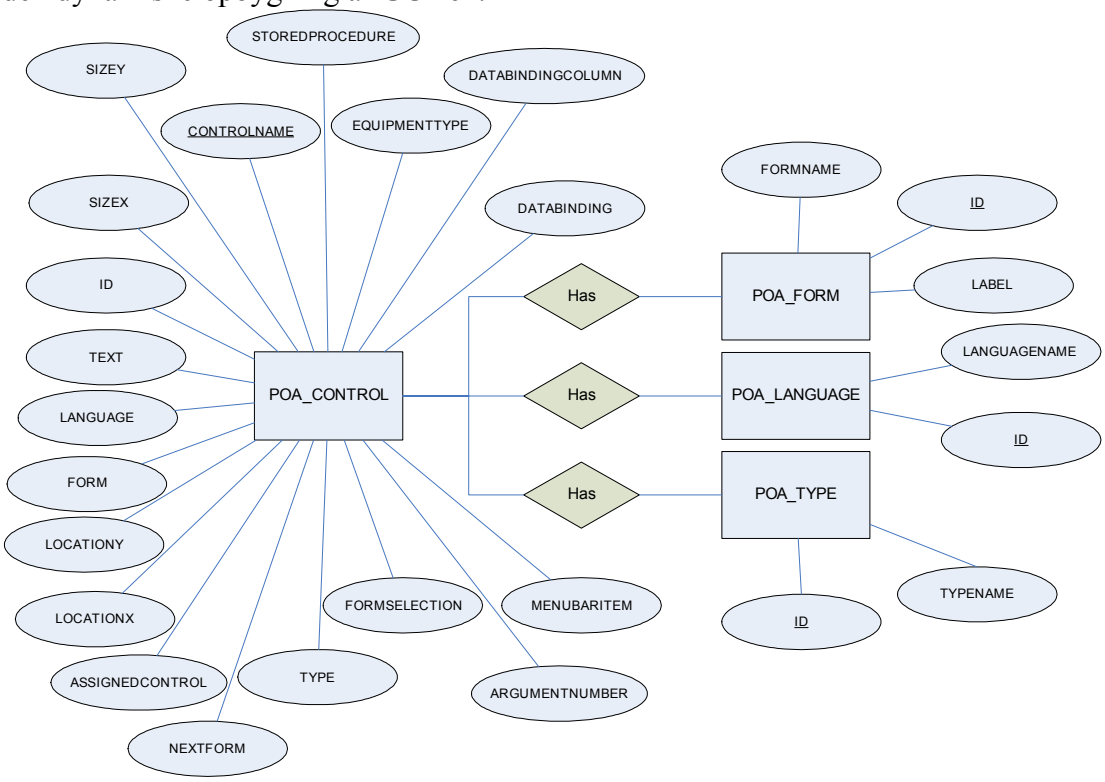
I det følgende afsnit gennemgås opbygningen af *Data Storage Layer* med udgangspunkt i de nævnte discipliner. Det overordnede mål med dette lag er at indfri kravene fra *Business Logic Layer* (afsnit 6.7). Disse krav anvendes som udgangspunktet for *Conceptual Database Design*. De to efterfølgende trin behandler så det databasespecifikke, som er nødvendigt for implementering af dette.

Således er der i de følgende afsnit både en beskrivelse af noget allerede foreliggende teknologi, som skal anvendes, samt noget nyt der skal føjes hertil og samspillet mellem disse to elementer.

De følgende afsnit om *Data Storage Layer* er altså en beskrivelse af de tilføjede elementer, og sammenspillet mellem disse og det eksisterende system.

6.8.1 Conceptual Database Design

Figur 23 viser et såkaldt ER-diagram for de ønskede tilføjelser til databasen. Dette beskriver, på et højt abstraktionsniveau, de designkrav til databasen, som omhandler den dynamiske opbygning af GUI'en.



Figur 23 – ER-diagram over de nødvendige tilføjelser til LMES+

ER-diagrammet på Figur 23 skal læses fra venstre mod højre. Af figuren ses hhv. `POA_FORM`, `POA_LANGUAGE`, `POA_TYPE` og `POA_CONTROL`. Hver af disse udgør et såkaldt *Entity Set*, og de har hver især et antal *Attributes* – fx ID.

`POA_CONTROL` er det centrale *Entity Set* og indeholder den data, der skal sendes videre til *Business Logic Layer* i forbindelse med dynamisk opbygning af GUI. Af figuren kan yderligere ses de relationer, der eksisterer mellem de forskellige *Entity Sets*. Disse er repræsenteret i form af såkaldte *Relationship Sets*. Som det kan ses, eksisterer der tre sådanne, som illustrerer relationerne mellem `POA_CONTROL` og de andre tre `Tables`. Fx kan det ses, at `POA_CONTROL` har en Has-relation til `POA_TYPE`. Dette skal forstås sådan, at de værdier, der defineres i `POA_CONTROLs` TYPE-felt er defineret ud fra den mængde af TYPES, som er defineret i `POA_TYPE`. Som det kan ses, er ID-feltet i `POA_TYPE` understreget, hvilket indikerer, at dette felt er *Primary Key* for denne `Table`. En relation er altid mellem et felt i en `Table` og *Primary Key* i en anden `Table`. Feltet, som har en relation til en anden `Tables` *Primary Key* siges at være *Foreign Key* i denne `Table`.

Ved at oprette `POA_TYPE` og `POA_LANGUAGE` som separate `Tables`, sikres det, at de forskellige informationer her eksisterer uafhængigt af, hvilke `Controls`, der på et givet tidspunkt er oprettet i `POAControl`.

6.8.2 Logical Database Design

Idet der tages udgangspunkt i ER-diagrammet på Figur 23 er det muligt at lave en mere detaljeret databasemodel kaldet *Relational database Schema*. I dette projekt anvendes som før nævnt en Oracle database, og i det følgende gennemgås, hvilke `Tables` og *stored procedures*, der skal oprettes for at indfri kravene fra *Presentation Layer*.

6.8.2.1 Tables

Som ER-diagrammet skitserer skal der oprettes fire `Tables` til brug ved opbygning af GUI'en. I forlængelse af Tabel 9 og Tabel 10 kan følgende krav for de fire `Tables` POA_FORM, POA_CONTROL, POA_TYPE og POA_LANGUAGE opstilles, hvor det fremgår hvilke datatyper, de forskellige felter skal have. De datatyper, der anvendes, er definerede i forhold til Oracle*s* PL/SQL-sprog, der, som nævnt, er en Oracle-specifik udvidelse af SQL-sproget.

Rækkenavn	Keys	Datatype	Nullable	Default value
ID (autoincrement)	Primary	NUMBER	No	0
FORMNAME	Unique	VARCHAR2(50)	No	
LABEL	Unique	VARCHAR2(50)	Yes	

Tabel 11 - POA_FORM-design

Rækkenavn	Keys	Datatype	Må antage nul-værdi	Default value
CONTROLNAME	Primary	VARCHAR2(200)	N	
TYPE	Foreign	NUMBER	N	
FORM	Foreign	NUMBER	N	
LANGUAGE	Foreign	NUMBER	N	
TEXT		VARCHAR2(2000)	Y	
ID (autoincrement)	Unique	NUMBER	N	0
NEXTFORM	Foreign	NUMBER	Y	
FORMSELECTION	Foreign	NUMBER	Y	
SIZEX		NUMBER	Y	
SIZEY		NUMBER	Y	
LOCATIONX		NUMBER	Y	
LOCATIONY		NUMBER	Y	
ASSIGNEDCONTROL		VARCHAR2(200)	Y	
STOREDPROCEDURE		VARCHAR2(200)	Y	
DATABINDINGCOLUMN		NUMBER	Y	
DATABINDING		VARCHAR2(200)	Y	
EQUIPMENTTYPE		VARCHAR2(200)	N	
MENUBARITEM		VARCHAR2(200)	Y	
ARGUMENTNUMBER		NUMBER	Y	

Tabel 12 - POA_CONTROL-design

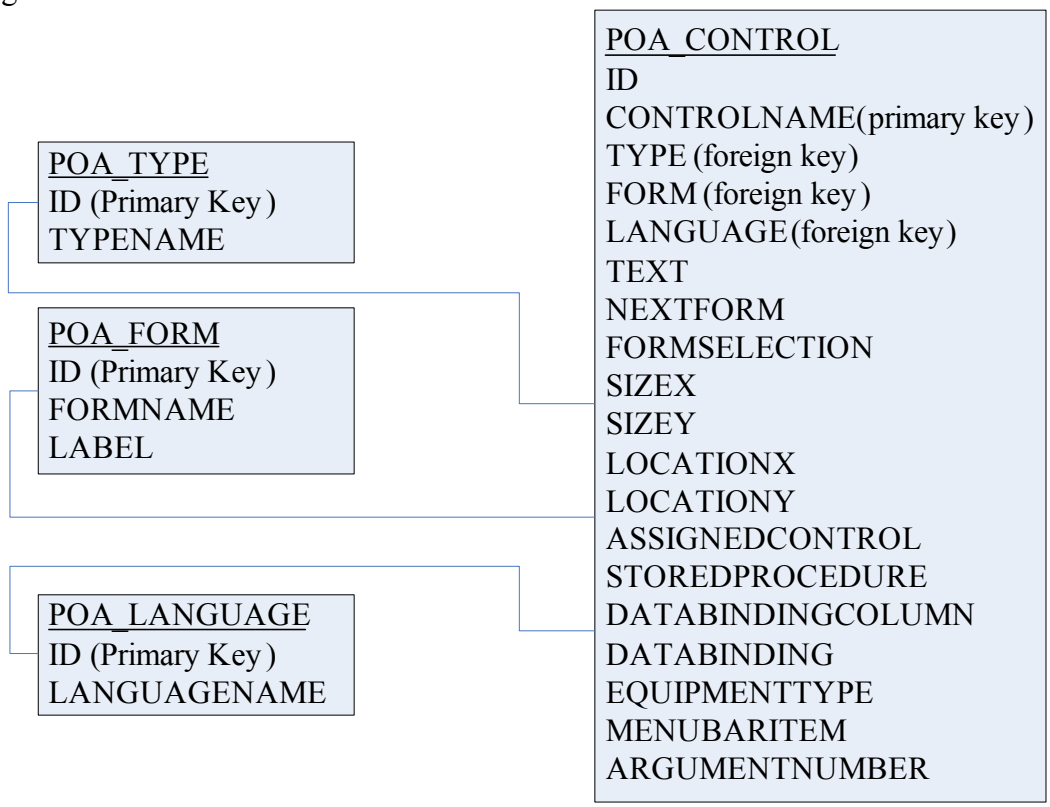
Rækkenavn	Keys	Datatype	Nullable	Default value
ID (autoincrement)	Primary	NUMBER	No	0
TYPENAME	Unique	VARCHAR2(100)	No	

Tabel 13 - POA_TYPE-design

Rækkenavn	Keys	Datatype	Nullable	Default value
ID (autoincrement)	Primary	NUMBER	No	0
LANGUAGENAME	Unique	VARCHAR2(50)	No	

Tabel 14 - POA_LANGUAGE-design

Figur 24 viser relationerne mellem de førnævnte Tables.



Figur 24 – Relational Schema

Af Figur 24 fremgår, at de respektive ID-felter er *Primary Key* for såvel POA_FORM, POA_TYPE og POA_CONTROL. Som det endvidere ses, skal ID-felterne autoinkrementeres. Til dette formål skal der implementeres en såkaldt `Sequence`, som skal kaldes `POA_AUTOINC_SEQ`. I denne skal størrelsen på inkrementeringen (i dette tilfælde 1) defineres. Ligeledes skal der angives minimum- og maksimumværdier. Disse sættes til hhv. 0 og uendeligt. Denne anvender en *cache*, der tildeler de processer, der benytter `Sequencen` de næste 20 tal fra den senest anvendte værdi. Dette gøres for at sikre, at de forskellige processer ikke inkrementerer til samme værdi. Dette medfører dog, at der kommer spring i de værdier, `Sequencen` returner, såfremt hele *cachen* ikke benyttes. Denne `Sequence` skal anvendes af alle `Tables` til at inkrementere deres ID-felt. Dette gøres ved, at der til de respektive felter tilføjes en `Trigger`, der kalder `POA_AUTOINC_SEQ` hver gang, der tilføjes en ny `Row`. Der skal derfor defineres fire `Triggers`, der skal kaldes `POA_CONTROL_AUTOINC_TRIGGER`, `POA_TYPE_AUTOINC_TRIGGER`, `POA_LANGUAGE_AUTOINC_TRIGGER` og `POA_FORM_AUTOINC_TRIGGER`.

6.8.2.2 Stored procedures

De *stored procedures*, der skal defineres for at servicere *Presentation Layer* skal defineres ved brug af PL/SQL-systemet (se afsnit 2.10). Ved lagring af *stored*

procedures i Oracle databasen, skal der oprettes en `Package` med navnet `I_POA`, hvor alle de *stored procedures* der tilhører POA-applikationen vil eksistere. På denne måde adskilles de *stored procedures*, som hører til POA logisk fra resten af de *stored procedures*, som er defineret i LMES+. En sådan funktionel inddeling i `Packages` går i øvrigt igen hele vejen i opbygningen af LMES+ databasen. I det følgende beskrives de nødvendige *stored procedures*. Der vil også blive præsenteret noget pseudokode for at give et indblik i, hvorledes *stored procedures* opbygges i PL/SQL

Som nævnt i `DataAccess` (afsnit 6.7.2) skal de have følgende navne:

1. `get_poa_gui`
2. `get_poa_formname`
3. `get_equipment`
4. `get_operation`

`get_poa_gui` skal anvendes til dynamisk opbygning af GUI'en. Hertil skal anvendes en outputparameter ved navn `poa_gui`, som er af typen `ref cursor`. Denne skal returnere alt indholdet fra POA_CONTROL-`Tablen`, hvor indholdet af TYPE-, LANGUAGE- og FORM-felterne erstattes af det indhold, de referer i hhv. POA_TYPE, POA_FORM og POA_LANGUAGE.

`get_poa_formname` skal ligeledes anvendes til dynamisk opbygning af GUI'en. Hertil skal anvendes en outputparameter ved navn `poa_gui`, som er af typen `ref cursor`. Denne skal returnere indholdet af POA_FORM.

`get_equipment` skal som nævnt anvendes til at hente udstyrsstatus. Til det formål skal anvendes en intputparameter ved navn `inputArgs`. Denne skal være af typen `table of varchar2(255) index by BINARY_INTEGER`. Dette betyder, at det skal være en PL/SQL `Table`, som er indeksérbar, hvilket gør det muligt at sende et `string` array fra `DataAccess`. Derudover skal anvendes en outputparameter ved navn `POARequest`, som skal være af PL/SQL-typen `ref cursor`. Denne skal returnere data afhængigt af, hvad der er defineret i intputparameteren.

`get_operation` anvendes som nævnt til at hente udstyrsoperationer. Til det formål skal anvendes en intputparameter ved navn `inputArgs`. Denne skal være af typen `table of varchar2(255) index by BINARY_INTEGER`. Derudover skal anvendes en outputparameter ved navn `POARequest`, som skal være af PL/SQL-typen `ref cursor`. Denne skal returnere aktive udstyrsoperationer på det udstyr, der er defineret i intputparameteren.

6.8.2.2.1 Pseudokode

Den følgende pseudokode tager udgangspunkt i `get_equipment`:

```
Package name: I_POA
  Procedure get_equipment( input:  (Table)equipment_Id,
                           Output: (refCursor)equipment_status)
    Begin
     SELECT ID, EQP_ID, ALIAS_TAG, NAME, VALUE
       FROM EQP_STATUS
      WHERE EQP_ID = equipment_Id
    End
End
```

Liste 5 - Pseudokode for get_equipment

Som det ses, skal der først og fremmest defineres, hvilken `Package`, den pågældende *stored procedure* tilhører. Som nævnt, anvendes til dette projekt en `Package` kaldet `I_POA`. Dernæst specificeres, hvilke input- og outputparametre, der anvendes af *stored proceduren*. Herefter begyndes koden til *stored proceduren* med et `Begin` *statement* og afsluttes med et `End` *statement*. Imellem disse *statements* angives det eller de `Select` *statements*, der ønskes afviklet. Programmet afsluttes med et `End` *statement*.

6.9 Applikationsinstallation

Som nævnt skal applikationen installeres på PDA'en ved, at administratoren tilgår en hjemmeside, hvor der ligger en installationsfil. En sådan installationsfil genereres ved, at der i *Visual Studio* oprettes et *Smart Device CAB Project* som kaldes `POA`. Dette bruges til at generere en CAB-fil, som kan bruges i *Windows CE 5.0* miljøer til installation. Efter at have oprettet *projectet*, skal dettes output defineres som værende *Primary Output* fra kompileringen af `POAPresentation`. Under *File System* for *projectet* defineres *Start Menuen* på PDA'en som værende installationslokation.

Når *Smart Device CAB Projectet* herefter bliver kompileret, vil der blive genereret en CAB-fil. Denne skal der laves en reference til på en webserver. Denne webserver skal kunne tilgås ved hjælp af PDA'en. Herefter vil det være muligt at installere og opdatere POA-applikationen ved at pege på CAB-filen på webserverens URL. Installationen sættes automatisk i gang, når dette gøres, og administratoren skal blot sige god for at installere og evt. overskrive en tidligere version. Efter endt installation, kan POA-applikationen startes ved at vælge den fra *Start Menuen* på PDA'en.

6.10 Prototype

Til at begynde med skal implementeres en simpel prototype, der har til formål at teste at al kommunikationen virker fra *Presentation Layer* til LMES+ og tilbage. Denne har kun til formål at sikre, at de forskellige lag fungerer hensigtsmæssigt, og der tages defor kun udgangspunkt i Hent Udstyrsstatus.

Dette indebærer, at *Presentation Layer* implementeres uden at gøre brug af dynamisk GUI generering, men derimod med anvendelse af fast definerede `Controls`. *Business Logic Layer* skal ligeledes implementeres så simpelt som muligt, hvilket bl.a. indebærer, at der kun skal oprettes et `OracleCommand`-objekt, som tager en `string` som input i stedet for `List`-objektet, som anvendes ved ved

den fuldt udbyggede implementering. På *Data Storage Layer* skal kun implementeres en enkelt *stored procedure*, der returnerer udstyrsstatus for det ønskede udstyr.

7 Implementering

I dette kapitel gennemgås implementeringen af systemet. I den følgende tekst gennemgås .NET specifikke tiltag, som omfatter ting, der ikke er udspecificeret i designspecifikationen. Derudover henvises der til den vedlagte CD-ROM med bilag, hvor applikationens kildekode kan findes i *Develop*-biblioteket. Herudover er vedlagt MSDN-dokumentation (POADocumentation.chm) og almindelig web dokumentation for `POAPresentation` og `POADataAccess`. Denne er genereret på baggrund af XML-kommentarerne i disses kildekode ved brug af NDoc programmet. Da det ikke i *Visual Studio* er muligt at generere XML-kommentarer for en *web service*, henvises der til kildekoden for denne. Generelt er der yderligere kommentarer at finde i kildekoden til de forskellige programdele. Den XML-genererede dokumentation findes i *Dokumentation*-biblioteket på bilags CD-ROM'en. For yderligere kommentarer henvises til selve kildekoden. På CD-ROM'en findes endvidere kildekode til *deployment*-projektet (ligger i *Develop\POA\POA*) og til en Windows-udgave af `POAPresentation`.

7.1 Visual Studio

I forbindelse med implementeringen er *Visual Studio*, som nævnt, blevet anvendt. Dette har givet en lang række muligheder, idet dette system automatiserer en række ting, som der ellers skulle have været taget eksplicit stilling til. Fx sørger *Visual Studio* for at generere alle de nødvendige protokol- og meddelelsesfiler, som anvendes i forbindelse med *web services*.

Visual Studio indeholder endvidere en *designer*, som er blevet anvendt til at konstruere forbindelsen til databasen, som det er blevet nævnt i afsnit 6.7.2. Denne har gjort det muligt at oprette `Command`-objekter ved brug af *drag-and-drop* funktionalitet. Denne *designer* giver yderligere mulighed for automatisk at oprette *strongly typed* `DataSet` vha. oprettelse af et XML-*schema*, som nævnt i afsnit 2.8.3. Dette er fx gjort i forbindelse med `dsPOAGUI` og indebærer, at det i `DataAccess` er blevet specificeret, hvilke `DataTables`, dette indeholder og hvilke `DataColumns`, `DataRelations` og datatyper, det indeholder. Dette har givet mulighed for, at der på *Presentation Layer* tjekkes for, hvor vidt de forskellige `DataRows` indeholder null-værdier og hvis dette er tilfældet istedet anvende en default-værdi.

Til debug-formål har *Visual Studio* en glimrende indbygget debugger. Det er, som det sædvanligvis er tilfældet, muligt at indsætte *breakpoints*, hvorefter applikationsafviklingen stopper ved disse, hvis der køres i debug-mode. Ved debug er det muligt at tjekke værdierne på primitive variable, såvel som objekter og grave ned i de objekter, som de måtte indeholde. Ligeledes er det muligt i løbet af debugningen at oprette nye *breakpoints*, ligesom det er muligt at skifte fra normal run-mode til debug-mode, hvis en applikation crasher under afviklingen.

8 Test

Dette kapitel beskriver testforløbet. Der er blevet testet sideløbende med implementering, og de opnåede testresultater er indgået som led i de efterfølgende iterationer, hvor der igen er blevet implementeret og testet. Til brug ved specificering af de forskellige tests er *TestDirector* anvendt, og det er ligeledes herfra den følgende dokumentation af testene stammer.

Der vil blive udført en række forskellige tests. Først og fremmest testes, at de enkelte delkomponenter virker. Dette gøres i en række modultests. Disse test udgør afprøvning af specifikke dele af applikationens funktionalitet. Der ønskes foretaget følgende modultest:

Lag	Test af
Presentation Layer	`MainMenuForm` og dynamisk GUI opbygning.
Business Logic Layer	`WSDataServices ServiceRequest()-` og `ServiceGUIRequest()`-metoder og generiske database komponenter.
Data Storage Layer	*Stored procedures* og tilhørende hjælpefunktioner i form af *triggers*.

Tabel 15 - Modultests

Testene på *Presentation Layer* og *Business Logic Layer* foretages ved, at der oprettes små testprogrammer, der gennemløber de forskellige komponenters funktionalitet, og *stored procedures* og *triggers* testes ved at afvikle dem på database serveren og verificere, at de producerer det ønskede output.

Efter at have udført modultests, foretages en såkaldt acceptance-test, hvor det samlede system afprøves, og det vurderes, om det udviklede system lever op til de krav, der er blevet fremlagt i *use-casesene* og dermed hvorvidt systemet indfrier slutbrugerens ønsker. Acceptance-test består, i forbindelse med dette projekt, af at verificere at *use-case #1, #2* og *#4* er indfriede i det omfang, det er blevet specificeret i kravspecifikationen.

For såvel de tests, der skal foretages af *Presentation Layer* og de forskellige dele af acceptance testens vedkommende, er der tale om tests, der skal afvikles på en PDA.

I forbindelse med test findes der mange forskellige fremgangsmåder og mange detaljeringsgrader. I forbindelse med dette projekt er der valgt at lægge vægten dels på modultest, hvor de forskellige større dele af de tre lag bliver testet og dels på acceptance-test, hvor den endelige applikation holdes op mod *use-casesene*. Denne prioritering skyldes det fokus opgaven har på funktionalitet. Hvis NNE på længere sigt ønsker at videreudvikle applikationen og gøre den produktionsmodnet, således at den vil kunne tages i anvendelse af Novo Nordisk, er der en lang række tests, der skal køres og en lang række krav, som applikationen vil skulle leve op til for at gøre den brugbar i forbindelse med farmaceutisk produktion. Disse tests og krav ligger dog uden for rammerne af denne opgave, da formålet har været at producere en prototype, som NNE kan bruge som inspiration til et evt. fremtidigt produkt.

Den dokumentation, der genereres som følge af at have udført disse tests er meget omfattende, og derfor bringes i selve rapporten kun dokumentation fra udførelse af acceptance-testen. Dokumentation af modultestene findes i form af RunReviewReport hjemmesiden, der ligger i *TestRapport*-biblioteket på bilags-CD-ROM'en.

8.1 Requirements

Som nævnt i afsnit 2.4 skal der defineres en række krav til de forskellige tests. Disse krav fremgår af Tabel 16. Af denne fremgår det, hvad kravet går ud på, og hvilke tests, der skal leve op til dette krav. Nogle af kravene har form af henvisninger til krav- og designspecifikationen i nærværende dokument.

Navn	Status	Beskrivelse	Tests der skal indfri dette krav
Use-case #1	Passed	Se use-case # 1 i kravspecifikation	Use-case #1
Stored procedure test	Passed	Test af alle *stored procedure*s i I_POA packagen skal være udført.	Dynamisk GUI Eksekvering af ServiceGUIRequest Eksekvering af ServiceRequest Use-case #1 Use-case #2
Database kommunikation test	Passed	DataAccess skal være testet	Dynamisk GUI Eksekvering af ServiceGUIRequest Eksekvering af ServiceRequest Use-case #1 Use-case #2
Stored procedure	Passed	Beskrivelse af *stored procedure*s defineres under afsnittet Data Storage i Design. Disse *stored procedure*s skal være implementeret.	Dynamisk GUI *stored procedure* Use-case #1 Use-case #2
Eksekvering af ServiceRequest test	Passed	ServiceRequest test skal være gennemført succesfuldt	Dynamisk GUI Use-case #1 Use-case #2
Eksekvering af ServiceGUIRequest test	Passed	ServiceGUIRequest test skal være gennemført succesfuldt	Dynamisk GUI Use-case #1 Use-case #2
WSDataService	Passed	Afsnit om WSDataService i design krav.	Dynamisk GUI Eksekvering af ServiceGUIRequest Eksekvering af ServiceRequest Use-case #1 Use-case #2
Presentation	Passed	Afsnit om brugergrænseflade i kravspecifikation. Afsnit om Presentation Layer i design krav.	Dynamisk GUI Use-case #2
Data Storage	Passed	Afsnit om Data Storage i design krav.	Trigger Use-case #2
Use-case #2	Passed	Se use-case # 2 i kravspecifikation	Use-case #2
Use-case #4	Passed	Se use-case # 4 i kravspecifikation	Use-case #4
Trigger test	Passed	Trigger test skal være gennemført succesfuldt	MainMenuForm Use-case #1 Use-case #2
Data Access	Passed	Afsnit om Data Access i design, herunder behovet for komponentudviklilng til generisk databaseopkobling.	Generiske database komponenter Use-case #2
MainMenuForm test	Passed	MainMenuForm test skal være gennemført succesfuldt	Dynamisk GUI Use-case #1 Use-case #2
WiFi forbindelse	Passed	Det specificerede WiFi skal være tilgængeligt og oppe at køre.	Dynamisk GUI MainMenuForm Use-case #1 Use-case #2 Use-case #4

Tabel 16 - Requirements

8.2 Acceptance-test

Acceptance-testen er den del af testforløbet, hvor det testes, om det system, der udviklet lever op til kravene i de definerede *use-cases*. En *use-case* kan give anledning til en eller flere *Acceptance-tests* afhængigt af dennes indhold.

En *Acceptance-test* er en såkaldt *Black Box* test. Det vil sige, at testpersonen kun tager stilling til om input genererer det forventede output og dermed ikke til, hvad applikationens indre maskineri stiller op med input for at generere output. Dette gør også, at denne test er velegnet til at afvikle sammen med slutbrugeren, da denne i mange tilfælde ikke har noget særskilt interesse i hvorledes applikationskoden virker, men derimod blot er interesseret i at kunne afdække, hvorvidt de krav, der blev stillet, er blevet indfriede. Således kan *Acceptance-test* på ingen måde stå alene, og det er nødvendigt forinden at have udført grundige *Modultests* og *Unit Tests*.

8.2.1 Use-case #1

Run Name: Run_6-14_18-4-21			
Test plan name:	Use-case #1	**Run Status:**	Passed
Test set name:	POA_draft	**Duration:**	1128
Execution date:	2006-06-14	**Execution time:**	18:26:18
Host:	NNE-PC6020	**Tester:**	atla
Domain:	NNE	**Project:**	LMES
Server:	http://appdkba137/tdbin		

Test Plan description:

Resumé
==========
Det skal testes, hvor vidt kravene i use-case #1 er opfyldte, og det dermed er muligt at hente status for udstyr.

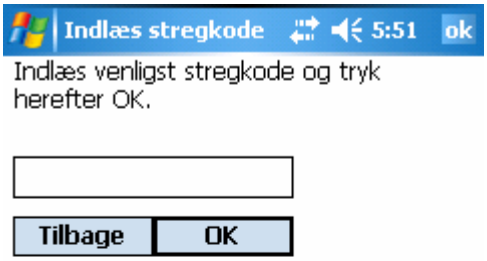
Beskrivelse
==========
Punkterne i beskrivelsen af use-case #1 gennemgås trinvis, idet POA-applikationen afvikles på en PDA.

Attachments
==========
Ingen

8.2.1.1 STEP 1 - VÆLG UDSTYRSSTATUS

Date	Description	Expected result	Actual result	Status
2006-06-14 18:05:28	POA-applikationen skal startes på PDA'en. Vælg udstyrsstatus som datavalg i hovedmenuen.	Skærmbilledet, hvor stregkode skal indskannes vises.	Se vedlagte screen dump for dokumentation af succesfuld afvikling.	Passed

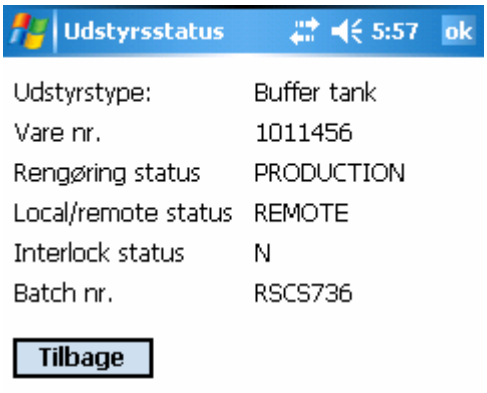
IndlæsStregkode:



8.2.1.2 STEP 2 - INDLÆS STREGKODE

Date	Description	Expected result	Actual result	Status
2006-06-14 18:06:17	Indlæs Tag-værdien 141A i stregkodefeltet og tryk OK	Skærmbillede med udstyrsstatus for det valgte udstyr vises: Udstyrstype: Buffertank Vare nr. 1011456 Rengøring status PRODUCTION Local/remote status REMOTE Interlock status N Batch nr. RSCS736	Der er indtastet 141A i stregkodefeltet og derefter trykket OK. Se vedlagte screen dump for dokumentation af succesfuld afvikling.	Passed

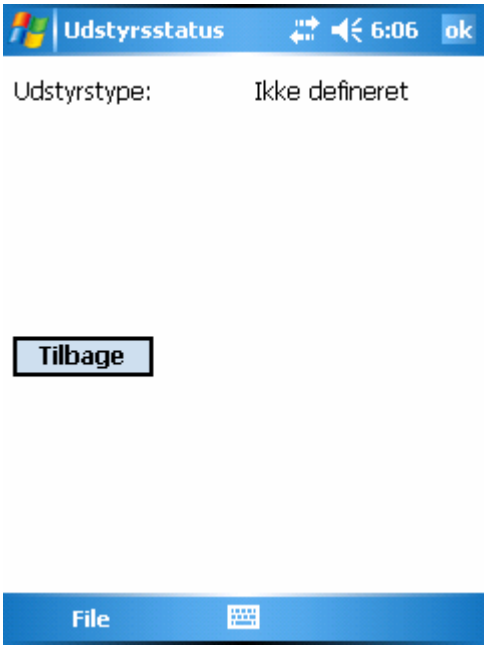
UdstyrsStatusResultat:



8.2.1.3 STEP 3 - ILLEGAL STREGKODE

Date	Description	Expected result	Actual result	Status
2006-06-14 18:08:44	Indtast i stregkodefeltet Tag-værdien X, der ikke findes i databasen.	Udstyrsstatusskærmbiledet vises uden nogen oplysninger. Udstyrstype: Ikke defineret	Se vedlagte screen dump for dokumentation af succesfuld afvikling.	Passed

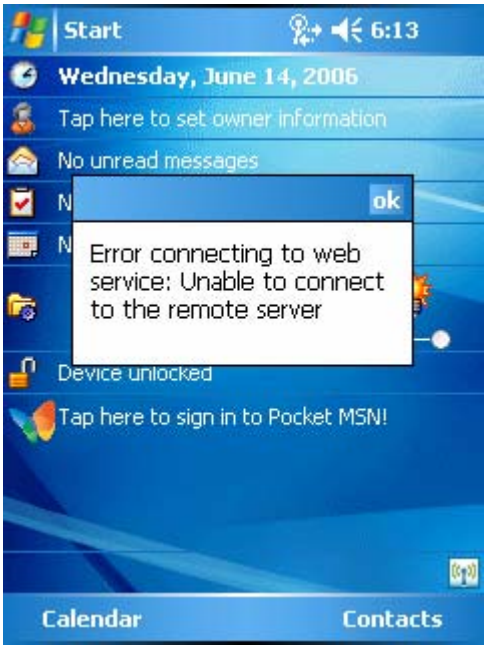
UdstyrsstatusIllegalInput:



8.2.1.4 STEP 4 - MANGLENDE WiFi FORBINDELSE

Date	Description	Expected result	Actual result	Status
2006-06-14 18:17:04	Afbryd WiFi forbindelse og forsøg at opstarte applikationen på PDA'en Tryk på OK, og applikationen skal afsluttes.	MessageBox med fejlmeddelelse præsenteres: Error connecting to *web service*: Unable to connect to the remote server	Se vedlagte screen dump for dokumentation af succesfuld afvikling.	Passed

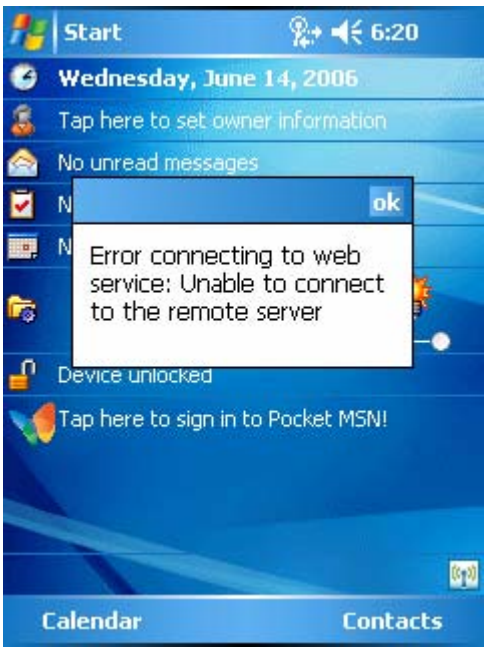
WiFiException:



8.2.1.5 STEP 5 - MANGLENDE FORBINDELSE TIL DATABASE / DATABASE NEDE

Date	Description	Expected result	Actual result	Status
2006-06-14 18:22:51	Afbryd forbindelsen mellem webserveren og Oracle databasen.	MessageBox med fejlmeddelelse præsenteres: Error connecting to *web service*: Unable to connect to the remote server	Forbindelsen mellem webserveren og Oracle databasen afbrydes.	Passed

WebServerException:



8.2.2 Use-case #2

Run Name: Run_6-14_18-43-3			
Test plan name:	Use-case #2	**Run Status:**	Passed
Test set name:	POA_draft	**Duration:**	193
Execution date:	2006-06-14	**Execution time:**	18:46:21
Host:	NNE-PC6020	**Tester:**	atla
Domain:	NNE	**Project:**	LMES
Server:	http://appdkba137/tdbin		

Test Plan description:
Resumé
==========
Det skal testes, hvor vidt kravene i use-case #2 er opfyldte, og det dermed er muligt at hente operationer for udstyr.

Beskrivelse
==========
Punkterne i beskrivelsen af use-case #2 gennemgås trinvis, idet POA-applikationen afvikles på en PDA.

Attachments
==========
Ingen

8.2.2.1 STEP 1 - HENT UDSTYRSOPERATIONER

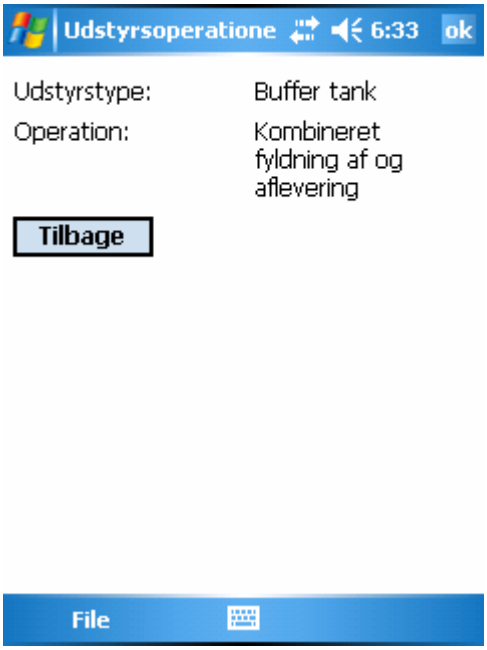
Date	Description	Expected result	Actual result	Status
2006-06-14 18:44:08	Vælg udstyrsoperationer som datavalg i Hovedmenuen på PDA'en.	Skærmbilledet, hvor stregkode skal indskannes vises	Se vedlagte screen dump for dokumentation af succesfuld afvikling	Passed

IndlæsStregkode:

8.2.2.2 STEP 2 - INDLÆS STREGKODE

Date	Description	Expected result	Actual result	Status
2006-06-14 18:44:27	Indlæs Tag-værdien 141A i stregkodefeltet og tryk OK	Udstyrsoperation vises for det valgte udstyr: Udstyrstype: Buffer tank Operation: Kombineret fyldning af og aflevering	Se vedlagte screen dump for dokumentation af succesfuld afvikling	Passed

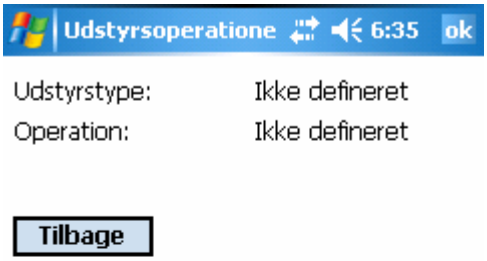
UdstyrsoperationResultat:



8.2.2.3 STEP 3 - ILLEGAL STREGKODE

Date	Description	Expected result	Actual result	Status
2006-06-14 18:44:44	Indtast i stregkodefeltet Tag-værdien X, der ikke findes i databasen.	Udstyrsstatusskærmbiledet vises uden nogen oplysninger. Udstyrstype: Ikke defineret Operation: Ikke defineret	Se vedlagte screen dump for dokumentation af succesfuld afvikling	Passed

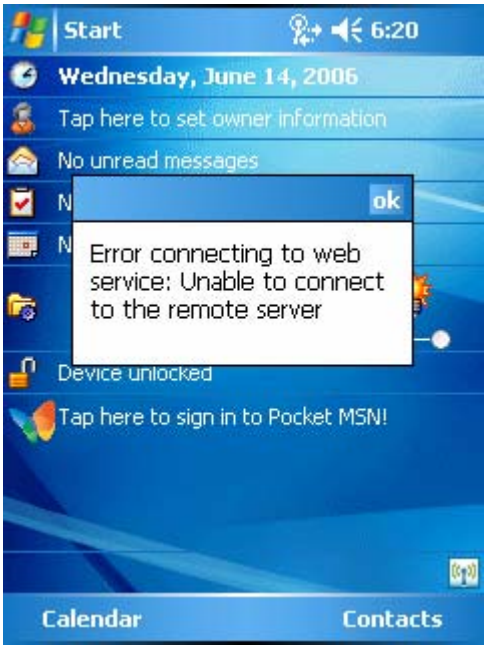
IllegalInput:



8.2.2.4 STEP 4 - MANGLENDE WiFi FORBINDELSE

Date	Description	Expected result	Actual result	Status
2006-06-14 18:45:55	Afbryd WiFi forbindelse og forsøg at opstarte applikationen på PDA'en Tryk på OK, og applikationen skal afsluttes.	MessageBox med fejlmeddelelse præsenteres: Error connecting to *web service*: Unable to connect to the remote server	Se vedlagte screen dump for dokumentation af succesfuld afvikling	Passed

Webserver exception:



8.2.2.5 STEP 5 - MANGLENDE FORBINDELSE TIL DATABASE / DATABASE NEDE

Date	Description	Expected result	Actual result	Status
2006-06-14 18:45:40	Afbryd forbindelsen mellem webserveren og Oracle databasen.	MessageBox med fejlmeddelelse præsenteres: Error connecting to *web service*: Unable to connect to the remote server	Forbindelsen mellem webserveren og Oracle databasen afbrydes. Se vedlagte screen dump.	Passed

Webserver exception:

8.2.3 Use-case #4

Run Name: Run_6-15_9-21-20			
Test plan name:	Use-case #4	**Run Status:**	Passed
Test set name:	POA_draft	**Duration:**	1531
Execution date:	2006-06-15	**Execution time:**	09:46:52
Host:	NNE-PC6020	**Tester:**	atla
Domain:	NNE	**Project:**	LMES
Server:	http://appdkba137/tdbin		

Test Plan description:
Resumé
==========
Det skal testes, hvor vidt kravene i use-case #4 er opfyldte. For at dette kan lade sig gøre, skal POA-applikationen være i stand til at deployes centralt.

Beskrivelse
==========
Punkterne i beskrivelsen af use-case #4 gennemgås trinvis. POA-applikationen deployes ved hjælp af en hjemmeside.

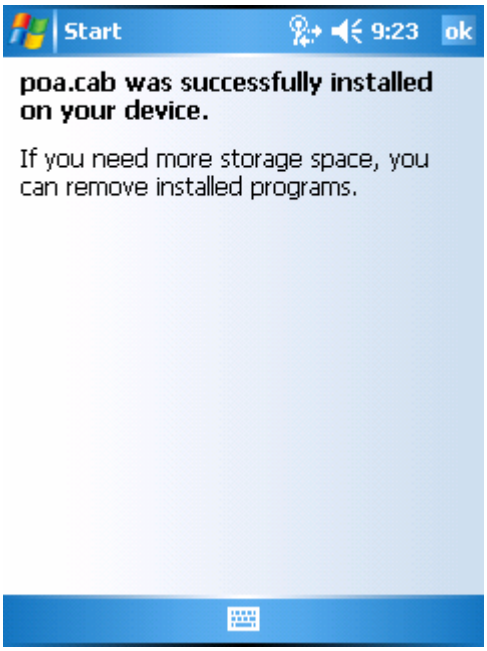
Attachments
==========
Ingen

8.2.3.1 STEP 1 - TILGÅ INSTALLATIONSMODULET

Date	**Description**	**Expected result**	**Actual result**	**Status**
2006-06-15 09:33:22	På PDA'en skal CAB-installationsfilen henten fra en hjemmeside, igennem Internet Explorer: http://10.15.14.6/deploy/poa.cab	CAB-filen downloades og POAPresentation_CF installeres	Hjemmesiden *http://10.15.14.6/deploy/poa.cab* tilgås, og der trykkes [Yes] for at downloade poa.cab - se screen dump #1. Som det fremgår af screen dump #2, blev applikationen korrekt installeret på PDA'en.	Passed

IEDeploy:

POAInstalled:

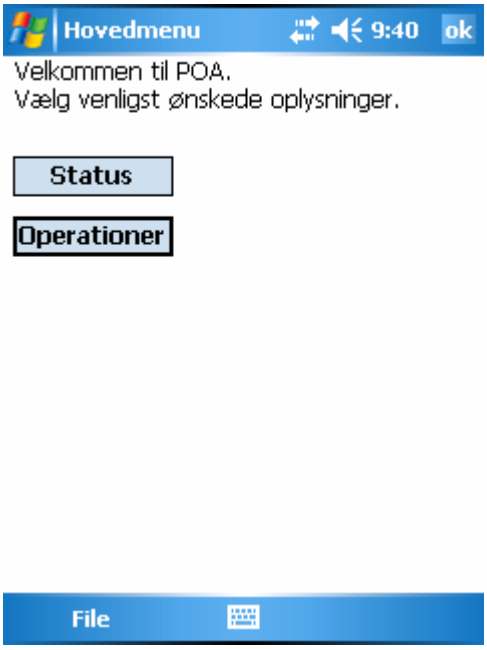


8.2.3.2 STEP 2 - START POAPRESENTATION_CF

Date	Description	Expected result	Actual result	Status
2006-06-15 09:46:48	Start POAPresentation_CF fra Start Menuen eller Program Files/ POAPresentation_CF på PDA'en.	POAPresentation_CF starter op og fungerer som specificeret.	Som det fremgår af screen dump #1, er POAPresentation_CF automatisk blevet tilføjet til startmenuen. Der trykkes på dette link for at starte applikationen. Det ses på screen dump #2, at applikationen fungerer korrekt.	Passed

Startmenu:

POAPresentation_CF:



9 Deployment

I dette kapitel gennemgås hvad der skal til for at POA-applikationen kan afvikles. Dette inkluderer dels, hvilke filer, der skal være kompilerede, hvilken hardware, der skal være til stede samt hvilke programmer, der skal køre. Dette afsnit kan betragtes som en opsætningsvejledning, der kan anvendes af administratorer. Der er imidlertid ikke givet en grundig gennemgang af, hvorledes de forskellige komponenter installeres, men i højere grad hvilke komponenter der er nødvendige.

9.1 Forudsætninger for afvikling af applikation

I de følgende afsnit gennemgås de vilkår, der skal være til stede for at POA-applikationen kan afvikles.

9.1.1 Database

Det er nødvendigt, at der er adgang til en Oracle database, der gør de nødvendige LMES+ og GUI-oplysninger tilgængelige. Hvis den anvendte database ikke allerede har de nødvendige POA-tabeller oprettet, henvises til *DataStorage*-biblioteket på bilags CD-ROM'en. Her findes en fil ved navn *POA.sql*, der indeholder de SQL-statements, der skal til for at oprette de nødvendige tabeller samt statements, der fylder dem op med det data, der er udviklet til dette projekt. Herudover findes en fil ved navn *StoredProcedures.txt*, som indeholder de *stored procedures*, der skal anvendes. Disse er eksporterede fra PL/SQL Developer version 6.0.2.880.

9.1.2 Programkompilering

Der er en række filer, der skal være kompilerede for at programmet skal afvikles. Kildekoden til disse filer findes på bilags CD-ROM'en i *Develop*-biblioteket. Alle filerne skal kompileres med *strong name*. Til dette formål er lavet en key-file, der hedder NNEPOAKEY.snk. Denne ligger i *Develop*-biblioteket på bilags CD-ROM'en og passwordet til denne fil er "Drumz1!".

9.1.2.1 DataAccess

`POADataAccess` skal være kompileret til en .dll-fil, som kan tilgås af `WSDataService`.

9.1.2.2 WSDataService

`WSDataService` skal være kompileret og skal afvikles på en webserver, der kan tilgås ved hjælp af PDA'en. I dette projekt er IP-adressen på webserveren hardcoded til at være *10.15.14.6*.

9.1.2.3 Presentation

`POAPresentation` skal være kompileret og installeret på PDA'en.

9.1.2.4 POADeployment

Deployment-projektet skal være kompileret, og den genererede .CAB fil skal refererences fra webserveren. Der er vedlagt en kopi af denne fil i *Develop*-biblioteket på bilag CD-ROM'en.

9.1.3 WiFi

WiFi-forbindelsen skal være aktiv. Dvs. der skal være tændt for *access pointet*, og dette skal være konfigureret med de indstillinger, som fremgår af WiFi afsnittet under supplerende krav (afsnit 5.7.10)

9.1.4 Webserver

Webserveren skal være *Internet Information Server 5.0*, og denne skal afvikles på en PC med *Microsoft Windows* installeret, der har adgang til både NNE's LAN-netværk og til WiFi *access pointet*. Denne skal køre og afvikle `WSDataService` for at applikationen kan fungere.

9.1.5 PDA

PDA'en skal køre *Windows CE 5.0*. Derudover skal den have installeret *.NET Compact Framework 2.0* og have WiFi netværkskort. Til sidst skal `POAPresentation` installeres ved at .CAB-filen tilgås via webserveren.

10 Fremtidige udvidelser

Dette kapitel omhandler en række muligheder for at udvide den eksisterende applikation. Der vil her blive diskuteret dels funktionalitetsmæssige udvidelser og dels tekniske udvidelser – herunder forbedringer af nogle af de eksisterende løsninger. Der skelnes her mellem funktionalitet, som det opleves af hhv. operatør og administrator og så mere underliggende udvidelser, som ikke direkte påvirker, hvorledes disse oplever at anvende applikationen.

10.1 Funktionelle udvidelser

10.1.1 Udstyrsoperationer

Use-case #2 er som nævnt kun blevet delvist implementeret. Det er muligt at udvide applikationen, således at det ikke kun er en enkelt operation, der bliver vist, men derimod alle, der er aktive for udstyret. Derudover kan det være ønskværdigt, at det skærmbillede, der præsenterer operationerne bliver opdateret regelmæssigt, således at operatøren kan følge med i udviklingen i det operationer, der bliver afviklet på et bestemt udstyr. At kunne præsentere flere operationer vil kræve en række ændringer i, hvorledes *Presentation Layer* opretter elementer, idet det vil være nødvendigt at tage stilling til, hvor meget data bliver returneret fra LMES+. Automatisk opdatering vil kunne implementeres ved, at der tilføjes en AUTO_UPDATE række i POA_CONTROLS tabellen, og at *Presentation Layer* tager stilling til denne oplysning.

10.1.2 Yderligere forespørgsler

Udover de forespørgsler, der kan serviceres i den nuværende implementering, kan det på længere sigt være ønskværdig med en række andre forespørgsler. Applikationen er søgt designet således, at en række af disse forespørgsler nemt vil kunne tilføjes ved blot at tilføje *stored procedures* til LMES+ databasen. Der kunne her fx være tale om at præsentere batch-relaterede oplysninger.

10.1.3 Yderligere GUI elementer

På længere sigt kan det tænkes at være ønskværdigt at præsentere andre elementer på GUI'en end dem, der er understøttet i den nuværende implementering. Dette vil kræve ændringer i *Presentation Layer*, men dette lag er søgt implementeret på en måde, så disse ændring i vid udstrækning vil have form af tilføjelser til funktionaliteten og ikke kræve, at der ændres grundlæggende på, hvordan laget fungerer.

10.1.4 Vedligeholdelse

Som nævnt er *use-case #6* ikke blevet implementeret. Dette kan tænkes gjort på en række måder. I og med, at *use-case #4* blev implementeret som en løsning, der anvender at operatøren via PDA'en tilgår en hjemmeside for at installere applikationen, er det også muligt ad denne vej at sørge for at den nyeste version findes på PDA'en. Dette lever imidlertid ikke op til kravene om automatisk opdatering, som fremgår af *use-casen*. Der findes forskellige løsninger til såkaldt *smart deployment*, hvor det vil være muligt at lade applikationen tilgå en URL idet den startes op for at tjekke, om der er en nyere version tilgængelig. For yderligere diskusion af dette henvises til [17][26].

10.2 Tekniske udvidelser

10.2.1 Historik

Det er muligt at udvide applikationen, således at der gemmes oplysninger om fx de *exceptions*, der bliver genereret i forbindelse med afvikling af applikationen. Dette

[26] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetcomp/html/AutoUpdater.asp

kræver ændringer en række forskellige steder i applikationen. Det vil fortsat være ønskværdigt, at alle *exceptions* bliver sendt op til *Presentation Layer*, således at brugeren kan tage stilling til dem, men det vil endvidere være nødvendigt at alle *exceptions* der opstår (også de, der opstår på *Presentation Layer*), ligeledes håndteres på *Business Logic Layer*. Her skal fejlmeddelelsen fra de enkelte *exceptions* sendes videre til *Data Storage Layer*, hvor der skal eksistere en table til at opbevare de modtagne *exceptions* samt en *stored procedure*, der står for selve modtagelsen af *exceptions* og indsættelse af disse i den relevante tabel.

10.2.2 Sikkerhed

I og med, at sikkerhedsaspekter ikke har været centrale i overvejelserne til dette projekt, er der en række ting, der vil kunne strammes op omkring på dette område.

10.2.2.1 Tjek af lovlige PDA'er

Som nævnt i *use-case #3* og afsnit 5.8.4.6, er det ønskværdigt at have mulighed for at tjekke hvorvidt de PDA'er der forsøger at tilgå systemet er godkendte. Dette kan gøres ved, at der i LMES+ databasen oprettes en tabel, der indeholder MAC-adresserne på godkendte PDA'er. Derudover skal der oprettes en *stored procedure*, der returnerer denne tabel til *Presentation Layer*. På dette lag skal denne *stored procedure* kaldes gennem *Business Logic Layer*, og på baggrund af den returnerede information samt den anvendte PDA's MAC-adresse skal det vurderes, hvor vidt det kan tillades at operatøren får lov at gennemføre sin forespørgsel.

10.2.2.2 Databasesikkerhed

For at højne sikkerheden, er det muligt at kryptere de oplysninger, der har med databasetilgangen at gøres. Dette kunne fx gøres ved at de *connection string* oplysninger, der findes i web.config-filen på *Business Logic Layer* krypteres med RSA krypteringsalgoritmer.

11 Konklusion

Der er blevet designet, implementeret og testet en applikation til brug ved overvågning af fabriksudstyr. Applikationen er et add-on til NNE A/S produktet LMES+, og den er blevet udviklet ved brug af en række *software engineering* metoder. Da der her er tale om et objektorienteret softwareprojekt, er der valgt at anvende RUP som specifik metode, og i forbindelse hermed er disciplinerne kravspecificering, designspecificering, implementering, test, deployment, og projektstyring blevet gennemført. I kraft af at RUP er blevet anvendt, er brugen af *use-cases* gennemgående for arbejdet med hver enkelt af disse discipliner. Der har i projektet også været brug fra databasedesign og til det formål er ligeledes blevet anvendt anerkendte designmetoder.

Projektet er blevet specificeret i samarbejde med NNE A/S. På baggrund heraf er der blevet defineret en række *use-cases*, og det er lykkedes at implementere en applikation, der lever op til kravene i disse. Den udviklede applikation er blevet holdt op mod disse krav ved gennemførsel af acceptance-tests, som specifikt har haft til formål at verificere, at denne indfrier kravene fra *use-casene*. For at sikre en struktureret afvikling af testene, er programmet TestDirector blevet anvendt.

Den endelige applikation fremstår som et helstøbt produkt, der tilbyder en delmængde af den endeligt ønskede funktionalitet og en dynamisk GUI. Således er applikationen af en sådan karakter, at NNE A/S vil kunne anvende den til at vurdere, hvorvidt der er tale et produkt, der skal arbejdes videre med. Applikationen er endvidere dokumenteret og implementeret på en måde, der har til hensigt at gøre det nemt at arbejde videre med og tilføje ny funktionalitet til denne.

Generelt set har projektforløbet været yderst tilfredsstillende, og arbejdet med udvikling af applikationen har været spændende og udfordrende. Et gennemgående problem undervejs i projektet har været anskaffelsen af en PDA med passende specifikationer. Dette skyldes kravet om at kunne afvikle *.NET Compact Framework 2.0* i sammenhæng med Novo Nordisks krav til industriel anvendelse af PDA'er. En række af PDA'er, der blev prøvet undervejs, var velegnede til industriel brug, men understøttede ikke *.NET Compact Framework 2.0*, herunder den første PDA, der blev afprøvet, Intermec CK30. Det endelige valg af PDA faldt på Intermec CN2B, der indfrier både de industrielle krav, kravet om stregkodeskanner og de softwaremæssige krav. Ligeledes er applikationen blevet testet på en PDA til kontorbrug. Her har der været tale om HP IPAQ HX2400. Denne vil ikke kunne anvendes til fabriksbrug og skal først udstyres med en stregkodeskanner, men den vil være anvendelig til testformål, idet den lever op til de softwaremæssige krav.

Projektarbejdet har givet såvel projektforfatterne som NNE A/S lejlighed til at beskæftige sig med håndholdt teknologi. Endvidere har det været en udfordring at arbejde med *.NET Compact Framework 2.0* og *.NET Framework 2.0*, da disse i skrivende stund stadig er nye på markedet, og megen af den eksisterende litteratur henvender sig til tidligere versioner af .NET platformen. I kraft af dette projekt har NNE A/S således fået udviklet en funktionel applikation, der kører under .NET 2.0 platformen og projektforfatterne har fået megen indsigt i, hvorledes denne platform fungerer, og hvad det vil sige at udvikle større software projekter hertil.

Anders Toft Larsen Torben Boeck Jensen

12 Referencer

[1] DAPI-LMES+ Batch Execution System, SDS Edition 005, NNE A/S
[2] DAPI-LMES+ Equipment Model System, Design Specification Edition 003, NNE A/S
[3] Stephen C. Perry: Core C# and .NET, Prentice Hall PTR, 2005
[4] Steven John Metsker: Design Patterns in C#, Addison-Wesley Professional, 2004
[5] Paul Yao, David Durant: *.NET Compact Framework* Programming with C#, Addison-Wesley Professional, 2004
[6] Karl J. Lieberherr: Controlling the Complexity of Software Designs, http://www.ccs.neu.edu/research/demeter/papers/icse-04-keynote/ICSE2004.pdf
[7] Craig Larman: Applying UML and Patterns – An Introductioin to Object-Oriented Analysis and Design and the Unified Process (2nd edition), Prentice Hall PTR, 2001
[8] Object Management Group – UML, http://www.uml.org/
[9] Alistair Cockburn: Basic Use Case Template, 1998, http://members.aol.com/acockburn/papers/uctempla.htm
[10] Juval Lowy: C# Coding Standard, IDesign Inc., 2005
[11] W3Schools online Web Tutorials, http://www.w3schools.com
[12] World Wide Web Consortium, http://www.w3.org
[13] Glenn Johnson: Programming *Microsoft* ADO.NET 2.0 Applications: Advanced Topics, *Microsoft* Press, 2005
[14] Object Oriented Analysis and Design, http://www.ooad.org/
[15] Carnegie Mellon University – *Software Engineering* Institute, http://www.sei.cmu.edu/
[16] DTU: IMM, layout, M.Sc theses, http://www2.imm.dtu.dk/teaching/thesis/
[17] MSDN: http://msdn2.*Microsoft*.com
[18] Raghu Ramakrishnan, Johannes Gehrke: Database Management Systems, McGraw Hill Higher Education, 2002
[19] S88: ANSI/ISA–88.01–1995, Batch Control Part 1: Models and Terminology

Bilag

Alle bilag til dette projekt findes på vedlagte bilags CD-ROM. Denne indeholder følgende:

- PDF-version af dette dokument
- DataStorage-bibliotek, der indeholder de filer, som er nødvendige for at oprette POA-databasen.
- Develop-bibliotek, der indeholder kildekode til POA.
- Dokumentation-biblioteket, der indeholder XML-dokumentation til POA-koden.
- TestRapport-biblioteket, der indeholder den samlede testrapport, som er blevet genereret i TestDirector.
- Specifikation af Intermec CN2B PDA (CN2Family_spec_web.pdf).