Department of Computer Science
Aalborg University
Selma Lagerlöfs Vej 300
DK-9220 Aalborg EAST
Denmark

A Mobile Solution for the Home Care Sector in Herning Municipality

DAT6
Aalborg University
June 2009

Group members: Lasse Soelberg | soeldk@cs.aau.dk
Peter G. Poulsen | pfnui@cs.aau.dk

Faculties of Engineering, Science and Medicine

Department of Computer Science

Title:
A Mobile Solution for the Home Care Sector in Herning Municipality

Project period:
DAT6,
2009-02-01 to 2009-06-11

Project group:
d620a

Group members:
Lasse Soelberg
Peter G. Poulsen

Supervisors:
Hua Lu

Abstract:

This project concerns the development of a mobile solution for the Home Care Sector in Herning Municipality. This system guides the employees through the day by providing a daily plan which contains all needed information. The project is motivated by two problems with their current browser based system; the client require access to the central system at all times and there is long response times when contacting the central system. We solve these problems by not using a browser based system, this allows us to store the data on the mobile clients. By storing the data on the client we lower the amount of communication needed and we solve the problem of needing a connection at all times.

Number of copies: 5

Number of pages: 109

Preface

This report is the result of our master thesis project for the DAT6 semester at Aalborg University. The DAT6 semester is the second of two semesters where we specialise in database technology. The first semester, DAT5, were used to develop a business intelligence solution for Herning Municipality. This semester we changed focus and are working on a new mobile system for Herning Municipality. This is motivated by problems within their current mobile system which we found out about during the development of the DAT5 project.

We would like to thank Herning Municipality for cooperating, providing data and help formulate a problem statement. We would also like to thank Hua Lu for supervising us during the development of this project.

Lasse Soelberg

Peter G. Poulsen

Contents

1 Introduction **9**
1.1 Scenario 9
1.2 Goals for our Solution 10

2 Analysis **13**
2.1 Daily Plan 13
2.2 Database 13
2.3 Recovery 26
2.4 Service 30
2.5 Mobile Client 31
2.6 Communication 37
2.7 Security 47

3 Design Overview **51**
3.1 Architecture 51
3.2 Models 52
3.3 Data Access Layer 55
3.4 Service 56
3.5 Client 57

4 Data Access **59**
4.1 Generating the Daily Plan 59
4.2 Writing Updates to the Database 66

5 Service **79**
5.1 Service Implementation 79
5.2 Configuration 80
5.3 HomeCareService Implementation 81
5.4 Models Namespace Implementation 84

6 Mobile Client **89**
6.1 User Interface 89
6.2 Communication 92
6.3 Storing Data 96
6.4 Models Namespace Implementation 100

7 Discussion and Conclusion **105**
7.1 Discussion 105
7.2 Conclusion 106

Bibliography **109**

Chapter 1

Introduction

This project concerns the development of a mobile solution for the Home Care Sector in Herning Municipality. For this project Herning Municipality have provided a dump of their database along with a guide to their current mobile system. This project is motivated by two problems within the current mobile system. The first is that the system is slow when communicating with the central system. Herning Municipality have told us that getting a response from the central system can take more than 30 seconds. The second problem is that the current mobile client requires a connection to the central system at all times. This gives problems since some citizens live in areas that doesn't have mobile coverage.

The solution has two main purposes; to provide the home care employee with a daily plan which contains all information needed for that day and updating the central system with delivered services.

1.1 Scenario

At the beginning of the work day the employees in the Home Care Sector of Herning Municipality receives a daily plan. A daily plan contains all the information the employee needs during that day. This includes the date of the daily plan, a work period and a list of citizens to visit. The list of visits contains information like name and address of the citizen along with a list of services which should be delivered.

The employee deliver the services described on the daily plan to the citizens. The delivered services have to be recorded in the central system so Herning Municipality can account for the used resources. The recorded information includes time spent on a visit, start time for the given visit etc.

1.1.1 Current System

Herning Municipality already have a mobile system for their Home Care Sector. Each employee have an HP iPAQ hw6915 Mobile Messenger pda. The current system is browser based which has both its strengths and weaknesses. The strengths and weaknesses of the current system will be discussed later, we are just going to describe the disadvantages we will be focusing on solving here.

One of the weaknesses are that their current system require a continues connection to be used. This is a problem since it is not all citizens in Herning Municipality that lives in a place with phone coverage. Another big issue is that it is slow in use. Response times larger than 30 seconds while communicating with the central system is not uncommon.

1.1.2 Limitations and Assumptions

We do not have time to implement a system with full functionality. We are only going to implement a system that have the most basic functions which is needed to generate a daily plan, get it to the employee and send the information back to the central system. The limitations and assumptions are:

- We are not going to implement many functions which the current system contains. This could for example be the ability to record breaks, move visits around and cancel visits on citizen requests. These functions can be added at a later time.
- We assume that we do not have to update the central system in real time. This is reasonable since the data is used to manage the used resources. This has to be done over a long time so data missing for the current day does not have much impact.
- We are not allowed to change the database structure. This is a limitation since our system have to be build on an already working database. If we changed the structure we might break their current systems; CARE and MobileCARE.

1.2 Goals for our Solution

We want to make a mobile system that solves the two problems with the current system. Namely it is slow and it requires a continues connection.

This means that the system have to be optimised in regards to response times so the employee does not have to wait for the system. It also have to be implemented so it does not need access to the central server at all times. The project focuses on implementing this for the standard scenario.

Besides the two goals we also have a number of criteria that we want to achieve with our solution.

- **Data Input at a Minimum:** The employee should not have to input much data into the system. This criteria should ensure that the employees in Herning Municipality would actually use the designed system.
- **Data Quality:** It is important to ensure high data quality within their database. This means that we cannot insert the same row multiple times and we have to insert valid data if possible.
- **Sensitive Data:** The data in the database is person sensitive. This means that it should not be possible to intercept the data and get a meaningful result. To ensure this criteria we have to ensure high security on the communication parts.

The solution will consist of three parts. The data access layer which handle the communication to the database. A service which handle communication between a mobile client and the data access layer. And finally a mobile client which is used by the Home Care employees.

Chapter 2

Analysis

This chapter contains an analysis of the key issues we will be focusing on. Section 2.1 will focus on the daily plan. Section 2.2 will focus on the database and how to retrieve the information needed for the daily plan. Section 2.3 will focus on recovery to ensure no data is lost within the system. Section 2.4 will focus on the service which binds our system together. Section 2.5 will focus on the mobile client and the requirement to our mobile application. Section 2.6 focus on the communication between the service and the mobile client. Section 2.7 focus on how to keep the data safe during the communication.

2.1 Daily Plan

This section will describe what information is needed on a daily plan. The section is based on the manual for Herning Municipality's current mobile solution. This should allow us to ensure that our system deliver the same information as the current system to the Home Care employees.

The information needed on the daily plan is shown in Table 2.1. The information needed on each visit is shown in Table 2.2. The information needed on each service is shown in table 2.3.

2.2 Database

Our scenario can be split into two separate tasks. The first is creating the daily plan for a given home care employee and the second is updating the database with the delivered services. These two tasks does not have anything in common except they use the same data which means the data extracted from the database will also be used to update the database.

Employee Name:	The employee can verify that the correct daily plan was received.
Date:	The employee can verify that the daily plan is for the correct date.
Work Period:	Used to show if the daily plan are for day(07-15), evening(15-23) or night(23-07).
List of Visits:	The visits the employee have to perform on the given day. This list can contain a random number of visits. Each visit contains an amount of attributes.

Table 2.1: Information Needed on the Daily Plan

Citizen Name:	Used to identify and address the citizen properly.
Citizen Address:	Used to find the citizen.
Citizen Key:	Describes where to find a key if needed.
Citizen City:	Used to find the citizen.
Citizen Postcode:	Used to find the citizen.
Citizen Phone:	Used to contact the citizen if needed.
Visit Start:	The time the visit is supposed to begin.
Visit End:	The time the visit is supposed to end.
List of Services:	The list of services the employee have to deliver to the citizen. The list can contain a random number of services. Each service have a set of attributes.

Table 2.2: Information needed for each Visit

Service Description:	Brief description of the service which have to be delivered.
Service Duration:	Duration of the service.
Service Details:	Detailed information for the service. This could be if the employee needs a tool to perform the service.

Table 2.3: Information needed for each Service

2.2.1 Retrieving the Daily Plan

To retrieve the necessary information from the database we need to access seven different tables. These are AID_LEVEL, AID_NEED_TYPE, AID_TIME_HISTORY, CLIENT, PERSONNEL, PROCEDURE_CODE and POSTCODE. The foreign key relationship between these tables can be found on Figure 2.1. An arrow from AID_TIME_HISTORY to PERSONNEL means that AID_TIME_HISTORY contains a foreign key constraint for PERSONNEL. The figure contains 9 tables, but CLIENT_MODULE and REFERRAL are only used to show that a foreign key exist all the way from AID_TIME_HISTORY to CLIENT, no data are retrieved from any of the two tables. The remainder of this section describes each of the seven tables and any problems the design of the tables might cause. Each description contains a table with 4 columns. First column is the attribute name. Second column is if the attribute is a part of the primary key either (Y)es or (N)o. Third column is if the attribute is null-able either (Y)es or (N)o. Fourth column is the data type of the attribute, CHAR(10) means CHAR(10BYTE).

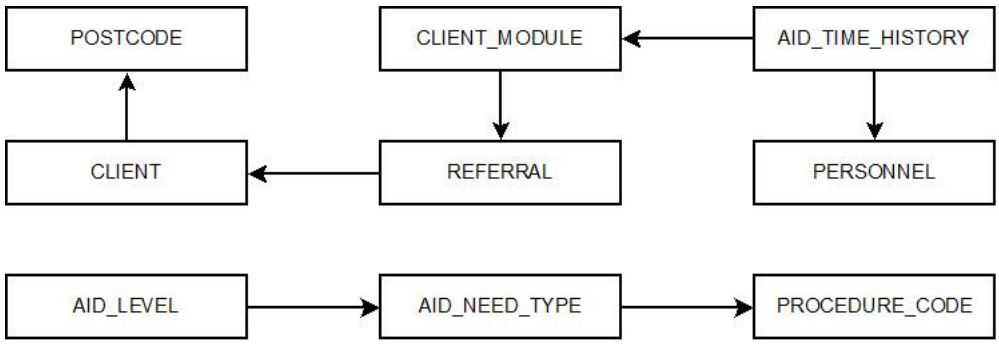

Figure 2.1: Table Relations Within the Database.

- **AID_TIME_HISTORY:** This table contains a row for each service which should be delivered over a given time period to a given citizen. This makes the table the ideal entry point for creating the daily plans, since there is a 1-to-1 correspondence between a row in this table and a service on the daily plan. The table contains the attributes INI, DATE_FR, DATE_TO and WEEKDAY_NO which identify all the services the employee should deliver on a given date. The remainder of the attributes found on Table 2.4 on the next page are either foreign keys or written on the daily plan. No foreign key constraint exist on MODULE_TYPE_NO, AID_NEED_TYPE_NO and AID_LEVEL even

though they are the primary key of the AID_LEVEL table. This missing foreign key is also the only drawback of this table since it allows for invalid values for each of these attributes.

Attribute	Primary Key	Null-able	Data Type
INI	N	Y	CHAR(10)
DATE_FR	Y	N	DATE
DATE_TO	N	Y	DATE
WEEKDAY_NO	Y	N	NUMBER
C_CPR_NO	Y	N	CHAR(10)
START_TIME	Y	N	CHAR(4)
NOTE	N	Y	VARCHAR2(355)
MODULE_TYPE_NO	Y	N	NUMBER
AID_NEED_TYPE_NO	Y	N	NUMBER
AID_LEVEL	Y	N	NUMBER

Table 2.4: AID_TIME_HISTORY Attributes.

- **AID_NEED_TYPE:** This table describes the different services that can be delivered to citizens. This could for example be cleaning or bathing. This table is used to get the description for each service that should be delivered. The needed attributes from this table can be found in Table 2.5.

  The table has one drawback it is inconsistently using the NEED_TEXT attribute. This means that the NEED_TEXT attribute can contain two different kinds of values. The first is an actual description of a service and the second is a procedure code. If it contains a procedure code the attribute PROCEDURE_CODE will contain a value.

Attribute	Primary Key	Null-able	Data Type
MODULE_TYPE_NO	Y	N	NUMBER
AID_NEED_TYPE_NO	Y	N	NUMBER
NEED_TEXT	N	Y	VARCHAR2(40)
PROCEDURE_CODE	N	Y	VARCHAR2(10)

Table 2.5: AID_NEED_TYPE Attributes.

- **AID_LEVEL:** This table describe how critical help is needed. This could for example be that someone needs 50 minutes of help with cleaning. The information gained from this table is the duration of each

service which should be delivered. The attributes of this table can be found in Table 2.6.

Attribute	Primary Key	Null-able	Data Type
MODULE_TYPE_NO	Y	N	NUMBER
AID_NEED_TYPE_NO	Y	N	NUMBER
AID_LEVEL	Y	N	NUMBER
DURATION	N	N	NUMBER

Table 2.6: AID_LEVEL Attributes.

- **CLIENT:** This table contains the information on all the citizens which are receiving or have received services. The table is used to gain all information regarding the citizens. This could for example be name and address. The needed attributes can be found on Table 2.7.

  The table contains one problem which is that POSTCODE, foreign key to the POSTCODE table, is null-able. This means that it is possible that it does not exist.

Attribute	Primary Key	Null-able	Data Type
C_CPR_NO	Y	N	CHAR(10)
NAME	N	Y	VARCHAR2(50)
SURNAME	N	Y	VARCHAR2(50)
ADDR_STRT	N	Y	VARCHAR2(55)
RES_PHONE_NO	N	Y	CHAR(12)
KEY_NO	N	Y	VARCHAR2(20)
RES_POSTCODE	N	Y	CHAR(4)

Table 2.7: CLIENT Attributes.

- **POSTCODE:** This table is used to translate postcodes into city names. The needed attributes can be found on Table 2.8.

Attribute	Primary Key	Null-able	Data Type
POSTCODE	Y	N	CHAR(4)
CITY	N	Y	VARCHAR2(40)

Table 2.8: POSTCODE Attributes.

- **PERSONNEL:** This table contains information on the employees. The table is used to extract the employee's name, NAME and SURNAME columns, for verification reason. The needed attributes can be found on Table 2.9.

Attribute	Primary Key	Null-able	Data Type
INI	Y	N	CHAR(10)
NAME	N	Y	VARCHAR2(34)
SURNAME	N	Y	VARCHAR2(34)

Table 2.9: PERSONNEL Attributes.

- **PROCEDURE_CODE:** This table is used to translate the procedure code from AID_NEED_TYPE into descriptions of the given service. The needed attributes can be found on Table 2.10

Attribute	Primary Key	Null-able	Data Type
PROCEDURE_CODE	Y	N	VARCHAR2(10)
DESCRIPTION	N	N	VARCHAR2(60)

Table 2.10: PROCEDURE_CODE Attributes.

The table designs gives us certain problems. These problems are described below with possible solutions.

2.2.1.1 Inconsistent Use of NEED_TEXT in AID_NEED_TYPE

The problem is that the NEED_TEXT attribute in AID_NEED_TYPE can contain both a procedure code and a description. If the first is the case the attribute PROCEDURE_CODE will contain a value otherwise it will contain null. To get the description of a procedure code we have to access the table PROCEDURE_CODE. The scenario is shown in Figure 2.2. There are three possible solutions for this problem:

- The first solution is to do a custom implementation to solve the problem. This could be to first queue AID_NEED_TYPE, then identify the procedure codes by using the PROCEDURE_CODE attribute. The last step would be to make a second query on PROCEDURE_CODE based on the identified procedure codes. This allow us to not access PROCEDURE_CODE at all if no procedure codes are identified.

- The second solution is to leave the procedure codes in the daily plan. This would assume that the employee knew what all the procedure codes meant which is unreasonable to believe.

- The last solution is described in [LP76] and uses the three order logic described in [Cod86]. This is achieved by using the outer join in SQL. The drawback of this approach is that we always have to check for null values even on not null-able attributes. This makes the application harder to maintain.

We have chosen to do a custom implementation. The reason for choosing a custom implementation is that the drawbacks of the other approaches are too much. The second approach would require all the employees to know all the procedure codes, which is unreasonable. The last approach could work but the join operation would make it hard to understand and maintain the code.

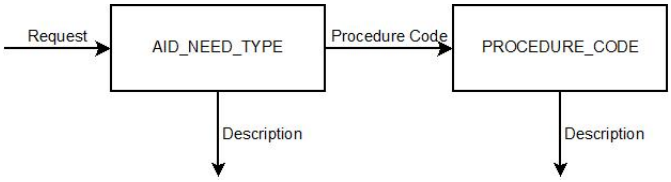

Figure 2.2: Inconsistent Use of NEED_TEXT.

2.2.1.2 No Foreign Key to AID_LEVEL

The problem of no foreign key constraint from AID_TIME_HISTORY to AID_LEVEL or AID_NEED_TYPE means that it is possible to have invalid values in AID_TIME_HISTORY. Invalid values have the consequence that service description and service duration can not be extracted from the database. The two values have different impact on the daily plan and will be treated as two different problems. Another problem can be derived from the missing duration values.

- The problem of a missing service description can be handled in two different ways. The problem occur if the combination of MODULE_TYPE_NO and AID_NEED_TYPE_NO is invalid.

  - The first is to remove the service from the daily plan. This gives the impact that the employee have no chance to deliver the service. Furthermore if services are removed there is a chance that an

empty visit could be left in the daily plan. In this case the visit should also be removed and citizens might complain because they were expecting a visit.

- The second is to leave the service on the daily plan. This would show the employee that something were supposed to be delivered. In this case it will be up to the employee to find out which service should be delivered.

- The problem of a missing duration can only be handled in one way since there is nothing to base a calculation on. The problem occur when the combination of MODULE_TYPE_NO, AID_NEED_TYPE_NO and AID_LEVEL is invalid.
  - The way is to leave a default value in case the value can not be extracted from the database. This allow the employee to deliver the service and to calculate the needed time by themselves.

- The last problem is derived from missing duration values for services. The problem consist of calculating the end time for the visits since no duration nor end time for each visit is present within the database. There are two possible ways to calculate the end time.
  - The first way is using the formula $EndTime = StartTime + \sum Service.Duration$. Where Service.Duration is the duration of a service which have to be delivered on the given visit. This approach has the drawback that it will not calculate the correct end time if services are missing their duration value. In case of missing duration values this calculation will give too short visits.
  - The second way is using the formula $EndTime = NextVisit.Start - Transport$. Where NextVisit.Start is the start time of the next visit and the Transport is the amount of time it takes to drive from the current visit to the next, this value can be extracted from the database. The drawback of this approach is that it require a Transport value to be present for all visits and the last visit will not have a next visit to base the calculation on. If the Transport value is missing in the database this calculation will give too long visits.

2.2.1.3 Null-able Citizen Address Attributes

The problem is that some of the attributes which describe the address of the citizen are null-able. If these values can not be extracted from the database

it will be impossible for the employee to find the given citizen. This problem only exist in theory since the database is filled with information from the National Register of Persons, which is updated automatically. This means that even if the attributes are null-able they should never in practise be left as null.

2.2.2 Updating the Database

The second part of the data access layer have to update the database with the delivered services. This information goes into the tables AID_DELIVERED and PP_REG. AID_DELIVERED has an entry for each delivered service and PP_REG has an entry for each of the following events: Start of day, end of day, transport, visit and pause. An entry could for example be that the employee started the day at 07:30 or that an employee visited a citizen from 08:24 to 08:42.

2.2.2.1 AID_DELIVERED

AID_DELIVERED contains 12 attributes which are all part of the primary key. The 12 attributes along with an explanation is shown in Table 2.11.

C_CPR_NO:	The CPR number of the citizen whom received the service. The citizen have to exist within the CLIENT table but no foreign key constraint exists.
REFERRAL_DATE:	The date of referral. It has nothing to do with the actual service and it can be extracted from the CLIENT table.
MODULE_TYPE_NO:	The main category of the delivered service. This could be a numeric value representing nursing. The value have to exist in the table MODULE_TYPE but no foreign key constraint exists.
AID_NEED_TYPE_NO:	Numeric value representing the service which have actually been delivered. The combination of MODULE_TYPE_NO and AID_NEED_TYPE_NO should exist in the table AID_NEED_TYPE but no foreign key constraint exists.

Table 2.11 – Continued on Next Page

Table 2.11 – Continued from Previous Page

AID_LEVEL:	Numeric value indicating the duration of the given delivered service. The combination of MODULE_TYPE_NO, AID_NEED_TYPE_NO and AID_LEVEL should exist in AID_LEVEL but no foreign key constraint exists.
WEEKDAY_NO:	A numeric value representing the given weekday the service were delivered. Monday is 1, Tuesday is 2 etc.
START_TIME:	The time of the day the employee started delivering the given service. This could for example be 1845.
PERSON_NO:	If multiple people were needed to deliver the given aid. The first employee have number 1, second 2 etc. Our system will not contain functionality to change this value and the default value 1 will always be used.
PLAN_DATE:	The date the service were supposed to be delivered. This could be different than the actual date if the citizen and employee agrees to change it.
DELIV_BY_INI:	A unique identifier for the given employee who delivered the service. The unique identifier consist of 10 digits.
DELIV_DATE:	The date the service were actually delivered. This could be different from the planned date in case the service were planned for 23:55 but delivered at 00:05.
PP_REG_ID:	DELIV_BY_INI, DELIV_DATE and PP_REG_ID should have been a foreign key to PP_REG according to the comments in the database. No foreign key constraint exists however.

Table 2.11: AID_DELIVERED Attributes.

2.2.2.2 PP_REG

The second table is PP_REG. This table contains 26 attributes, 3 of them are part of the primary key, namely INI, REG_DATE and ID. This table also have attributes which are not used. These attributes are CAR_TYPE and TRIPCOUNTER. All attributes except those which are not used are explained in Table 2.12

INI:	Unique identifier for the given employee who participated in the event. The identifier consist of 10 digits and it should exist in the PERSONNEL table but no foreign key constraint exists.
REG_DATE:	The date the event occurred.
ID:	Unique number assigned to each event to make them unique for the given day. This have nothing to do with the actual event.
EVENT_TYPE_ID:	Used to describe the different events. This is a number between 1 and 5. 1 - Start day, 2 - End day, 3 - Transport, 4 - Visit, 5 - Pause.
START_TIME:	The time of the day when the given event started.
DURATION:	The duration of the given event in minutes.
C_CPR_NO:	The CPR number of the citizen if the event is a visit. Null otherwise. The CPR number should exist in the CLIENT table but no foreign key constraint exists.
PAUSE_ID:	Unique number identifying which kind of pause were taken, only filled if the event was a pause. Null otherwise. We will not implement the functionality to record pauses.
END_TIME:	The time of the day when the given event ended.
DISP_START_TIME:	The time of the day when the given event was suppose to start.

Table 2.12 – Continued on Next Page

Table 2.12 – Continued from Previous Page

MODULE_TYPE_NO:	The type of visit, nursing, home aid etc. Only filled if the event is a visit. Null otherwise. The value should exist in the MODULE_TYPE table but no foreign key constraint exists.
ORIG_DATE:	The date the event was suppose to occur.
ACT_DATE:	The date the event actually occurred.
ORIG_START_TIME	The time of the day when the given event was suppose to start.
ACT_START_TIME:	The time of the day when the given event started.
ORIG_DURATION:	The duration the event was supposed to last.
ACT_DURATION:	The duration the event lasted.
CLIENT_ABORTED:	Flag to indicate if client aborted the visit. We will not implement the functionality to let clients abort a visit.
PDA_CHANGED:	Flag to indicate if the change were made in mobileCARE. In our case always negative.
CARE_CHANGED:	Flag to indicate if the change were made in CARE. In our case always negative.
APPROVED_CHANGE:	User who approved the change.
LAST_MODIFIED_INI:	Employee who last modified the entry. In our case this will be the employee who performed the event. The value should exist in PERSONNEL but no foreign key constraint exists.
LAST_MODIFIED_TIME:	The date when the last modification happened.
ON_BEHALF_OF_INI:	Ini of the employee the modification were done in behalf of, if someone else did it. This value should exist in the PERSONNEL table but no foreign key constraint exists.

Table 2.12: PP_REG Attributes.

2.2.2.3 The Information Needed to Populate the Tables

We need to receive enough data from the mobile client to fill out the above two tables. For this purpose we have identified the minimum amount of data we need to sent. The reason we only want to sent the minimum amount is that we want to lower the communication cost. The overall needed data can be found on Table 2.13 and the data for each visit can be found on Table 2.14.

EmployeeID:	The unique identifier of the employee. This is needed as insert value and to retrieve the same information which created the daily plan.
StartDayTime:	The time of the day when the employee started working. This is needed in PP_REG as one of the events.
EndDayTime:	The time of the day when the employee stopped working. This is needed in PP_REG as one of the events.

Table 2.13: Needed Overall Update Data from the Mobile Client.

StartTime:	The time of the day when the employee started the visit. This is needed in PP_REG.
EndTime:	The time of the day when the employee stopped the visit. This is needed in PP_REG.
PlanDate:	The date the visit were planned to happen. This is needed as insert value and to retrieve the needed data from other tables.
ActualDate:	The date the visit actually happened. This is needed in PP_REG as both REG_DATE and ACT_DATE, furthermore it will also be the one of the foreign key values in AID_DELIVERED.
CitizenCPR:	The CPR number of the citizen which the visit were delivered to. This is needed to ensure that the extracted data from other tables are assigned to the correct visits. It is also needed as insert value for PP_REG.

Table 2.14: Needed Update Data for each Visit from the Mobile Client.

2.2.2.4 Problems With Updating

The design of the two tables gives us some problems. These problems will be outlined in the remainder of this section and solutions to each problem will be proposed.

Missing External Foreign Keys: This problem is a lack of foreign keys to other tables. This could for example be CLIENT or PERSONNEL. This problem has two different solutions based on the attributes in question.

- The first solution is for the tables which have foreign keys in AID_TIME_HISTORY. These attributes will be considered to correct since we are extracting them from AID_TIME_HISTORY. This means that we do not have to do anything further for these values. Examples of such values are C_CPR_NO and INI.

- The second solution is for the tables which does not have foreign key constraints in AID_TIME_HISTORY. This means that the data might be invalid. In these cases we can insert the values extracted from AID_TIME_HISTORY since we have no possible ways to deduct what the values are for the invalid data. Examples of such attributes are MODULE_TYPE_NO, AID_NEED_TYPE_NO and AID_LEVEL.

Missing Foreign Key on AID_DELIVERED: This problem is a missing foreign key constraint on AID_DELIVERED to PP_REG. This foreign key constraint were meant to exist, which can be seen from the comments on the attributes in AID_DELIVERED. There is only one solution to this problem; to ensure that the data is inserted into PP_REG first. This allow us to refer the rows from AID_DELIVERED to the events in PP_REG.

2.3 Recovery

There are two reasons for implementing data recovery. These are explained below.

- The first reason is to ensure fast response time for the clients. This is in the case of multiple clients requesting an update at the same time. It will not be possible to insert all the request within a limited time range so we have to store the data while waiting for it to be inserted into the database.

- The second reason is to ensure that no data is lost in case of failures. The different types of failures are explained in the next chapter. Our recovery scheme will only cover transaction failures and system failures.

The remainder of this section is based on the information from [Gra81] and [HR83].

2.3.1 Failures

The failures can be split into three categories; transaction failure, system failures and disk failure. This section will describe the categories along with the data loss in case of a failure.

- **Transaction Failure:** Transaction failure happens when a condition within the system means that the transaction could not happen. This could happen if the connection to the database is lost, for example if the database crashes. This kind of failure does not give any data loss since the data will still be present within the system.

- **System Failure:** System failure happens when the system shutdowns unexpectedly. This could happen if the power turned off during a thunderstorm. This kind of failure suffers from data loss as all volatile storage, like cache and main memory, is lost.

- **Disk Failure:** Disk failures happens when the non-volatile storage stops working. This could happen if a fire destroys the entire system. This kind of failure gives the most data loss as both volatile and non-volatile storage is lost.

2.3.2 Recovery Techniques

This section will describe the recovery technique we are going to use. This recovery scheme will not cover disk failures since techniques to approximate stable storage is outside the scope of this project. This means that this section will focus on system failures since transaction failures can be handled by repeating the insert operation.

There are two possible recovery schemes; Time-Domain Addressing and Log-Based Recovery. We have chosen to use Log-Based Recovery since we do not need the feature of getting a view of how the database looked at a given point in time. The Log-Based Recovery scheme builds around two operations; Undo and Redo. In our case we will only need the Redo operation since we are only inserting into the database. This means that our insert operation have to be idempotent so we can be sure that a row is only inserted one time no matter how many times the insert operation is done.

The log stores all the data received from the clients. The data is stored in this log file until it is successfully committed to the database. After each commit we can do a checkpoint and remove the already committed data from the log file. This means that only a minimum have to be redone in case of a system failure.

2.3.3 Concurrency

The log file on the server will be accessed by all the clients and a high degree of concurrency will be going on. This section will focus on the solution to handle the concurrent access to the shared resource; the log file. This section is based on [Han72].

The way to handle the many update requests are by using mutual exclusion to ensure that only one thread access the log file at any given time. There are two different kinds of threads which will try to access the file. The first is the update requests from the clients which will write their update information into the log file. The second kind is the thread which will insert the data into the database. None of these threads will be given priority to the log file since the read operation is done infrequently since high data quality has to be ensured. This means that the insert operation will be time consuming since many checks have to be done and data have to be extracted from the database.

2.3.4 Implementation

The log can be implemented with different approaches. The remainder of this section will focus on the strengths and weaknesses of each approach. This will be illustrated using an example with an UpdateObject containing two delivered visits. The used data can be found in Table 2.15.

UpdateObject		
EmployeeID:999999999	StartDayTime:0930	EndDayTime:1045
First Visit		
StartTime:0934	EndTime:1004	PlanDate:01-10-2008
ActualDate:01-10-2008	CitizenCPR:9999999999	
Second Visit		
StartTime:1010	EndTime:1040	PlanDate:01-10-2008
ActualDate:01-10-2008	CitizenCPR:9999999998	

Table 2.15: Example Data.

2.3.4.1 Text File

One possibility is implementing the log as a plain text file. This is optimal regarding disk space since the bare minimum can be stored. The solution has some drawbacks since all parsing have to be done by us. Furthermore it will be hard to maintain the solution if changes are happening to the input.

An example of storing the example update object as a plain text file can be found in Figure 2.3.

```
0930,1045,9999999999,633584160000000000,633584160000000000,9999999999,0934,1004,
633584160000000000,633584160000000000,9999999998,1010,1040
```

Figure 2.3: Result using Plain Text Recovery.

2.3.4.2 XML File

Another possibility is to implement the log as an XML file. This gives some overhead in storage as all the tags for each object are stored. The optimal part of this approach is that reading and writing to XML is supported by the .NET framework. This means that it will be easy to maintain and update in case of changes to the update objects. An example of the stored UpdateObject can be found on Figure 2.4.

```
<?xml version="1.0" encoding="utf-8" ?>
- <Root>
  - <UpdateObject StartDayTime="0930" EndDayTime="1045" EmployeeID="9999999999">
    - <Visit>
        <ActualDate>633584160000000000</ActualDate>
        <PlanDate>633584160000000000</PlanDate>
        <CitizenCPR>9999999999</CitizenCPR>
        <StartTime>0934</StartTime>
        <EndTime>1004</EndTime>
      </Visit>
    - <Visit>
        <ActualDate>633584160000000000</ActualDate>
        <PlanDate>633584160000000000</PlanDate>
        <CitizenCPR>9999999998</CitizenCPR>
        <StartTime>1010</StartTime>
        <EndTime>1040</EndTime>
      </Visit>
    </UpdateObject>
  </Root>
```

Figure 2.4: Result using XML Recovery.

2.3.4.3 Conclusion

We have decided to use the XML implementation of the log file. The reason is that the storage overhead caused by using XML should rather small since the update objects are removed from the file after being successfully inserted into the database. Furthermore the support of the .NET framework will make it easier to maintain and update the recovery.

2.3.5 Recovery Based Problem

Recovery gives us one problem because of a poorly defined primary key in PP_REG. The problem is that the primary key can not be used to ensure uniqueness. This means that the primary key allows for us to insert the same data multiple times in case of a redo action after a crash. Table 2.16 shows the primary key attributes of two events done by the same employee on the same date. It can be seen that the first two attributes have the same value. The ID is assigned by our application. This means that it can not be assigned before we know which values are already in use and if some of the rows have already been inserted.

Event 1		
INI:9999999999	REG_DATE:08-10-01	ID:1
Event 2		
INI:9999999999	REG_DATE:08-10-01	ID:2

Table 2.16: Recovery Problem Example.

The solution is to use other attributes when checking for uniqueness. The chosen attributes are INI, REG_DATE, EVENT_TYPE_ID, START_TIME, END_TIME and C_CPR_NO. These attributes can ensure uniqueness in almost all cases.

2.4 Service

The service binds the entire system together. It communicates with both the mobile client and the database.

2.4.1 Requirements

Even though the service is important, there aren't many requirements for it, since most of the functionality is implemented in either the client or the data access layer. The requirements we have identified are:

1. Platform independent communication.

2. Implementation should be hosted in an existing system.

3. Easy to develop and maintain.

2.4.2 Our Choice

The service implementation should be able to run on the already existing servers at Herning Municipality. The current system is hosted in a Windows server environment which gives us three different options to implement a service in:

1. PHP
2. Classic ASP
3. .NET

The existing environment already hosts .NET pages and services and thus have support for developing solution based on the .NET framework. This makes the choice easy for us, since we are already familiar with .NET development using C# and Visual Studio.

2.5 Mobile Client

An important part of our solution is the mobile client. It is through this client that the users will interact with the system, and as such it is a very important part of our solution.

In this section we will start with examining the already existing solution. We will then define requirements for our solution. And finally we will decide on an implementation language.

2.5.1 The Current System

The existing mobile solution that is used by Herning Municipality is browser based. The devices used are HP iPaq hw6915 Mobile Messenger, which is shown in Figure 2.5. The operating system used is Windows Mobile 5.0 and the browser is Mobile Internet Explorer. Figure 2.6 shows the main menu of the existing system.

2.5.1.1 Advantages with the current system

There are two main benefits of using a browser based system. The first is that the clients don't have to be updated with new versions of the system, since it resides entirely on a server. If there is an update to the system, all it takes is to update the server and all clients will automatically be using the new version.

Figure 2.5: HP iPAQ hw6915 Mobile Messenger.

Figure 2.6: Main menu for the existing mobile client solution.

Another advantage is that all you need on a client is a browser, which, at least in theory, makes it easy to change platform. The reason why it is only in theory is because the html output is specifically generated towards a 240x240 pixel screen, which is what the HP iPAQ hw6915 uses.

2.5.1.2 Disadvantages with the current system

There are two major disadvantages with using a browser-based solution. The first one is that there always have to be network access. No network access means no contact to the server and thus the system is unusable. This is one of the problems that Herning is facing with the current system. There are not coverage throughout the Municipality.

Another disadvantage is that there needs to be transferred more data across the network, since all markup for the page has to be delivered and not only the data to display, and this can be a problem if the transfer rate of the network isn't high enough.

2.5.2 Requirements

There are several requirements that a client for the system should fulfil.

1. It should be useable on different devices, such that the Municipality are not limited in their choice of mobile device.

2. It has to be able to communicate with a central service.

3. It should not rely on a continues network connection.

4. It should be possible to close the client without loosing information and on restart of the client it should continue where it left.

2.5.3 Implementation Language

There are several possibilities for choosing a programming language for the client implementation. The operating system on the existing mobile devices is Windows Mobile 5 which natively supports programs written in C++ and C#/VB using the .NET Compact Framework. Hewlett Packard has also added a Java Micro Edition virtual machine to the system.

Being Windows Mobile there is also the possibility to add further runtime environments to the system if needed, which for example can add support for using Python as the implementation language. We will however only focus on the more established C-style languages namely C++, C# and Java.

In the following we will examine the three possibilities before deciding on a language.

2.5.3.1 C++

C++ compiles directly into machine code. This can make the solution very efficient, both in terms of execution time and the amount of system memory used. It as also a well supported language, so there exists a lot of support for it, both in regards to tools, Visual Studio for example, and in regards to good information on the internet.

One of the biggest problems with native code (C/C++) is the use of memory. Especially when it comes to cleaning up the memory. Since it is an unmanaged environment there is no garbage collection and this puts the responsibility of cleaning up memory on the developers. This can result in memory leaks which can have severe side effects on mobile devices. Especially Nokia phones have had problems where memory allocated to objects that weren't cleaned up would continue to be used by those objects, even though the phones have been turned off and on, eventually rendering the phone useless when all memory is used and with no way of clearing the memory unless the phone is sent to the manufacturer for a hard reset.

Another problem with using C++ is the lack of portability of the compiled programs. Programs are compiled specifically to the platform they are run on. This means that programs made for Windows Mobile 5.0 won't necessarily work on newer or older versions of Windows Mobile, let alone on devices not using Windows Mobile.

2.5.3.2 C#

For a managed environment Microsoft has .NET Compact Framework, which makes it possible to use either C# or Visual Basic as implementation language. Being a managed environment, .NET takes care of all memory allocation as well as using garbage collection for cleaning up the memory. This removes any issues of persisting memory leaks across restarts of the client software.

Programs written in C# compiles to MSIL (MicroSoft Intermediate Language), which is the byte code language for the .NET framework. Using this intermediate language makes programs somewhat portable. They can be run on any version of Windows Mobile that has .NET Compact Framework installed, but not on any other architecture.

Being a modern language, C# comes with features not present in the other options, for example C# Generics[MSD05] and LINQ[MSD09]. The

language also features an extensive standard class library, even though it is not as huge as the one for the full version of .NET.

2.5.3.3 Java Micro Edition

Java Micro Edition (Java ME, formerly J2ME) is a widely used language for platform independent programs running on various mobile devices. Almost all devices comes with a Java environment (for example the Symbian platform) or has the possibility of installing one (for example the Windows platform). The only platform today that doesn't support Java is the iPhone OS for Apple's iPhone. This makes Java ME very portable, since it runs on pretty much anything.

Given its large distribution, Java ME has a lot of support. There are a lot of information on the internet as well as good tool support. Both Eclipse[1] and NetBeans[2] are good choices for development environments. There also exist many open source projects that can add support for features not included in the language, such as JSON support[3].

A problem with Java ME is that it is an ageing platform. It is based on Java 1.3, which has been around since the early 2000's. So no support for newer language constructs, such as Generics which was introduced with Java 1.5. Its standard library is also limited compared to Java Standard Edition.

2.5.4 Conclusion

Our choice of implementation language should adhere to the requirements described in Section 2.5.2. Requirements 2-4 are fulfilled for the three different choices, they are all capable of communicating on a network and storing data locally. The difference is in portability, how easy it is to develop in the given language and how much prior experience we have with it.

2.5.4.1 C++

If the only requirement for the client was to make it as efficient as possible, C++ would be the best choice, since it is unmanaged and thus doesn't sit on top of another framework. Development is more difficult than with a managed language since the developers need to take care of garbage collection themselves.

[1] http://www.eclipse.org
[2] http://www.netbeans.org
[3] http://www.json.org

Portability is not high with a C++ based solution. It is tied very much to a single platform. There are no guarantees that a program will run on different versions of Windows Mobile. Especially not future versions, since they can have changed features that will break existing programs.

As a final note most of our development efforts during our studies have been done in other languages than C/C++, making our development experience with unmanaged languages low.

2.5.4.2 C#

C# as the implementation language seems as a good choice when it comes to features within the language and in the standard class library. Development can be done inside Visual Studio which have a positive impact on productivity.

The biggest downside is that it has to be run on a Windows platform, although that includes Windows CE, Windows Mobile 5, Windows Mobile 6 and any future versions of Windows Mobile.

Our programming experience with C# is high, it has been the main language for development during our time at the university. However that has been in relation to desktop and server development. We have never done any development for Windows Mobile devices.

2.5.4.3 Java Micro Edition

A big strength of Java ME is that it is platform independent and widely supported by mobile manufacturers. This makes it a good choice when it comes to developing highly portable mobile applications.

Java ME is not as feature rich as C# and the standard library is not as extensive as the .NET version, which will mean a higher development effort has to be put into it.

We have a lot of programming experience with the Java platform, which also includes previous experience with Java ME development.

2.5.4.4 Summary

Table 2.17 shows how the three languages evaluates according to our criteria. X denotes fully support, ÷ denotes does not support and + denotes partial support.

	C++	C#	Java
Support	X	X	X
Portability	÷	+	X
Language features	+	X	+
Managed code	÷	X	X
Programming experience	÷	+	X

Table 2.17: Evaluation criteria for the mobile client implementation language.

2.5.4.5 Our Choice

Our choice for implementation language has fallen on Java ME. This is mainly due to two things: Firstly it is the most portable option and secondly we have prior experience with it. And especially the portability aspect is very compelling, since our communication solution with more work can be extended into a general solution that can be used on almost any mobile device, new as well as old.

2.6 Communication

The communication protocol used for the system has to be fairly lightweight, since it has to function on a mobile device. It should be platform independent, since the client and service will be implemented in different programming languages. In addition it would also be nice to have a solution that is simple to develop and maintain. There are several ways that this can be accomplished in. We will take a look at the following three different ways:

1. Custom build network protocol.
2. SOAP based XML Web Services.
3. RESTful services.

To help determine which approach is the most suitable one for our purposes, we will examine the strength and weaknesses for each different way, by the use of an example based on a use case from the project.

2.6.1 Example

The use case we base our example on, is for a user to retrieve a daily plan. By providing a user name to the service, the users daily plan for the current day will be returned.

Table 2.18 shows the elements used to describe a daily plan. Table 2.19 shows the elements used to describe a single visit. Both daily plan and visit are kept more simple in this example that they will be in the actual implementation. This is because the example only needs to help us determine which approach to use and as such it is too time consuming to make a full implementation of the use case.

Name	Description
Username	Username of the employee
Name	Name of the employee
Date	The date the daily plan is valid
Visits	A list of the citizens the user has to visit

Table 2.18: The elements describing a daily plan.

Name	Description
Citizen	The name of the citizen
StartTime	The planned start time for the visit
EndTime	The planned end time for the visit

Table 2.19: The elements describing a visit.

For the purpose of this example, we have created 2 daily plans that are stored in an XML file. An excerpt of this file is shown in Figure 2.7, where it is the daily plan for the fictitious user John Doe that is used. The date for the visit is set to the current date. Figure 2.8 shows the C# class used to represent a daily plan in the system and Figure 2.9 shows the C# class for a visit.

In order to examine the communication that happens between clients and servers, we use the tool Fiddler[4], which is a proxy that logs all in- and outgoing traffic on HTTP(S) and allows the user to inspect the data in details.

2.6.2 Custom Build Solution

The main benefit of using a specialised custom protocol, is that it can be made very efficient, both in terms of the size of the information sent across the network as well as how many resources are used to process the information.

[4]http://www.fiddler2.com/fiddler2/

```
<dailyplan>
  <username>12AB</username>
  <name>John Doe</name>
  <visits>
    <visit>
      <citizen>Frederik Hansen</citizen>
      <starttime>08:00</starttime>
      <endtime>08:50</endtime>
    </visit>
    <visit>
      <citizen>Lisbeth Tyregod</citizen>
      <starttime>09:00</starttime>
      <endtime>10:30</endtime>
    </visit>
    <visit>
      <citizen>Kit Krum</citizen>
      <starttime>10:35</starttime>
      <endtime>11:10</endtime>
    </visit>
  </visits>
</dailyplan>
```

Figure 2.7: Excerpt from a XML file containing daily plans.

```
public class DailyPlan
{
    public string Name { get; set; }
    public string Date { get; set; }
    public List<Visit> Visits { get; set; }
}
```

Figure 2.8: The DailyPlan class.

```
public class Visit
{
    public string Citizen { get; set; }
    public string StartTime { get; set; }
    public string EndTime { get; set; }
}
```

Figure 2.9: The Visit class.

Building a custom solution comes with some major disadvantages though. There is a lot more development involved, since the communication stack has to be made specifically for the solution and thus increasing the complexity of the solution quite substantially. There are also no existing frameworks that can be used to speed up development. Another complication is that there are no inherent platform independence, since the solution isn't build using well-known frameworks.

We have chosen not to implement a version of our example using this approach. Simply because it is a too time consuming task to build it. There are many decisions to make, some of which are: Which protocol to use? TCP/IP? UDP?. Which format should the response have? XML? HTML? binary?. What about the client? Is Fiddler enough or does we have to also develop a client to see a result?

2.6.3 SOAP Based XML Web Services

XML Web Services (WS-*) are a well proven technology for making platform independent communication using the HTTP protocol. It uses a WSDL file to describe the services that it offers. From this file a client is then able to see how a request should be formed, in order to use the services.

It is very easy to make an XML Web Service on the .NET platform. A simple one, as the one used in our example, is literally done in a matter of minutes.

2.6.3.1 Retrieve a Daily Plan

The method to retrieve a daily plan is shown in Figure 2.10. It opens the XML file containing the daily plans, finds the plan that matches the supplied user name and then creates a DailyPlan instance containing the name and list of visits for that user. The Date element is set to the current date. A new instance of the Visit class is created for each visit belonging to the daily plan.

In order to send values along with a request, in our case the user name of a specific user, the value has to be specified within the soap envelope sent to the server. Figure 2.11 shows the request send to the server to get the daily plan for the user with username 12AB. The response from the server is shown in Figure 2.12.

```
[WebMethod]
public DailyPlan getDailyPlan(string username)
{
    XDocument partsXML = XDocument.Load(Server.MapPath("dailyplans.xml"));

    var dailyplan = from plan in partsXML.Descendants("dailyplan")
                    where plan.Element("username").Value == username
                    select new DailyPlan
                    {
                        Name = plan.Element("name").Value,
                        Date = DateTime.Now.ToShortDateString(),
                        Visits = (from visit in plan.Descendants("visit")
                                  select new Visit
                                  {
                                      Citizen = visit.Element("citizen").Value,
                                      StartTime = visit.Element("starttime").Value,
                                      EndTime = visit.Element("endtime").Value
                                  }).ToList()
                    };

    return dailyplan.FirstOrDefault();
}
```

Figure 2.10: Method to retrieve a daily plan.

```
POST /dat6/service1.asmx HTTP/1.1
Host: cs.soelsoft.dk
Content-Type: text/xml; charset=utf-8
SOAPAction: "http://company.com/getDailyPlan"
Content-Length: 360

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <getDailyPlan xmlns="http://company.com/">
      <username>12AB</username>
    </getDailyPlan>
  </soap:Body>
</soap:Envelope>
```

Figure 2.11: The entire request message sent to the service in order to invoke the getDailyPlan method with the username value of 12AB.

```
HTTP/1.1 200 OK
Date: Mon, 04 May 2009 11:07:58 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
X-AspNet-Version: 2.0.50727
Cache-Control: private, max-age=0
Content-Type: text/xml; charset=utf-8
Content-Length: 717

<?xml version="1.0" encoding="utf-8"?><soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"><soap:Body><getDailyPlanResponse
xmlns="http://company.com/"><getDailyPlanResult><Name>John Doe</Name><Date>04-
05-2009</Date><Visits><Visit><Citizen>Frederik
Hansen</Citizen><StartTime>08:00</StartTime><EndTime>08:50</EndTime></Visit><Vi
sit><Citizen>Lisbeth
Tyregod</Citizen><StartTime>09:00</StartTime><EndTime>10:30</EndTime></Visit><V
isit><Citizen>Kit
Krum</Citizen><StartTime>10:35</StartTime><EndTime>11:10</EndTime></Visit></Vis
its></getDailyPlanResult></getDailyPlanResponse></soap:Body></soap:Envelope>
```

Figure 2.12: The response message recieved from the xml web service when the getDailyPlan method is invoked with the username value of 12AB.

2.6.4 RESTful Web Services

Before looking at RESTful web services, we should first take a look at REST itself. REST is short for **RE**presentational **S**tate **T**ransfer and was first introduced by Roy Fielding in his doctoral dissertation in 2000 [Fie00]. Fielding was one of the authors of the HTTP/1.0 specification and the primary architect behind HTTP/1.1. REST is an architectural style and not a model. This means that it follows principles and not rules. The four principles behind REST are:

- **Resources.**
  Resources are the primary abstraction of information in REST. A resource is identified through a URI (**U**niform **R**esource **I**dentifier). The URI does not have to be static, it can be dynamic, for example the URI "http://cs.soelsoft.dk/dat6/Service1.svc/dailyplan/12AB" will return the daily plan for the user 12AB. If there are changes in visits to perform on the current day in the underlying system, the URI does not change.

- **Uniform Interface.**
  To manipulate resources REST uses the verbs defined by the HTTP protocol, more precisely the four verbs GET, POST, PUT and DELETE (the HTTP 1.1 protocol also defines the verbs OPTIONS, HEAD, TRACE and CONNECT). GET is used to retrieve information, such as getting a list of parts. POST is used to update already existing resources, for example the price of a part, PUT is used to add new

resources or completely replace existing resources, DELETE is used to delete resources.

- **Descriptive Messages.**
  A message in REST should contain all information needed to understand the message. This is done with the use of the HTTP header, for example if the message contains a JPEG image, the headers Content-type is set to "image/jpeg".

- **Stateful Interactions Through Hyperlinks.**
  All interaction with resources are stateless. This means that every request should contain all the information needed to understand the request. All interactions happens through hyperlinks, so if a request returns information about another resource, the URI for the resource should be included.

2.6.4.1 The Example Implementation

The implementation of the example uses Windows Communication Foundation (WCF) as a framework for the service. Services in WCF can, with a little effort, be designed to use a REST friendly way of communication. The service consist of a config file that describes the HTTP endpoint, which is shown in Figure 2.13. An interface that describes the methods to use and finally an actual implementation of the interface. The address to the service is "http://cs.soelsoft.dk/dat6/Service1.svc".

2.6.4.2 Retrieve a Daily Plan

Figure 2.14 shows the interface that belongs to the parts list resource. It defines that a client should access the resource via HTTP GET, that the URI for the resource is the service address with "dailyplan/{username}" appended, where {username} is a variable that automatically gets used in the signature, and finally that responses will be formatted as JSON objects[5].

Figure 2.15 shows the actual implementation of the interface. The implementation is very similar to the SOAP based approach. The only difference is in getting the physical address for the XML file.

The request sent to the server is shown in Figure 2.16, and Figure 2.17 shows the response.

[5]Visit http://json.org for an explanation of JSON

```
<system.serviceModel>
  <behaviors>
    <endpointBehaviors>
      <behavior name="RESTFriendly">
        <webHttp/>
      </behavior>
    </endpointBehaviors>
    <serviceBehaviors>
      <behavior name="MyServiceTypeBehaviors" >
        <serviceMetadata httpGetEnabled="true" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <services>
    <service name="WcfWebService.Service1">
      <endpoint address="" behaviorConfiguration="RESTFriendly" binding="webHttpBinding"
        contract="WcfWebService.IService1" />
    </service>
  </services>
</system.serviceModel>
```

Figure 2.13: an excerpt of the web.config file that shows the definition of the REST enabled endpoint for the service.

```
[OperationContract]
[WebGet(UriTemplate = "dailyplan/{username}",
    ResponseFormat = WebMessageFormat.Json)]
DailyPlan RetrieveDailyPlan(string username);
```

Figure 2.14: The interface that defines the method to retrieve a daily plan.

```
public DailyPlan RetrieveDailyPlan(string username)
{
    XDocument partsXML = XDocument.Load(
        HostingEnvironment.ApplicationPhysicalPath + "dat6\\dailyplans.xml");

    var dailyplan = from plan in partsXML.Descendants("dailyplan")
                    where plan.Element("username").Value == username
                    select new DailyPlan
                    {
                        Name = plan.Element("name").Value,
                        Date = DateTime.Now.ToShortDateString(),
                        Visits = (from visit in plan.Descendants("visit")
                                  select new Visit
                                  {
                                      Citizen = visit.Element("citizen").Value,
                                      StartTime = visit.Element("starttime").Value,
                                      EndTime = visit.Element("endtime").Value
                                  }).ToList()
                    };

    return dailyplan.FirstOrDefault();
}
```

Figure 2.15: The implementation to retrieve a daily plan.

```
GET /dat6/service1.svc/dailyplan/12AB HTTP/1.1
Host: cs.soelsoft.dk
```

Figure 2.16: The entire request message sent to the RESTful service in order to call the getDailyPlan method with the username value of 12AB.

```
HTTP/1.1 200 OK
Date: Mon, 04 May 2009 11:24:07 GMT
Server: Microsoft-IIS/6.0
X-Powered-By: ASP.NET
X-AspNet-Version: 2.0.50727
Cache-Control: private
Content-Type: application/json; charset=utf-8
Content-Length: 247

{"Date":"04-05-2009","Name":"John Doe","Visits":[{"Citizen":"Frederik
Hansen","EndTime":"08:50","StartTime":"08:00"},{"Citizen":"Lisbeth
Tyregod","EndTime":"10:30","StartTime":"09:00"},{"Citizen":"Kit
Krum","EndTime":"11:10","StartTime":"10:35"}]}
```

Figure 2.17: The response message received from the RESTful service when the getDailyPlan method is called with the username value of 12AB.

2.6.5 Conclusion

There are 3 main requirements that our choice should fulfil: It should be platform independent, it should be fairly lightweight and it should be simple to develop and maintain.

2.6.5.1 Custom Build Solution

The custom build solution has the potential to be very lightweight, since the size of message can be kept as low as possible. Also the parsing of the messages can be made very efficient, since it only has to parse fixed cases.

The biggest problem with a custom build solution, is that it is exactly that: Custom build. That makes it complex to develop and maintain and this is not what we are looking for.

2.6.5.2 SOAP Based XML Web Service

The biggest advantage with a WS-* web service is that it is a well-known way of making an online service and thus that exists good tooling support for it. The example service is created literally within a few minutes. This makes development and implementation, particularly of the service itself very fast and easy.

The problem comes on the client side though. Java Micro Edition does not have built in support for WS-* web services, just as there is no built in XML parsers. There are an extension to the specification available that

makes Java Micro Edition able to use WS-* services (JSR 172), but since this has to be implemented by the phone manufacturer and only newer devices have this, we have chosen to disregard it. This means that we have to use either our own solution or import an already built one (it could for example be kSOAP 2[6]).

Another issue with a WS-* based solution, is that it introduces significant overhead. The communication requests and responses from Section 2.6.3, shows that there is a lot of data to transmit when using SOAP. Parsing XML files can also be a problem for devices with limited resources, simply because of the lack of memory and processing power.

2.6.5.3 RESTful Web Service

The support for making RESTful services, especially using WCF, has increased a lot recently. It doesn't take much more effort to setup a new RESTful based service compared to a WS-* based service. This makes it easy to build and maintain the service.

The only thing needed on the client to communicate with the service is the use of HTTP, and all Java Micro Edition implementations are capable of this, since it is part of the specification of CLDC 1.0 that it supports HTTP communication. There is a limitation though, and that is that only the verbs GET and POST are supported which limits the options on the server side, since operations using PUT and DELETE has to be defined through the use of POST.

WCF gives the developer two choices in regards to the format of requests and responses for the service, namely XML or JSON. As described in the previous section, Java Micro Edition does not contain an XML parser, however that is also the case regarding a JSON parser. So no matter if the choice ends with being XML or JSON, there needs to be implemented a parser for it. But since the JSON format is more lightweight than XML, so is the resulting parser making it more suited for limited devices.

With regards to the messages sent using JSON, the requests and responses in Section 2.6.4 shows that the overhead is limited.

2.6.5.4 Summary

To help us make a decision, we have made a table where we have evaluated the three different ways for communication. X denotes fully support, ÷ denotes does not support and + denotes partial support.

[6]http://ksoap2.sourceforge.net/

	Custom	WS-*	RESTful
Use of existing server framework	÷	X	X
Use of existing client framework	÷	+	+
Lightweight messages	X	÷	+
Lightweight parsing of messages	X	÷	+
Simple to build and maintain	÷	X	X

Table 2.20: Evaluation criteria for communication choice.

2.6.5.5 Our Choice

Since the clients have to run on limited devices, our emphasis is on keeping the messages and parsing of the messages as light as possible. This suggest that we should choose a custom build solution, however this choice will add a lot of complexity to the project. Using a RESTful service will give us an existing framework to use, as well as keeping the message overheard to a minimum, especially when using JSON as the message format. For this reason we have chosen RESTful services as our communication architecture.

2.7 Security

Since the information that we have to send between the client and service contains sensitive information about citizens (such as social security number, name and address), we have to protect the data during transmission.

There are two security issues to look at: First, what kind of encryption to use and second, the transmission format.

2.7.1 Encryption

There are many encryption algorithms that have been developed through the years. Notable algorithms includes the Data Encryption Standard (DES), Triple DES and Advanced Encryption Standard (AES).

DES was developed in the 1970's and are no longer considered secure. Triple DES is an enhancement to DES that uses the DES cipher three times on each block, in order to increase the key size without making a new algorithm. AES is based on the Rijndael algorithm and is described in [DR02].

AES is the recommended encryption standard for the US Government and is approved for SECRET with 128 bit keys and TOP SECRET with 192 and 256 bit keys[7]. For these reasons, we have chosen to use AES for our solution.

2.7.2 Implementation

There are no cryptography implementations in the class library for Java ME, therefore we have to either develop our own implementation of AES or use a third party implementation. We have chosen to use a third party implementation.

The implementation we have chosen is made by a group called The Legion of the Bouncy Castle[8]. We have chosen their implementation because they have both Java ME and C# implementations as well as being free of charge for use and distribution.

2.7.3 Transmission Format

Another security issue are regarding the format of the transmitted data. If we use the DailyPlan class from Section 2.6 as an example, we can choose to encrypt the values for each attribute or we can choose to encrypt the entire object.

If we choose to encrypt the values of each attribute, an example of what a resulting json object would like are shown in Figure 2.18. A huge disadvantage with this approach is that we actually disclose the nature of the data that is send, even though the actual values is unreadable. An advantage though is that the implementation especially on the service is simple for the communications part, since WCF is able to automatically convert objects send and received to and from json.


```
{"Date":"GwXRd8HYuPIm0-Dotdd","Name":"N7KfoQF","Visits":
[{"Citizen":"4MPcygL","EndTime":"u0_dPj","StartTime":"MPcygL"},
{"Citizen":"jFXpGwX","EndTime":"P3Ixfw","StartTime":"HlUd2y"},
{"Citizen":"WXRd8","EndTime":"MrdePm","StartTime":"Xadg_Ww0"}]}
```


Figure 2.18: A DailyPlan with encrypted attribute values.

Another approach is to encrypt everything. The advantage of this method is that besides being encrypted there isn't disclosed any information about

[7]The policies for using AES in the US Government can be found here: http://csrc.nist.gov/groups/ST/toolkit/documents/aes/CNSS15FS.pdf

[8]http://www.bouncycastle.org/

the nature of the data. A big disadvantage is that we cannot take advantage of WCF's capabilities for converting objects to and from json and thus would have to make a custom transport protocol which would complicate the service implementation.

A third option is to combine the two. By making a simple object that is used as a carrier for any encrypted data, we will still retain the ability to use automatic conversion as well as not disclosing information about the nature of the encrypted data. Table 2.21 shows the only element belonging to the EncryptedData class and Figure 2.19 shows an example of what an EncryptedData objects json representation could look like.

Attribute	**Data type**
Data	String

Table 2.21: The element describing the EncryptedData object.


```
{"Data":"2MHIVZJepcbnAgpWPn7P3IxfwN1kHQJW4caSbC4MPcygLF1ZyXadg_Ww0-
N38kR_U7spml5Hku0_dPjde6LHx0WHlUd2yeJ4jfTt2FAtOzOQK6KcSzPXtNnSmoH-
4vqr-tTYEEuiIw7iwDZx7lh7vegdaQco8erWLNVVDjnqpP9TCk835-
wASLKQw0xBw60M17Jmag2BIx3LcGPWn0I328vrg0N1oV"}
```


Figure 2.19: A EncryptedData object with a completely encrypted DailyPlan.

We have chosen to use an EncryptedData object as transmission format, since not only will it allow us to use WCF's built in conversion ability, but it will also give us a central place to do decryption and encryption, such that objects added to an EncryptedData instance will be automatically encrypted and automatically decrypted when retrieved again.

Chapter 3

Design Overview

In this chapter we will give an overview of the design of the solution. We will describe the objects that is needed for communication between the client and the service, since they have to be implemented in different languages and run on different platforms.

We will also describe the interfaces that the service expose, such that we can built a client that uses the service.

Figure 3.1 shows an overview of the solution. An employee uses a mobile device to interact with a service at Herning Municipality. The service does all communication with a database.

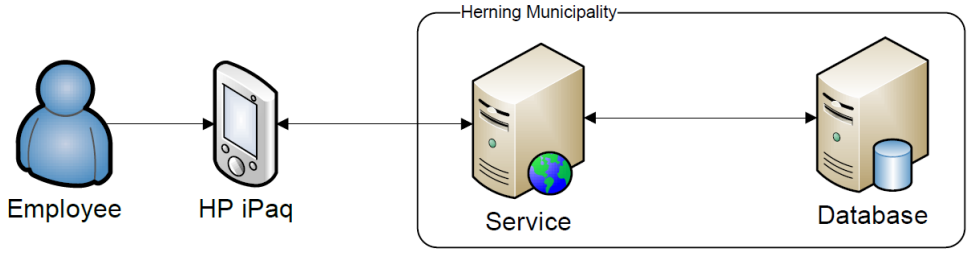

Figure 3.1: Overview of the solution

3.1 Architecture

Our solution consists of three main parts: A data access layer, a service and a client. Each with its own namespace, which is d620a.HomeCare.DAL, d620a.HomeCare.Service and d620a.HomeCare.Client respectively. There is a fourth part that contains the objects that are passed between the other

three namespaces. This part belongs in the namespace d620a.HomeCare.Models. The four namespaces and their dependencies are shown in Figure 3.2.

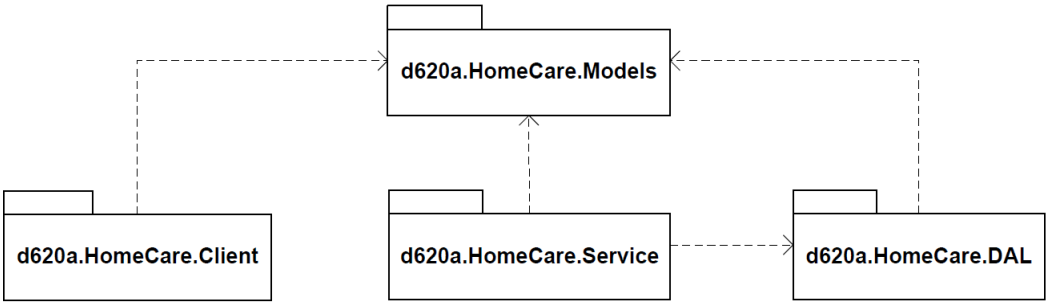

Figure 3.2: The four namespaces and their dependencies

In the following sections we will take a more detailed look at the four different parts.

3.2 Models

The Models namespace contains the classes and interfaces that are used for communication between the other 3 parts. In this section we will take a look at the members of this namespace.

3.2.1 Data Access

We use the Repository pattern for access to the data. This pattern makes it easier for us to test the system during development, since the service can be developed independent from the data access layer and just use a simple implementation of the interface to provide data. The class diagram of the interface is shown in Figure 3.3. The interface defines three methods:

- **GetDailyPlan:** This method retrieves daily plan for the given employee id on the given date.
- **UpdateDatabase:** Updates the repository with the provided UpdateObject. Returns true if the update was successful.
- **GetEmployeeId:** Returns the employee id that belongs to the user with the supplied username and password. Returns null if no user exist.

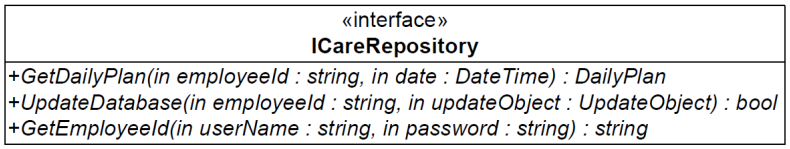

Figure 3.3: The ICareRepository interface

3.2.2 Daily Plan

We use three classes to describe the daily plan for an employee. These classes are shown in Figure 3.4. A daily plan is built by the data access layer and sent to the client.

- **DailyPlan:** Has information related to the daily plan. Such as the employee's name, the date and the list of visits that the employee has to do.
- **Visit:** Has information related to a specific visit. This information includes who to visit, where they live and the time frame the employee should be at the citizen. It also contains a list of the tasks that the employee has to do.
- **Task:** Has information related to a single task, such as what to do and how long it is scheduled to take to do.

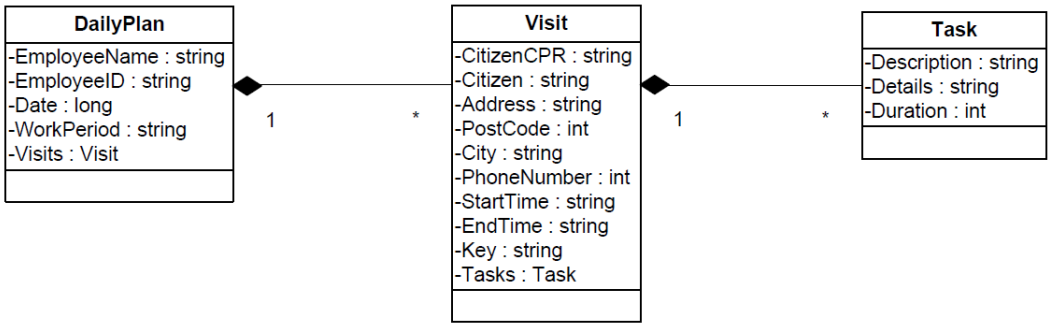

Figure 3.4: The classes used to describe a daily plan.

3.2.3 Update Object

We use two classes to describe the information that needs to be updated, after an employee has carried out his/her daily plan. Figure 3.5 shows the

two classes. It is the client that builds the UpdateObject and then sends it to the data access layer through the service.

- **UpdateObject:** Stores start- and end times for the watch as well as list of the visits that has been carried out.
- **DeliveredVisit:** Has information regarding a performed visit. Information includes who was visited, start and end times of the visit and the date. The reason for both a planned day and a actual day, is that the date shifts if the actual start time of the visit is 00:05, but was planned to start at 23:55.

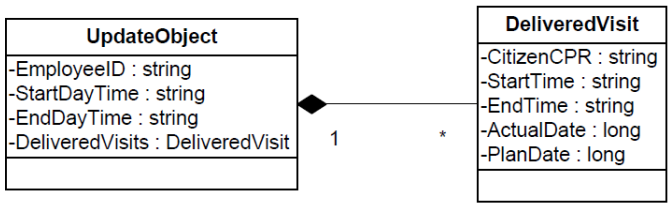

Figure 3.5: The classes used to describe the information to update in the database.

3.2.4 Login

The LoginUser class is used to store an employee's login credentials. It is send encrypted to the service to allow the service to verify that the employee has the rights to perform the requested action. Figure 3.6 shows the class.

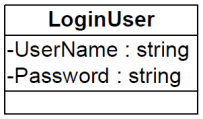

Figure 3.6: The class used to describe the login credentials of a user.

3.2.5 Encrypted Data

Two classes are used to hold and encrypt/decrypt objects that should be transmitted. These two classes are shown in Figure 3.7.

- **EncryptedData:** Contains the object to be transmitted. The object is serialised to and from json as well as encrypted and decrypted automatically.

- **Encryptor:** Does the actual encryption and decryption on byte arrays.

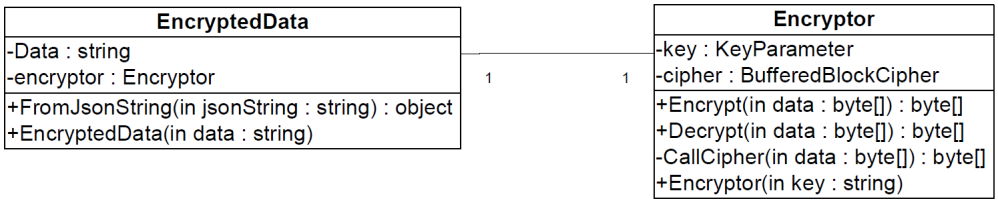

Figure 3.7: The class used to describe the data that are transferred across the network.

3.3 Data Access Layer

The data access layer has two main purposes. To generate the daily plan and to update the database with new information.

3.3.0.1 Generating the Daily Plan

The daily plan is generated based on two parameters; EmployeeID and Date. The output of the generation is either a daily plan, which contains all the correct information for the given date, or a completely empty daily plan, if the request happens on a day without work. There are several issues to consider.

- **Information Quality:** The information have to be as correct as possible. This is possible in many cases but certain design flaws within the database means that it will not always be. In the case of missing information default values are used which allow the employee to do their own interpretations of what is needed.
- **Response Time:** The response time should be good. This is a must since we are working on making a system which solve the problems of the current system.

3.3.1 Updating the Database

The database is updated based on an UpdateObject received from the mobile client. The output is a Boolean which indicate if the object was successfully received and stored. There are several issues to consider.

- **Data Quality:** The data inserted into the database has to be of the highest quality. High quality means that rows are only inserted one time and the inserted data is valid. Always inserting valid data might not be possible since the database already contains invalid data. In these cases the inserted values will be the same as the invalid values.

- **Data Loss and Response Time:** The system should be fast to respond that the data have been received and stored. This is not possible if inserting directly into the database. Instead an intermediate recovery system is used to ensure that no data is lost and give fast response time. This can be done since the data only have to be inserted into the recovery system before a response is generated.

3.4 Service

The service binds the entire solution together. It is responsible for communicating with both clients and the data access layer. The data access layer is used through the ICareRepository interface, described in Section 3.2.1. The following sections outline requirements for the implementation of the service.

3.4.1 Service Interface

The service needs to expose two methods. One method is for retrieving a daily plan and the other is for receiving an update to store. This section contains the needed information to access those two methods.

3.4.1.1 Daily Plan

This is the method that retrieves a daily plan. The information needed to access this method is as follows:

- **URI:** "dailyplan/login", where login is an EncryptedData json string containing a LoginUser object for the user to retrieve the daily plan for. The EncryptedData object should be sent as a url safe base64 string, without any padding.

- **HTTP Verb:** GET

- **Response:** The response should be an EncryptedData object containing a DailyPlan instance.

- **Status Codes:** 200 OK, if the daily plan has been successfully created. 403 Forbidden if the login credentials are wrong. 400 Bad Request if the login cannot be cast into a LoginUser object. 500 Internal Server Error if there are other errors.

3.4.1.2 Update

This is the method that updates the database with an UpdateObject. The information needed to use this method is as follows:

- **URI:** "update/login", where login is an EncryptedData json string containing a LoginUser object for the user to retrieve the daily plan for. The EncryptedData object should be sent as a url safe base64 string, without any padding.
- **HTTP Verb:** POST
- **Request:** The object received should be an EncryptedData object containing an updateObject instance.
- **Status Codes:** 200 OK, if the update has been successfully added. 403 Forbidden if the login credentials are wrong. 400 Bad Request if the login cannot be cast into a LoginUser object. 500 Internal Server Error if there are other errors.

3.4.2 JSON

Since we are using WCF for the communication part for the service, the framework takes care of serialising objects to and from json, as long as those objects are a part of the signature of the service methods. However, since the Data attribute within the EncryptedData object contains a json string representation of another object, we need to be able to convert to and from json objects.

Instead of making our own implementation of a json parser, we have chosen to use an Open Source parser called Json.NET, which is available from Microsoft's community site for Open Source software[1].

3.5 Client

The client is the only means for interacting with the service. It will be implemented in Java and built for the Java Micro Edition platform. There are

[1] http://json.codeplex.com/

three main areas of requirements to look at: The user interface, communication and storing of data.

3.5.1 User Interface

The users of the systems are employees in the home care sector at Herning Municipality in Denmark. This means that we will implement the user interface in Danish, since we cannot guarantee that the users understand English. Also the data stored in the database is in Danish and it will only confuse people to be presented with different languages.

3.5.2 Communication

In order to not freeze the UI during communication sessions, the communication should happen in a separate thread. To keep the user informed of the communication process, a waiting screen should be displayed. It is important that the screen always shows some form of progress, such that the user don't think the system has crashed.

3.5.3 Storing Data

There are several items that needs to be stored in persistant storage on the client. The login credentials, the daily plan received from the service and the information that makes up the UpdateObject.

The reason for storing these informations in persistant storage is to allow the client application to be closed without loosing information. It will not be good if the users cannot make a phone call without loosing all data.

3.5.4 JSON

Java Micro Edition does not have built in support for json. So in order to use json objects, we have to either built a json parser or use a third party parser. We have chosen to use the org.json.me parser which is available on the json.org website[2]

[2]Link to the implementation: http://www.json.org/java/org.json.me.zip

Chapter 4

Data Access

Section 2.2 analyses the database provided by Herning Municipality. This database have some design problems, which the data access layer should handle. Section 2.2 is split into two pieces; analysis of the data retrieval to generate the daily plan and analysis of the update procedure. This chapter will also be split into these two piece.

4.1 Generating the Daily Plan

This section will focus on the generation of the daily plan. The generation require three queries; MainQuery, NeedTypeQuery and ProcedureQuery. An example will show how the daily plan is generated. The example is shown for an employee with one visits and the visit contains two services. This is a simplified example and should only be considered like this. The input to GetDailyPlan method is shown in Table 4.1.

Attribute	Value
EmployeeID	9999999999
Date	01-10-2008

Table 4.1: GetDailyPlan Input Data.

Attribute	Value
EmployeeName	
Date	01-10-2008
WorkPeriod	

Table 4.2: DailyPlan object.

Before the first query is executed the DailyPlan object is created and the date is added, it is not possible to add WorkPeriod and EmployeeName at this time since we don't know the values. This leave the DailyPlan object shown in Table 4.2. No visits or services are created yet. Each object will be represented as a table containing two or three columns. The first column will be the attribute name, the second will be the current value of the attribute

and the last if present will be the source. The source column is only used when executing queries to show which attributes contains the object values within the database.

4.1.1 MainQuery

The MainQuery is based on INI, DATE_FR, DATE_TO and WEEKDAY_NO from AID_TIME_HISTORY. These four attributes can be used to identify all services which have to be delivered by a given employee (INI) on a given weekday (WEEKDAY_NO) from date (DATE_FR) to date (DATE_TO). This means that all services which satisfy the following $INI = EmployeeID$, $DATE\_FR \leq Date \leq DATE\_TO$ and $Date.WeekDay = WEEKDAY\_NO$ have to be delivered by the given employee on the given day. A note on this part is that DATE_TO is null-able so if the value is null it is considered to be larger than date.

The MainQuery extract data from AID_TIME_HISTORY, CLIENT, PERSONNEL and POSTCODE and is based on the analysis in Section 2.2. This means that joins are only done on foreign key constraints. The query created by our example can be found on Listing 4.1. Based on this query all visits and services are created on the DailyPlan object. This means that the example DailyPlan object can be found on Table 4.3, the Visit object on Table 4.4, the first Service object on Table 4.5 and the second Service object on Table 4.6. Furthermore a list of objects containing the values MODULE_TYPE_NO, AID_NEED_TYPE_NO and AID_LEVEL are created. These values can be directly mapped to the services on the daily plan. The values are used by the next query.

Listing 4.1: Main Query.

```
select CLIENT.C_CPR, CLIENT.NAME, CLIENT.SURNAME, CLIENT.
    KEY_NO, CLIENT.ADDR_STRT, CLIENT.RES_PHONE_NO,
    AID_TIME_HISTORY.START_TIME, AID_TIME_HISTORY.
    MODULE_TYPE_NO, AID_TIME_HISTORY.AID_NEED_TYPE_NO,
    AID_TIME_HISTORY.AID_LEVEL, AID_TIME_HISTORY.NOTE,
    PERSONNEL.NAME, PERSONNEL.SURNAME, POSTCODE.POSTCODE,
    POSTCODE.CITY
from AID_TIME_HISTORY join CLIENT on AID_TIME_HISTORY.
    C_CPR_NO = CLIENT.C_CPR_NO join POSTCODE on CLIENT.
    POSTCODE = POSTCODE.POSTCODE join PERSONNEL on
    AID_TIME_HISTORY.INI = PERSONNEL.INI
where AID_TIME_HISTORY.INI = '9999999999' and
    AID_TIME_HISTORY.DATE_FR <= '08-10-01' and (DATE_TO >= '
    08-10-01' or DATE_TO IS NULL) and WEEKDAY_NO = 3
order by START_TIME
```

Attribute	Value	Source
EmployeeName	NoName	PERSONNEL.NAME PERSONNEL.SURNAME
Date	01-10-2008	
WorkPeriod		

Table 4.3: DailyPlan object.

Attribute	Value	Source
CitizenCPR	9999999999	CLIENT.C_CPR_NO
Citizen	NoName	CLIENT.NAME CLIENT.SURNAME
Address	NoWhere Street 99	CLIENT.ADDR_STRT
PostCode	9999	POSTCODE.POSTCODE
City	NoWhere	POSTCODE.CITY
PhoneNumber	99999999	CLIENT.RES_PHONE_NO
StartTime	0930	AID_TIME_HISTORY.START_TIME
EndTime		
Key	None Needed	CLIENT.KEY_NO

Table 4.4: Visit object.

Attribute	Value	Source
Description		
Duration		
Details	None	AID_TIME_HISTORY.NOTE

Table 4.5: First Service object.

Attribute	Value	Source
Description		
Duration		
Details	Something	AID_TIME_HISTORY.NOTE

Table 4.6: Second Service object.

4.1.1.1 AddWorkPeriod

The method AddWorkPeriod is invoked after the MainQuery is executed and the result added to the DailyPlan object. This method will add one of three possible work periods to the DailyPlan object. The possible WorkPeriods can be found in Table 4.7 which also contains an English translation of each possible value. The WorkPeriod is assigned based on the Visits contained in the DailyPlan object. Each visits StartTime is counted within one of these ranged. The range with the most StartTimes are assigned as the work period for the DailyPlan object. From our example we only have one Visit object which have StartTime 0930. The WorkPeriod is assigned the value Day since the only visit is in this range. This gives the DailyPlan object shown in Table 4.8.

Danish	English	Period
Dag	Day	07:00-15:00
Aften	Evening	15:00-23:00
Nat	Night	23:00-07:00

Table 4.7: WorkPeriod Translation.

Attribute	Value
EmployeeName	NoName
Date	01-10-2008
WorkPeriod	Day

Table 4.8: DailyPlan object.

4.1.2 NeedTypeQuery

The NeedTypeQuery is based on MODULE_TYPE_NO and AID_NEED_TYPE_NO from AID_LEVEL. The reason it is not based on AID_LEVEL as well is that we have found invalid AID_LEVEL values in AID_TIME_HISTORY. This query is generated based on the list generated by the MainQuery and it is only generated if our DailyPlan contains any services. The statements used for this query are $AID\_NEED\_TYPE\_NO = Task.AidNeedTypeNo$ and $MODULE\_TYPE\_NO = Task.ModuleTypeNo$.

The query extract data from AID_NEED_TYPE and AID_LEVEL since a foreign key constraint exist. An example of the query based on our example can be found on Listing 4.2. Only the two example Service objects are modified by the result of this query. The two new Service objects can be found on Table 4.9 and Table 4.10. Table 4.10 shows the second Service object contains a procedure code. This procedure code is added to a list which is used to generate the last query.

Listing 4.2: Getting Task Information.

```
select AID_NEED_TYPE.MODULE_TYPE_NO, AID_NEED_TYPE.
    AID_NEED_TYPE_NO, AID_LEVEL.AID_LEVEL, AID_NEED_TYPE.
    DESCRIPTION, AID_LEVEL.DURATION, AID_NEED_TYPE.
    PROCEDURE_CODE
from AID_NEED_TYPE join AID_LEVEL on (AID_NEED_TYPE.
    MODULE_TYPE_NO = AID_LEVEL.MODULE_TYPE_NO and
    AID_NEED_TYPE.AID_NEED_TYPE_NO = AID_LEVEL.
    AID_NEED_TYPE_NO)
where (AID_NEED_TYPE.MODULE_TYPE_NO = 1 and
    AID_NEED_TYPE_NO = 15) or (AID_NEED_TYPE.MODULE_TYPE_NO
    = 1 and AID_NEED_TYPE_NO = 19)
```

Attribute	Value	Source
Description	Cleaning	AID_NEED_TYPE.NEED_TEXT
Duration	20	AID_LEVEL.DURATION
Details	Something	

Table 4.9: First Service object.

Attribute	Value	Source
Description	AAF6	AID_NEED_TYPE.NEED_TEXT
Duration	10	AID_LEVEL.DURATION
Details	Something	

Table 4.10: Second Service object.

4.1.2.1 CalculateEndTime

The method CalculateEndTime is called after executing and storing the result of the second query. This method use the duration of the Service objects to calculate the EndTime for each visit. The formula used is $EndTime = StartTime + \sum Service.Duration$ where $Service.Duration$ is the duration of a service which have to be delivered on the given visit. Since the two Service objects in our example has durations 20 minutes and 10 minutes respectively, it will take the employee 30 minutes to finish the visit. This value is added to StartTime to calculate the EndTime for the visit. The updated Visit object can be found in Table 4.11.

Attribute	Value
CitizenCPR	9999999999
Citizen	NoName
Address	NoWhere Street 99
PostCode	9999
City	NoWhere
PhoneNumber	99999999
StartTime	0930
EndTime	1000
Key	None Needed

Table 4.11: Visit object.

4.1.3 ProcedureQuery

The ProcedureQuery is based on the PROCEDURE_CODE attribute from the PROCEDURE_CODE table. The query is generated using the list of procedure codes generated by NeedTypeQuery and it is only generated if the list is not empty. It uses the statement $PROCEDURE\_CODE = procedurecode$.

The query extract data from PROCEDURE_CODE. This is done to translate all the Service objects with procedure codes as description. In our example this is only the second Service object. An example based on our example can be found on Listing 4.3. The updated second Service object can be found in Table 4.12.

Listing 4.3: Procedure Code Translation.

```
select PROCEDURE_CODE, DESCRIPTION
from PROCEDURE_CODE
where PROCEDURE_CODE = 'AAF6'
```

Attribute	Value	Source
Description	Medicine	PROCEDURE_CODE.DESCRIPTION
Duration	10	
Details	Something	

Table 4.12: Second Service object.

4.1.3.1 SortVisitsForNight

The method SortVisitsForNight is invoked after executing ProcedureQuery and updating the DailyPlan object. This method is quite simple and it is only executed if the WorkPeriod of the DailyPlan object is Night. The method switch around the Visits so those before midnight are first and those after midnight are after. Since our DailyPlan object example is for Day this method is never invoked. Table 4.13 shows 4 visits on a DailyPlan object with WorkPeriod night. Table 4.14 shows the same DailyPlan object after this method is invoked.

Visit	StartTime
Visit 1	0010
Visit 2	0030
Visit 3	2315
Visit 4	2345

Table 4.13: Before.

Visit	StartTime
Visit 3	2315
Visit 4	2345
Visit 1	0010
Visit 2	0030

Table 4.14: After.

4.1.4 Testing

The three methods executing the queries are private methods to our data access class, which also implements the interface described in Section 3.2.1. The private methods have been tested with unit tests using reflection. This has been done to ensure correct behaviour of each individual part. The overall public method, namely GetDailyPlan from the interface, have also been tested to ensure correct end to end results. However it has not been tested with all possible combinations of inputs since it should only test the interaction between the different private methods.

A goal for our solution is that it has to be fast. It has been tested and a daily plan can be created in less than 10 sec. We have not tested for multiple users at the same time so the time could be higher if the system went live.

4.1.5 Recovery

There is no recovery implemented on this part of the data access layer. The reason for this is that the response time will be too slow if the employees have to wait for the system to recover. Instead the employee should request the daily plan at a later time when the system should have recovered.

4.1.6 Problem Solutions

In section 2.2 certain problems were outlined with ideas to possible solutions. This section will focus on which solutions were chosen along with their implementations.

4.1.6.1 Inconsistent Use of NEED_TEXT in AID_NEED_TYPE

We have implemented our own solution to the problem of inconsistent use of NEED_TEXT in AID_NEED_TYPE. The implementation uses the queries found on Listing 4.2 and Listing 4.3. The result of the first query is analysed for procedure codes, done by checking if the PROCEDURE_CODE attribute is null. The identified procedure codes are used to generate the second query which will translate the procedure codes into descriptions.

4.1.6.2 Missing Foreign Key to AID_LEVEL

The problem of the missing foreign key constraint from AID_TIME_HISTORY to AID_LEVEL gave three problems.

- The problem of a missing description is left to the employee to solve. This means the services will be included on the daily plan even if the description can not be extracted from the database. This was chosen to give the employee a chance to deliver the services if it is possible to identify.
- The problem of missing duration values could only be solved by using a default value. The value chosen is 0 so each time an employee finds a service with 0 as duration they have to chose how much time to spend on it.
- Calculating the end time for visits uses the formula $EndTime = Start\ Time + \sum Service.Duration$ where Service.Duration gives the duration value for a service which have to be delivered on the given visit. The drawback was that missing duration values would count as 0 which would give too short visits. It will be left to the employee to know that the end time will be wrong if any services have a default value as duration.

4.2 Writing Updates to the Database

This section will focus on the design and implementation of the second part of the solution; updating the database with new data. This part consist of a

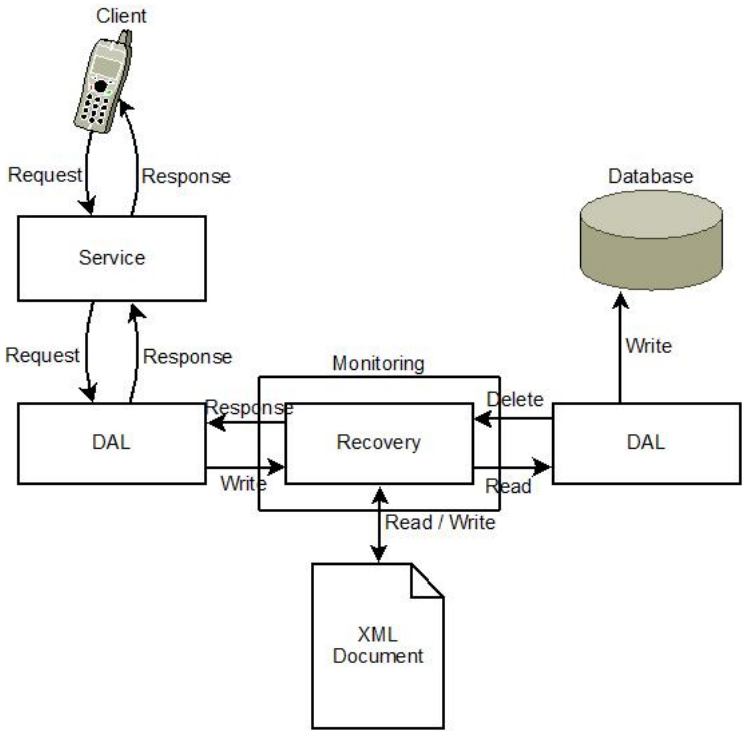

Figure 4.1: Overview of the Recovery Process.

recovery scheme and the actual updating method.

4.2.1 Recovery

It is important that the clients get a fast response when trying to update the database. For this reason we have decided to implement a recovery scheme. The recovery scheme writes the UpdateObject received from the clients to an XML document and respond when this action is successful. This gives the faster response time as the user does not have to wait for the data to be successfully inserted into the database. Another thread reads from the given XML document and insert the data into the database. This is illustrated on Figure 4.1.

The illustration shows that the recovery class contains three public methods. The incoming update requests uses the write method and the background thread uses the read and delete methods. To speed up the process for the clients the write method returns a Boolean indicating if the received UpdateObject was successfully written to the XML document. The Boolean is returned to the client and if the data was successfully written it can be removed here. The background thread reads the first node from the XML

document with the read method and if it is successfully inserted into the database the delete method will remove the node from the XML document.

Multiple clients can request updates at the same time and the background thread will be reading and deleting as well. This means that many threads will try to access the XML document concurrently. To avoid this problem of concurrency we have implemented the Recovery class as a singleton. This allow us to monitor the instance using the static methods of the Monitor class, which is a part of the standard library. The Monitor class ensures that only one thread can access the given object at any time.

4.2.1.1 XML Structure

The XML document is structured with a root node at the top. Each child of the root node is an UpdateObject. The UpdateObject node contains three attributes, which are EmployeeID, StartDayTime and EndDayTime from the UpdateObject. The children of an UpdateObject node are Visit nodes. A Visit node has 5 children; StartTime, EndTime, PlanDate, ActualDate and CitizenCPR from the DeliveredVisit object. The structure of the XML document can be found on figure 4.2. However as we use Log-Based Recovery, it is important that all operations which insert data into the database is idempotent. The reason for this is that the same operations can be executed many times because of system failures.

4.2.2 Implementation

The implementation of the database access is using a separate background thread. This thread reads from the XML document and if the update node is successfully inserted into the database, it is removed from the XML document. The process of getting the update node inserted into the database consist of six steps which are explained below. This process is only started if an update node is present in the XML document.

- **Transforming the UpdateObject:** This step transform the UpdateObject into objects which contains the same attributes as PP_REG and AID_DELIVERED.
- **Inserted Into PP_REG:** This step checks if the UpdateObject have already been inserted into PP_REG. This is needed because our recovery scheme has to be idempotent.
- **Delivered Services:** This step retrieve which services were delivered on each visit.

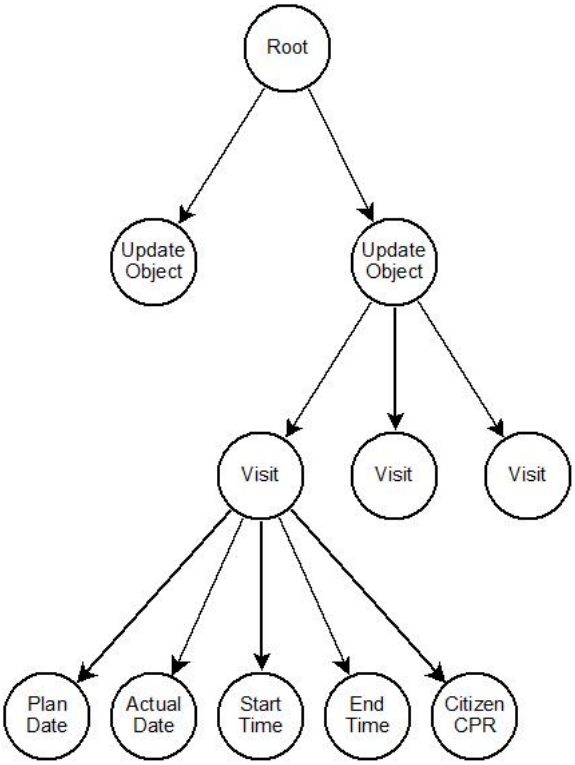

Figure 4.2: XML Structure.

- **Calculating the Duration:** This step calculate how much time were assigned to each visit. This is done by adding the duration of all the delivered services.
- **Inserted Into AID_DELIVERED:** This step checks if the UpdateObject has already been inserted into AID_DELIVERED. This along with step 2 should ensure that our insert operation is idempotent.
- **Inserting the UpdateObject:** The last step insert the objects which are not marked as AlreadyInserted into the database.

This section uses two local types of objects, namely PP_REG objects and AID_DELIVERED objects. These objects contains the same properties as the attributes in the table of the same name. The properties of PP_REG objects and AID_DELIVERED objects can be found in Figure 4.3. The objects does not contain the same amount of properties as the number of attributes in the table. The reason for this is that some of the information can be shared and also some of the attributes are the same. For example on PP_REG objects there are no reason to have more than two start time value, one for the actual start time and one for planned start time, even though the table contains four different start time attributes.

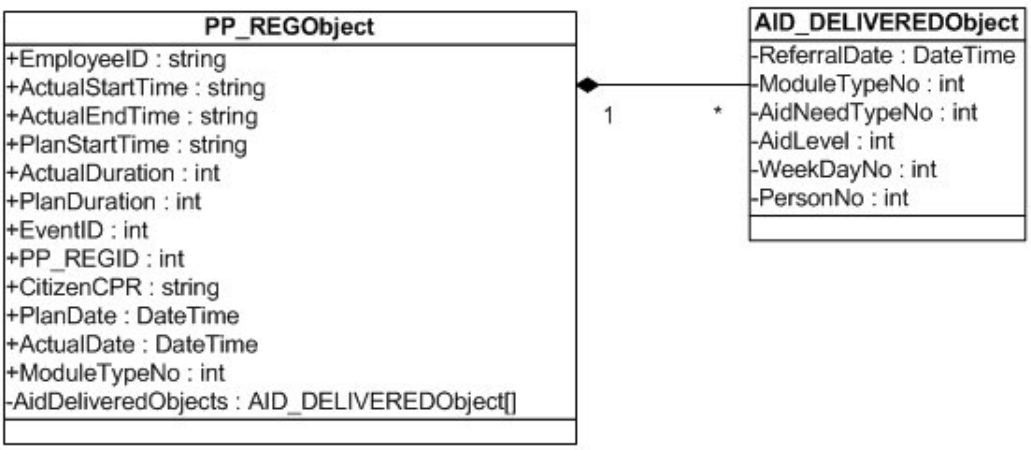

Figure 4.3: UML Diagram.

The remainder of this section will go through the six steps of the insert process. An example will illustrate each of the six steps. The example will use the UpdateObject found in Table 4.15 and the DeliveredVisit object found in Table 4.16. This UpdateObject will generate five PP_REGObjects and two AID_DELIVEREDObjects. We will only show PP_REGObjects 1 and 3

since the remaining three PP_REGPObjects will have the same information filled in on each step as PP_REGObject 1.

Attribute	Value
StartDayTime	0922
EndDayTime	1010

Table 4.15: UpdateObject.

Attribute	Value
StartTime	0932
EndTime	1004
CitizenCPR	9999999999
PlanDate	633584160000000000
ActualDate	633584160000000000

Table 4.16: DeliveredVisit.

4.2.2.1 Transforming the UpdateObject

The first step transforms the UpdateObject into a list of PP_REG objects. It generate a PP_REG object for the StartDayTime entry, two PP_REG objects for each visit(one is the visit and the other is the transport to the visit) and two more entries for the EndDayTime(one for the end of day and one for transport from last visit). This means that if the UpdateObject contains x visits it will generate $2x + 3$ entries for the PP_REG table. PP_REGObject 1 can be found on Table 4.17 and PP_REGObject 3 can be found on Table 4.18.

Attribute	Value
EmployeeID	9999999999
ActualStartTime	0922
ActualEndTime	0922
PlanStartTime	
ActualDuration	0
PlanDuration	
EventID	1
PP_REGID	
CitizenCPR	
ActualDate	01-10-2008
PlanDate	01-10-2008
ModuleTypeNo	

Table 4.17: PP_REGObject 1.

Attribute	Value
EmployeeID	9999999999
ActualStartTime	0932
ActualEndTime	1004
PlanStartTime	
ActualDuration	32
PlanDuration	
EventID	4
PP_REGID	
CitizenCPR	9999999999
ActualDate	01-10-2008
PlanDate	01-10-2008
ModuleTypeNo	

Table 4.18: PP_REGObject 3.

4.2.2.2 Removing Already Inserted PP_REG Objects

This step is done to ensure an idempotent insert process. It is done by querying PP_REG and mark already inserted PP_REGObjects. It is not possible to identify the already inserted objects on the primary key since it consist of INI, REG_DATE and ID. The first two attributes are the same for all events done by a given employee on a given day and the last attribute is assigned by the application. Instead of using the primary key we have decided to use the attributes INI, REG_DATE, EVENT_TYPE_ID, START_TIME, END_TIME and C_CPR_NO. This will ensure uniqueness in almost all cases. An example where it would not ensure uniqueness is if the employee click though the daily plan in less than a minute. In this case all the transport events will have the same values for all the attributes.

This step also assign PP_REGID to all PP_REGObjects. The query used for our example can be found in Listing 4.4. The updated PP_REGObjects can be found on Table 4.19 and Table 4.20.

Listing 4.4: PP_REG Query

```
select ID, REG_DATE, EVENT_TYPE_ID, START_TIME, END_TIME
from PP_REG
where INI = '9999999999' and (REG_DATE = '08-10-01')
```

Attribute	Value
EmployeeID	9999999999
ActualStartTime	0922
ActualEndTime	0922
PlanStartTime	
ActualDuration	0
PlanDuration	
EventID	1
PP_REGID	1
CitizenCPR	
ActualDate	01-10-2008
PlanDate	01-10-2008
ModuleTypeNo	

Table 4.19: PP_REGObject 1.

Attribute	Value
EmployeeID	9999999999
ActualStartTime	0932
ActualEndTime	1004
PlanStartTime	
ActualDuration	32
PlanDuration	
EventID	4
PP_REGID	3
CitizenCPR	9999999999
ActualDate	01-10-2008
PlanDate	01-10-2008
ModuleTypeNo	

Table 4.20: PP_REGObject 3.

4.2.2.3 Extracting the Missing PP_REG Information

This step extracts the PlanStartTime and ModuleTypeNo for each PP_REGObjects with EventTypeID 4. It also creates a list of AID_DELIVEREDObjects for each visit (EventTypeID 4). The query used for this step can be found in Listing 4.5. The updated PP_REGObject with EventTypeID 4 can be found in Table 4.22. An example of a AID_DELIVEREDObject can be found in Table 4.21.

Listing 4.5: Data Extract Query.

```
select C_CPR_NO, REFERRAL_DATE, START_TIME, MODULE_TYPE_NO,
    AID_NEED_TYPE_NO, AID_LEVEL, WEEKDAY_NO
from AID_TIME_HISTORY
where INI = '9999999999' and ((DATE_FR < '08-10-01' and (
   DATE_TO > '08-10-01' or DATE_TO is null))
```

Attribute	Value
ReferralDate	05-10-2008
ModuleTypeNo	1
AidNeedTypeNo	4
AidLevel	4
WeekDayNo	3
PersonNo	1

Table 4.21: AID_DELIVEREDObject.

Attribute	Value
EmployeeID	9999999999
ActualStartTime	0932
ActualEndTime	1004
PlanStartTime	0930
ActualDuration	32
PlanDuration	
EventID	4
PP_REGID	3
CitizenCPR	9999999999
ActualDate	01-10-2008
PlanDate	01-10-2008
ModuleTypeNo	1

Table 4.22: PP_REGObject 3.

4.2.2.4 Calculating the PlanDuration for Visits

This step calculates the PlanDuration based on the planned duration for each visit. This means that this value can be invalid if AidLevel values are invalid. The calculation is done based on the following formula $PlanDuration = \sum Service.PlanDuration$ where Service.PlanDuration is the allocated time for a service that had to be delivered on the given visit. An example of the query can be found in Listing 4.6. The result of the query only modify

Attribute	Value
EmployeeID	9999999999
ActualStartTime	0932
ActualEndTime	1004
PlanStartTime	0930
ActualDuration	32
PlanDuration	30
EventID	4
PP_REGID	3
CitizenCPR	9999999999
ActualDate	01-10-2008
PlanDate	01-10-2008
ModuleTypeNo	1

Table 4.23: PP_REGObject 3.

PP_REGObjects with EventTypeID 4. The updated object which satisfy this can be found in Table 4.23.

Listing 4.6: AID_LEVEL Query.

```
select DURATION
from AID_LEVEL
where (MODULE_TYPE_NO = 1 and AID_NEED_TYPE_NO = 4 and
    AID_LEVEL = 4)
```

4.2.2.5 Removing Already Inserted AID_DELIVERED Objects

This step is the final part to ensure that the insert process is idempotent. This step queries the AID_DELIVERED table to find the already inserted AID_DELIVERED objects. An example of the used query can be found on Listing 4.7. This step could have been done by the database since all attributes are part of the primary key. We have decided to include the step since changes might happen to the database in the future.

Listing 4.7: AID_DELIVERED Query.

```
select C_CPR_NO, START_TIME, MODULE_TYPE_NO,
    AID_NEED_TYPE_NO, AID_LEVEL, DELIV_DATE
from AID_DELIVERED
where DELIV_BY_INI = '9999999999'
```

4.2.2.6 Inserting the Data into the Database

The sixth step inserts all objects, which were not identified as already inserted by step 2 and step 5, into the database. This is done by normal insert into statements. The commit statement is executed after all rows have been inserted into both tables. The last thing done by this step is to remove the XML node from the XML document if it was successfully committed to the database. An example of a PP_REG object insert can be found on Listing 4.8 and a AID_DELIVERED object insert can be found on Listing 4.9. The query found in Listing 4.8 is modified if EventTypeID is different from 4. The reason for this is that EventTypeID 4 is a visit and therefore got for example a visited citizen attached. These values are just left as null in case of other events.

Listing 4.8: Insert Into Query 1.

```
insert into PP_REG(INI, REG_DATE, ID, EVENT_TYPE_ID,
    START_TIME, DURATION, C_CPR_NO, END_TIME,
    DISP_START_TIME, MODULE_TYPE_NO, ORIG_DATE, ACT_DATE,
    ORIG_START_TIME, ACT_START_TIME, ORIG_DURATION,
    ACT_DURATION, CLIENT_ABORTED, PDA_CHANGED, CARE_CHANGED,
     LAST_MODIFIED_INI, LAST_MODIFIED_TIME)
values ('999999999', to_date('08-10-01', 'yy-mm-dd'), '3',
    '4', '0932', '32', '9999999999', '1004', '0930', '1',
    to_date('08-10-01', 'yy-mm-dd'), to_date('08-10-01', 'yy
    -mm-dd'), '0930', '0932', '30', '32', 'n', 'n', 'n', '
    9999999999', to_date('08-10-01', 'yy-mm-dd'))
```

Listing 4.9: Insert Into Query 2.

```
insert into AID_DELIVERED(C_CPR_NO, REFERRAL_DATE,
    MODULE_TYPE_NO, PLAN_DATE, AID_LEVEL, AID_NEED_TYPE_NO,
    WEEKDAY_NO, START_TIME, PERSON_NO, DELIV_BY_INI,
    DELIV_DATE, PP_REG_ID)
values ('9999999999', to_date('08-10-05', 'yy-mm-dd'), '1',
     to_date('08-10-01', 'yy-mm-dd'), '4', '4', '3', '0932',
     '1', '9999999999', to_date('08-10-01', 'yy-mm-dd'), 3)
```

4.2.3 Testing

This part of the data access layer is tested using unit tests. To ensure no old data is stored in the XML file or in the database all inserted data is removed from these before starting the tests. We will test if the data is correctly inserted into the XML file, if the insert operation is idempotent and if the data is correctly inserted into the database. We have tested the following:

- **Correct Insertion into XML File:** This is tested using the three public methods of the Recovery class. We insert two UpdateObjects into the XML document. We read the first UpdateObject and checked if it contained all the correct data. Afterwards we remove the first UpdateObject from the XML file and read the next UpdateObject and also checked if it contained the correct the information.

- **Idempotent:** We tested that our insert operation was idempotant by inserting the same UpdateObject twice into the XML file. The background thread should insert the first UpdateObject into the database but should identify the second as already inserted. The database was queried to check if it contained the right amount of rows or too many.

- **Correct Insertion into Database:** We tested that an UpdateObject inserted into the XML file also gave the right amount of rows and that the rows contained the right information in database. This was done by inserting the UpdateObject into XML file, let the background insert it into the database and query the database for the information back. We had to let the test thread sleep while waiting for the background thread to insert the information into the database.

4.2.4 Problem Solutions

In section 2.2 certain problems were outlined with ideas to possible solutions. This section will focus on which solutions were actually used along with their implementations.

4.2.4.1 Missing External Foreign Keys

The missing foreign keys can be put into two categories. The first is those that have a foreign key constraint in AID_TIME_HISTORY. The second is those that does not.

- The first kind are already ensured to exist. The reason for this is that we insert the data which were extracted from AID_TIME_HISTORY. AID_TIME_HISTORY contains the foreign key constraints so the extracted data is correct. Therefore it can be inserted into the new tables without doing any checks.

- The second kind can not be ensured to exist since the foreign key constraint does not exist in AID_TIME_HISTORY. This means that we might work with invalid data in the first place. It is possible for us

to verify if the data is valid or not but we don't have any possibility to insert other values in case of invalid data. The reason for this is that we have no chance to detect what the right values should be and the attributes are not null-able. Examples of these attributes are MODULE_TYPE_NO, AID_NEED_TYPE_NO and AID_LEVEL.

4.2.4.2 Missing Foreign Key on AID_DELIVERED

The problem of no foreign key constraint from AID_DELIVERED to PP_REG is solved by inserting all rows in PP_REG before inserting anything into AID_DELIVERED. The structure chosen for this is to create overall PP_REG objects and have the AID_DELIVERED objects inside the PP_REG objects. This means that the foreign key can be created from the overall data in PP_REG and we do not assign wrong values as foreign key.

4.2.4.3 Recovery Based Problem

The problem of a purely defined primary key in PP_REG is solved by using other attributes to ensure uniqueness. The attributes chosen are INI, REG_DATE, EVENT_TYPE_ID, START_TIME, END_TIME and C_CPR_NO. It is still possible to have multiple rows with the same information if the employee just click all the way though the daily plan in less than one minute. However all the rows will still be inserted since we are not doing a commit before after all rows are inserted, so these rows containing the same information will not be set as already insert and therefore get different ID's assigned.

Chapter 5

Service

As described in Section 2.6, we will make a RESTful service using Windows Communication Foundation. To learn how to build a service, we have been using a blog series made by Rob Bagby, who is a developer evangelist at Microsoft.[Bag08]

The two basic components to the service, is the interface, IHomeCareService, and the implementation of the interface, HomeCareService. Figure 5.1 shows a class diagram with these 2 elements.

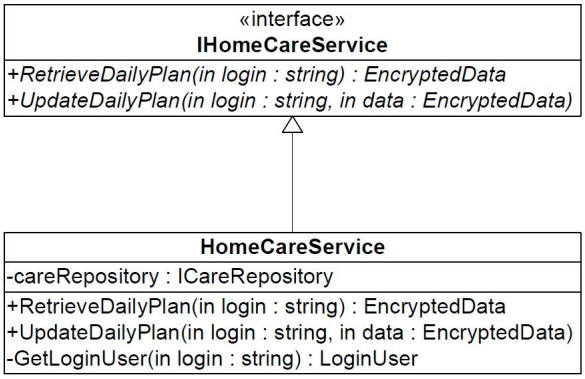

Figure 5.1: The interface and class that makes up the service.

5.1 Service Implementation

There are 3 parts to look at for the implementation of the service. The configuration of the service through the web.config file and the IHomeCareService interface, the implementation of the IHomeCareService interface and the implementation of the d620a.HomeCare.Models namespace. An overview

of the classes and interfaces used in the implementation is shown in Figure 5.2.

The remaining sections in this chapter will explore the implementations of these three parts.

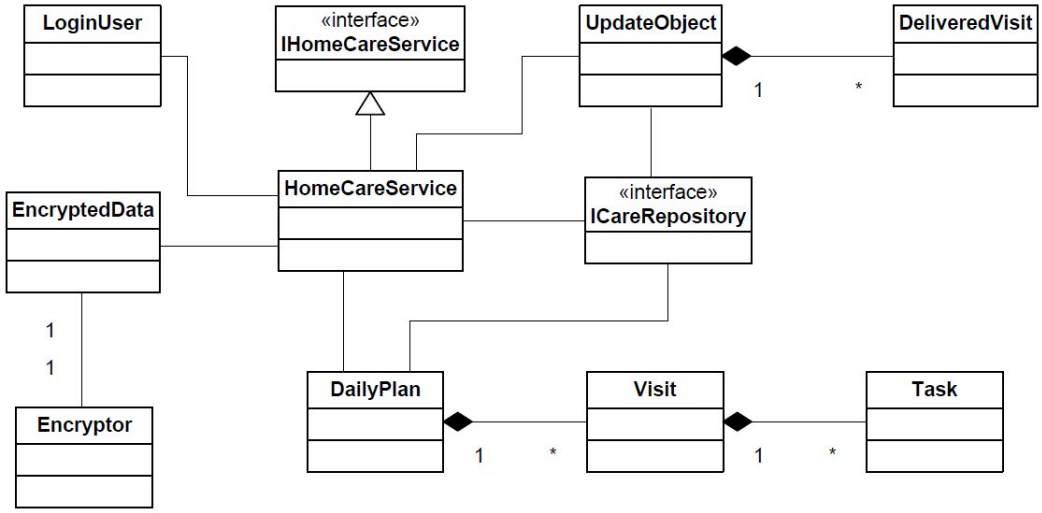

Figure 5.2: Overview of the classes and interfaces used in the service implementation.

5.2 Configuration

Section 3.4.1 describes the behaviour that the service should adhere to. Some of the behaviour's are implemented through the IHomeCareService interface.

5.2.1 Daily Plan

Figure 5.3 shows the method declaration to retrieve a daily plan, and as can be seen, it follows what was described in Section 3.4.1. What the method attributes means is that the method is a part of the service (defined by the OperationContract attribute). If there is a GET request (defined by the WebGet attribute) that matches the address "HomeCareService/dailyplan/{login}" (defined by UriTemplate), invoke the RetrieveDailyPlan method with {login} as input parameter and a EncryptedData object as return parameter. The return parameter should be formatted as a json object (defined by ResponseFormat).

```
[OperationContract]
[WebGet(UriTemplate = "dailyplan/{login}",
    ResponseFormat = WebMessageFormat.Json)]
EncryptedData RetrieveDailyPlan(string login);
```

Figure 5.3: The interface code for the RetrieveDailyPlan method.

5.2.2 Update

The method declaration to receive an update is shown in Figure 5.4. Again, the OperationContract attribute means that the method is a part of the service. The WebInvoke attribute defines that if a POST request matches the address "HomeCareService.svc/update/{login}", the method named UpdateDailyPlan should be invoked with {login} and the POST'ed EncryptedData elements as input parameters. The received EncryptedData element should be formatted as a json object.

```
[OperationContract]
[WebInvoke(Method = "POST",
    UriTemplate = "update/{login}",
    RequestFormat = WebMessageFormat.Json)]
void UpdateDailyPlan(string login, EncryptedData data);
```

Figure 5.4: The interface code for the UpdateDailyPlan method.

5.2.3 Web.config

The basic configuration of the service, happens through the web.config file. This is where the service is told to use a RESTful style or a more traditional XML Web Service style. More specifically it is in the section system.servideModel that the service configuration happens. Figure 5.5 shows the configuration needed for our service implementation.

5.3 HomeCareService Implementation

The HomeCareService class implements the IHomeCareService interface and thus comtains the actual implementation of the service. There are three methods in the class: The two interface methods and a method to convert the encrypted login string to a LoginUser class.

```
<system.serviceModel>
  <behaviors>
    <endpointBehaviors>
      <behavior name="RESTFriendly">
        <webHttp />
      </behavior>
    </endpointBehaviors>
    <serviceBehaviors>
      <behavior name="MyServiceTypeBehaviors">
        <serviceMetadata httpGetEnabled="true" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
  <services>
    <service behaviorConfiguration="MyServiceTypeBehaviors"
             name="d620a.HomeCare.Service.HomeCareService">
      <endpoint address="" behaviorConfiguration="RESTFriendly"
                binding="webHttpBinding"
                contract="d620a.HomeCare.Service.IHomeCareService" />
      <host>
        <baseAddresses>
          <add baseAddress="http://localhost:80/" />
        </baseAddresses>
      </host>
    </service>
  </services>
</system.serviceModel>
```

Figure 5.5: An excerpt from the web.config file, showing the configuration of the service endpoint.

5.3.1 Daily Plan

Three parts from the d620a.HomeCare.Models namespace are needed for the service to process the request for a daily plan. The three parts are: Daily Plan (see Section 3.2.2), Login (see Section 3.2.4) and Encrypted Data (see Section 3.2.5).

Figure 5.6 shows the implementation of the RetrieveDailyPlan method. The method is where the HTTP status codes from Section 3.4.1 is implemented. If the received login string cannot be converted to a LoginUser instance, the status code 400 (BadRequest) will be returned. The employeeId is retrieved form the data access layer and if there is no employeeId for the given user, the status code 403 (Forbidden) is returned. Next the daily plan for the user is retrieved for the current day. If a DailyPlan instance is returned from the data access layer, the status code is set to 200 (OK) and a EncryptedData object is returned containing the DailyPlan instance.

```
public EncryptedData RetrieveDailyPlan(string login)
{
    OutgoingWebResponseContext outResponse = WebOperationContext.Current.OutgoingResponse;

    LoginUser user = GetLoginUser(login);
    if (user == null)
    {
        outResponse.StatusCode = HttpStatusCode.BadRequest;
        return null;
    }
    string employeeId = careRepository.GetEmployeeId(user.UserName, user.Password);
    if (String.IsNullOrEmpty(employeeId))
    {
        outResponse.StatusCode = HttpStatusCode.Forbidden;
        return null;
    }

    if (employeeId != null)
    {
        DailyPlan plan = careRepository.GetDailyPlan(employeeId, DateTime.Now);
        if (plan != null)
        {
            outResponse.StatusCode = HttpStatusCode.OK;
            return new EncryptedData(plan);
        }
    }

    outResponse.StatusCode = HttpStatusCode.InternalServerError;
    return null;
}
```

Figure 5.6: The implementation code for the RetrieveDailyPlan method.

5.3.2 Update

The UpdateDailyPlan method also requires three parts from the Models namespace. It uses the Login and Encrypted Data parts, like the method to retrieve a daily plan, and then it uses the Update Object part (see Section 3.2.3).

The implementation of the UpdateDailyPlan method is shown in Figure 5.7. Retrieving the LoginUser instance and the employeeId is the same for this method as for the UpdateDailyPlan method. After an employeeId has been retrieved from the data access layer, the UpdateObject instance is retrieved from the EncryptedData input parameter and the employeeId and UpdateObject is used as input parameters to the UpdateDatabase method of the data access layer. If the update has been performed successfully, the status code 200 (OK) will be returned, otherwise the status code 500 (Internal Server Error) will be returned.

```
public void UpdateDailyPlan(string login, EncryptedData data)
{
    OutgoingWebResponseContext outResponse = WebOperationContext.Current.OutgoingResponse;

    LoginUser user = GetLoginUser(login);
    if (user == null)
    {
        outResponse.StatusCode = HttpStatusCode.BadRequest;
        return;
    }
    string employeeId = careRepository.GetEmployeeId(user.UserName, user.Password);
    if (String.IsNullOrEmpty(employeeId))
    {
        outResponse.StatusCode = HttpStatusCode.Forbidden;
        return;
    }

    UpdateObject update = data.FromJsonString(typeof(UpdateObject)) as UpdateObject;
    bool result = careRepository.UpdateDatabase(employeeId, update);
    if (result)
    {
        outResponse.StatusCode = HttpStatusCode.OK;
        return;
    }

    outResponse.StatusCode = HttpStatusCode.InternalServerError;
    return;
}
```

Figure 5.7: The implementation code for the UpdateDailyPlan method.

5.4 Models Namespace Implementation

Most of the classes in the Models namespace have trivial implementations only consisting of properties with get and set methods. As an example of this, Figure 5.8 shows the complete implementation of the Task class.

There are two classes of interest to look at in the Models namespace. And this is the EncryptedData and Encryptor classes. These two classes are responsible for conversion and encryption of objects and as such it is important that their behaviour is the same on both the service and the client. In the following we will outline the C# implementation of these classes, which is based on the UML class diagram from Figure 3.7.

5.4.1 EncryptedData

Before the EncryptedData class can be transmitted through the service, it has to be serialised. This happens by decorating the class with the DataContract attribute. The members of the class that should be serialised has to be decorated with the DataMember attribute. This also makes it possible to keep methods and properties in the class that is local to the implementation but shouldn't be part of the serialised result.

```
namespace d620a.HomeCare.Models
{
    public class Task
    {
        // Description of what have to be done, like cleaning
        public string Description { get; set; }
        // Further details, if available
        public string Details { get; set; }
        // The duration of the given task, like 15 min
        public int Duration { get; set; }
    }
}
```

Figure 5.8: The implementation code for the Task class.

5.4.1.1 Data property

When the service receives an EncryptedData json object, it will call the Data property's set method to populate it with the data. This will also decrypt the data. When the service sends an EncryptedData object it will call the property's getter to retrieve the data to send. Figure 5.9 shows the implementation of the property.

```
[DataMember]
public string Data
{
    set
    {
        CheckInitialisation();
        try
        {
            byte[] s = encryptor.Decrypt(UrlBase64.Decode(value));
            data = enc.GetString(s);
        }
        catch (Exception e)
        {
            data = e.Message;
        }
    }
    get
    {
        return data;
    }
}
```

Figure 5.9: The implementation code for the Data property in the Encrypted-Data class.

5.4.1.2 FromJsonString

This method returns an object instance with the provided type based on the json string stored in the data attribute. The work itself happens through the

use of the third party library Json.NET. Figure 5.10 shows the implementation of the method.

```
public Object FromJsonString(Type type)
{
    return JsonConvert.DeserializeObject(data, type);
}
```

Figure 5.10: The implementation code for the FromJsonString method in the EncryptedData class.

5.4.2 Encryptor

It is the Encryptor class that is responsible for the encryption and decryption processes. It uses the AES engine made by the Bouncy Castle group[1]. What follows is an explanation of the methods in the Encryptor class.

5.4.2.1 Constructor

The constructor take a key as input parameter. The key has to be the same on both the client and service side for the encryption/decryption process to be successful. The AES engine is instantiated and the key is converted into a byte array that contains 32 elements. This gives a key size of 256 bit, which is the most secure key size that is supported by AES (it also supports 128 and 192 bit keys). If the input key doesn't contain 32 characters, 0 will be added as padding until there is exactly 32 characters. Figure 5.11 shows the implementation of the constructor.

5.4.2.2 CallCipher

CallCipher is the method that performs the actual encryption/decryption. It sets up the output byte array before the AES engines ProcessBytes method is called. This methods processes the input data byte array into the output byte array. The implementation is shown in Figure 5.12.

5.4.2.3 Encrypt

The Encrypt method takes an unencrypted byte array as input parameter and returns an encrypted byte array. It initialises the AES engine to do encryption with the key that was provided in the constructor. The first parameter in the Init method is used to determine if the engine should perform

[1] http://www.bouncycastle.org/

```
private Encryptor()
{
    cipher = new PaddedBufferedBlockCipher(
        new CbcBlockCipher(
        new AesEngine()));

    System.Text.UTF8Encoding enc = new UTF8Encoding();
    byte[] b = enc.GetBytes(keyString);
    byte[] keyByteArray = new byte[32];
    for (int i = 0; i < keyByteArray.Length; i++)
    {
        if (i < b.Length)
        {
            keyByteArray[i] = b[i];
        }
        else
        {
            keyByteArray[i] = 0;
        }
    }

    this.key = new KeyParameter(keyByteArray);
}
```

Figure 5.11: The implementation code for the constructor in the Encryptor class.

```
private byte[] CallCipher(byte[] data)
{
    int size = cipher.GetOutputSize(data.Length);
    byte[] result = new byte[size];
    int olen = cipher.ProcessBytes(data, 0, data.Length, result, 0);
    olen += cipher.DoFinal(result, olen);

    if (olen < size)
    {
        byte[] tmp = new byte[olen];
        Array.Copy(result, 0, tmp, 0, olen);
        result = tmp;
    }

    return result;
}
```

Figure 5.12: The implementation code for the CallCipher method in the Encryptor class.

encryption or decryption. If the value is true it will encrypt and if it is false it will decrypt. The implementation is shown in Figure 5.13.

```
[MethodImpl(MethodImplOptions.Synchronized)]
public byte[] Encrypt(byte[] data)
{
    if (data == null || data.Length == 0)
    {
        return new byte[0];
    }

    cipher.Init(true, key);
    return CallCipher(data);
}
```

Figure 5.13: The implementation code for the Encrypt method in the Encryptor class.

5.4.2.4 Decrypt

The Decrypt method is very similar to the Encrypt method. The only difference is that it initialises the AES engine to do decryption by setting the boolean value in the Init method to false. The implementation is shown in Figure 5.14.

```
[MethodImpl(MethodImplOptions.Synchronized)]
public byte[] Decrypt(byte[] data)
{
    if (data == null || data.Length == 0)
    {
        return new byte[0];
    }

    cipher.Init(false, key);
    return CallCipher(data);
}
```

Figure 5.14: The implementation code for the Decrypt method in the Encryptor class.

Chapter 6

Mobile Client

The mobile client is the means for interacting with the service. In this Chapter we will explore the development of the client, starting with a presentation of the user interface before discussing the implementation of the communication and storage parts and ending with the implementation of the Models namespace.

6.1 User Interface

On program startup, it will ask for the users credential. Figure 6.1 shows the screen where the user types in his or her username and password. After pressing the Login button, the user will be asked if it is okay to use airtime, this screen is shown in Figure 6.2. Figure 6.3 shows the screen that is displayed while the client communicates with the service.

When a successful response has been received from the service the received daily plan is shown to the user. Figure 6.4 shows a daily plan for an employee, where there are two citizens to visit.

To see the details associated with a visit, the user can mark a visit and press the details button. Figure 6.5 shows an example of the details belonging to a visit.

Finally when all visits has been carried out, the service is contacted with the update information. Figure 6.6 shows the response for a successful update.

6.1.1 DailyPlanForm

Besides displaying the plan to the employee the DailyPlanForm form is also used to guide the employee through the workday. This is done through

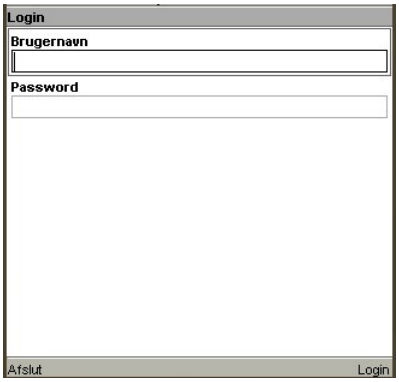

Figure 6.1: The login screen.

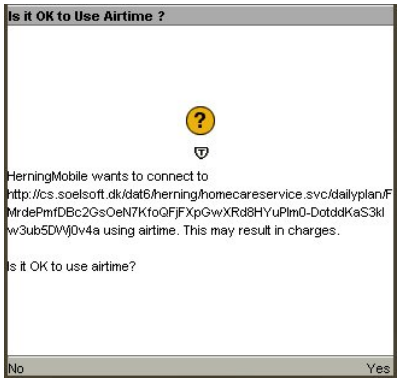

Figure 6.2: Request for airtime.

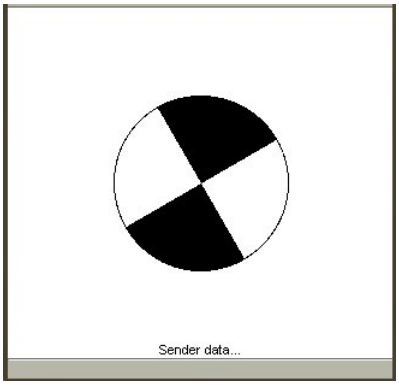

Figure 6.3: The waiting screen shown during communication.

Figure 6.4: The daily plan screen.

Figure 6.5: The visit screen.

Figure 6.6: The response screen after a successful update.

the button in the lower right corner in Figure 6.4, which will change name depending on which state the form is in. The button can have four different names, which are shown in Table 6.1.1 along with the English translations of the names.

Button Name	**In English**
Start Vagt	Start Watch
Start Besøg	Start Visit
Afslut Besøg	End Visit
Afslut Vagt	End Watch

Table 6.1: The different button values

6.1.1.1 The Different States

The DailyPlanForm can be in several different states, which are shown in Figure 6.7. The form is in the initial state when it is first displayed to the user. When the employee chooses to begin his or her watch, the time of day is recorded and the form transitions to the watch state. Here there is a check to see if there are any citizens to visit, that haven't had a visit yet. If so, the buttons value changes to "Start Besøg" and the form transitions to the Transport to Citizen state and lasts until the employee starts the actual visit. This happens by pressing the button which will record the time of day for the start of the visit, change the button value to "Afslut Besøg" and transitions the form to the Visit state. When the employee has carried out all the tasks associated with the visit, the button is pressed which records the time of day and transitions the form back to the Watch state.

Finally, when there are no more visits to do, the button value changes to "Afslut Vagt" and the form transitions to the Transport to End state, which lasts until the employee has returned to the starting point. At this time the button is used for the last time, which will record the time of day, start the procedure of sending the updates to the service and transition the form to the final state.

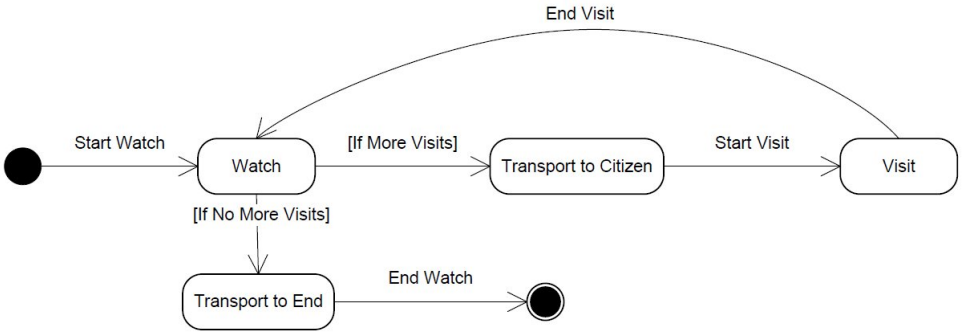

Figure 6.7: The state diagram for the daily plan screen.

6.2 Communication

All communication with the service is initiated through the Herning midlet class. To avoid having the communication freeze the program execution, the midlet delegates the job to a worker thread that does the actual communication. Figure 6.8 shows the class diagram for the ConnectionWorker class as well as the methods from the Herning midlet class that is used for communication.

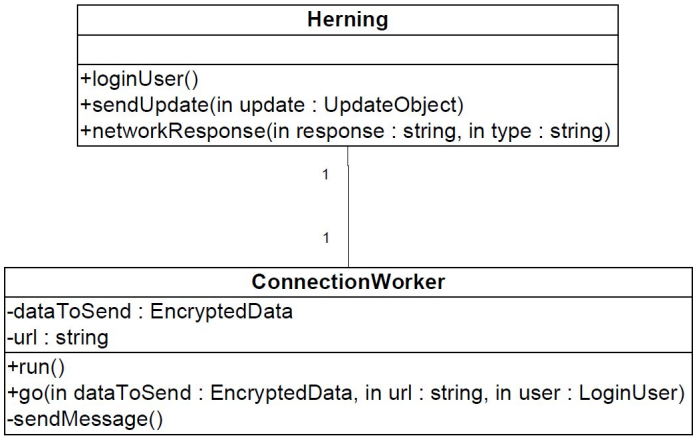

Figure 6.8: The class diagram for the CommunicationWorker class.

6.2.1 Herning Midlet

The midlet contacts the service on two different occasions. The first one is to retrieve the daily plan for the user, this task is handled in the loginUser method. The second occasion is when the update has to be send to the service

and this is handled by the sendUpdate method. Finally the responses from the service are handled by the networkResponse method. In the following we will take a closer look at the three methods.

6.2.1.1 loginUser

The loginUser method is invoked when the user presses the login button on the login screen. The method retrieves a LoginUser instance and a daily plan from storage. If the entered username and password matches those from the LoginUser instance and the daily plan exists, the daily plan will be loaded, otherwise a call to the ConnectionWorker's go method will start the process of retrieving the plan from the service. Figure 6.9 shows the implementation of the loginUser method.

```
private void loginUser()
{
    String username = userNameField.getString();
    String password = passwordField.getString();
    loginUser = recordStore.readLoginUser();
    String plan = recordStore.readDailyPlan();
    if (loginUser != null && loginUser.getUserName().equals(username) &&
            loginUser.getPassword().equals(password) && plan != null) {
        setDailyPlan(plan, false);
    }
    else {
        loginUser = new LoginUser(username, password);
        recordStore.saveLoginUser(loginUser);
        worker.go(null, ConnectionWorker.RETRIEVE_DAILY_PLAN, loginUser);
    }
}
```

Figure 6.9: The loginUser method in the Herning midlet class.

6.2.1.2 sendUpdate

The sendUpdate method is invoked when the employee has finished the watch. It takes an UpdateObject instance which will be added to an EncryptedData object that is used as input parameter for the ConnectionWorker's go method. Figure 6.10 shows the implementation of the sendUpdate method.

```
public void SendUpdate(UpdateObject update) {
    EncryptedData data = new EncryptedData(update.toJson());
    worker.go(data, ConnectionWorker.SEND_UPDATE, loginUser);
}
```

Figure 6.10: The sendUpdate method in the Herning midlet class.

6.2.1.3 networkResponse

The networkResponse method is invoked from the ConnectionWorker class when there has been a successful response from the service. Depending on if the response is caused by a request for a daily plan or an update, it will either display the daily plan or mark the current plan as completed. Figure 6.11 shows the implementation of the method.

```
public void networkResponse(String response, String type)
{
    if (type.equals(ConnectionWorker.RETRIEVE_DAILY_PLAN))
        setDailyPlan(response, true);
    else if (type.equals(ConnectionWorker.SEND_UPDATE))
    {
        recordStore.CompletedDailyPlan();
        setAlert("Svar","Oplysningerne er modtaget.", ALERT_INFO, getDailyPlanForm());
    }
}
```

Figure 6.11: The networkResponse method in the Herning midlet class.

6.2.2 ConnectionWorker

The ConnectionWorker class runs in its own thread, such that it won't freeze the midlet during communication sessions with the service. As long as there are no requests for the service it will be in the wait state and it's only when there is a message to send that it will go into an active state.

The notification to enter the active state happens in the go method, which is also where the data needed to make the communication is entered. The implementation of the go method is shown in Figure 6.12.

When the worker enters the active state, it will call the sendMessage method, which takes care of the actual communication.

6.2.2.1 sendMessage

The sendMessage method starts by creating a connection instance with the information received through the go method. It will also switch the display to

```
public synchronized void go(EncryptedData dataToSend, String url, LoginUser user)
{
    EncryptedData login = new EncryptedData(user.toJson());
    this.dataToSend = dataToSend;
    this.url = url;
    this.url = this.url + "/" + login.getEncryptedData();
    notify();
}
```

Figure 6.12: The go method in ConnectionWorker class.

show the waiting screen to keep the user informed about the communication process. Figure 6.13 shows this initialisation part.

If the invocation happens because there is an update to send to the service, the UpdateObject has to be send as a POST attachment. Figure 6.14 shows the steps taken to send an update.

If the service's response to the request is successful, it will return an HTTP status code with value 200. This will cause the method to invoke the networkResponse method in the Herning midlet with appropriate values. Figure 6.15 shows the implementation.

```
connection = ( HttpConnection )Connector.open(
        URL_TO_SERVICE + url.replace('.', ' ').trim(), Connector.READ_WRITE );
waitCanvas.setMessage("Sender data...");
midlet.switchDisplayable(null, waitCanvas);
connection.setRequestProperty("User-Agent",
    "Profile/MIDP-2.0 Configuration/CLDC-1.1");
```

Figure 6.13: Initialisation of the HTTP connection.

```
if (dataToSend != null)
{
    requestBody = dataToSend.toJson();
    connection.setRequestMethod(HttpConnection.POST);
    connection.setRequestProperty("Content-Type", "application/json");
    connection.setRequestProperty("Content-Length", ""+requestBody.length());
    dos = connection.openDataOutputStream();
    dos.write(requestBody.getBytes("UTF8"));
}
```

Figure 6.14: Sending an update.

```
if (connection.getResponseCode() == HttpConnection.HTTP_OK)
{
    if (url.startsWith(SEND_UPDATE))  {
            midlet.networkResponse(responseMessage.toString(), SEND_UPDATE);
    } else if (url.startsWith(RETRIEVE_DAILY_PLAN))
    {
        EncryptedData retrievedData = new EncryptedData("");
        retrievedData.fromJson(responseMessage.toString());
        midlet.networkResponse(retrievedData.getData(), RETRIEVE_DAILY_PLAN);
    }
}
```

Figure 6.15: A successful response received from the service.

6.3 Storing Data

We need to be able to store the daily plan received from the service as well as all the time of each event (start and end of the watch and each visit). If these values only exist in memory, it will not be possible to shut down the client without loosing the data. A possible scenario where the program could be shut down is when the employee makes a phone call.

To store the data we will use the Record Management System (RMS) that comes with Java Micro Edition. It is a database system that allows a Java program to use persistent storage[1].

Our storage solution consists of two classes. HomeCareRS, which is the main class for interacting with the record store and RecordStoreWorker, which executes the actual inserting of data in the record store in a separate thread.

6.3.1 HomeCareRS

There are three different things we need to store in persistent storage:

1. The LoginUser instance associated with the current employee.
2. The DailyPlan received from the service.
3. The UpdateObject that has to be sent to the service.

6.3.1.1 LoginUser

We need to store the LoginUser instance so the employee doesn't have to enter the login credentials again when the update is send to the service. It

[1]For an introduction to RMS visit
http://developers.sun.com/mobility/midp/articles/databaserms/

is also compared to the user name and password that an employee inputs in order to verify if the employee already has a saved daily plan.

The method to save the LoginUser instance is invoked from the loginUser method in the Herning MIDlet class (the method is shown in Figure 6.9) when it has been determined that there does not exist a previously stored daily plan. Figure 6.16 shows the implementation of the method that save a LoginUser instance.

```
public void saveLoginUser(LoginUser loginUser) {
    try {
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        DataOutputStream dout = new DataOutputStream(bout);
        byte[] data;
        dout.writeUTF(loginUser.getUserName());
        dout.writeUTF(loginUser.getPassword());

        dout.flush();
        data = bout.toByteArray();

        worker.saveItems(LOGIN_USER, data);
      }
      catch (Exception e) {
           display.setCurrent(new Alert(e.getMessage()));
      }
}
```

Figure 6.16: The method to store a LoginUser instance in the record store.

The method to retrieve a LoginUser instance from the record store is invoked at the beginning of the loginUser method in the Herning MIDlet class. The implementation is shown in Figure 6.17.

6.3.1.2 DailyPlan

The DailyPlan instance is stored as the JSON string representation that is received from the service. We have chosen this strategy since we don't need to change the information in the object and a single string is a lot easier to handle than complex objects. Only having a single string to store and retrieve makes the implementation trivial, except for one thing. Each DeliveredVisit belonging to the UpdateObject is stored for itself, and the records that will contain them have to be added before they can be used and that happens in the method that saves the daily plan. The save method is shown in Figure 6.18.

```
public LoginUser readLoginUser() {
    String username = null ,password = null;
    try {
        byte[] data = new byte[100];
        ByteArrayInputStream bin = new ByteArrayInputStream(data);
        DataInputStream din = new DataInputStream(bin);

        rs.getRecord(LOGIN_USER, data, 0);
        din.reset();
        username = din.readUTF();
        password = din.readUTF();
      }
      catch (Exception e) {
            display.setCurrent(new Alert(e.getMessage()));
      }
    if (username != null && password != null)
        return new LoginUser(username, password);
    return null;
}
```

Figure 6.17: The method to retrieve a LoginUser instance from the record store.

```
public void saveDailyPlan(String dailyPlan, int numOfVisits)
{
  try
  {
      worker.saveItems(DAILY_PLAN, dailyPlan.getBytes());
      while (rs.getNumRecords() < DELIVERED_VISITS + numOfVisits)
          rs.addRecord(null, 0, 0);
  }
  catch (Exception e)
  {
      display.setCurrent(new Alert(e.getMessage()));
  }
}
```

Figure 6.18: The method to a DailyPlan instance in the record store.

6.3.1.3 UpdateObject

The UpdateObject is divided into two parts, one containing the information related to the watch and one part that is related to each visit. This causes the save method for the UpdateObject to only be called twice. The first time is to save the start time for the watch and the second time is to save the end time for the watch.

Having each visit have its own records limits the amount of data that we need to write every time there is an update. As with the updateObject, every DeliveredVisit needs to be stored twice. The first time to store the initial data (such as the citizen's cpr number) and the start time of the visit. The second time is to store the end time of the visit. If a watch contains 10 visits we will only store each visit twice instead of 20 times.

6.3.2 RecordStoreWorker

The RecordStoreWorker is responsible for storing data in the record store. The class extends the Thread class, such that the storing of data happens in its own thread, and thus not freezing the user interface when writing to persistant storage. Figure 6.19 shows the class diagram for the RecordStoreWorker class.

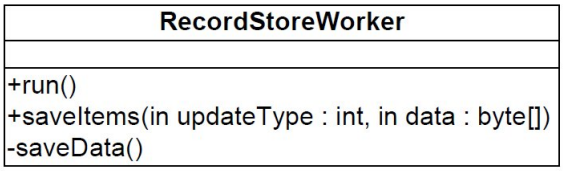

Figure 6.19: The class diagram for the RecordStoreWorker class.

The run method is inherited form the Thread class and is used to put the thread in a wait state until some data has to be stored in the record store. It will the call the saveData method and return to the wait state.

The saveItems method is invoked from the HomeCareRS class whenever there is data to store. The inputType parameter is used to determine which record to update, it can either be LoginUser, DailyPlan, UpdateObject or a DeliveredVisit. The data byte array is the actual data to store. The method signals the thread to enter the active state which will call the saveData method.

The saveData method stores the data in the record store. The implementation is shown in Figure 6.20.

```
private synchronized void saveData()
{
    try {
        rs.setRecord(updateType, data, 0, data.length);
    }
    catch(RecordStoreException e) {
        e.printStackTrace();
    }
}
```

Figure 6.20: The saveData method in the RecordStoreWorker class.

6.4 Models Namespace Implementation

The implementation of the Models namespace is not as simple for the client as it is for the service. Even though most classes just consists of private variables with getters and setters, it is not enough. The problem comes when the classes has to be made into a json string representation. The service uses .NET reflection through the Json.NET component to automatically create a json string from an object instance or an object instance form a json string. Java does not have the reflection ability, so the creation of strings and objects have to be implemented into each class that needs the functionality.

In the following we will take a closer look at these changes, as well as examine the Java implementation of the EncryptedData and Encryptor classes.

6.4.1 Changes to the Models Namespace

Four classes has to be changed to add this behaviour. Figure 6.21 shows the updated DailyPlan class diagram, Figure 6.22 shows the updated UpdateObject class diagram, Figure 6.23 shows the updated EncryptedData class diagram and finally Figure 6.24 shows the updated LoginUser class diagram.

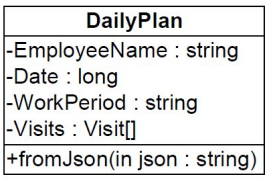

Figure 6.21: The changed class diagram for the DailyPlan class.

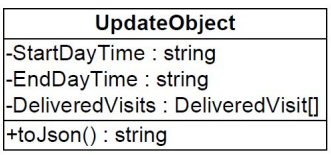

Figure 6.22: The changed class diagram for the UpdateObject class.

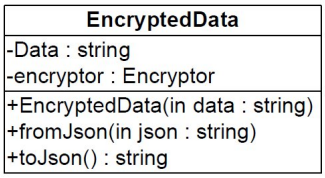

Figure 6.23: The changed class diagram for the EncryptedData class.

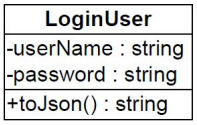

Figure 6.24: The changed class diagram for the LoginUser class.

6.4.1.1 fromJson

The fromJson method has been added to two classes: DailyPlan and EncryptedData. It takes a json string as input and populates the class variables with the values from the json string. Figure 6.25 shows the fromJson implementation in the EncryptedData class.

In this case there is only a single attribute to populate: data. A JSONObject instance is created from the json string input parameter. The data attribute is then set to the value of the attribute named Data in the JSONObject instance.

```
public void fromJson(String json){
    data = null;
    try {
        JSONObject jsonObject = new JSONObject(json);
        data = jsonObject.getString("Data");
    } catch (JSONException e){
        e.printStackTrace();
    }
}
```

Figure 6.25: The fromJson method in the EncryptedData class.

6.4.1.2 toJson

The toJson method has been added to the EncryptedData and UpdateObject classes. The method creates a json object representation of the class and returns the string representation of the json object. Figure 6.26 shows the toJson implementation in the EncryptedData class.

```
public String toJson() {
    JSONObject jsonObject = new JSONObject();
    try {
        jsonObject.put("Data", data);
    } catch (JSONException e){
        e.printStackTrace();
    }
    return jsonObject.toString();
}
```

Figure 6.26: The toJson method in the EncryptedData class.

6.4.2 Encrypted Data

The EncryptedData object is responsible for encrypting and decrypting json string representations of the objects that are send to and received from the service. The encryption happens in the constructor, and decryption happens in the getData method. Figure 6.27 shows the constructor and Figure 6.28 shows the getData method.

```
public EncryptedData(String data){
    encryptor = Encryptor.getInstance();
    try
    {
        byte[] request = encryptor.encrypt(Strings.toUTF8ByteArray(data));
        this.data = new String(UrlBase64.encode(request));
    } catch (CryptoException ce){
        ce.printStackTrace();
    }
}
```

Figure 6.27: The constructor in the EncryptedData class.

```
public String getData()
{
    String result = null;
    try
    {
        byte[] s = encryptor.decrypt(UrlBase64.decode(data));
        result = Strings.fromUTF8ByteArray(s);
    } catch (CryptoException ce)
    {
        ce.printStackTrace();
    }
    return result;
}
```

Figure 6.28: The getData and setData methods in the EncryptedData class.

6.4.3 Encryptor

The Java implementation of the Encryptor class is very similar to the C# implementation for the service. The differences are in the way to convert a string to a byte array (Java has a getBytes method on the String object, .NET uses a encoding class in the System.Text namespace) and the way to synchronise the encrypt and decrypt methods. Besides these two differences, the implementations are identical.

As an example Figure 6.29 shows the Java implementation of the encrypt method and by comparing it to the C# implementation from Figure 5.13, the only difference is that the Java implementation uses the synchronize keyword for synchronisation where the C# implementation uses an attribute.

```
public synchronized byte[] encrypt(byte[] data)
        throws CryptoException {
    if (data == null || data.length == 0) {
        return new byte[0];
    }

    cipher.init(true, key);
    return callCipher(data);
}
```

Figure 6.29: The encrypt method in the Encryptor class.

Chapter 7

Discussion and Conclusion

This chapter contains discussion and conclusion of the project.

7.1 Discussion

This section will discuss alternative solutions to problems introduced within this project along with the reasons why these have not been chosen. Furthermore the limitations and assumptions will be discussed.

7.1.1 Missing Functionality

The system we have developed only have the most basic functionality. It can only work with a daily plan for the current day. And only with the information that has been planned for it. It is not possible to change information belonging to the plan. For example if the employee has to make an unscheduled visit to another citizen, the visit cannot be added to the daily plan. The same is true for the services to deliver. It is not possible for the employee to change them using our system.

Other missing functionality includes the ability to make a phone call to the citizen that the employee is about to visit and to be able to see the daily plan for another day.

7.1.2 Database Structure

We are not allowed to change the database structure since Herning Municipality currently have their own system running on it. However changing the structure of the database would have allowed us to improve the insertion a lot since many of the problems are from design problems within the database. An example is missing foreign keys which even allow for bad data quality.

7.1.3 Updating the Database

We are currently only updating the database when the work day ends. This could give the potential for higher data loss if a disk failure occurs on the mobile client towards the end of the work day. To avoid this data loss it could be possible to send the current UpdateObject to the service, whenever there is added information and the device has network coverage.

7.2 Conclusion

We have developed a mobile solution. The mobile solution allows the employees in Herning Municipality to request a daily plan for their current work day. At the end of the day the central server is updated with the new data.

The solution has only been tested in our own development environment and not in the production environment at Herning Municipality. Because of this, we cannot say how our solution performs during communication sessions compared to the existing solution.

7.2.1 The Goals

We had two goals for the system: It should be more responsive than the current system and it should not have the need for a continues network connection.

7.2.1.1 Responsive System

The existing system have long response times for each server request. And since it is browser based, every action in the system issues a new server request. Our solution to this problem is to limit the number of server requests. We only access the server twice: The first access is to retrieve a daily plan containing all information related to the employees workday, the second is to send the data that needs to be updated.

If the time problems comes from either the network or the database access, then our request for a daily plan will not be faster than the existing system, since there is no way to generate a daily plan without accessing the database. However we only have to do this once for each workday, since we store the entire plan locally on the employee's mobile device and any subsequent request, until the update is send back to the server, is performed locally and thus will be instant.

When we send the update back to the server, we store the update information in a log file at the server. When the log file has been saved, the

client is informed that the update has been received, before the update gets added to the database. This way the database access does not influence the response time and the user should be getting a faster response.

7.2.1.2 No Continues Connection

With our solution the employee only have to access the service at the beginning and end of the workday. During the period where the employee visits the different citizens, our solution doesn't require a network connection and thus it doesn't matter if the citizens live in places that does not have network coverage.

7.2.2 Criteria

We defined three criteria that we would like our solution to achieve: Minimal data input, high data quality and protection of sensitive data.

7.2.2.1 Data Input at a Minimum

This criteria have been achieved to the extent that the employee only have to input a user name and a password to receive a daily plan. The client program then guides the employee through the workday, where all the employee has to do is to choose the next action (for example Start Visit or End Visit). The update is sent to the server automatically when the workday has come to an end.

7.2.2.2 Data Quality

This criteria have been harder to work with since the data quality in their current database is not high. Examples of this is that certain tables allows for invalid data which means that in certain cases it would not be possible to extract the duration for a service from the database. The criteria have been fulfilled to the point that we ensure that each row is only inserted one time. Furthermore we also ensure that the data quality in the database is not getting any worse. This means that the places where we only have access to invalid data we do not have any other options but to insert this invalid data into the database again.

7.2.2.3 Sensitive Data

This criteria have been fulfilled to the extent that we have used 128 bit AES encryption for all data sent between the web service and the mobile client.

This means that if the information is intercepted during transmission, no useful data about the citizens can be retrieved by an attacker.

Bibliography

[Bag08] Rob Bagby. Demystifying the code, rest in wcf blog series index, 2008.

[Cod86] E. F. Codd. Missing information (applicable and inapplicable) in relational databases. *SIGMOD Rec.*, 15(4):53–53, 1986.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[Fie00] Roy T. Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[Gra81] Jim Gray. The transaction concept: Virtues and limitations (invited paper). In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 144–154. IEEE Computer Society, 1981.

[Han72] Per Brinch Hansen. Structured multiprogramming. *Commun. ACM*, 15(7):574–578, 1972.

[HR83] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[LP76] M. Lacroix and A. Pirotte. Generalized joins. *SIGMOD Rec.*, 8(3):14–15, 1976.

[MSD05] MSDN. An introduction to c# generics, 2005.

[MSD09] MSDN. Language-integrated query (linq), 2009.