Object / Data Source Mapping Layer

Uffe Dejligbjerg

Kgs. Lyngby 2003
IMM-THESIS-2003-45

Object / Data Source Mapping Layer

Uffe Dejligbjerg

Kgs. Lyngby 2003

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

IMM-THESIS: ISSN 1601-233X

Acknowledgements

I have reserved this page for appropriate and inappropriate, sentimental and unsentimental words of thanks.

First of all, I would like to thank my pregnant wife who caringly stood by me through these last seven months even though I have spent most of my time in the basement writing this thesis and developing the software framework. I would like to thank her very much for holding on to me without ever complaining – at least not about the project.

Thank you to all the people with whom I have discussed this project, with a special thanks to the following.

- **Bjarne Poulsen** – my supervisor, who helped me find literature and gave me valuable suggestions about the structure and content of this thesis all the while constantly reminding me that delivery dates were just around the corner.
- **Vibeke Harder Dejligbjerg** – my wife, who somehow found the time to proofread my thesis after normal work hours, and repeatedly helped me reformulating and rewriting my rather cryptic and sometimes incomprehensible phrases and sentences.
- **Henrik Skydtsgaard Nielsen** – who always could come up with a feature to add to the framework whenever he did not run application tests.
- **Jannich Perch Jensen** – who always stood by with moral support and pleasant company whenever a coffee break was emerging or the pizza delivery came by the office.
- **Lise Bach Clausen** – who found the time to read and comment on my thesis even though she had no relation to my work area.

Abstract

DEJLIGBJERG, UFFE: Object / Data Source Mapping Layer: Technology behind and development of a common access layer for different data sources. (Under the direction of Bjarne Poulsen, IMM, DTU and in collaboration with MAN B&W Diesel A/S)

The purpose of this research project is to investigate the possibility of creating a common access layer that connects to different data sources. I have set out to explore the possibilities of creating and implementing a framework supporting and improving efficiency in application development.

The proposed architecture provides a mapping of data properties from data sources to an object-oriented programmatic layer that reflects data schemas explicitly in its class structure which can be accessed directly by the business logic. This means that application developers will be able to concentrate on analysing and designing the business layer instead of writing database access source code or managing data sources and queries.

The framework architecture will make data access generic so that the business layer is not filled with data source specific source code. This will eliminate the need to embed SQL statements into business layer source code while isolating the business layer from changes made to the implementation of data sources. Finally, this will introduce an object-oriented data structure, which will improve maintainability and allow developers to focus on a more object-oriented design process.

The development process will be inspired by *The Rational Unified Process*, as prescribed by Jan Sørensen, MAN B&W Diesel A/S [8] according to the standard project guide for software development. This project will make use of these techniques in handling and planning the process, but since this project has a technical theme it will not be a study in process handling.

Kgs. Lyngby, August 15th 2003

Uffe Dejligbjerg, s938132

Resumé

DEJLIGBJERG, UFFE: Objekt-orienteret tilknytningslag for datakilder: Omhandler teknologier og idéer bag udviklingen af et fælles tilgangslag til forskellige datakilder. (Under vejledning af Bjarne Poulsen, IMM, DTU og i samarbejde med MAN B&W Diesel A/S)

Formålet med dette projekt er at undersøge mulighederne for at udvikle et fælles tilgangslag, som forbinder forskellige typer af datakilder i ensartede tilgangsmetoder. Jeg vil udforske principperne i at skabe og implementere en programmatisk struktur, som understøtter applikationsudvikling med forøget effektivitet.

Arkitekturen knytter datakildeegenskaber fra datakilder til et objekt-orienteret programmatisk lag som reflekterer dataskemaer eksplicit i sin klassestruktur, som kan tilgåes direkte fra implementationen af forretningslogikken. Dette betyder, at applikationsudviklere kan koncentrere sig om udelukkende at analysere og implementere forretningslogik i stedet for at skulle vedligeholde forespørgsler i samt adgang til databaser i deres applikationskildekode.

Løsningens arkitektur gør tilgangen til datakilder generisk, så der ikke eksisterer dataspecifik kildekode i forretningslaget. Dette vil fritage udviklere for at placere datakildespecifikke udtryk, f.eks. SQL-udtryk, i forretningslaget, samtidig med at det vil isolere forretningslaget fra ændringer der foretages i implementationen af datakilder. Endeligt vil det tilføje objekt-orienterede principper i databaseprogrammeringen. Dette vil lette vedligeholdelsen af forretningslaget og tillade udviklere at fokusere på objekt-orienterede designprocesser.

Selve udviklingsprocessen er inspireret af *The Rational Unified Process* som foreskrevet i projekthåndbogen, der er udviklet til brug for softwareudvikling i MAN B&W Diesel A/S af Jan Sørensen [8]. Processen bruges til projekthåndtering og –planlægning, men da projektet primært omhandler en teknologisk udvikling, vil selve principperne bag udviklingsprocessen kun blive sporadisk behandlet.

Kgs. Lyngby, 15. august 2003

Uffe Dejligbjerg, s938132

Table of Content

- ACKNOWLEDGEMENT………………...

- ABSTRACT…………………………….. i

- RESUMÉ IN DANISH……………………... ii

- TABLE OF CONTENT……………….. iii

1 INTRODUCTION……………………………………………………………………….1-3

1.1 BUSINESS RELATIONS……………………………………………………………………….. 1-3
1.2 PROJECT DESCRIPTION………………………………………………………………………. 1-5
1.3 STAKEHOLDERS AND USER DESCRIPTIONS………………………………………………….... 1-9
1.4 PRODUCT OVERVIEW…………………………………………………………………….. 1-12
1.5 PRODUCT FEATURES……………………………………………………………………... 1-14
1.6 CONSTRAINTS……………………………………………………………………………. 1-16
1.7 DOCUMENT OUTLINE…………………………………………………………………….. 1-17

2 PROJECT PLANNING……………………………………………………………2-3

1.1 PROCESS DESCRIPTION……………………………………………………………………. 2-3
1.2 THE RATIONAL UNIFIED PROCESS………………………………………………………… 2-4
1.3 IMPLEMENTING THE PROCESS…………………………………………………………….... 2-7
1.4 PROCESS WORKFLOW…………………………………………………………………… 2-10
1.5 SUMMARY………………………………………………………………………………... 2-13

3 REQUIREMENT SPECIFICATIONS…………………………………………... 3-3

3.1 PROCESSING REQUIREMENTS……………………………………………………………… 3-3
3.2 INTENDED AUDIENCE AND READING GUIDE…………………………………………………. 3-4
3.3 FUNCTIONAL REQUIREMENTS………………………………………………………........................ 3-5
3.4 NON-FUNCTIONAL REQUIREMENTS………………………………………………………… 3-16
3.5 ROLES…………………………………………………………………………………… 3-19
3.6 REQUIRED DOCUMENTATION…………………………………………………...................... 3-20
3.7 ACCEPTANCE TEST AND CRITERIA…………………………………………………………. 3-21
3.8 SUMMARY………………………………………………………………………………... 3-22

4 TECHNOLOGIES………………………………………………….................................. 4-3

4.1 THE .NET PLATFORM…………………………………………………………………………….. 4-3
4.2 .NET COMPONENTS…………………………………………………………………………….4-5
4.3 ENVIRONMENT…………………………………………...4-8
4.4 DATA SOURCES…………………………………………………………………………...4-11
4.5 DESIGN PATTERNS……………………………………………………………………….4-14
4.6 SUMMARY…………………………………………………………………………………...4-17

5 ANALYSIS……………………………………………..5-3

5.1 PURPOSE……………………………………………………………………….............................. 5-3
5.2 STRATEGY ANALYSIS…………………………………………………………………………... 5-4
5.3 GENERIC OBJECT-MODELLING OF DATA SOURCE……………..5-9
5.4 ARCHITECTURAL SIGNIFICANT DESIGN PACKAGES…………………………...................................5-15
5.5 SUMMARY…………………………………………………………………………………...5-17

6 DESIGN…………………………………………...6-3

6.1 PURPOSE……………………………………………………………………….............................. 6-3
6.2 THE GENERIC OBJECT MODEL…………………………………………………………………... 6-4
6.3 DESIGN REPRESENTATION……….. 6-14
6.4 SUMMARY…………………………………………………………………………………...6-18

7 DESIGN…………………………………………...7-3

7.1 IMPLEMENTATION STRATEGY………………………………..7-3
7.2 FRAMEWORK EXAMPLES……………………………………………………………….............. 7-6
7.3 TESTING…………...7-13
7.4 SUMMARY…………………………………………………………………………………...7-17

8 CONCLUSION…………………………………...8-3

8.1 PURPOSE……………………………... 8-3
8.2 GOALS…………………………………………………………... 8-3
8.3 SUMMING UP….. 8-4
8.2 EVALUATION OF ESSENTIAL RESULTS…………………………..8-5
8.3 SUMMARY OF CONTRIBUTIONS..8-6
8.4 FUTURE IMPROVEMENTS………………………………………………………........................... 8-7

- BIBLIOGRAPHY……………………………...C

- APPENDIX……………………………………... II

A CLASS DIAGRAM………………..III

B ENTITY SEQUENCE DIAGRAMS………………………………..V

C INTERFACE CONTRACTS.. XI

D INITIALISING METADATA IN THE FRAMEWORK …………...XIX

E DATA SCHEMAS...XXI

CHAPTER 1

Introduction

1 INTRODUCTION 3

1.1 BUSINESS RELATIONS 3

1.1.1 Historical View 3

1.1.2 Needs and Opportunities 4

1.2 PROJECT DESCRIPTION 5

1.2.1 Project Purpose 5

1.2.2 Project Scope 6

1.2.3 Project Delimitations 7

1.2.4 Definitions, Symbols, Acronyms and Abbreviations 8

1.2.5 References 8

1.3 STAKEHOLDERS AND USER DESCRIPTIONS 9

1.3.1 Infotek – The IT Department 9

1.3.2 System Analyst and Software Architect 10

1.3.3 Application Developers 10

1.3.4 Basic Infrastructure 11

1.4 PRODUCT OVERVIEW 12

1.4.1 Product Perspective 12

1.4.2 Assumptions and Dependencies 12

1.4.3 Cost and Pricing 13

1.5 PRODUCT FEATURE 14

1.5.1 Abstraction 14

1.5.2 Database Portability 14

1.5.3 Maintainability 14

1.5.4 Faster Development 14

1.5.5 Error-Free Development 15

1.6 CONSTRAINTS 16

1.7 DOCUMENT OUTLINE 17

1 Introduction

1.1 Business Relations

1.1.1 Historical View

Since the early days of computing, the MAN B&W Diesel A/S, formerly B&W Motorfabrik A/S, has pursued a leading role in the use of the brute force of calculating power. And not just for the fun of it. As a company relying heavily on thermo dynamical and material engineering the process of calculations has been the key foundation on which the company is built. That and ingenuity.

It started with punched cards via tube based calculators till today's supercomputers. And with it, it's massive amount of stored data. Today not just calculations are done on computer. Everything is stored on computer; drawings, engine data, sales data, documents and the like. And it is even stored in revisions to keep a historical track record. As computers were introduced to more and more departments and persons, the number of different types of data grew. And with it came redundant data. Every new application stored a number of data, and often it overlapped.

With the introduction of the Microsoft .NET platform, MAN B&W Diesel A/S wanted to set a standard for developing the applications that supported the business. It was already realised that the huge number of different data sources storing business dependent data was a problem for the organisation. As the programmers used an array of different tools and data sources, every application was designed as a vertical solution. In short this means that the application is designed to solve a specific problem for a specific company or department. Reuse was nowhere to be found neither in source code nor in data storage.

Today, developers at MAN B&W Diesel A/S are in a process of defining the n-layer component based architecture on which future development is to be based. This includes defining a standard for storing data in databases. But some systems are already built and in operation and other systems are brought in from the outside to the domain needs to be connected to in-house applications.

1.1.2 Needs and Opportunities

In the MAN B&W Diesel A/S organisation today, all business critical systems run on a mainframe system. Except from SAP R/2, all systems have been developed by the company itself and most of these are already well beyond twenty-five years of age. All data is stored on the same mainframe platform as the applications, but due to its age the mainframe platform and the corresponding applications are currently being replaced. This replacement has caused data to be scattered throughout multiple systems and possible data sources due to lack of data planning.

Other than the mainframe systems, there are only few business supporting applications. Most of these have non-critical functions like recording clock ins and outs or giving an extra Windows GUI to existing mainframe data. But all these applications have been developed as vertical solutions. All applications have there own persistent storage, component layer and GUI interface. There has been little or no reuse of application source code or data.

The IT organisation has decided to use only the Windows operating system as client platform. On the server level all application and web servers run the Windows platform when replacing the mainframe applications. Only a few database systems will run Sun Solaris and maybe Linux in the future, but these non-Windows platforms are only for data storage.

This overall upgrade process is currently taking place. The largest application transitions are the shift from SAP R/2 to the Windows/Oracle-based SAP R/3 and the replacement of the home built Engine part lists and drawings management system to a full-scale implementation of a PLM system called TeamCenter Engineering developed by EDS. On the whole this transition in technology is a welcome opportunity to clear up data storages and the surrounding applications.

1.2 Project Description

The basics of this project are to find a developing tool that favours the reuse of data in the organisation. Data reuse is a key objective in the current development program of the IT division as the organisation sets out to clear up in the redundant data that exists in various applications in the company. Furthermore, the project must enhance the development of data-handling applications.

A decision has been made to investigate the prospects of implementing an Object / Data Source Mapping Layer that unites many different data sources in a common access layer. This type of technology is not new to software architects, but now it has to prove its value and capabilities in the development of applications within the framework of MAN B&W Diesel A/S.

Even though many implementations of this kind of mapping layers exist on the market today, the technology is new to MAN B&W Diesel A/S; hence we cannot decide up front whether to buy an existing framework or to home-grow our own. As mentioned before, the organisation has a history of developing business critical applications within the organisation instead of buying standard software. Due to this fact, this project must outline the technologies behind an Object / Data Source Mapping Layer in order to form the necessary general view of the technology.

1.2.1 Project Purpose

The project Object/Data Source Mapping Layer sets out to create an object oriented class foundation for accessing data in the software development organisation of MAN B&W Diesel A/S. This class foundation - also called a framework - must support the current data sources used by the organisation and at the same time it has to be so extensive that it can be used by the application developers.

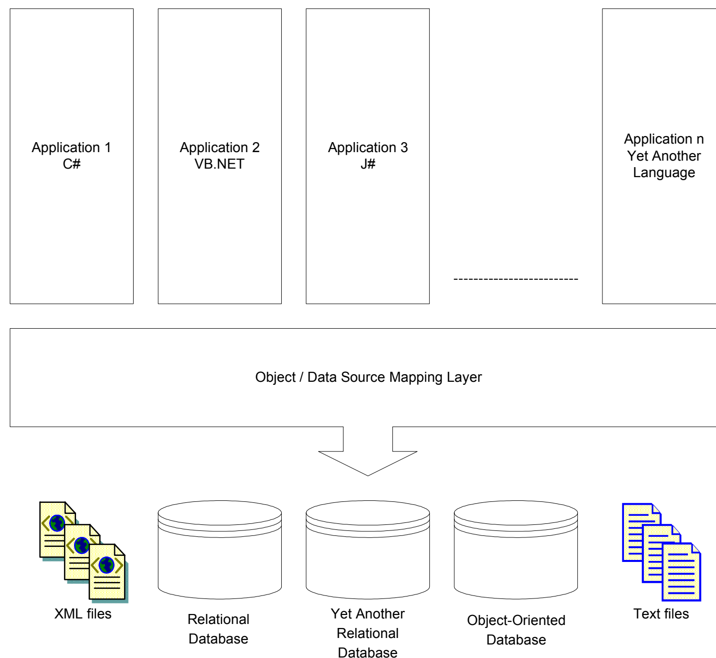
On the market today there are products which implements an Object / Relational Mapping framework, and there is a possibility that one of these products can fill out the requirements specified by the IT Department in the organisation. Therefore this project includes an end-evaluation of whether to try, buy and tailor an existing Object / Data Source Mapping product or to continue the development of an internal product with special functionally.

1.2.2 Project Scope

The end result of this project consists of two parts; the first part is an analytical undergoing of the technical principals within an object superstructure on top of different data sources. This includes an introduction to the various types of data sources currently in use in the MAN B&W Diesel A/S IT infrastructure and some of those planned for the future.

How object modelling on data sources works in terms of technical aspects will be explained in general in order to establish a foundation on which a solution can be evolved. The general ideas will then be placed in the context of the choice of platform that MAN B&W Diesel A/S uses in its organisation. In this case the cutting-edge technology of the .NET platform from Microsoft.

The other part is an implementation of the ideas proposed in the analysis. This implementation will demonstrate the functionality through the demo-application which is specified in the test schema. Here the framework will have to prove that the concepts work in a practical example. This will also provide a better sense of how the framework actually works to the involved parties in the project. A sketch of the idea is seen on figure 1.1.



Project planning being a part of this project it is not performed as a study in itself. Instead it is merely used as a tool to support the process of writing this thesis. MAN B&W Diesel A/S has chosen to follow the Rational Unified Process when doing project management in the software development organisation. This is the direct reason for choosing this particular process tool in the project.

In connection to an internal software project called LIME Project, Jan Sørensen has outlined a standard for project planning and execution [8] in Infotek, which is the name of the umbrella

organisation that covers all IT related aspects of MAN B&W Diesel A/S. This document is based on the Rational Unified Process and forms the basis for process handling in this thesis.

1.2.3 Project Delimitations

The most significant delimitation is the given timeframe of this project. Companies that offer products with this kind of functionality have whole staffs of architects and developers in order to keep the product up-to-date in terms of features and support. This fact will of course be considered when a final analysis of the actual needs of MAN B&W Diesel A/S is being carried out. Still, this is not an argument not to proceed with this thesis. The objective is to gather information about the technology behind an object/data source mapping layer as well as implementing a prototype.

Since the project will deal more with specifying the features needed and go through technologies connected to it than implementing a full-scale solution, there will be a down-scoping of the implemented features. This means that all features in the requirements will be rated and some cut off in the first, preliminary version that this project concludes in.

1.2.4 Definitions, Symbols, Acronyms and Abbreviations

DTU	-	Technical University of Denmark
MBD	-	MAN B&W Diesel A/S
Infotek	-	Umbrella organisation covering all IT related departments at MBD
RUP	-	Rational Unified Process
VS.NET	-	Visual Studio .NET
PLM	-	Product Lifetime Management
LIME	-	Lifetime Management of Engines
UML	-	Unified Modeling Language
CRUD	-	Create, Retrieve, Update and Delete
[Number]	-	Numbers in square brackets referees to a source in the bibliography.

1.2.5 References

BJP	-	Bjarne Poulsen IMM, Technical University of Denmark
HAM	-	Hans O. Mortensen, Head of Dept. 9535 9535 Tech. Adm. Systems, MAN B&W Diesel A/S
JAS	-	Jan Sørensen, Project Manager 9535 Tech. Adm. Systems, MAN B&W Diesel A/S
NIG	-	Niels Garde, Web Developer 9550 Business Systems, MAN B&W Diesel A/S
UFD	-	Uffe Dejligbjerg 9535 Tech. Adm. Systems, MAN B&W Diesel A/S
HSN	-	Henrik Skydtsgaard Nielsen Software Engineer, Microsoft Business Solutions

1.3 Stakeholders and User Descriptions

According to RUP [5], in one form or another, many different parties have interests in a project. This list of involved parties is in short known as the stakeholders. The stakeholders are the funding authorities, users, project managers, developers and so on all related to a project. They may play different roles in the execution of the project plan, but still they are groups of people that are related to the project. In order to understand the needs of the stakeholders we have to define them. This will provide us with an identical understanding of the parties involved in this project and their interests.

1.3.1 Infotek – The IT Department

Infotek is the sole fund holder in this project; hence in the end it is the management of Infotek that decides whether the project lives or ends – and if it lives, in which direction the project goes. Of course the final decision is highly influenced by the other stakeholders, but this stakeholder may be considered as the most important one. Infotek also holds the responsibility towards the rest of the organisation and the board of directors.

Since Infotek holds both funds and the direct responsibility for the project, the members obviously favour non-technical issues and features. They have no rush in introducing new technologies unless the department gains some value that can be measured in the financial balance. This includes numbers in staff and currency. Whether this needs to be achieved on a short or a long-term basis only management can decide, as they have the financial overview of the company. Even though a project can cut staff and save money in the long run by investing now, management requires that the money is present for the investment, and that the organisation is ready for a change in habits.

The most important of these requirements is to cut down on development costs. Like most other departments in various companies around the world, Infotek have a list of tasks to do, services to provide and a limited bag of money to support it. The easier it is to develop the needed applications and provide the services, the lesser money is required; hence it accumulates a larger profit. This calls for a higher level of programming and a robust and easy to use framework on which to found the architecture and built the applications.

With a high-level programming tool and a supporting framework that eases the production of the application, every resource becomes more productive. This fact reduces the need for

resources, e.g. the staff of developers. Instead of cutting back on staff, a cut down on time-to-market could be desirable. This could be combined with adding more features to applications, increasing the list of deliverables or adding more services to the department area. This will facilitate additional growth possible for the company.

1.3.2 System Analyst and Software Architect

The System Analyst organises and formulates the requirements while trying to understand the technological constraints and risks. The Software Architect then composes the architectural blueprint on which to build the proposed solution. During this thesis I will play both roles. However, I will co-operate with my employer, Infotek, on the task as System Analyst.

The first thing to do is to capture all the requirements that the department could possibly think of. And just as important, capture all the requirements they did not come to think of. Especially the last part can be rather difficult since the analyst has to read between the lines and through context in order to catch all the needs that the clients – in this case Infotek – have.

While gathering the requirements there should be no constraints on any technical or architectural solution. This means that a choice between exploring new ideas of tomorrow or using only proven technologies should be postponed until touched upon in the analysis and combined with a proposed solution in the design. Still, due to the nature of this project as a thesis for a Masters Degree at DTU, I will try to explore the possibilities of new technologies while aiming for an expandable and robust architecture that is easy to use and maintain for future application developers.

1.3.3 Application Developers

When assigned to a project, a team of application developers must implement a proposed solution within a given timeframe. The developers are evaluated on the fulfilment of required features and functionality and whether they lived up to the milestones set during the project. It is needless to say, that the more source code is prepared beforehand, the easier it is to be in time for the deliverables.

This implies that application developers welcome the introduction of higher-level programming tools. Each new level of abstraction in the development tools gives an

advantage in time consumption, hereby reducing the time needed to implement a solution. A better architectural structure in the frameworks on which solutions are built, also eases the implementation load.

1.3.4 Basic Infrastructure

The Basic Infrastructure department is in control of all computers and related products in the organisation. They deploy software, new clients and servers. So when a new piece of software needs to be installed onto clients in the organisation or a new server need to be connected, this department is behind the show. Furthermore, they are in control of creating new databases and securing the network in general.

The members of this department will therefore require that the product is easy to use, configure and deploy. And there should be no or only simple support afterwards. With more that 1000 client PCs, 200 servers and a large wired set of switched cable net on MBD, there are enough daily tasks already to handle.

1.4 Product Overview

This product – or proposed solution – will integrate access to different data sources under one object-oriented framework. It will provide a set of common access methods by wrapping data sources with different characteristics into a uniform set of objects. This will enable application developers to change between different data sources that implement the same data schema, without changing any source code. Only an update of a configuration file that names the new data source instead of the old one is required.

Furthermore, the data schemas of the data sources will be reflected in the object model where it will enhance the general view and understanding of existing data in the domain. This helps application developers to implement functionality without having to look up documentation about data schemas time and time again. Hopefully it also helps reduce the task of memorising data schemas.

1.4.1 Product Perspective

This product sets out to be the only framework for accessing data in a development environment. In time, when fully implemented, it will eliminate the need to use different file handling libraries, database access libraries like ADO, ADO.NET or other tools to create connectivity to data sources.

Gradually over time, new data sources will be introduced and installed in the organisation. Data will be migrated from one platform to a new one. This will not be a problem when using this framework as long as the data schema is migrated along with the data. It requires only one single point of implementation of the data source before all applications are running on the new data server. All application source code is reusable.

1.4.2 Assumptions and Dependencies

In this report we must assume that a broad spectre of data sources currently exists and that these data and corresponding data schemas will have to exist in their current form. This implies that no need for data migration or conversion due to the Object / Data Source Mapping Layer will be accepted. Instead the proposed framework will have to extend data as they are. This collection of existing data sources can be assumed to be running on very different operating systems and implemented on various platforms.

The platform on which the framework must run can be limited to Microsoft Windows 2000/XP. This is a political decision made by the management of IT Infrastructure in order to simplify the requirements to future software and application integration. This decision require that all other platforms, operating systems and applications must be phased out – a process that has already come a long way. Therefore architecture and development can be assumed to be legacy-free without the need to support current applications, utilities or systems. All efforts can be put into forming a new basis for future development.

This thesis is carried out without any resources from MBD worth mentioning. As described in the project scope and delimitations this limitation causes a need for cutting of some features and functionality. Therefore this project has no dependencies on or constraints towards resources and deliverables from MBD.

1.4.3 Cost and Pricing

Initial investigation and preliminary development in this project is carried out as a Masters Thesis at DTU; hence the cost to this project will be very limited. The project is conducted parallel to a full-time job with a normal portfolio of projects and cost is therefore reduced to unpaid overtime and holidays. Only a few co-ordinating meetings can be regarded as a real cost.

When evaluating the project, we must assume that a final implementation of the full framework will require resources and make demands on funds. I therefore propose a mid-way evaluation to be conducted in the fall after the evaluation of the project to decide whether to continue the project or take another development path.

1.5 Product Feature

Before drawing up the full requirements, I will now give a short introduction to the features of this product to provide a sketched survey of the solution.

1.5.1 Abstraction

To help application developers, the application source code will be completely isolated from the source code needed to access data sources. Instead, the product will provide common access behaviour that exposes data schema only as some kind of meta-data. Therefore, changes in the implementation of the framework or in the data sources will not cascade into changes in the application source code.

1.5.2 Database Portability

Portability is obtained, so that only changes in the data schemas may result in changes in the application source code. Changing database providers is only a matter of changing the database type in the mapping layer and copy the data schema and data to the new provider. No application source code changes are required.

1.5.3 Maintainability

Only a simple behavioural interface is available to application developers. This abstraction from data source logic gives a clean cut between layers, making it easier to implement new ways of accessing data and correcting errors without changing application source code. In addition, due to the abstraction of the persistence layer, a change in the data providers does not cascade into changes in the application source code.

1.5.4 Faster Development

Since the data schema is reflected in the framework and no data source access source code has to be written, application programmers can focus on the business problem at hand and the development of data schemas and business logic. Furthermore, since the interface to the framework is object-oriented, application developers can take better advantage of object-oriented concepts instead of thinking in a relational manner.

1.5.5 Error-Free Development

As the data sources are encapsulated in an object layer with a set of common access methods, the application developers will be less likely to introduce errors when accessing the data models. Moreover, as the framework is tested and the source code that accesses data is only written once, new errors are less likely to be introduced. Instead, developers can focus on getting the data schema and business logic right.

1.6 Constraints

As in all other companies over a certain size, there are restrictions of a political nature. These restrictions does not need to point to a solution that is proven to be market leading. It can be beneficial even though it implies some constraints or lost possibilities. The major advantage of these political decisions is to tidy up the organisation and standardise the tools used in the business processes.

On a technical level it has been decided, that all future implementations and developments are to follow Microsoft's technical guidelines and best practices. This does not mean that all other components and development frameworks are banned; it just requires them to be fully compliant with Microsoft standards as well.

In the eyes of system architects and application developers, this decision requires all development to be carried out on the development tools from Microsoft. Furthermore, all future development – including this project – must be founded on the newly introduced .NET platform with the main programming language set to be C#.

1.7 Document Outline

This project is organised in the following chapters:

Abstract

Outlines the synopsis of the project and introduces the reader to the theme of the thesis.

Chapter 1 – Introduction

Provides an introduction to the project and the background for the requirement specification contained in chapter 3. It includes a definition of all abbreviations and naming of people with relations or contributions to this project.

Chapter 2 – Project Planning

Gives an overview of how the project is handled and specifies the deliveries during the project.

Chapter 3 – Requirement Specifications

Lists the functional and non-functional requirements for the project. Furthermore, it provides a reading guide and defines the roles and user profiles related to the project.

Chapter 4 – Technology

Defines the technologies used in the project and explains the design patterns used in the architecture and design.

Chapter 5 – Analysis

Contains an analytical discussion based on a set of papers that defines multiple architectural strategies. The chapter concludes in a conceptual model for a solution.

Chapter 6 – Design

Forms a design on the basis of the analytical solution. This evolved solution forms the basis for the implementation.

Chapter 7 – Implementation and Test

Since this project forms a prototype, implementation and test is constituted in the same chapter. Gives an overview of how the design is implemented and an application test of the solution.

Chapter 8 – Conclusion

A final summary based on the chapter summaries in the paper.

Appendix

Contains additional text and information to the project.

Literature List

Contains references to papers and literature on a list form.

CHAPTER 2

Project Planning

2 PROJECT PLANNING ... 3

2.1 PROCESS DESCRIPTION ... 3

2.2 THE RATIONAL UNIFIED PROCESS ... 4

2.2.1 Iterative Development ... 4

2.2.2 Visual Software Models ... 6

2.3 IMPLEMENTING THE PROCESS ... 7

2.3.1 Architectural Process ... 7

2.3.2 The Spiral Model ... 8

2.3.3 Phases and Milestones ... 8

2.4 PROCESS WORKFLOW ... 10

2.4.1 Project Phases ... 10

2.4.2 Project Deliverables ... 11

2.5 SUMMARY ... 13

2 Project Planning

2.1 Process Description

This project consists of different elements. From the outside, the project only appears to be a software project. But in order to make the journey from idea to final product without just groping in the dark, it is worth the effort to lay out the path that the project must follow.

Having recognised the need to define a path guiding the project from beginning to end one has to attack the problem of defining and describing the project in a well-documented software development process. Instead of trying to figure it all out on your own, the process description can be based on the experience of others.

In order to achieve a solid and well-proven template on which to mould a process, it is wise to built your process template on the experience of others. In MBD, the experience has been brought in from the definitions collected of the Rational Unified Process. Based on this, JAS has written a document [8] that defines a template path on which to found the processes of software development projects.

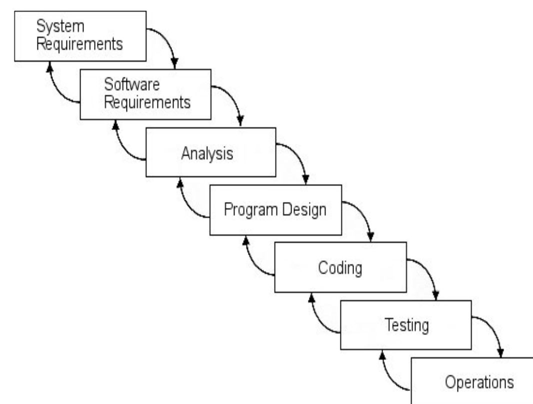
2.2 *The Rational Unified Process*

The RUP is many things that all support the process of developing software. It is a book, a development method, a set of guidelines and a software tool. It is an all-in-one process framework that helps you define a development process and documents it underway using the UML standard. It is a general and comprehensive utility that can be used as a platform for process planning and description.

While the Rational Unified Process can be used out of the box by following all the steps contained in the framework, it is not my intention to follow it blindly, generating useless work and producing documents that add only little or no value to the project. Instead, the RUP must be modified, adjusted, and expanded in order to consider the specific needs, constraints, and culture of the organisation implementing the framework. In this organisation, the focus is on the development methods and guidelines in the RUP whereas the software is not being used.

2.2.1 Iterative Development

Most software development processes still follow the principals of the well-known Waterfall Model according to which the development process consists of a strict sequence of phases: requirement analysis, design, implementation/integration, and test. This inefficient approach dictates the completion of one phase before starting another. In extreme cases, it utilises only one group of resources at a time, leaving all the others idle. Furthermore it postpones testing until the very end of the process. This can allow structural or implementation errors to go unnoticed until the end of the project lifecycle, posing a serious threat to the deadline.



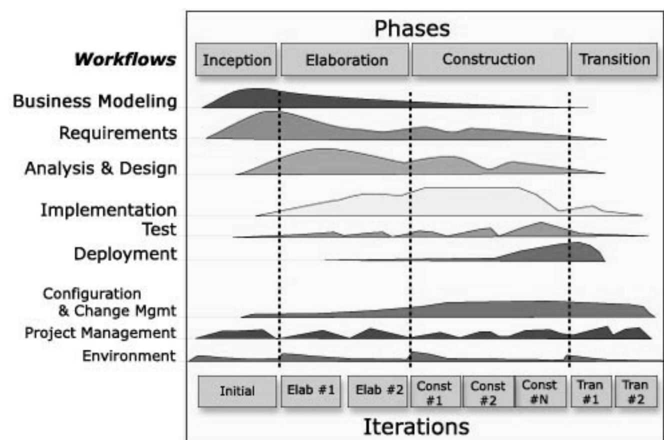
RUP, on the other hand, represents an iterative approach that implies a number of advantages. Let's regard the first step, the requirements. Throughout the entire development process, the scenario of a fixed set of non-changing requirements is not very likely. While the project idea matures and evolves, it must be possible to reflect these changes in the requirements.

Thus, an update of the requirements is a reoccurring event which will trigger a review and update of all the following parts of the process. This development method in which every pass-through is adding a little to the project is exactly what makes it an iterative process.

The iterative process provides the project management with a means of making tactical changes to the project during the process. It makes it possible to reschedule the release date by reducing functionality or to base the implementation on components from another supplier or even on another technology. It even allows different groups to identifycommon solutions to similar problems during the planning.

Just like the Waterfall model, the RUP specifies a number of workflows. But instead of completing one workflow before starting another, all are executed in parallel. Now, with all these workflows coexisting while depending on one another, not all workflows demand the same amount of effort and resources on a given time. E.g. a tester will not be able to test the product when it consists of a mere hundred pages of requirements standing on a shelf.

In the guidelines for the RUP, a workload is estimated for the different types of workflows in different stages or phases of a project. The figure shows that the workloads peaks sketch the same ladder down through the model as the Waterfall model. But no workflow is completed until the end phase – here called the Transition. This does not mean that deadlines cannot be set. It does mean, however, that changes can be made after deadline.



This is why project managers often avoid using the iterative approach, since it can be regarded as a kind of endless and uncontrolled process. But this is exactly what the RUP offers; a set of tools and guidelines to control an apparently uncontrollable process. The number, duration, and objectives of the iterations can be carefully planned, and the tasks and responsibilities of participants must be very well defined. Some reworking of workflows can takes place between two iteration steps but this is also controlled by the RUP.

2.2.2 Visual Software Models

Models are sketches of reality. They help us to understand a problem and outline its solution. Without a simple sketch, we would be unable to understand large and complex systems. This is where the Rational Unified Process provides a set of tools and model templates that are the descriptive language of software modelling.

This language called the Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing, and documenting the elements of a software system. It provides the standard means for writing the blueprints of the solution. It covers conceptual ideas such as business processes and system functions, as well as concrete items like classes written in a specific programming language, database schemas, and reusable components.

While UML provides an ability to express various models, it does not instruct the user to use all models at the same time. The different types of model templates can be regarded as different kinds of views on the same problem and this is why there is some overlapping. Therefore, different problems require a different combination of UML models to describe it.

2.3 Implementing the Process

When a system is based on handling business processes and user interactions it is appropriate to analyse the needs by creating use-cases to define the requirements. Normally, a system has a number of user types called Actors. Every type of Actor uses the system to perform a number of actions. A use-case is an UML notation that describes such a series of actions that the system performs in order to return some result of value to an actor.

Now, this description fits the analysis and description of a number of business processes that must be supported by an application. But in this case, according to the user profiles in the requirement this proposed solution aims at constructing a framework that supports Application Developers and Database Designers in their development process therefore there is no use-cases to define since no business processes are involved.

2.3.1 Architectural Process

When designing a system, use-cases can guide you through requirements, analysis, design, implementation and tests. The approach is fine if the project sets out to implement business rules on an existing architecture already implemented in the infrastructure. But use-cases can only define user interaction with a system; it cannot define the inside of the software engine. So, when starting from scratch an architectural approach is required.

The architecture can be regarded as a common vision that gives the involved parties a clear perspective of the whole system, which is necessary to design the system. Is must define the most important elements of the proposed system e.g. grouping subsystems in systems and sketching their mutual dependencies. In other words, it defines the foundation on which the system can be understood, built and evolved in the future.

The main focus of early iterations is to produce and validate a software architecture. In the initial development cycle, this takes the form of an executable architectural prototype that gradually evolves, through subsequent iterations, into the final system.

Since this project deals with extending an existing framework by adding new classes and functionality to its structure and the basics of all future development, the focus in this project will be architectural driven.

Moreover, the nature of this product is conceptual and abstract combined with complex technologies, which calls for a solid and understandable architecture that explicitly defines structures, dependencies and interfaces, and which is not degraded over time due to the implementation of new functionality.

2.3.2 The Spiral Model

The project plan for this project aims at following the Spiral Model which is implemented in the RUP. The Spiral Model is a process model developed to address the problems discussed in the Waterfall Model in the earlier stages of the software life cycle. This is achieved by iterating through stages in a spiral where every stage is repeated once every roundtrip.

In the first stage, the objective, alternatives and constraints are defined in the requirements set for the project. The second stage handles the analysis where the alternatives and constraints are evaluated in a design before proceeding. The third stage is the implementation phase where the design is transformed into a practical solution. The fourth and last stage is the test phase where the current progress is evaluated in order to be able to plan the next roundtrip.

This way of performing the same stages again and again provides flexibility to the process where errors encountered during the implementation can be corrected in the requirements and analysis. The only real problem is that it does not specify an end stage. This implies that the process is infinite and that the requirements must specify a goal where the solution covers enough functionality.

In this project, one of the two goals is specified in the requirements and rate functionality. I have chosen to rate the functionality is in four categories. Out of these four categories only two are planned for implementation. All new features will be categorised in the latter non-implemented categories; hence the project scope will not grow.

2.3.3 Phases and Milestones

In large, co-operating environments every iteration or cycle in the spiral model must be defined as a kind of milestone. If not, the control of the process will be lost. In this case however, I have chosen not to specify any cycles. The only reason why I can do this is that I am the only one involved in this project.

Instead, I have chosen to take the numbers of cycles needed to meet the requirements. This means that every time I have encountered a problem that the analysis or implementation could not handle, I finished the cycle and took another one.

Still, this way of controlling the process does not mean that there are no milestones or deliverables. These are defined and must be kept. The milestones are arranged in the following table.

Activity Name	Duration	Start Date	Finish Date
Interfaces	10 days	19 / 05 / 2003	30 / 05 / 2003
Functional Test Procedures Ready	15 days	02 / 06 / 2003	20 / 06 / 2003
Full Class Implementation	5 days	23 / 06 / 2003	27 / 07 / 2003
Access to a Data Source	5 days	30 / 06 / 2003	04 / 07 / 2003
First Live Data Object	10 days	07 / 07 / 2003	18 / 07 / 2003
Framework Implementation Test	5 days	21 / 07 / 2003	25 / 07 / 2003
Demo Application Implemented	5 days	28 / 07 / 2003	01 / 08 / 2003
Total Implementation	55 days	06 / 01 / 2003	01 / 08 / 2003

2.4 Process Workflow

2.4.1 Project Phases

As described earlier, the RUP defines four overall phases in a spiral software process. The Spiral Model describes how focus is shifted in the course of the different phases of a project. These four phases is defined in the RUP model as Interception, Elaboration, Construction and Transition. Together, they describe how the entire product progresses through time.

In the first phase, *Interception*, the goal is to define the true objectives of the project and capture the requirements in a structured way. In this phase, I considered various ideas and possible architectures and evaluated them against each other. The schedule of the milestones and deliverables was rather diffuse in this phase. However, at the end of the interception phase they started to clear up. In short this phase constitutes the setting up of the overall requirements from where the rough sketches of the architecture are drawn. The project objective was defined and a preliminary project plan was sketched.

In the second phase called *Elaboration*, the understanding of the problem at hand was established. This resulted in a refined architectural plan and a more detailed and specific project plan. Furthermore, since the architecture was evaluated more risk areas were eliminated as the technical barriers became known and most of them were overcome. The analysis resulted in a preliminary design which started the actual implementation in the first prototype.

The third phase, *Construction*, is not that different to the second phase. Still, the architecture and design was improved, but also stabilising in their form. Parallel to this, the implementation of the prototype grew from a simple solution to an advanced one. The behaviour of the prototype was tested through the functional test plan and by the end of the phase; test users could actually use the solution with some reservations.

In the last phase, *Transition*, feedback from the test users resulted in minor changes in the feature list, and the more advanced features were implemented in the prototype. The test users received a new version of the prototype almost every day with corrected errors and new features. In this phase, a test of the prototype was conducted, and the product was evaluated against the objective and requirements. This resulted in the overall thesis conclusion.

2.4.2 Project Deliverables

Like the milestones, delivery dates are specified to ensure that the process only takes the time specified before the project was opened. The project will result in the delivery of three parts: The thesis report, the software product and a demo application. The implementation is specified in the milestones set in section 2.3.4, Phases and Milestones, and the delivery of the thesis must follow this plan. This resulted in the delivery dates in the table below.

Deliverables	Action	Responsible	Date
Conceptual Modelling	The objectives of the project must be formulated in a conceptual idea. This conceptual idea will form the starting point from which the project evolves	UFD	31-01-2003
Review of Conceptual Modelling	MBD and DTU review and approve of the conceptual idea.	BJP, JAS	14-02-2003
Project Plan	In this project plan, a number of milestones are set. These must be defined along with the methodology and guidelines which the process must follow.	UFD, JAS	28-02-2003
Review of Project Plan	An evaluation of the proposed milestones and project structure.	BJP	14-03-2003
Requirement Definition	From the conceptual idea, the requirements are defined in a document. These requirements must specify the necessary features and the scope that delimits this project.	UFD, JAS	28-03-2002
Review of Requirements	User Profiles and Stakeholders as defined in the project review the requirements	BJP, JAS	11-04-2003

Deliverables	Action	Responsible	Date
Analysis & Design	The analysis of the problem and a design proposal must be completed.	UFD	30-05-2003
Review of A&D	A review of the analysis and design proposal.	JAS, HSN, BJP	13-06-2003
Implementation	The implementation of the design with the required functionality must be completed.	UFD	27-06-2003
Code review	This phase must run parallel with the implementation; hence the same delivery date.	UFD	27-06-2003
Test	Test of the implementation must be completed and the last errors found.	UFD, HSN, NIG	18-07-2003
Delivery	Full delivery of the project with implementation and documentation.	UFD	15-08-2003

2.5 Summary

MAN B&W Diesel A/S has chosen the Rational Unified Process as the primary tool for handling and describing software development processes. It is an implementation of the iterative approach prescribed by the Spiral Model. In this project, I have used an adaptation of this process framework described in a project process guide made by the IT department at MAN B&W Diesel A/S.

When carrying out a development project, a strict process model is important – even when the project is conducted by only one person as is the case here. In order for the project to be completed on time and to fulfil the requirements set out, milestones and deliveries must be kept. These milestones and delivery dates are defined in this chapter.

CHAPTER 3

Requirement Specification

3 REQUIREMENT SPECIFICATIONS ... 3

3.1 PROCESSING REQUIREMENTS ... 3
3.2 INTENDED AUDIENCE AND READING GUIDE ... 4
3.3 FUNCTIONAL REQUIREMENTS ... 5
3.3.1 Encapsulation of Data in an Object-Model ... 5
3.3.2 Runtime Environment ... 5
3.3.3 Extending the .NET Platform ... 6
3.3.4 Support of Multiple Persistent Data Sources ... 6
3.3.4.1 Flat Files ... 7
3.3.4.2 Spreadsheets ... 7
3.3.4.3 Relational Databases ... 8
3.3.4.4 Object-Oriented Databases ... 8
3.3.4.5 Encapsulation of Data in Application API's ... 8
3.3.4.5.1 SAP R3 ... 9
3.3.4.5.2 TeamCenter Engineering ... 9
3.3.4.6 Semistructured Data ... 9
3.3.5 Queries and Selections ... 10
3.3.6 Transactional Support ... 10
3.3.7 Fully Configurable Setup ... 10
3.3.8 Graphical Utility ... 10
3.3.9 Rating the Requirements ... 11
3.4 NON-FUNCTIONAL REQUIREMENTS ... 16
3.4.1 Usability and Rapid Development ... 16
3.4.2 Reliability and Supportability ... 17
3.4.3 Resilience towards Extensions ... 17
3.4.4 Capacity ... 18
3.4.5 Data Reuse ... 18
3.5 ROLES ... 19
3.6 REQUIRED DOCUMENTATION ... 20
3.7 ACCEPTANCE TEST AND CRITERIA ... 21
3.8 SUMMARY ... 22

3 Requirement Specifications

3.1 Processing Requirements

Before addressing the development of a system, you must define what the system is all about. Who needs it? What is needed? And how are you planning to use it? To put it in the words of Doug Rosenberg [6, pp. 122, line 13]:

> *Simply stated, a* ***requirement*** *is a user-specified criterion that a system must satisfy*

Now, this quote contains three unknown quantities – users, criterions and the system. These quantities must be defined in a requirement specification, though, not necessarily in that order.

A criterion is not just a criterion. Criterias can be divided in different kinds. In this thesis, we shall differ between functional, non-functional and test requirements. The join of the two sets of requirements called functional and non-functional obviously gives a full set. While the functional requirements are strict and precise demands for functionality, the non-functional describes the ideas behind the required solution. Ideas like performance or usability.

In this chapter, we will conduct a thorough investigation of the demands for a data access framework. This investigation will result in a prototype where the most significant and important requirements are implemented in a working solution. On the basis of this solution, the MBD organisation must decide whether to continue the development cycle for this solution in order to improve functionality and implement new features, or to scan the market for already built products. In the both cases, this chapter specifies the requirements for such a product.

3.2 Intended Audience and Reading Guide

This paper describes a development process from setting up requirements to finalising a software prototype. It will include discussions about the chosen strategies and technologies used. It is assumed that the reader has basic knowledge about software development processes; still the thesis includes a description of the process strategy used in chapter two.

The project is based on the newly released technology platform of .NET, the reader will get an explanation of concepts in chapter four. However, a little knowledge about the concepts of object-oriented programming and storing data in data source is required to get full benefit of the text.

Chapter five, six and seven deal with the full analysis, design, implementation and test and are of a technical nature. In order to fully understand these chapters the reader must have knowledge about UML and software architecture. Since they deal with the core content of this thesis, these chapters are of a technical nature. Non-technical readers can skip these chapters and proceed straight to the implementation and conclusion.

3.3 Functional Requirements

Functional requirements capture the intended behaviour of a system. This behaviour is expressed in a physical form as functionality, tasks or perhaps services that the system is required to execute in operation.

When specifying a system of a certain size, it is useful to rate requirements. A rating will enable developers to focus on base functionality before moving on with features that will improve the basic concept of the system. Such a rating will also help differentiate the various product versions in a product line if needed for marketing reasons.

In order to get a release quickly it is necessary first to concentrate efforts on core functionality. This will result in a stripped down version that only contains basic functionality, and additional features can then be added in future releases. The tricky part is to distinguish between basic and future functionality, but once a decision is made a quick release will provide future development with valuable feedback from users with hands-on experience.

3.3.1 Encapsulation of Data in an Object-Model

The fundamental concept of this project is to construct an object-oriented abstraction of data located in different data sources. Since these data sources do not implement an object-oriented architecture, this framework must encapsulate data in objects. This will enable developers to work with object-oriented analysis and design, without having to build a bridge between programmatic ideas for each new project.

Therefore it is required, that all data structures and storage behaviour are constructed in a uniform way. No requirements are given to the number or type of behavioural methods needed, since it depends entirely on the analysis and chosen analysis model. But objects must somehow reflect the value name definitions of the data stored in the data sources like FirstName, LastName and Age, instead of renaming values like value1, value2 …value-N.

3.3.2 Runtime Environment

On both the client and the server side, the MBD organisation has chosen to concentrate all efforts on the Microsoft platform. Given this political choice, all client computers in the organisation run the Window operating system and so must the proposed solution. On the

server side, most of the servers run the Windows Server operating systems. In this case however, the server side can be regarded as a homogenous Windows environment due to the fact that non-Windows machines handle the task of database server and not application server. This gives us a requirement for the solution to run on the Windows platform on the server side, but with the possibility to access several platform database servers.

3.3.3 Extending the .NET Platform

As described in the introduction, the chosen platform for future development of applications in MBD is the new .NET platform from Microsoft. This makes it an absolute prerequisite for the proposed solution to be based and implemented on the .NET platform.

Furthermore, it has to be fully integrated with the .NET framework and function as an extension of its original functionality. A simple add-on is not enough. It is imperative that the solution works transparently in the framework and the enclosed development package to give application developers full advantage of the solution.

To achieve transparency, all introduced types must extend the standard types in the .NET framework and be fully compatible with the WinUI and WebUI controls. This includes adding new project specific data containers to UI controls like any data containers implemented in the .NET platform.

3.3.4 Support of Multiple Persistent Data Sources

A persistent data source is a data storage that holds data infinitely until a request for deletion is specifically made. There is no data loss due to power failure, exposure to light or other suddenly occurring incidents. It implements the basic CRUD (Create, Read, Update and Delete) functionality, hence giving access to the data stored in the persistent data source.

In MBD, a broad variety of persistent data sources is used. Some are up-to-date with the technical level of the industry and some are well over the prime of their life. Due to the application infrastructure all data cannot simply be migrated to a new type of data storage. It will take a lot of effort and resources to rewrite some of the old applications that are still in use. Still data must be accessible by the newly developed applications.

This requires the framework to support a full spectrum of data sources from simple files and spreadsheets to semi-structured data like XML and relational databases. Furthermore, an encapsulation of some API giving access to application data is needed.

3.3.4.1 Flat Files

Support for flat files containing simple lines of data must be implemented in the framework. These simple data sources are mostly used to data exchange between systems where a direct data link for some reason cannot be implemented.

The flat files must support both character-separated and fixed-length data fields. The most commonly used type of character to separate data fields in a flat file is a semi-colon. This type of file called a .cvs file is the lowest common denominator for data exchange between spreadsheets.

Fixed-length data fields are the common format used for export of data from the mainframe. This data structure reflects how data physically are stored on disks inside the mainframe system. Therefore it is easy to export data in this format. In this format, data can be added to a flat file by sequentially copying the raw content of records from the mainframe.

3.3.4.2 Spreadsheets

Since the Microsoft platform has been chosen as a basis for the entire organisation, the spreadsheet used is Excel in various versions. Support for Excel-spreadsheets is only loosely required. It is a nice-to-have, since all spreadsheets can save documents as comma-separated files, already supported by the framework.

To make it easier for people to maintain these data in spreadsheets, and to stop the constant conversion of documents between different file formats, a support for Excel-spreadsheets is required.

3.3.4.3 Relational Databases

The most commonly used and important platform for storing data is undoubtedly Microsoft SQL Server. This is used for all new software projects to store stores business critical data. It is therefore required that this data source is supported.

As an alternative to the MS-SQL data platform, the organisation has begun to look at MySQL running on cheap Linux servers. This database is rich on functionality and costs only a fraction of the MS-SQL. It has its limitations e.g. it does not support stored procedures, but it is fast, and with an object layer on top, the stored procedures are obsolete anyway. Support for this data source is required.

Some large systems are in the process of being implemented in the organisation. These systems use the Oracle database as data storage and runs on the Sun Solaris operating system. It is required that the proposed solution supports this data source. However, as the database contains data structures that are integrated with the above systems, it has to be possible to lock the data source as read-only to prevent corruption of data.

It is worth mentioning that the Sybase data storage system is still used in running applications. It has been decided to out-phase this system of the organisation and migrate data to other platforms. Therefore, it is not a requirement that Sybase is supported.

3.3.4.4 Object-Oriented Databases

No database of this type has yet been implemented in the organisation. But the object-oriented database should be kept in mind when designing the architectural structure of the framework in order to extend the solution with support for this kind of data source.

3.3.4.5 Encapsulation of Data in Application API's

Today there are two major applications containing business-relevant data. These data can be accessed by the developing team either by wrapping application API's into the solution or by enabling the solution to access the data storage directly.

3.3.4.5.1 SAP R3

In the financial area, SAP R/3 is being implemented. This system contains economical data like orders, invoices, customer and contact information. These data needs to be accessible from other business applications. Due to the licensing politic of SAP, it requires a license per employee to allow so-called “anonymous access” to the data-layer. Therefore a data-extraction project will synchronise data from SAP R/3 with a data schema implemented on an MS-SQL Server.

3.3.4.5.2 TeamCenter Engineering

Product data is handled by a PLM system for handling product data called TeamCenter Engineering, previously iMAN, which has been developed by EDS. The system has an API to extract data, but it would be preferable to have this system integrated in this framework to achieve an identical interface for data across all data platforms.

3.3.4.6 Semistructured Data

Today and in the future, MBD expects the exchange of data between platforms and applications to be based on XML – a semistructured data format. In particular, the exchange of order information, product specifications and even drawings will be exported, sent and received in this format.

In order to be able to built applications that can fully benefit from this form of data exchange, we wish this framework to implement support for XML documents. Since data stored in XML format is not a part of the general data models in MBD but are used solely for data exchange, this requirement is non-critical.

In fact, it might be considered that XML documents are not a part of the persistent layer since it is primarily intended for data exchange. However, as the format is easy to use when designing data schemas for implementation in small systems put together and as it will be a key exchange format, support for this data format is required.

3.3.5 Queries and Selections

In order to benefit from an object layered persistent data source, the ability to request certain data is essential. There is no need to store data for later usage if it cannot be found at will. This calls for a feature that enables selection of data through queries into data storages.

Essentially, queries can be differentiated in simple and advanced queries. Simple queries selects a set of data based on an simple index, while advanced queries are based on various conditions and even combine data from different parts of a data schema.

3.3.6 Transactional Support

Since the data access solution will be implemented in all future applications in the organisation, it will be used as a parallel tasking and multi-user solution. In order to be able to handle this type of requests without leaving the data inconsistent because of hardware or software failure, the transaction concept must be supported by the solution.

For application testing and prototype purposes, transaction managing is not a critical requirement, but a transactional management must be implemented in the final solution. It will therefore have to be considered in the architecture and design.

3.3.7 Fully Configurable Setup

Deployment and management of data sources are handled by a different department than the one who develops applications. This implies that a re-configuration of application connections to data sources is needed during operation. Therefore, data source setup must be configurable through a setup file.

3.3.8 Graphical Utility

Setup for the framework to connect to data sources must have a GUI-oriented utility. This is important for the availability of data to the application developers. The utility must enable the developers to automatically retrieve data schemas from data sources to initialise the access layer with the required meta-data or likewise. Since data schemas can be reflected manually in the configuration manner that the solution requires, this utility is not critical for the system.

But in order to use the solution on large, existing data sources, it is required to exist. If not, too many errors will be introduced when trying to setup data schemas.

3.3.9 Rating the Requirements

After listing the requirements in words, a summary is presented in the table below. In the table, requirements are given id's and explained in brief along with a risk evaluation and a requirement rating.

The risk defines the importance of a requirement to the whole solution and the critical key issues. This establishes a general view of how to prioritise development resources and may even cause the solution to be redefined due to impossible barriers.

The requirement ratings set the importance of the requirement and hereby define the order in which the features must be implemented. The ratings are as follows:

1. Key requirements
2. General requirements
3. Non-critical requirements
4. Nice-to-have requirements

Id	Requirement	Details	Risks	Rating
1	Create framework skeleton	This project is centred on the implementation of an object framework. An implementation of this object-structure is required before any requirements can be fulfilled.	High Since it is the foundation of the project, an architectural mismatch will cause it all to fail.	1

Id	Requirement	Details	Risks	Rating
2	Integration to standard .NET types	Integration with .NET type is essential for the demonstration of the framework capabilities. This is what distinguishes a usable solution from a simple, non-usable implementation.	Low An easy requirement, since all data types can be extended.	1
3	Support for different data sources	Support for different data sources must be rated to set an order of implementation.	Medium Due to the nature of this product being a prototype, it is not imperative for all types of data sources to be implemented.	
3.1		- Flat files		3
3.2		- Relational databases		1
3.3		- Object-oriented databases		4
3.4		- Semistructured data		3
3.5		- Data behind application API		3
4	Flat files	The different flat file types are rated here:	Low Old and known technology that supports a limited behaviour.	
4.1		- Character separated fields		3
4.2		- Fixed length fields		3
4.3		- Spreadsheets		4

Id	Requirement	Details	Risks	Rating
5	Relational databases	There are different types of relational databases running in the organisation. Support for these types is rated in accordance with the requirements.	Medium Known technology, but it supports an advanced query language that differs between the various types. Still it has to be wrapped in a similar behavioural encapsulation.	
5.1		- Microsoft SQL Server		1
5.2		- MySQL		1
5.3		- MS Access		2
5.4		- Oracle		2
5.5		- Sybase		4
6	Object-oriented databases	There are no requirements to actually implement support for a certain type of object-oriented database. Therefore this requirement cannot be rated	Low None exists in the organisation and the requirement is only nice-to-have. No real risk.	None
7	Encapsulation of data in application API's	The two types of applications specified in the requirements do not have to be implemented in this prototype. There are still cost issues which determine how a connection has to be made.	Medium There is a risk that it is impossible to wrap the application data access behaviour into the same behaviour as other data sources.	
7.1		- TeamCenter Engineering		3
7.2		- SAP		4

Id	Requirement	Details	Risks	Rating
8	Semistructured data	Data represented in a semistructured format like XML will only be used when exchanging data between systems. Since no data schemas today are implemented in this format, this requirement is non-critical	Low The requirement is non-critical, and a worst case scenario would specify the use of DOM or SAX as framework for this kind of data. Still, with XPath, XML-QL queries and ADO.NET support for XML it may very well be possible to implement.	3
9 9.1 9.2	Queries and selections	The ability to select data through queries is essential for the usability of persistent data storage. Still, the requirement of implementing queries will be split in two with different ratings. - Simple selections - Advances selections	Low/Medium Simple selections can be implemented on all types of data sources. But there is a risk that simple data sources cannot implement the advanced due to their nature as simple.	 1 2
10	Transactional support	Support for transaction is essential for heavy use with multiple clients in order to maintain data integrity. But it will only be rated 3 in this requirement, since it is considered an add-on feature to the basic framework.	High There is a chance that the data object structure will not support the execution of a list of stored transactions.	3

Id	Requirement	Details	Risks	Rating
11	Full configurable setup	In order to be able to switch between data sources without recompiling source code, the configuration file must be implemented from the beginning.	None This is a low risk area, since the technology of handling files is well-known and used in many cases prior to this project.	1
12	Graphical utility	A graphical utility that guides you through the setup makes it easy to handle the framework configuration. The setup of data schema meta-data can be handled manually. But without the ability to debug data schemas, it is almost impossible not to implement errors as well without an automation tool.	None This is a low risk area, since the framework will function with or without this utility	2

3.4 Non-Functional Requirements

A non-functional requirement is a soft requirement opposite to the functional requirements. By soft I mean, that the requirements do not address the tasks which the system must perform. It does not consider the input/output. Rather, it describes the look and feel of the system in relation to other systems and users.

These non-functional requirements can be just as important as the functional ones. Let's take an example. The functional requirements describe a car as a box with four wheels, a steering device, doors and windows, an engine and an engine control board. This sounds like a car but in this example no one bothered to specify usability, so the steering device is in the opposite end of the box as the engine control. And the car breaks down after 100 kilometres due to the lack of reliability. It cannot be repaired since no requirements were made to supportability.

The car is still a car and has all the functionality and features of a normal car. But the basic idea of the car as a usable and reliable instrument to be used in the everyday life of an average family would only have been described in the non-functional requirements.

Since these requirements tend to be weaker than the functional requirements, it is necessary to connect these requirements to the acceptance test. Here it is possible to specify a test-environment in which a number of specific tests must be executed.

For a pre-release of a system, some of the non-functional requirements can be overlooked. It is possible to build a system of black boxes that later on can be optimised for speed, reliability and capacity. However, these demands can also be deeply founded in the system architecture, which can make it almost impossible to correct the errors later without redesigning the whole system.

3.4.1 Usability and Rapid Development

The objective of rapid application development is to quickly build high quality programs with low maintenance costs. In order to meet the requirement of rapid development without introducing implementation errors, usability is a key feature. As such, the two intentions of rapid development and usability are interrelated.

Developers must find the solution very intuitive if it is to support rapid application development. An intuitive approach will also provide a fast learning curve and make it possible for developers to work with the solution without any or only a short introduction. This enables MBD to increase productivity and achieve more readable program source code.

With a readable and easy-to-use rapid application development framework, a cut down in development time when implementing new systems is the desired effect. In fact it is a key criterion of success, since this is what will make the framework worth developing when measured in a cost/benefit analysis. A faster replacement of the old application is also an important objective. When reuse of a persistent data source is possible, and an interface to the layer is at hand by the click of a button, only the GUI and business rules need to be converted.

3.4.2 Reliability and Supportability

The framework must be a reliable platform to perform the task as a persistent layer and at the same time support all newly developed data accessing applications in MBD. This must be achieved by a careful analysis of the key problem, a good design based on the previous analysis and a thorough testing of the solution.

When developed, the framework must be constantly expanded and re-tested while demands grow. Then, when introduced, the framework sets out to play a vital role in the software developing departments and consequently also in the eyes of the users. If an error is introduced, potentially all applications will experience faults. The result will distrust of the solution.

3.4.3 Resilience towards Extensions

Future developers in the organisation must be able to extend the framework without having to worry about the effect of the change in the rest of the system. Since it must be possible to implement new functionality in the system, the interfaces towards the consumer or application developer must be clearly defined. It is difficult to test for this requirement but nevertheless it has to be taken into consideration when defining the architecture.

3.4.4 Capacity

The hardware requirements follow the standard requirements set by Microsoft to run applications on the .NET framework. According to MSDN, the framework runs on most Windows operative systems; hence it has the requirements needed to run on one of these systems.

Supports all of the .NET Framework except Microsoft ASP.NET	Supports the entire .NET Framework
Windows 98	Windows 2000 (all versions - no Service Packs required)
Windows 98 SE	Windows XP Professional
Windows ME	
Windows NT 4.0 (all versions - Service Pack 6a required)	
Windows XP Home Edition	

(Link: MSDN)

All data sources have different requirements towards hardware scaling. Due to unknown parameters such as data complexity, size and usage, no specific requirements can be made about data servers or network capacity. In this project it can be assumed that data sources are sufficiently scaled.

3.4.5 Data Reuse

It is required that the framework supports the data sources already in operation in the organisation. It is neither possible nor feasible to migrate all data from today's data sources to new ones. Such a task will require that all data applications are rewritten and all data transferred at once which requires too many resources in only a short period of time. The decision made by management is that a full migration is a non-tenable solution

As such, the proposed solution does not have to require a non-redundancy in entities offered by data sources. It must be able to handle the fact that two or more entities with the same name and structure can be present in different data sources. The application developer and database designer have to make sure that redundant data is eliminated. But the framework must help developers get a broader view of things to prevent the introduction of multiple, redundant entities in more data schemas. This must be achieved without having to browse through large documentation and multiple shelves of books and binders.

3.5 Roles

The user roles described in the introduction are listed below. These reflect the different tasks that need to be handled during the processes of this project. The same person may perform several roles.

- Requirement Specifier
- System Architect
- Application Developer
- Database Designer
- Quality Assurance Tester
- Support Function
- Deployment and Maintenance

3.6 Required Documentation

Some documentation is required to this solution and the process of development. As described in the Introduction, this project follows a plan developed by JAS from MBD that has derived from RUP specifying a number of document parts. This report will follow this template and will thus constitute the documentation for the progress of the project.

Furthermore, an API documentation for the solution is required. A description of classes, methods and properties is required for the application developer to be able to use the framework. Also, in order to enable the role of System Architects to expand the functionality in the future, documentation about internal variables and methods must be documented. In the environment provided for .NET development by Microsoft, source code comments can be compiled into an API documentation document. This or a similar type of documentation is required.

3.7 Acceptance Test and Criteria

In the case of this project it is rather difficult to define an acceptance test and criteria since development is done in-house. The objective must be to achieve a good design on which to build the functionality needed. There is no reason for requiring a lot of functionality if the price is a poor design.

The acceptance criteria are to implement all the "Key Requirement" and "General Requirements" on top of a durable design. Afterwards, the design will be evaluated in MBD before continuing the project with the implementation of the remaining requirements.

Since this project do not deal with an application of any kind, the only GUI-part of the project is rated as 3 or a "non-critical requirement and therefore it will not be implemented at this state. With no application running, no ordinary user test can be performed. Application Developers will conduct an evaluation of the API.

To prove the functionality, two different kinds of application test must be performed. The first application performs functional testing during development and has a data schema and application source code designed for this purpose only.

The other application test implements a legacy data source in the organisation and substitutes existing application behaviour. As demo application we will use the DAISY application from the Intranet. DAISY is an abbreviation for Diesel Address Index SYstem and functions as a web address index. This will show whether the solution can fulfil the role as the only type of data access from every application in the organisation.

3.8 Summary

The chapter includes a thorough requirement study which describes the functionality needed in an object-oriented common data access system. The requirements are rated according to their importance to the project.

The requirements for a data access framework in the software development department are gathered and rated according to their importance to the project. This requirement study can be used as a basis for choosing a future strategy for data source implementation in the software development organisation.

CHAPTER 4

Technologies

4 TECHNOLOGIES ... 3
4.1 THE .NET PLATFORM ... 3
4.1.1 What is .NET? ... 3
4.1.2 The .NET Framework ... 3
4.2 .NET COMPONENTS ... 5
4.2.1 COM and End of DLL-Hell ... 5
4.2.2 Assemblies ... 5
4.2.3 Namespaces ... 7
4.2.4 Error Handling ... 7
4.2.5 ADO.NET ... 7
4.3 ENVIRONMENT ... 9
4.3.1 Runtime Environment ... 9
4.3.2 Development Tools ... 10
4.3.3 Enterprise Template-Based Projects ... 11
4.4 DATA SOURCES ... 12
4.4.1 Flat Files ... 12
4.4.2 Relational Databases ... 13
4.4.3 Object-Oriented Databases ... 13
4.4.4 Semistructured Data ... 13
4.4.5 Data Behind Application API's ... 14
4.5 DESIGN PATTERNS ... 15
4.5.1 Creational Patterns ... 15
4.5.1.1 Abstract Factory ... 15
4.5.1.2 Singleton ... 15
4.5.2 Structural Patterns ... 16
4.5.2.1 Façade ... 16
4.5.3 Behavioural Patterns ... 16
4.5.3.1 Iterator ... 16
4.5.3.2 Mediator ... 16
4.5.3.3 Strategy ... 17
4.6 SUMMARY ... 18

4 Technologies

4.1 The .NET Platform

For a long time, .NET was only a concept introduced by Microsoft. Many people had an opinion about it but only a few really knew something about it. When asking what .NET was, you would receive a wide range of answers, only few of them accurate and all of them conflicting with each other. Today, the common claim made about .NET is that it is a Java rip-off but even though there are some resemblances the truth is somewhat different.

4.1.1 What is .NET?

.NET is a software platform. It is a language-neutral environment for writing programs and software components that can operate and communicate in an easy and secure way. Rather that targeting a particular combination of hardware and operative system, programs will instead target .NET, and will run wherever .NET is implemented.

But .NET is also the name given to various parts of software built on the .NET platform. Together they form a collection of products like Visual Studio .NET and services like Passport which is built on top of the platform. The components that make up the .NET platform are collectively called the .NET Framework.

4.1.2 The .NET Framework

The .NET Framework has two main parts namely the Common Language Runtime (CLR) and a hierarchical set of class libraries. The CLR is best described as the execution engine of .NET. It provides programs with a runtime environment. The most important features of the platform are:

- New low-level assembler-style language, called Intermediate Language (IL).
- Memory management, including garbage collection.
- Checking and enforcing security restrictions on the running code.
- Loading and executing programs, with version control.

Before I can go into details, a few terms that will be introduced in this chapter have to be defined:

Managed / Unmanaged Code Managed Code is source code that targets .NET and which contains self-describing metadata. While both managed and unmanaged code can run in the runtime environment, only managed code contains the information that allows the runtime environment to guarantee safe execution and interoperability between programs and components.

Managed Data: The CLR provides the environment with safe memory management facilities greatly helped by the introduction of a garbage collector. Languages like C#, VB.NET, JScript.NET use Managed Data by default. Others like C++ do not. The CLR also constrain the features available to a language, for instance C++ looses multiple inheritance.

In order to keep a common type-foundation, the CLR uses the Common Type System (CTS). This type-foundation ensures that all classes across all languages are compatible with each other. This compatibility is achieved by describing types in a common way which also includes error handling through exceptions. Through the CTS the runtime environment can ensure that source code does not attempt to access memory that is not allocated.

All .NET source code is compiled into IL code instead of native assembly code. During execution in the runtime environment the CLR uses Just-In-Time (JIT) compilation. Every time a method are called within an executable, it gets compiled into native assembly code. Since the compilation only happens the first time a method is called, the JIT compiler only performs its task once. After the JIT compilation, the compiled source code is stored in a cache.

4.2 .NET Components

Most people who have developed applications for the Windows operative system have experienced how difficult it has been to control the different versions of a COM-component. Sometimes the registration of one component that would make one application work causes other applications to fail. The problem occurs when more dynamic link library files, also called DLL-files, with the same name are installed on the same machine without being fully compatible.

4.2.1 COM and End of DLL-Hell

Originally, it was the idea that one component could support multiple applications and that deploying an update simply was to overwrite one old file with a new one with the same interface and behaviour after errors had been fixed. Now think of a situation where the installed applications knew the existence of a multiplication error in a shared component and therefore compensated for it in their layer. When an updated version of the component that fixed this error was installed, the applications would perform wrong calculations. This situation is known amongst windows developers and administrators as the DLL-hell.

Since this mismatch was of a structural nature where versioning was not in the design, it required a whole new infrastructure in the operative system to correct the problem. From the time where every new component with the same name had to be 100% backwards compatible, and at the most introduce a few new methods and features, the new infrastructure needed to implement support for more applications using different versions of the same component.

To solve these problems, Microsoft first had to solve the problem of every component having to register itself in a central registration database as this makes maintenance almost impossible due to lack of overview. This could be solved with a decentralised strategy where every application describes which component and component version to use.

4.2.2 Assemblies

The infrastructural solution to these problems is the .NET platform. This platform has introduced a new type of library file, named DLL like the previous one. On the inside, however, it has nothing to do with the old COM components. The new library file is called an assembly which is self-describing towards the system. The description lies inside the

assembly as metadata and is called the manifest. The manifest contains information about the identity, references, exported types and the attached resources. In fact one assembly can span over multiple files. This is a design feature aimed for WAN usage where only the needed parts of an assembly are transferred through the network; hence saving bandwidth.

Assemblies can be installed in two different ways: As private and shared. The private assembly applies only to one application and is isolated from the rest of the machine on which it is deployed. Since it is only visible to one application it is physically located in the same folder structure as the application. When using private assemblies there are no constraints on the naming, other than it has to be a unique name in the application. Even though version numbering and other identities are stored in the manifesto, the knowledge is never used. The developer knows the program and has full control over the files located in its folder structure.

The shared assembly can be used by multiple applications throughout the machine. The .NET framework is an example of a set of shared assemblies where all .NET applications share the same installation. The shared assemblies are typically stored in a container called GAC for Global Assembly Store, but can be placed anywhere on the machine. Any user with administrator rights can install the assembly on the machine by simply copying the file into the GAC.

Compared to the private assemblies, the shares must have a unique name to the whole machine. Furthermore, since they are shared the GAC must have some kind of assurance that the assembly is what is appears to be, and not a hacked version implementing a backdoor or virus into the system. This is achieved by the concept of strong naming based on an encrypted set of keys that identifies assemblies to applications. The compiler signs the assemblies using this key, which is afterwards used by the GAC to verify that its content is not changed.

Versioning of assemblies are based on a version number. This number consists of four sets of digits separated by periods; major, minor, built and revision. Assemblies with the same major and minor number will normally be regarded as compatible since the two latter numbers are used to identify small fixes. The .NET framework will automatically call for the file with the highest of these numbers. It is now possible to set up an application to use only one or more specific versions of an assembly

4.2.3 Namespaces

In general, a namespace is a unique identifier that organises a set of names in an unambiguous way. In .NET a namespace is a feature connected to the assemblies. It is a simple, logical grouping of related interfaces, classes, structures and even other namespaces. It enables the assemblies to be organised in a semi-structural way like files in folders on a file storage system. The .NET framework is located in the namespace *System.** that contains all functionality.

4.2.4 Error Handling

One of the major problems with the COM architecture was that the environment did not implement a standard mechanism to raise and handle errors. The most commonly used method was to evaluate return values like *hresult*, boolean values or the Visual Basic *Err* object.

The .NET framework provides the architecture to handle errors in a standard and consistent way through the concept of exceptions. An exception is an unexpected behaviour that occurs during execution of code. The exceptions can be thrown across processes and machines since they are a part of the CLR. In fact, the CLR can automatically convert an exception into an *hresult* enabling the .NET error-handling feature to extend to old COM components as well. The base class called Exception must be extended to create custom exceptions. Hereafter, they are fully integrated into the error handling architecture.

4.2.5 ADO.NET

ADO.NET is a system class structure integrated in the .NET framework to provide access to data sources. Because web services are a key objective for Microsoft, the data structure of ADO.NET is XML based to support marshalling of data objects through e.g. the SOAP protocol (Simple Object Access Model), which is a protocol for sending packets of XML.

The class structure for accessing data in data sources is based on the façade pattern. Data sources are wrapped into a uniform interface to make them seem similar. When data is retrieved, the data structures in ADO.NET are disconnected. To provide this, there are two key objects: the DataReader and DataSet.

The DataReader provides a read-only stream of data that is lightweight and fast for forward, sequential data reads. The functionality resembles the cursor pattern, but since data is streamed, the backwards traversing is not allowed – just like the SAX model for XML handling.

The DataSet object provides a disconnected representation of data from a data source. It can contain numerous tables, and their relations and constraints. In fact, it is a snapshot of a selected part of a relational database.

As data are represented in the same structure internally in the DataSet as they are in relational databases, there are still a mismatch between the object-oriented approach of the .NET programming structure and the representation of persistent data. Furthermore, accessing specific data in the DataSet requires knowledge about the data schema as a string containing the table or field name is used to references data.

4.3 Environment

Before analysing the requirements, it is important to fully define the environment in which the solution is being built. The runtime part of the environment is defined in the requirements, but an introduction to the tools and platforms used in the analysis, design and implementation workflows are also required.

4.3.1 Runtime Environment

The runtime environment is very specifically defined in the requirements. All client platforms in the MBD organisation are based on Windows 9x, 2000 and XP, and even as this thesis is being written, all the 9x and 2000 are migrated to XP as described in the requirements, chapter 3.3.2. This strategy is obviously to combine forces in the support groups to be able to concentrate the effort on only a narrow set of platforms. This strategy also provides certain advantages to the software development department.

The single-client platform strategy enables the application developers to focus on a narrow set of development tools and components. There is no need to go for the lowest common denominator to gain portability when there is only one platform to support. When the platform is the Microsoft Windows XP operative system as in this case, it is obvious to say .NET. According to Microsoft, this is the state-of-the-art development platform and framework of tomorrow.

Furthermore, the Windows operative system combined with the .NET platform forms a coherent whole that spans from operative system to business applications, all providing the possibility to enhance and integrate new, home-grown applications to the common system. If the product portfolio of Microsoft supports most needs and all other needs can be fulfilled either by developing applications on the platform or buying third part products based on the same platform, the choice is obvious. Never before in history has Microsoft proven the market wrong – it holds a powerful development and support organisation and implements new technologies like web services and security in their products.

On the server side on the other hand, the strategy is not single-platform. The main goal is to use cheap “windows-boxes” as servers, and the majority of servers are Windows 2000 Server based. But some tasks are better solved by other platforms, for instance the TeamCenter Engineering application uses an Oracle database server which runs faster on a Sun server

running the Sun Solaris platform. Furthermore, some tests on the use of Linux as a server platform is performed. In general, the server layer consists of machines running the Windows platform, but the database server layer in particular can run multiple platforms.

As for now, the choice of database platforms is:

- MS SQL Server (Windows)
- MySQL
- Oracle
- MS Access
- Sybase

I have listed the Sybase database in the list of active database platform because this platform was the chosen database of yesterday and therefore many applications are based on data in these systems. But it has been decided to stop all use of Sybase by migrating data to the MS SQL platform and future development will NOT deal with this platform in MAN B&W Diesel A/S.

4.3.2 Development Tools

Since all development is targeting the Microsoft .NET platform, the obvious choice of tool is the Microsoft Visual Studio .NET. During this project, I have used the first version of VS.NET with the version 1.0 .NET Framework. A migration to the newer release called Visual Studio .NET 2003 running the .NET Framework version 1.1 which was released during this project is planned in the MBD organisation, but not yet installed. I therefore chose to continue development on the “old” platform.

The Visual Studio .NET is a full-scale integrated environment for developing applications, databases and components for either standalone or distributed applications. It integrates source code-writing editors with the tools needed to compile, package and deploy application and services on other computers and servers. In addition, Microsoft delivers documentation both as online help as well as a full MS Developers Network (MSDN) help website.

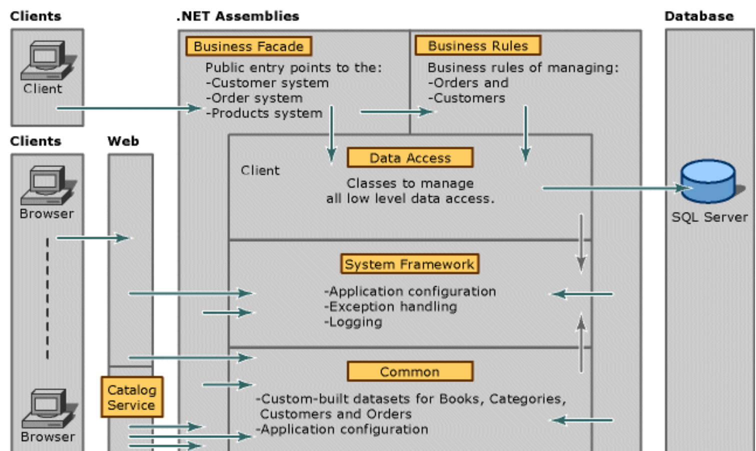
VS.NET also includes a tool called Visio that provides the necessary diagramming support for UML specifications and documentations of a VS.NET solution. Unfortunately, it does not

support reverse engineering, only one round-trip from diagram to source code or from source code to diagram; hence you cannot update the source code from your retrieved diagrams. Tools that support full-scale reverse engineering in Visual Studio, like Rational XDE, are known to me as well as other developers in MBD but are not used as an integrated tool.

4.3.3 Enterprise Template-Based Projects

The primary tool to use is the Visual Studio .NET when developing .NET applications. As mentioned, it provides developers with an integrated development environment. It also defines a set of architectural templates from which solutions can be structured. As MBD has decided to follow the .NET strategy all applications must also follow these templates.

These templates describe a layered model where applications can be distributed over six different layers. The layers are sketched on the figure below [16] and every layer can call objects and methods in other layers according to the arrows. The top layer is the presentation layer that can be either a Web or Windows GUI layer or a web service. The business layer is divided into two separate layers with the façade in front of the rules. This façade layer can hide changes in business objects by keeping an unchanged interface towards the presentation layer.



The internal parts are the Data Access, System Framework and Common layer. The Data Access layer controls access to data sources and are the only layer that can do so. The System Framework is a layer that all layers can access. It typically handles errors and configuration. The Common layer resembles the System Framework layer because all layers can access it. But it is primarily used for common type definitions and enumerations.

Even though this model "only" describes six layers, each layer can contain multiple layers internally – just like the Russian Matryoshka doll. Therefore, the architecture actually describes an n-tier layered software design.

4.4 Data Sources

In short, Data Sources are sources of data. The term covers multiple types of database services from where data can be retrieved, saved and/or deleted. Databases are not limited to cover only the computer-based types that we think of today. In fact, they derive from paper based file drawers. This can also be seen from Webster's dictionary, where the definition is:

> *Database: A usually large collection of data organized especially for rapid search and retrieval (as by a computer)*

Thus, databases have existed for a long time for organising knowledge worth keeping, probably throughout most of the known history. Today, the term is closely connected to computers, since they provide fast and reliable ways of storing, searching and retrieving data.

Even though paper based storage is commonly used even today, this kind of data source is obviously not taken into account since this thesis covers a software solution. But also legacy mainframe data sources are disregarded since this kind of data storage is being phased out, as explained in the requirements in chapter 3. In the following we will only go through the relevant types of data sources.

4.4.1 Flat Files

This kind of data source only contains unstructured and type-less data. Entities are stored in files, typically with one relation per file. Data are only indexed by line number and the format is not primarily made for data storage. It therefore requires a great deal of implementation to get data source behaviour, and queries are not efficient unless they involve a selection upon line number, only.

Old database systems also use files as storage but they also include an index system. These file-based databases implement data source behaviour on top of the indexed files but are still rather slow when querying data. Generally, this kind of data source is not used today but can still occur in older systems. In fact, even though MS Access has a relational engine on top of the data storage, it is still a file-based database. In spite of this, I will regard Access as a relational database in this project, due to its look and feel.

4.4.2 Relational Databases

Where the file-based data sources originate in something everybody have access to – a file system – the relational database is based on a mathematical concept. In order to keep data integrity, the database needs to take control over which data goes in and which comes out again. Relational databases or RDBMS (Relational DataBase Management Systems) represent a reliable data storage system with full database behaviour including advanced query functionality.

With higher control comes higher maintenance. Usability and control is always a trade-off. The more control you want over things, the more difficult they are to setup and configure. However, most databases come with administration tools that make it easier, and if you require data integrity and query speed there is no alternative to this kind of control.

4.4.3 Object-Oriented Databases

After the introduction of object-oriented programming that combines data and behaviour, the OO-concept also spread to databases. This means that inheritance and encapsulation is possible, and that objects in the programming world can easily be made persistent in data storages without mapping data to different data structures and vice versa.

The ODBMS (Object-Oriented DataBase Management Systems) are not commonly used today even though most large programs use object-oriented techniques. Therefore, data are still mapped from relational to object structures. This mapping to different data structures is called "impedance mismatch". The lack of the impedance mismatch in ODBMS' gives them a performance advantage over RDBMS', especially when working on complex data structures. Impedance mismatch obviously slows down performance on complex data because of the processing needed to map from one data structure to another.

4.4.4 Semistructured Data

Semistructured data sources have been developed as a dynamically typed data model that allows representing of data with loosely defined or irregular structures – a schema-less description format – in which the data is less constrained than usual in databases.

The XML format is a mark-up language for structured and semistructured data, which were developed by the WWW Consortium (W3C) as a standardised format in which to transmission data and their relationships in platform independent computer documents. The format can act as a mediator for accessing and distributing data across computer platforms and networks.

XML is a license free and open standard. Yet, it has achieved a strong vendor support amongst software companies. The syntax is simple, since it derives from SGML just like the HTML standard. Furthermore, the format uses metadata, which allows it to be self-describing – a feature which has been picked up by assemblies in the .NET platform.

On top of the XML standard, multiple types of query engines and languages have been developed. These query languages (e.g. XML-QL) and path expressions (e.g. XPath) can locate and select data in XML documents in the same way as SQL does it in relational databases.

4.4.5 Data Behind Application API's

Some applications hide the database used as data storage. As a replacement for direct access, they provide access to data through a business layer, which hides the original data schema. Often, they map data into business entities instead of the original data schema.

Still other applications may need to access these data. Since data access behaviour is different for each specific application, there is no need to generalise. Each case must be treated like one the four major types: File, relational, object or semistructured data sources.

4.5 Design Patterns

A design pattern is a description of a best practise. When designing software, developers depend on experience to define the best architecture and design to solve a problem. Design patterns are in fact experiences explained as a set of best practises. One of the best-known books about this subject is the "Elements of Reusable Object-Oriented Software", also known as "The Gang of Four" [14].

Based on this book, I shall here give a short introduction to the design patterns used in the design. A full description of the patterns is given whenever the pattern is used in a specific case. According to [14], there are three kinds of pattern: Creational, Structural and Behavioural patterns. Creational patterns describe how objects should be created, Structural patterns how objects should be defined and Behavioural patterns specify how objects should behave.

4.5.1 Creational Patterns

4.5.1.1 Abstract Factory

The Abstract Factory pattern describes a way to create instances of abstract classes from a matched set of related concrete subclasses. The pattern is often used when implementing different windowing systems with similar functionality, but the analogy to different databases with similar functionality is not that far off.

The subclasses in the Abstract Factory pattern all inherit the same base class but there is nothing to prevent some of the subclasses from implementing behaviour that differ from the other subclasses. However, if you want an external consumer to use an arbitrary instance of one of the subclasses, they will only know the behaviour specified in the interface e.g. the base class.

4.5.1.2 Singleton

The Singleton pattern ensures that only one instance of a class is created. All objects that use an instance of that class then use the same instance. In case of a multithreaded design, you need to implement the class with concurrent thread synchronisation to ensure that only one instance of the class is created.

4.5.2 Structural Patterns

4.5.2.1 Façade

The Façade pattern provides uniform access behaviour to a subsystem of objects. This pattern helps to simplify a complex set of object behaviour through an easy-to-use interface, hereby allowing changes in the implementation of the subsystem without requiring changes in the source code in front of the façade. This enhances the abstraction level but still it does not prevent programmers to cast objects into the deeper subclass it actually represents.

4.5.3 Behavioural Patterns

4.5.3.1 Iterator

The Iterator pattern is a commonly used pattern. It defines an interface, which declares the methods needed to access objects in a collection sequentially. This enables programmers to access data contained in lists and collections using a standard interface, and hides the details about how the data structures are actually implemented.

The implementation of the Iterator pattern is abstract so that a consumer accesses data independently of the collection class. The iterator is also known as a cursor in a data source.

4.5.3.2 Mediator

This pattern models a class whose object is responsible for controlling and coordinating the runtime interactions of a group of other objects. It helps to encapsulate behaviour in a separate mediator object like the Façade. The Mediator ensures a loose coupling between objects by keeping them from referring to each other explicitly. Objects on either side of the Mediator do not need to know about each other, they just need to know their Mediator. The Mediator pattern uses one object to coordinate state changes between other objects. It centralises the logic needed for one object to manage state changes of other objects instead of distributing the logic over all of the other objects.

4.5.3.3 Strategy

The Strategy Pattern describes how to encapsulate related algorithms into classes that are subclasses of a common super class. This allows objects to vary their selection of algorithm over time. In essence, the pattern decouples an algorithm from the consumer, and encapsulates it in a separate class. This separate class shares its façade with other implementations of other algorithms and thus makes it possible for the consumer object to switch algorithms.

4.6 Summary

In this chapter some of the technicalities surrounding this project are reviewed. Most of the subjects, e.g. the different types of data sources, are known to any reader with programmatic experience but the .NET framework – the foundation of this thesis – is relatively new to most people. This is why I have outlined the key subjects (assemblies, namespaces and Common Language Runtime) in this chapter.

The core of all future plans inside Microsoft resides on the .NET platform [15, pp.29]. This implies that all future products coming from Microsoft Corporation will be founded on .NET. As the majority of mainstream business applications are developed by Microsoft and most other applications need to integrate towards them, the .NET platform is bound to prevail.

CHAPTER 5

Analysis

5 ANALYSIS ... 3

5.1 PURPOSE ... 3
5.2 STRATEGY ANALYSIS ... 4
5.2.1 Introduction to papers ... 4
5.2.2 Encapsulating Data Access ... 5
5.2.3 Architectural Strategies ... 6
5.3 GENERIC OBJECT-MODELLING OF DATA SOURCES ... 9
5.3.1 The Basic Concepts ... 9
5.3.2 Structure of the Persistence System ... 10
5.3.3 Extending .NET Data Types ... 11
5.3.4 Object Modelling on Non-Object Data ... 12
5.3.5 Mapping ... 13
5.3.6 Queries and Selections ... 13
5.4 ARCHITECTURAL SIGNIFICANT DESIGN PACKAGES ... 15
5.4.1 One Package Design ... 15
5.4.2 Extending the Package ... 15
5.5 SUMMARY ... 17

5 Analysis

5.1 Purpose

The purpose of this chapter is through careful analysis of papers and ideas to find the strategy and the analytical tools needed to develop a robust design of a data access framework solution. Analysing the requirements and then refining and structuring them into a specification of the system will achieve this.

To do so, we must choose a design strategy for the solution. Some strategies are already given by requirements, but the "missing links" must be established with a more precise and detailed explanation of the system in a conceptual model. This project is primarily based on the work of Scott W. Ambler in the papers [2] and [11] and the proposed solution model is founded on these papers. Other papers by Ambler and others have also contributed with valuable information and ideas.

Since the analysis part of the project consists of abstractions and laying out the overall structure, it is worth noticing that some requirements are better solved in the design and implementation sections. This could be issues concerning the implementation platform, e.g. programming language, legacy systems and purchased components.

The analysis will also deal with the conceptual delivery model – the way that the product is presented as a package. This will include a definition of all elements contained in a "boxed" product and a description of how they interact.

5.2 *Strategy Analysis*

5.2.1 Introduction to papers

As stated in the beginning of this chapter, this analysis is primarily based on the conceptual work of Scott W. Ambler. On the website www.agiledata.org, Ambler has published a large number of papers centred on software development techniques and database implementation techniques.

The two primary sources of information and ideas are the papers [3] *The Fundamentals of Mapping Objects to Relational Databases* and [11] *The Design of a Robust Persistence Layer For Relational Databases*. But other papers are also worth mentioning here.

Scott W. Ambler, The Fundamentals of Mapping Objects to Relational Databases

> This paper analyses the concepts of "bringing data professionals and application developers together". It reflects on the problems of mapping objects into relational data storages and semistructured data sources like XML. It explains in details how to connect the two different worlds and where to be careful in the design.

Scott W. Ambler, The Design of a Robust Persistence Layer For Relational Databases

> A must read paper for anyone designing a robust object-oriented persistence layer. It only handles relational databases but especially the passage where he specifies a design has been very useful for this thesis.

Wolfgang Keller, Object/Relational Access Layer

> Keller does not build a solution instead he focuses on outlining the key problems of designing an object layer on top of relational data sources. According to Keller, along with every problem there is a solution proposal. This paper has its centre of attention on the analysis part.

Mark L. Fussell, Foundations of Object Relational Mapping

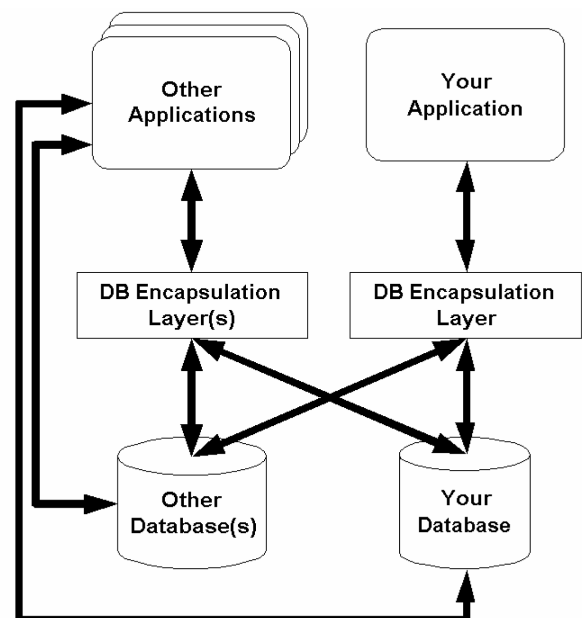
When modelling relational data onto objects or vice versa, there are a lot of conceptual and abstract definitions to keep track of. Mark L. Fussell outlines all of these concepts in this abstract paper.

5.2.2 Encapsulating Data Access

Interfaces are important factors when encapsulating functionality. The idea behind encapsulation is to hide an implementation behind a well-defined set of functions. This enables later changes in the implementation and its related components without affecting other components in the system. That is, as long as the interface to this component does not change.

This is exactly the basic idea of this project; to encapsulate the behaviour of different data sources in a new, common interface. Since we have multiple applications accessing a variety of data source types, the encapsulation layer must capture the behaviour of every single data source and hide it behind a common façade.

Scott W. Ambler has written a paper about this problem [3], where he discusses the problems related to encapsulation data access. He addresses several scenarios, like having multiple applications connecting to a single database through a single access layer. The one scenario which resembles this scenario the most is: having multiple applications accessing multiple data sources through multiple data source access layers.



Before moving on, I would like to extend Ambler's paper with the scenario of having multiple applications accessing multiple data sources through a single data source access layer, since this is the strategy that I will take. It is similar to the figure to the right [3] but the DB Encapsulation Layer(s) are now

multiple instances of the same encapsulation layer. Currently there is no implementation of a DB Encapsulation Layer in the MBD organisation, so there is no possible reuse. Therefore, there is no possibility of gaining any of the savings in developments costs and maintenance burden that Ambler proposes.

5.2.3 Architectural Strategies

Ambler lays down four different approaches or strategies from which to construct a data source access layer [3]. The four are:

- Brute force
- Data access objects
- Persistence frameworks
- Services

As these strategies have various advantages and drawbacks, neither of them are the best practise in any case. Instead, they can be used for different types of projects by helping to specify a design that fulfils the requirements set out.

Strategy: The simplest strategy is *Brute force.* The approach is that business objects access data sources directly, e.g. by submitting SQL queries to the database. This means that all the data source behaviour needed is implemented directly in the business layer. It is not a real “encapsulation strategy” since it allows accessing the data sources directly, but it is fast to implement, especially in the .NET framework, where business objects can call data sources through ADO.NET. Furthermore, there are no limitations to how you access data, so there is no development time wasted waiting on other developers to implement data source behaviour. The downside is found in the mix-up between business implementation and data access.

Advantages: Fast to implement. No limitations in the type of data access allowed.

Drawbacks: No clean cut between business and data access layer.

Strategy: A step further up the ladder is the d*ata access objects*, which is real encapsulation of data source behaviour. The strategy is to create objects that expose an interface where business behaviour is implemented on the data source. This implies a layered architecture since business logic is no longer accessing data directly. This approach is easy to implement, and with a common return type set the solution can be fully integrated into a programming framework. On the downside the data access object strategy is still rather resource consuming to maintain. As there is no common-interface strategy, developers have to be tutored and restricted by coding standards.
Advantages: Easy to implement. Supports layered architecture.
Drawbacks: Resource consuming to implement and maintain. No common interface strategy. Time consuming, tutoring needed.

Strategy: With the *persistence framework* strategy we find full encapsulation of data source behaviour from the business objects. This involves accessing data through a generic framework, where the data is mapped to data sources through the use of metadata. When business logic requests data, the database access source code is generated based on reflection from the metadata. The persistence framework can either be an implementation, where business logic uses metadata to call standard methods, or the framework can implement the metadata in a compiled form. The framework can be designed as an implicit framework, where data is saved by the framework without performing a call to a save method, or it can be designed as explicit framework where the consumer must to call the framework for read/write behaviour. This type of access layer is easy to use for the business logic developer, since all behaviour is the same regardless of the data source used or the data schema accessed, and developers can concentrate on data schema and business logic design instead of writing data access source code. On the downside, a large framework needs to be implemented before even the simplest behaviour is accessible.
Advantages: Generic design. Easily maintained metadata. Easy to use.
Drawbacks: Large infrastructure to maintain.

Strategy: The last strategy is *services*. Currently the most hyped way of accessing data is web services, but there are other architectures that act as services as well, the most known of which are CORBA, CICS, DCOM and EDI. With the use of services, there is a complete disconnection between the data access layer and business logic. Instead, data is transferred to an object by e.g. an XML document or other data transport documents from the service. The object state is then updated from the data. This process of converting an object into data and vice versa is called Marshalling and Unmarshalling. We do not know how the service retrieves or saves data behind the "service curtain". The advantage of this strategy is that one implementation is potentially platform independent and that the architecture of web services is becoming an industry standard. On the downside, application developers are likely to need help from a database developer in order to find and access the services.
Advantages: Platform independent. Architecture is an industrial standard.
Drawbacks: No standard for finding data sources and viewing their data structures.

Having taken these four strategies into consideration, I expect the persistence framework to satisfy the requirements. In particular, because Ambler emphasises that this approach is suitable for accessing a variety of data sources, including relational databases, XML data sources and non-relational databases. Furthermore, it shows an architectural design that supports implementation of wrapping legacy data sources like TeamCenter Engineering or SAP to make them appear similar to the other data sources.

Another important issue is that this solution is supposed to make data access faster and easier for application developers and thus increase their productivity. This is possible through a generic framework based on metadata, which describes the data schema. With a solution like this, all effort can be concentrated on designing business rules based on business analysis and implementing these rules in a data schema and business components. There will be no more need for every developer to debug cryptic SQL statements or other means of query languages over and over again due to changes in data models, or to implement the same access methods several times.

5.3 Generic Object-Modelling of Data Sources

Encapsulation describes the concept of hiding implementation details and expose a controlled behaviour to a consumer. Behaviour is the look and feel that a consumer sees exposed. It represents an interface to the mechanisms that a consumer is allowed to interact with. This interface is the only way of interacting with an object, and all knowledge about the state of the object is gained from its behaviour. “State” is the values contained in the object that together defines the object state

5.3.1 The Basic Concepts

When representing data, Mark L. Fussell [12, pp. 8] introduces us to the following terms that I have chosen to use in this report: Attributes, Attribute Values, Relations and Tuples.

An attribute is the name of a property or characteristic that is a part of something. In this project that something is an object that represents data and the attribute is a property of a data object and is considered to be a container for a value on the object. The container also offers the behaviour needed to access the attribute.

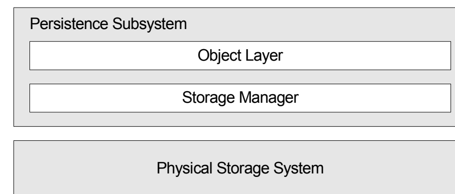
The attribute container contains an attribute value. The purpose of the attribute is to hold a value that makes the attribute specific. An attribute without an attribute value is only an abstract definition. The value must respect the data type that is defined by attribute. In this case, as is the case with XML values like (CTYPE), the attribute holds a type-less value that can be converted according to a specific need.

A relation is a descriptor that describes a relationship between attributes. It is a truth predicate that defines a set of attributes that exists somewhere. Take your living room; it has attributes like size, location and colour. This is a relation: There exists a room with the size, location and colour of your living room.

A tuple is a value that is based on a relation. When referring to your exact living room, a tuple has attribute values in the attributes described by the relation. This tuple now describes the characteristics of a specific room. If two tuples contain the same attribute values in any order based on the same relation, they are considered equal.

5.3.2 Structure of the Persistence System

This project must define a persistence system, and the overall architecture proposed consists of three layers, as displayed by Wolfgang Keller [7, pp.8]. The two top-layers make up the Persistence Subsystem, which contains both the Object Layer and the Storage Manager, while the third layer handles the actual access to various types of data sources.



The top layer is the Object Layer, which contains the object representation of data in the framework. It encapsulates data in an object-oriented structure and offers the needed behaviour to access data both as single tuples and as a selection of tuples from a relation in a data source. Furthermore, it holds information about the data schema it represents and references to the data source in which its data has been made persistent.

In the middle layer is the Storage Manager. Through the manager, the data contained in the object are mapped into a data source; hence it handles the "impedance mismatch" described in section 4.4.3. If the Storage Manager is set to handle all communication between the upper and the lower layer, these two layers does not need to know anything about each other and the implementations can be changed and expanded without having to rewrite all layers. The mediator design pattern offers this kind of behaviour by separating the Object Layer and the Physical Storage System. I will therefore build the Storage Manager according to this pattern as described in section 4.5.4.2.

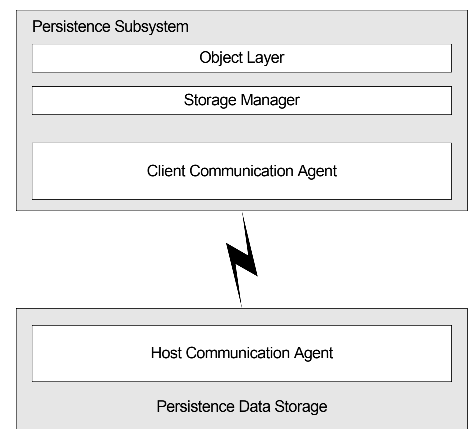
The lower layer controls management of concrete data sources and offers the functionality necessary to make the object layer retrieve, save and delete data from these. It is the Physical Storage System that offers the persistence behaviour needed to access the data sources. Since the Storage Manager offers a way for the Object Layer to work with the Physical Storage System without any of them knowing about the other, it acts as a mediator, which is also referred to as a broker [11].

When deciding on system architecture, a choice must be made of where to place the interface to the persistence mechanism for the consumer to access. Ambler has decided to put the interface in the Object Layer [11, pp. 11] but one could argue that this exposure of the behaviour is not in the interest of application developers.

When designing an n-tier application with a business logic and GUI layer, developers would like to encapsulate all persistence behaviour in complex business behaviour. But still the data needs to be transferred to the GUI layer. When the interface is on the objects, this behaviour will be exposed to the GUI designers. For a total encapsulation, this is not satisfactory.

In this project though, I have chosen the same interface strategy as Ambler except for the fact that the Object Layer only forwards all behavioural methods to the Storage Manager. Therefore, it is possible to remove the behaviour from the data objects and let the consumer place the call directly to the Storage Manager instead. Placing the interface directly on the data objects gives a simpler object structure to the consumer, and that is the reason for taking this approach.

The proposed three-layer design actually consists of four layers. The Persistence Data Storage is seldom located on the same machine as the Persistence Subsystem. Therefore, an intermediate layer is introduced to handle network communication between the Persistence Subsystem and the Persistence Data Storage. This layer is often handled by ODBC or a similar technology. On a schematic level this gives us a layered architecture as seen on the figure to the right [7, pp. 11].



5.3.3 Extending .NET Data Types

In the requirements, this solution is specified to extend the .NET framework and its types. This implies that all class definitions in the object layer extend and make use of the data types in the Common Type System in the .NET framework as required in section 3.3.3.

To have support for single and multiple tuples, the object containing a tuple must derive from the basic .NET type called *object*. This enables tuple objects to be stored in the various collection types made available in the namespace System.Collection. In fact, this requirement is easily fulfilled since all class definitions in .NET automatically derive from this object.

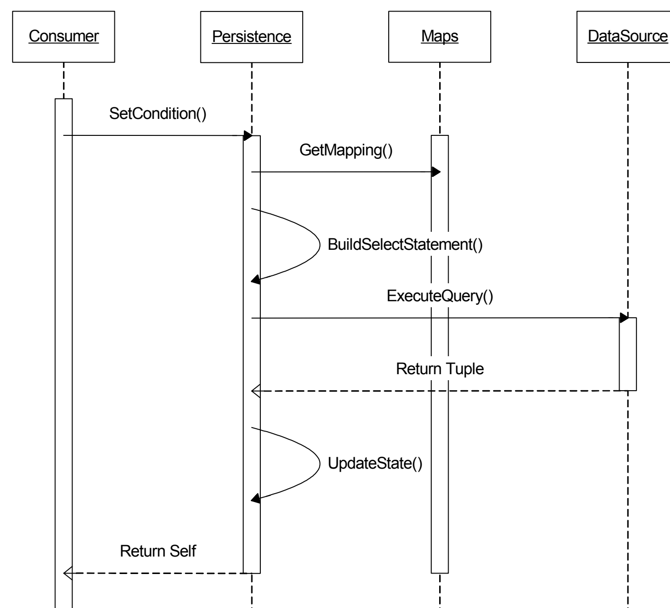
When working with a data selection, it is only natural that the set of tuples are represented as a set of tuple objects. To define two different kinds of interfaces for accessing a tuple, whether it has been part of a selection or it has always been only one tuple, would seem unnatural to the application developer. Therefore a selection – or set – of tuples is represented as a list of tuple objects. From this list of tuple objects, a single tuple object can be obtained and all Collection behaviour, like the iterator pattern, can be used.

A tuple object represents a relation and consequently it contains attributes. These attributes will support all types of the Common Type System; hence the full solution can be used across all .NET languages. Moreover, all definitions are made as Enumerations in order to avoid common errors like misspelling of configuration strings, and to keep definitions well organised.

5.3.4 Object Modelling on Non-Object Data

In this project, I am operating with the concept of encapsulating different non-object-oriented data sources in an object-oriented framework. This encapsulation strategy implies that data is represented in an object-oriented structure to the consumer. This strategy can be implemented in more than one way.

One approach is to give the objects an identity where tuple in the data source can only be represented once in memory. This holds the advantage that data cannot become old or "dirty". All changes made to the attribute values in the tuple representation take place in the same memory area. When implementing this feature in a multi-threaded or multi-user environment, a parallel-programmatic locking mechanism must be implemented to stop multiple consumers from accessing the same resource, e.g. a data attribute, at the same time.



Another approach is to let all objects have there own identity. This opens for the possibility to have multiple representations of the same tuple in memory. This is the approach chosen for this project. The locking mechanism must then be placed on the Storage Manager to prevent multiple objects from updating the same data at the same time.

5.3.5 Mapping

"Mapping" defines the rearrangement of data from one structure A to another structure B. This is what must to be done when switching data from a relational to an object-oriented environment cf. the "impedance mismatch" in section 4.4.3. To structure data in an object-oriented way, I have chosen to represent a relation as a class. This means that relations are mapped to classes.

Every instance of the class contains a tuple, which means that the relation object has the state of a tuple. These data object instances of the relation classes will inherit behaviour from abstract subclasses, and will only implement the data schema based on a kind of metadata corresponding to the relation as the requirements specify in section 3.3.1.

5.3.6 Queries and Selections

Essentially, a query is a means of retrieving a selection of data based on a set of conditions. In order for the solution to support queries of data, it must provide the behaviour and data structure needed to specify such conditions. When conditions are set, they will be used to select a single tuple or to select and delete multiple tuples. The conditions that the solution must support are:

- Equals
- Contains
- BeginsWith
- EndsWith
- GreaterThan
- LessThan

Initially, these are the conditions that will be supported by the framework. Of course it is easy to think of more advanced conditions like phonetically versions of the first four conditions. But this is left to be possible future enhancements.

When the conditions are set, the query may result in more that one selected tuple; hence the Object / Data Source Mapping Layer must provide a way of representing such a selection of tuples. To distinguish a single tuple from a set of tuples, I have chosen to create separate base classes for the two cases. A selection of tuples is in essence a collection of tuples, and we already have an object representation of a single tuple. Therefore, a selection of tuples is a container class that contains a list of tuple objects and supports iteration behaviour to traverse and get single objects.

5.4 Architectural Significant Design Packages

5.4.1 One Package Design

The statement "one package design" requires that the prototype be packed into one assembly. All methods are packed in one library or package called *DataLib.dll* that can be referenced by .NET projects inside Visual Studio .NET. In real life, this strategy can be difficult to implement since third part components linked to an assembly cannot be embedded into the new binary file. However, as few assembly files as possible is required.

In this case, the .NET framework does not directly support the MySQL and Oracle databases, and add-on products must be installed on the systems that access these data sources. The installers needed for supporting these data sources will be included in the final product.

Along with the binary assembly file, an executable file called *ManagementConsole.exe* exists. This is the .NET GUI based utility that can generate class-files from the data schema in a data source. Furthermore, it can maintain and alter the configuration file needed for the framework to contact data sources.

5.4.2 Extending the Package

In order to extend the solution with new features and behaviour, it must be added into the source code, and the DataLib.dll package must be recompiled and linked. This will result in a new version of the assembly, which must then be deployed. As long as the existing interfaces are not altered or deleted, the data source reflection classes and applications which are built on top of the DataLib.dll needs no recompilation.

The data sources are implemented through the façade strategy. This strategy implies that the actual implementation of specific instances of a data source could be put in separate assemblies. These assemblies would then inherit the persistence storage interfaces and abstract classes from DataLib.dll. The mediator which handles communication between data objects and the persistence storage would then become aware of which available data sources could be contacted through a configuration file.

This architecture is not implemented in the current design but since the façade pattern is used in the wrapping of data sources, it is possible to extend the framework with this feature in the future. This would enable external parties to implement their data sources into the framework. The reason for not taking the approach now is that since the intended usage of the framework is only inside the MBD organisation, no external parties will be implementing data sources into the system. Furthermore, it would require more effort in the implementation phase.

5.5 Summary

The analysis has determined that the best strategy for implementing a data access layer is the persistence storage strategy. This strategy defines an encapsulation of persistence data in an object structure available to application developers. The object structure reflects the data schema provided by various types of data sources.

The solution is based on a generic approach where a framework offers abstract behaviour that is concretised by metadata. This type of generic approach gives a single point of maintenance for accessing data from data sources. There is only one solution in which to extend with new features, and those features are immediately available to all other data sources. With the automatic generation of metadata that reflects concrete data sources, accessing data is an easy job for application developers.

In this chapter, all proposed solutions were dealt with on a conceptual level. No regards towards implementation details were handled in any way. This will be conducted in the next chapter. Still the line of approach is now laid out, and the design can derive directly from the conclusions appearing in this analysis.

CHAPTER 6

Design

6 DESIGN......3

6.1 PURPOSE......3

6.1.1 From Strategy to Design......3

6.2 THE GENERIC OBJECT MODEL......4

6.2.1 Persistence Layer......7

6.2.2 Managing Layer......9

6.2.3 Object Layer......10

6.3 DESIGN REPRESENTATION......14

6.3.1 Designing Interfaces......14

6.3.2 UML Contracts and System Sequence Diagram......14

6.3.2.1 Entity......15

6.3.2.2 EntitySet......16

6.3.2.3 EntityCursor......16

6.3.2.4 EntityAttribute......17

6.3.2.5 EntityTransaction......17

6.3.2.6 PersistenceException......17

6.4 SUMMARY......18

6 Design

6.1 Purpose

It was concluded in section 5.2 that the strategy of designing a persistence framework fulfilled the requirements set for this project. A strategy, however, is not the same as a solution. Therefore, this chapter sets out to create a software design that follows the persistence framework strategy and outline it with the tools provided to us by the UML standard.

6.1.1 From Strategy to Design

In this design the only layer exposed to the consumer is the top layer. This layer contains data and data access behaviour in an object-oriented structure. The bottom layer handles all communication with and queries to data sources. In the middle layer, a manager connects the two other layers so that the top and bottom layer knows nothing of each other. The theory behind this design is explained later in this chapter.

When designing a persistence layer that exposes a common behaviour towards a consumer, it makes sense to aim for a generic design. This requires the development of a generic object structure that is based on metadata describing how data are stored, altered, retrieved and deleted. In short, how the behaviour is initialised.

The UML tools used are primarily Contracts plus Sequence and Collaboration Diagrams that provide three different views on the behaviour of the framework. When addressing behaviour, this chapter will be based on the exposed behaviour, but it will also show how the internal object communications are handled. This should leave the reader with a broad enough view of things to obtain a full understanding of the solution.

6.2 The Generic Object Model

The framework extension consists of an interrelated set of classes, which provides the structure for implementing persistence behaviour behind the interfaces. This structure only shows the behaviour accessible by consumers. More specialised behaviour is implemented internally, all of which will be explained in chapter 7.

Since data is stored in persistent storage systems or data sources and only reflected in accessible objects when needed by a consumer, the objects must have a way to reference attributes in the data source. This is done by a direct mapping between the attributes on a data object to an attribute in a relation on the data source. It is assumed that the data type of the attribute is a primitive type like integers or primitive-alike like dates or strings. The mapping is then done through the parameters contained in the metadata of the data object.

While the direct mapping only represents a single entity in the data schema, there is also the need to reflect one-to-one, many-to-one, one-to-many and many-to-many. In the eyes of the consumer, the one-to-one is considered a special case of the many-to-one relationship, just as the one-to-many is considered a special case of the many-to-many relationship. This enables us to construct the interface in a more simple way.

The relationship data can be obtained for a single tuple object as a new selection and for a tuple set object through the concept of views. Based on the data contained in the objects, related data are retrieved from the data source either as a new tuple object or as a tuple set object. From one of these objects the relationship process can continue. The mappings supported by this object framework are:

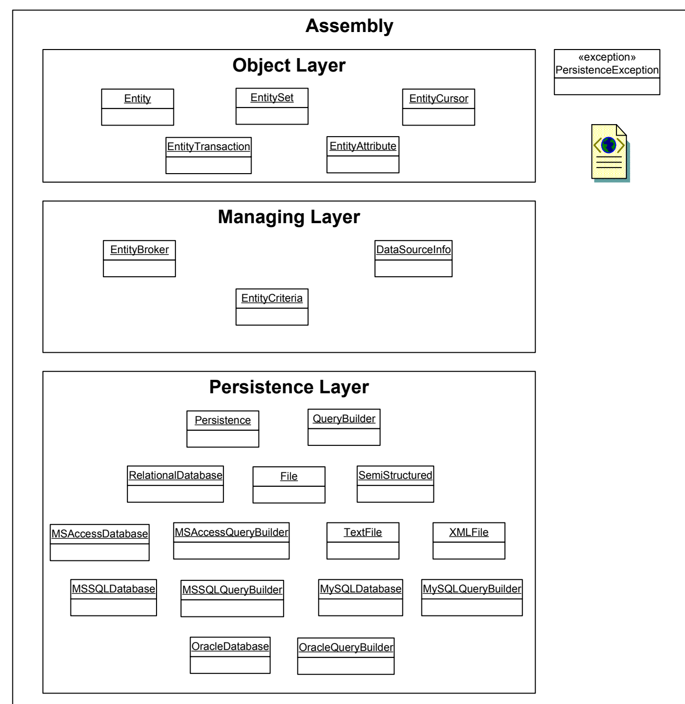
- One-to-one
- One-to-many
- Many-to-one
- Many-to-many
- Views or Aggregated Relations

A foreign key is the reference that combines two relations in a relationship. This foreign key references one tuple in the secondary relation from the primary relation. But from the secondary relation it actually references multiple tuples; hence the abstraction mentioned

before where one-to-one and many-to-one are represented in the same way to the consumer. In an object-oriented structure it is designed as a relationship that returns a single related tuple.

A mediator relation represents the many-to-many relationship between two relations. This relation constructs a one-to-many relationship from each of the two relations to the mediator. This fact makes it hard for an auto-generator to recognise the relationship since none of the two relations on either side of the mediator know each other. Still, when it is recognised, the object structure treats it is a singular relationship that returns a set of related tuples.

A view is a virtual relation that consists of a set of attributes, which is instantiated as tuples. However, a view does not exist as a stored set of data values in a data source. In this framework, views are treated as an aggregated mapping. The concept covers the mapping of multiple objects into one a relation that spans multiple entities on the data source. This type of aggregated object contains the attributes from more that one data object, and a tuple in this object covers a junction of the tuples of the data objects from which it origins.



As shown on the figure to the right, the solution consists of three layers. Each of these layers contains a number of classes. A full class diagram can be found in Appendix A but the classes are outlined here in their respective layers.

As you may remember from chapter 5, the top layer called Object Layer contains all interfaces exposed to consumers. This means that the other two layers are only internally represented. For a further explanation of the classes contained in the framework, please see the following table:

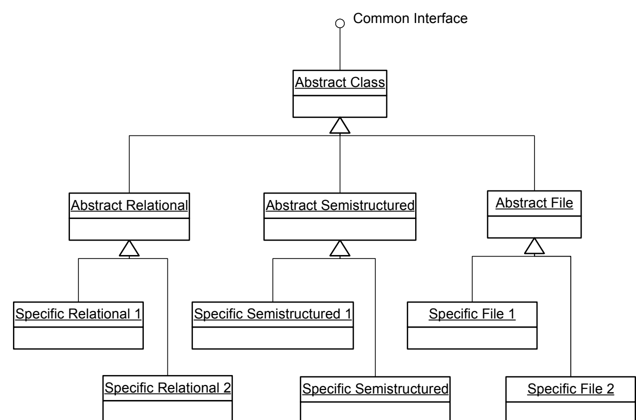
Class	Description	Access
Entity	The Entity class encapsulates the behaviour needed to instantiate a single tuple from a persistent entity as an object. The class publishes the necessary methods to make the object persistent.	Public
EntitySet	Contains a collection of tuples from a persistent entity selected through a set of selections and provides the necessary behaviour to access these data.	Public
EntityCursor	Provides a direct access to a specific tuple in an entity in a data source and the behaviour to traverse through the tuples contained in the persistence entity.	Public
QueryBuilter	Constructs the insert, update, delete and select statements needed to access data in a persistent data source – no matter if it is accessed by SQL or other query methodologies. The query is built on the information encapsulated in the Entity and EntitySet classes.	Internal
Persistence	A class hierarchy that encapsulates means of accessing flat files, relational databases, legacy systems, program API's, object-oriented databases or other types of data sources. The class layer wraps the complicated technologies such as ADO.NET, ODBC and other class libraries in order to protect the applications from changes in these.	Internal
EntityBroker	Handles the connection and possibly a caching mechanism to the Persistence Layer from the Entity/EntitySet encapsulation.	Internal
EntityTransaction	This class provides the framework with the functionality needed to support transactions and even nested transactions in the Persistence Layer.	Public
PersistenceException	When errors occur, this class extends the exception mechanism in the .NET framework to handle persistence specific errors.	Public

This overview has introduced the main classes of the framework. Let us now go through the framework starting from the bottom and working our way up through the layers.

6.2.1 Persistence Layer

In order to support a generic object design, consumer objects must see all provider objects as exposing the same interface. For this reason, I have explored the world of design patterns as described in section 4.5 and chosen to incorporate the Strategy Pattern in my solution. This pattern lets the framework wrap behaviour in a new common behaviour that is configurable through an outside object. Actually, it behaves a bit like a conjunction between the Façade Pattern and the Factory Pattern, which will be explained below.

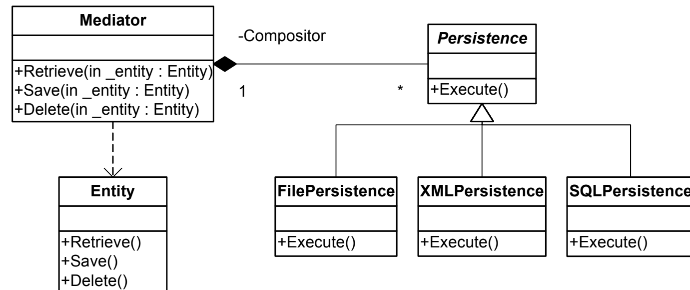
The Façade Pattern specifies how to combine a set of objects with different behaviour in an object structure that exposes common behaviour through a common interface. In this case, the set of different objects are represented by a variety of data sources. The design resembles a tree structure with class definitions as nodes and inheritance as branches, where the root node defines the interface. In the diagram to the right, you can see the structure.



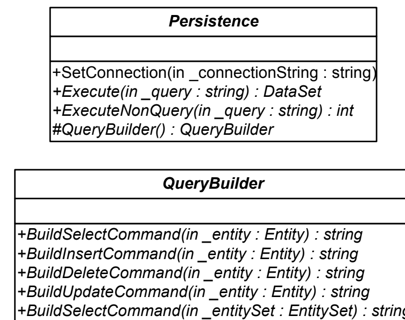
Through this design other objects can operate on a data source knowing that it keeps the UML contract specified for the interface. In fact, this enables a given object to access data from a data source in a generic way when having knowledge about the interface only and without being able to create an instance of the real implementation of the data source.

The question now is: If an object only knows the interface, how does an object obtain an instance of a specific data source? This is where the Factory Pattern comes into play. This pattern has knowledge about all the data sources and can by an outside specification produce an instance of the right object behind the interface. The outside action towards the factory can be regarded as a behavioural context.

Combining and refining these two patters will result in the Strategy Pattern. This pattern allows a control class e.g. the data source factory to choose a strategy or an instance of a specific type of data source communication class on the basis of an external context. The context is here defined by the metadata that specifies the behaviour of the data object. A diagram of the strategy pattern is taken from the book Design Patterns by "The Gang of Four" [14, pp. 315] and modified to this case in the figure above.

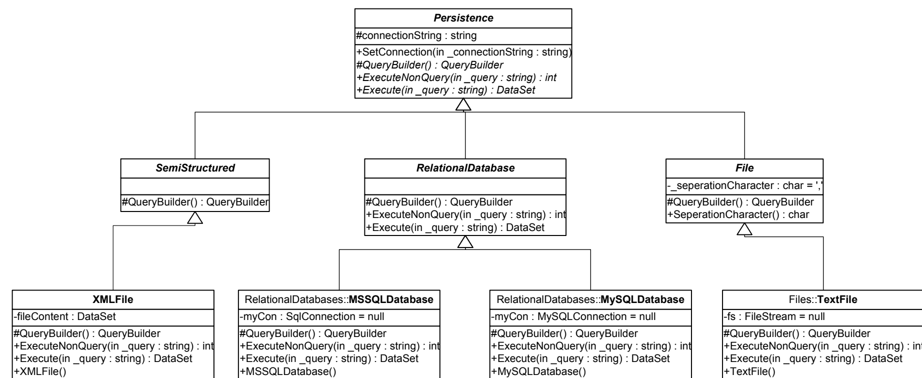


Since various data sources have various ways of querying data, every data source encapsulation consists of two classes. One for handling data source communication and a helper class for generating the necessary query-string based on the conditions set by the consumer. These conditions are transferred from the top layer to the bottom layer by the middle layer. It will be explained later through UML diagrams exactly how all objects communicate.



The Persistent Storage Layer implements a class hierarchy that encapsulates the behaviour of different types of data sources in a common interface. The full encapsulation must be taken with a pinch of salt, since various data sources implements different levels of behaviour. E.g. flat files generally provide less functionality than a relational database since it only implements sequential reading and writing of data without indexing and advanced search capabilities.

In general, the design follows the Strategy Design Pattern and consists of three layers. The top layer – the Persistence class –implements the abstract interface for the mechanism in a general way. Since it is abstract it cannot be instantiated and plays only the role of a common interface. The middle layer breaks down the different types of data sources in order to gain the advantages of source code reuse. The three classes, Semistructured, Relational Database and File are abstract implementations that implement the general behaviour of these types.



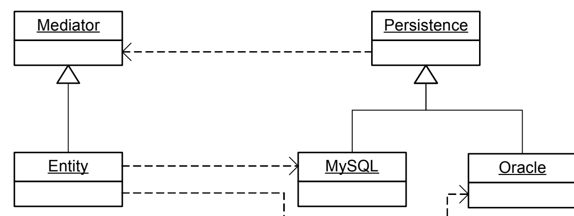
The bottom layer contains the actual implementation of data source behaviour. This is where the real and differentiated behaviour of various types of persistence mechanisms are wrapped into the common class framework.

In order to speak the language of a specific data source, every implementation of a data source in a class has a helper class called QueryBuilter. This class is based on reflection and implements a generic way of CRUD behavioural access to a persistent mechanism. The attributes or values come from the collections of data fields in the Entity class for a relational database the output is usually an SQL statement. Only the EntityBroker class can invoke this part of the class framework, which maintains connections to the data sources.

6.2.2 Managing Layer

The Managing Layer consists of two important entities that handle most of the mechanisms along with some support classes and structures. The main class is the Broker class. Being the main functionality, it handles the task of being a Mediator between the Object Layer and the Persistence Layer. Furthermore, it handles the Factory Pattern all classes in the Persistence Layer and exposes persistence behaviour for the data objects.

As a mediator, the class handles all communication between the top and bottom layer. Some refers to the mediator as a broker, but the essence of the pattern is that none of the layers knows about each other's existence. The mediator handles data requests from data objects towards the Persistence Layer by retrieving conditions from data objects and



sending it to the correct query builder in the Persistence Layer. This results in a query that returns a selection of data that is stored on the data object. The structure can be seen on the above figure, taken from [14, pp. 276] and adapted to this particular case.

The factory pattern is a part of the mediator. Whenever a data object needs to access the Persistence Layer, the context or state of the data object requests the mediator to access a specific type of data source. In this case, the consumer is the mediator since it handles all persistence behaviour. In fact, the three tasks of being mediator, adapter and factory could be divided into three separate classes. I have chosen to keep all functionality in one class. Perhaps the design would be more transparent if the class were divided into two separate classes. The class is not externally exposed; a change in the interface would pose no risk to the external interface. On the other hand, since this design poses no drawbacks, I have chosen to keep it as for now.

DataLib::**EntityBroker**
#EntityBroker() +Initialise(in _filename : string) -GetConnection(in _connectionName : string) : IPersistence -ReleaseConnection(in _persistence : IPersistence) +RetrieveEntity(in _entity : Entity) +RetrieveEntity(in _entitySet : EntitySet) +SaveEntity(in _entity : Entity) +SaveEntity(in _entitySet : EntitySet) +DeleteEntity(in _entity : Entity) +DeleteEntity(in _entitySet : EntitySet)

The mediator handles all configurations of data sources by accessing a configuration file containing data source definitions. This is used to initialise data sources when factorising data communication classes. The configuration file is an XML file and can be located where needed. This approach enables supporters to add, alter or remove data sources without having to recompile and redistribute the program package. As a default, the configuration file could be located in a centralised spot, and data sources would only have to be added in one file. Alternative multiple files can be distributed along with multiple applications.

6.2.3 Object Layer

The Object Layer consists of data classes that contain descriptive metadata and data values as well as expose persistence behaviour towards the consumer. Because of these features, this layer defines the look and feel of the product. Thus, the interface to these classes must be accurate and fulfilling. Otherwise the application developers will see a lack of functionality or worse, new functionality cannot be added without changing the existing interfaces. Applications developed on top of the framework would be obsolete and they would have to be rewritten, recompiled and redeployed.

The Object Layer consists of four exposed class definitions: EntityAttribute, Entity, EntitySet and PersistenceException, where Entity and EntitySet are abstract class definitions designed to be extended later based on data schemas. I'll get back to this approach and an alternative one later in this part of the chapter. These four classes alone define the behaviour available to application developers. The transaction manager should be added in the future as described in section 6.1.3.

The first of the four classes is the EntityAttribute. This class represents a single attribute from a relation in a data source. It is used and initialised both by the Entity and the EntitySet classes and contains an attribute value, which can be retrieved and altered. Furthermore, it handles the condition for a single attribute that is used to generate queries in the persistence layer.

EntityAttribute
+Equals(in _value : object) : IEntity +LessThan(in _value : object) : IEntity +GreaterThan(in _value : object) : IEntity +Contains(in _value : object) : IEntity +BeginsWith(in _value : object) : IEntity +EndsWith(in _value : object) : IEntity +Name() : string +Value() : object +ToString() : string +clearCriteria() : IEntity

The Entity class represents a relation in a data source and knows its own origin. This knowledge is defined through the metadata, which is generated on the basis of a data schema implemented on a specific data source and contains only one tuple representation of a relation. To store the tuple, it contains a collection of EntityAttributes and information about which attributes make up the identity or key of the tuple. This identity or key enables the class to reference the tuple in the data source.

Entities::***Entity***
+Retrieve() +Save() +Delete()

Entities::***EntitySet***
+Retrieve() +Save() +Delete()

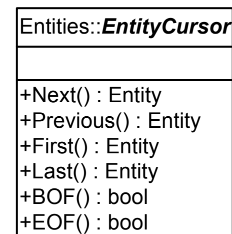
An attribute could have been used as object identifier to support only instance of a specific tuple object in memory. This attribute could contain an auto-incremented number or line number combined with the relation name for data sources based on a file or it could be GUIDs used as unique identifier. As the framework has to support legacy data schemas that use combined attributes as primary keys, this option must be opened for.

When performing a query that returns a selection, an application developer would expect to receive a list or collection of Entity objects. This list is contained in the EntitySet class. It handles the behaviour needed to iterate through a collection of tuples. Overall, the mechanisms are almost similar to the Entity class but extended with the behaviour needed to manage a list of Entities.

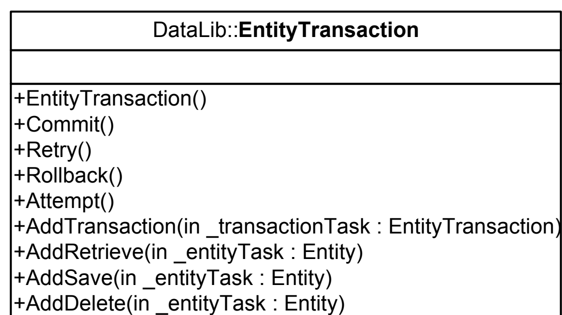
The two latter classes will be implemented as abstracts without object constructors. The idea is that the metadata defines the relation in a data source in a generic object. The metadata is in fact a class definition which is named by the relation it represents and inherits behaviour from the abstract class. This approach enables the framework to implement the data schema in a class definition, hence exposing the schema to the application developers that use the framework.

Another approach could be to let the Entity and EntitySet classes be real object types and construct the metadata as a separate structure that could be parsed by the Entity and EntitySet classes to initialise its state. The difference is, whether to let the metadata inherit behaviour from an abstract class as I have chosen, or to let metadata and behavioural mechanisms be defined separately.

The framework can be extended with a new class called EntityCursor that is implemented in the same way as the EntitySet. But instead of actually containing a selection of tuples, it could implement behaviour to scan to first, last, next and previous tuple in a relation based on a data source. But since it is not stated in the requirements that such behaviour is essential to the framework, it has not been implemented in this prototype.



In a future release, the framework must be extended with the implementation of the TransactionManager as specified by the requirements. The interface has already been specified but it has not been scheduled for implementation. The TransactionManager manager will structure all calls from the object layer to the Persistence Layer in a single transaction. Transactions can be nested inside larger transactions and it supports features like Retry and Rollback in case of a failure. Furthermore, it support the Attempt behaviour for a preliminary try to see if the transaction could be valid before an actual Commit is sent. In order for application developers to be able to access the manager, it must be placed in the upper layer.



The last class is the PersistenceException class. This class handles all error reporting from the persistence system as a whole. The class is based on the .NET exception handler class, which is extended with behaviour specific for this framework. As for now, this one class handles all types of exceptions, and error messages are compiled into the package. Since exceptions are intended for application developers only, error messages in one language is considered enough.

6.3 Design Representation

In this project, I have chosen to follow the architectural strategy of designing an explicit persistence framework as described in section 5.2.3. This strategy implies that a class structure is constructed as described in the analysis. But as the analysis resulted in an overall structure, the design must extend it with a specific interface.

6.3.1 Designing Interfaces

The interface has to be as persistent as possible, so that no defined behaviour is changed in future versions as stated in the requirements section 3.4.3. This is particularly important as the assembly must be referenced by a multiple set of applications, and a large change in the interface will cascade a rewriting and a recompilation of every application. This persistency in the interface is hard to achieve, although it can be done. The approach chosen here is to design the interface towards the consumer as large-behavioural methods, thereby limiting the functionality available to only a few methods.

These do-it-all methods also make the framework easy to use. Since they are based on the state of the exposed objects, the consumer only have to have an object initialised with the right metadata and filled with the correct attribute values or conditions. The methods will then handle the rest. The metadata is filled out by a graphical utility that auto-generates it from a data schema read from a data source.

6.3.2 UML Contracts and System Sequence Diagram

The classes have been defined in this chapter pp. 5 and 6. But these definitions only cover the larger object entities and their interrelation. In order to establish how behaviour has to be implemented, we need UML Contracts and System Sequence Diagrams.

A UML Contract identifies the state changes of a system or object when an operation or method is executed. Effectively, it will define what a single operation does. Operations are taken from the system interaction diagrams and every exposed behavioural method is described in a Contract. In brief, the Contract defines the preconditional expectations and the state to expect after the transition that occurs during execution.

Contracts are defined for the exposed classes only according to [13, pp. 179]. Since internal classes, interfaces and behaviour have no consequences for a consumer of this framework, their behaviour is can be decided by the implementer of the framework. All contracts can be found in appendix C but a few key Contracts will be introduced in this section.

Contracts are black box definitions. They define what goes in and what comes out. In order to review the dependencies between the classes and how they call each other, we need UML interaction diagrams. These action diagrams show how classes are interrelated inside the black boxes and how an object executes an interface operation.

System Sequence Diagrams show how events occur sequentially due to input from an external source, e.g. a consumer. They are a set of timeline drawings of interrelated systems. In this case, these systems are interrelated classes. Actions connects systems is a sequential order. They connect timelines, to show the type of occurring action. The System Sequence Diagrams for this system can be found in appendix B.

6.3.2.1 Entity

The entity class offers the persistence behaviour needed to retrieve, save and delete tuples in a data source. This behaviour covers the CRUD behaviour introduced in chapter 3. The Save() operation conducts both creation and updating of tuples. When an object is not yet made persistent, the tuple must be created whereas an existing tuple must be updated. Therefore, two contracts were made for the Save() operation.

Contract Entity:Save (new tuple)

Operation:	Save()
Responsibilities:	
Cross Reference:	Save – existing tuple
Exceptions:	If the key attribute values were not set and the key was not auto-generated, or the data source could not be contacted an exception was thrown.
Preconditions:	A value was set for one or more attribute values and the connection string to the data source was loaded.
Postconditions:	The tuple was made persistent in the appropriate data source and the key attribute values were updated with the generated key, if the key was set to be auto-generated. The key attribute values were set to read-only.

Contract Entity:Save (existing tuple)

Operation:	Save()
Responsibilities:	
Cross Reference:	Save – new tuple, Retrieve
Exceptions:	If the data source could not be contacted, an exception was thrown.
Preconditions:	One or more attribute values were changes since the last retrieval. Since the entity was retrieved before this update, it could be assumed that the connection string was set correct.
Postconditions:	The entity was retrieved and the attribute values were set to contain the attribute values from the updated tuple.

6.3.2.2 EntitySet

The EntitySet class implements the same operations as the Entity class. Therefore, I shall not comment on all the Contracts. To be able to compare the Save() operations of both classes, I will show the Contract for updating a selection of entities with one or more new attribute values.

Contract EntitySet:Save

Operation:	Save()
Responsibilities:	
Cross Reference:	Retrieve
Exceptions:	If the data source could not be contacted, an exception was thrown.
Preconditions:	A selection was retrieved and one or more attribute values were set. Since the entity was retrieved before this update, it could be assumed that the connection string was set correct.
Postconditions:	The selection of entities was updated and the attribute values were set for each entity to contain the attribute values from the updated tuple.

6.3.2.3 EntityCursor

The EntityCursor is not implemented. Therefore, I will not comment on the contracts which can be reviewed in Appendix B.

6.3.2.4 EntityAttribute

EntityAttributes are classes which are referenced as properties in the metadata class implementations. Value can be retrieved and stored in the container as attribute value through the property called value. Since this class is only a container for values and meta-data, there is no behaviour to express in an activity diagram.

6.3.2.5 EntityTransaction

The EntityTransaction works on entities where objects are added to a list for future execution of behavioural operations. Instead of executing the Save() operation on an entity, the object is added to a transaction through the AddSave(_object) method. When a number of entity operations are added to the list, the whole transaction can be executed by one method.

Prior to this batch execution, it could be relevant to know if the operation would succeed. This can be tested through the Attempt() operation. The operation executes the transaction without altering the tuples in the data source. This is specified in the following Contract.

Contract EntityTransaction:Attempt

Operation:	Attempt()
Responsibilities:	
Cross Reference:	None
Exceptions:	If a problem was reported by an added entity, an exception was thrown.
Preconditions:	One or more entities were added to the transaction.
Postconditions:	The actions were attempted on the data source without changing the state of any tuples.

6.3.2.6 PersistenceException

This class entity contains no behaviour. It extends the .NET class Exception to provide a mean of handling error messages specific to this framework. Therefore, it acts primarily as a wrapper which makes it possible to add persistence specific behaviour in the future.

6.4 Summary

The architecture has now made the transition from a conceptual model to a concrete specification ready for implementation. The class structure is defined and placed in relation to the three layered model and behavioural Contracts and Sequence Diagrams are created in UML specifications.

Furthermore, it has been described how the framework functions internally. It is a generic design where data schemas are automatically implemented as class definitions which inherit generic persistence behaviour from abstract framework classes.

The automatic implementations of data schemas are created through a utility which reads data schemas from a data source. Through automatic source code generation it builds a set of files containing class definitions and stores them in a directory on a disk. The generic layer and the specific mapping layer are two different assemblies but the mapping assembly, *DataSchemaName.dll*, which implements a data schema, inherits generic behaviour from the generic assembly, *DataLib.dll*.

CHAPTER 7

Implementation and Test

7 IMPLEMENTATION AND TEST 3

7.1 IMPLEMENTATION STRATEGY 3

7.1.1 Implementing the Framework 3

7.1.2 Integration to Demo System 4

7.2 FRAMEWORK EXAMPLES 6

7.2.1 The Basis 6

7.2.2 The Configuration File 6

7.2.3 Management Console 8

7.2.4 The Entity Object 9

7.2.5 The EntitySet Object 12

7.3 TESTING 13

7.3.1 Demo Data Sources 13

7.3.2 Functional Testing 14

7.3.3 Application Testing 15

7.4 SUMMARY 17

7 Implementation and Test

7.1 Implementation Strategy

The strategy is to create an assembly that implements the design proposed in chapter 6. This assembly is developed using Visual Studio .NET and following the process plan proposed in chapter 2. As stated in this plan, the implementation and functional test is closely connected procedures that are executed in parallel.

The test procedure relies on the assumption the first classes implemented are the behavioural interfaces that are exposed to the consumer. These shallow class definitions are auto-generated from the class diagram made in Visio, so my strategy will be to implement the class hierarchy without actual implementations of algorithms behind.

With the class structure in place, the Management Console program is constructed to enable automatic reflection of data sources on the Object / Data Source Mapping Layer framework. This utility can read the data schema from a test database and generate the classes needed for the functional test program to function. On top of these generated classes, the functional test program is implemented.

When the functional test program and the data source Management Console is ready to use, the implementation of behavioural algorithms can commence. During the entire implementation phase, test procedures guarantee that the behaviour of the implementation is consistent with the intentions.

7.1.1 Implementing the Framework

The object layer is implemented as thin objects. Where thick objects combine state and behaviour in class definitions, thin objects only hold data structures. At first, the object layer of this framework resembles thick objects as it provides access to behavioural methods. But as explained in chapter 6, the data object only forwards calls to hidden objects managed by a mediator. Due to this fact, the data objects are a relatively thin construction. This in essence saves memory and execution time whenever an object is allocated and deallocated.

The data schema from a data source is reflected in the definitions of the actual data objects. The Management Console generates these objects by extracting the data schema from the data source through a service class called Data Source Info. This class only contains forward declarations towards the EntityBroker and it only exists to expose the behaviour since the Managing Layer containing the EntityBroker class is hidden inside the assembly. How the class representing a data relation is constructed and how it initialises the attributes on top of the object layer can be seen in appendix D.

All communication from data objects to behavioural objects is handled by the mediator, which is implemented as a singleton. The singleton design pattern enables the runtime system to allocate only one instance of a class in the memory space and provides us with a single point of reference when handling data source connections. The implementation of a singleton in .NET is conducted by hiding the class constructor and providing a public and static variable initialised with an instance of it self.

```
Class Singleton
{
      protected Singleton() { ..do some algorithm here.. };
      public static readonly Singleton Instance = new Singleton();
}
```

If a cache-object were introduced to handle data source connections, the EntityBroker would not have to be a singleton. This would probably be a better solution since it would remove some of the resource sharing issues known from parallel programming. A number of locking mechanisms should also be implemented in the singleton where data object from different execution threads access the same methods.

7.1.2 Integration to Demo System

This Object / Data Source Mapping Layer framework is a prototype as stated in the requirements chapter 3.1. In order to prove its capabilities towards the organisation, a demo system has been implemented using the prototype on top of an existing data source. Currently, all information concerning companies, contact persons, addresses and contact information are stored in the DAISY system. DAISY is an abbreviation of Diesel Address Index SYstem. Based on the DAISY database, a search and view application has been developed using the Datalib.dll assembly and the files generated by the Management Console.

The integration is implemented to give decision makers a tool from which to evaluate the solution. The evaluation will conclude with a decision as to buy an existing product or to continue development of this solution. As non-technical users in MAN B&W Diesel A/S already knows the data contained in the DAISY database, they have previously come across parts of the product and can more easily relate to the solution.

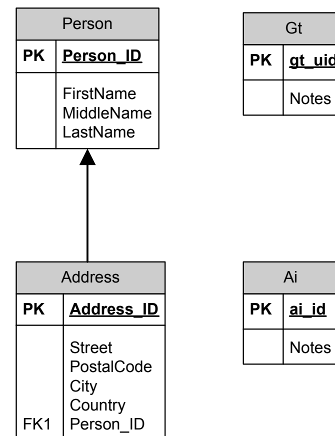
Developers on the other hand can review the application source code and see how the framework is used in a practical example. Of course, the same thing is possible with the functional test program, but there is still a difference between test examples and examples taken from the real world. This strategy provides both possibilities.

7.2 Framework Examples

The application developed for functional testing is an example of framework usage. Though not primarily a demo application, the data model used in the implementation can be used to demonstrate how the framework is used in practical applications.

7.2.1 The Basis

The data model in the test application consists of four relations of which two has a joined relationship. For instance, a person can have multiple addresses in which case the address relation has a many-to-one relationship with the person relation. Together, the two joint data relations form a separate section of the data model.



Each of the three sections in the data model contains a type of key. The person/address relation uses keys that consist of standard attributes. When a new tuple is added to the data source, the consumer must provide a unique value for the insert not to fail.

The "Gt" relation uses a GUID (global unique identifier) as primary key. This is a unique identifier that is generated to create a unique identity for an entity. A key is guaranteed to be unique no matter how many times you try to generate new ones or how many different machines you try to generate it on. Not all data sources support this data type, but since the algorithm is implemented in the .NET framework, and the key can be stored in an attribute consisting of 36 characters, this framework actually extends the data source functionality.

Last but not least, the key of the "Ai" relation is an integer that is automatically incremented for every insert. This is supported by all relational and file-based data sources. The attribute is often an integer of various lengths and the value is incremented by one for every insert. In fact the data type could be any kind of enumerator that can be incremented in steps.

7.2.2 The Configuration File

The framework must be able to contact a data source at some point in time to retrieve, save or delete data. In order to do so, it must be able to open a connection to the data source based on

the name given in the data object that reflects a data relation. This string name must be translated into a connection string that the persistence layer can use. The connection string must contain information like: Network address, real name of data source, user credentials to establish contact to data sources and etc.. These data are stored in an XML based configuration file, which has a specific format.

This is an example of a configuration file that follows the schema places in Appendix D.2:

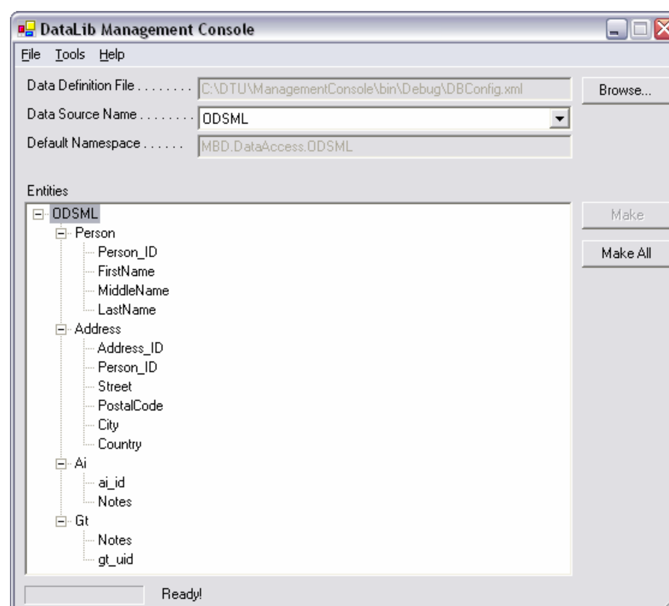
```
<?xml version="1.0" encoding="utf-8" ?>

<Configuration>
   <DataSources>
      <DataSource type="MSAccess">
         <ConnectionName>ODSML_MSACCESS</ConnectionName>
         <ConnectionString>
            Provider=Microsoft.Jet.OLEDB.4.0; Data Source=c:\DTU\odsml.mdb
         </ConnectionString>
      </DataSource>
      <DataSource type="MySQL">
         <ConnectionName>ODSML_MYSQL</ConnectionName>
          <ConnectionString>
           driver={MySQL ODBC 3.51 Driver};
           server=localhost;uid=root;pwd=1234;database=ODSML;option=NUM
          </ConnectionString>
      </DataSource>
      <DataSource type="MSSQL">
         <ConnectionName>ODSML</ConnectionName>
         <ConnectionString>
            Data Source=localhost; Initial Catalog=ODSML; User Id=sa_odsml;
            Password=1234
         </ConnectionString>
      </DataSource>
      <DataSource type="Oracle">
         <ConnectionName>ODSML_ORACLE_NOT_TESTET</ConnectionName>
         <ConnectionString>
            Provider=msdaora; Data Source=ODSML; User Id=sa_odsml;
            Password=1234
         </ConnectionString>
      </DataSource>
   </DataSources>
</Configuration>
```

The configuration file must be located in the default directory for the Datalib.dll assembly to be able to access it. The default location depends on the type of application that is being developed. Web applications running on the Internet Information Server use \Winnt\System32 as default location whereas windows applications use the directory containing the executable.

7.2.3 Management Console

The Management Console is responsible for the reflection of the data model into the generic object framework. This reflection is done by auto-generation of a class structure that inherits behaviour from the generic Entity and EntitySet classes. This results in two classes generated per data relation in the data source. In the future, when the Entity Cursor is also implemented, the utility also generates a class on top of it, resulting in a total of three classes per relation.

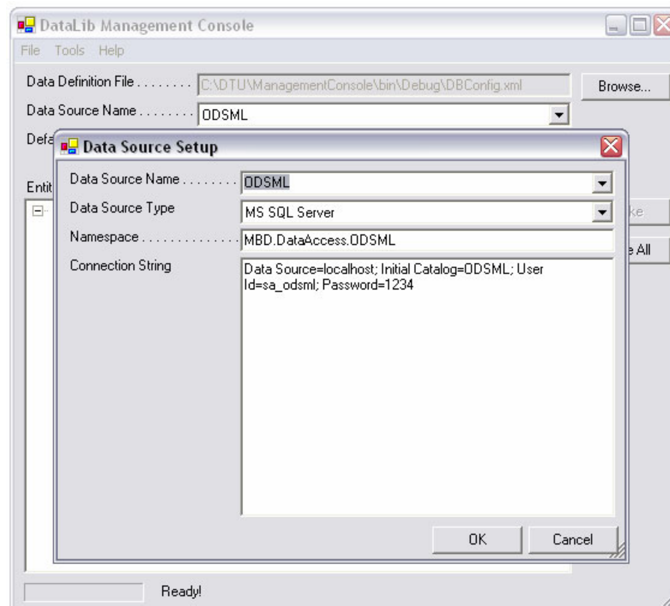
Executing the ManagementConsole.exe file, will launch the utility as a windows program. By default, the configuration file located in the same directory is used, but it is also possible to browse for other configuration files. The screen that appears looks like this:



The Management Console enables users to view data relations and their attributes from a data source in a semi-structured way using a tree control. At program start, the defined data sources can be found in the dropdown box called “Data Source Name”. When a data source is selected, the default namespace is filled out and the tree view is populated with the relations and attributes.

To the right of the tree view there are two buttons. These buttons are activated when a data relation is selected. The “Make” button generates the class definitions, which reflects the selected data relation. The “Make All” button generates all class definitions for the entire data source.

To configure data sources, the menu Tools->Data Source Setup allows consumers to edit the current selected configuration file. Changes can be discarded or saved into the XML configuration file.



When the Management Console creates a number of class files, it would only be natural if it would also compile them into an assembly. This is not yet implemented but would be a natural extension of the utility.

7.2.4 The Entity Object

In this solution, the Entity object is the reflection of a relation in a data source. It contains a single tuple that is populated with attribute values and offers persistence behaviour. It goes without saying that the implementation of the Entity class keeps the contracts defined in Appendix C.

When a data source is reflected into the object layer, object instances of a class with the same named as the relation represent tuples. Let's review a few examples based on the test database defined in section 7.2.1 for the functional test program. In this example, the Person relation is used.

When creating a new Person, the first thing to create is a new object instance of the class Person. On this object, the attribute values are filled out and the object is saved into the data source.

```
Person pers = new Person();
pers.Person_ID = 140374;
pers.FirstName = "Uffe";
pers.MiddleName = "";
pers.LastName = "Dejligbjerg";
pers.Save();
```

Now, the new tuple is stored on the data source. Remember that if an error occurred, an exception was thrown. This exception must be caught by enclosing the "Save()" command into a try-catch structure.

```
try
{
      pers.Save();
}
catch (PersistenceException pe)
{
      // Do some error handling
}
```

In order not to loose oneself in details, I will not include the try-catch constellation in the following examples. When a tuple has been created, it can always be retrieved again later. In the next example, I will show how to find the above tuple, modify it and save the changes. Everything is conducted on an object of the type Person.

```
Person pers = new Person();
pers.Person_ID.Equals(140374);
pers.LastName.Equals("Dejligbjerg");
pers.Retrieve();

pers.MiddleName = "NewMiddleName";
pers.Save();
```

Of course, a tuple can also be deleted from the data source. When deleting a single tuple, the delete command is executed on a singe-tuple object. It only makes sense to delete the data on an object, if the object actually represents a tuple. If not, a deletion must be conducted on the basis of a set of conditions that may affect multiple tuples. The process will be conducted by the multiple-tuple object called EntitySet instead.

To delete a tuple already retrieved, the procedure is:

```
Person pers = new Person();
pers.Person_ID.Equals(140374);
pers.LastName.Equals("Dejligbjerg");
pers.Retrieve();

// Some other source code

pers.delete();
```

This source code removes the tuple from the data source.

The relationships which an Entity is a part of can be located through the framework. Unfortunately, those relationships are not yet auto-generated by the utility. Basically, there are two types of implementation: the many-to-one and one-to-many defined in section 6.2.

In the following I shall use the two relations Person and Address for exemplifying the above. The Person has a one-to-many relationship with the Address relation. One person can have multiple addresses. This must be implemented in the schema classes for Person as:

```
// one-to-many relation
public AddressSet Relation_Address()
{
      AddressSet relationset = new AddressSet();
      relationset.Person_ID.Equals(Person_ID);
      return relationset.Retrieve();
}
```

And for the Address, the many-to-one relationship is implemented as:

```
// many-to-one relation
public Person Relation_Person()
{
      Person relation = new Person();
      relation.Person_ID.Equals(Person_ID);
      return relation.Retrieve();
}
```

The difficulty part of implementing this feature in the metadata class representation is to extracting the information about relationships from the data sources. However, this problem can be solved within a minor timeframe.

7.2.5 The EntitySet Object

The EntitySet object represents multiple tuples in a data source. Working with selections of tuples is conducted by working an EntitySet object. The object must be an instance of the same class name as the relation on which the tuples are based. Still using the test database from the functional test program, I shall go through examples of usages of the Address relation. The multiple-tuple object reflecting the Address relation is called AddressSet.

To retrieve a selection of data through the AddressSet object, a set of conditions must be added to the object. A combination of different conditions is allowed.

```
AddressSet addresses = new AddressSet();
addresses.PostalCode.GreaterThan(2800);
addresses.Country.Equals("Denmark");
addresses.Ascending(addresses.PostalCode);
addresses.Retrieve();
```

This will result in a selection of Address objects. If the data is to be displayed in a UI control, the object can be added to such as a data source. Let's add the selection to a DataGrid:

```
dataGrid.DataSource = addresses.DataSource;
```

If instead, the program must iterate through the selection in order to perform the method "Convert()" on every object, it can be done like this:

```
foreach (Address addr in addresses.Elements)
{
            Convert(addr);
            addr.Save();
}
```

As opposed to the Entity objects, the EntitySet object can be used to make a batch-delete based on a set of conditions. Through this deletion, multiple tuples will be removed from the data source. To delete all addresses where the postal code is between 2800 and 3500, the procedure is the following:

```
AddressSet addresses = new AddressSet();
addresses.PostalCode.GreatherThan(2800);
addresses.PostalCode.LessThan(3500);
addresses.Delete();
```

7.3 Testing

This project is required to conclude in a functional prototype of the Object / Data Source Mapping Layer. No real product release is planned and the product does not have to be included in the production of applications. Because no product is released, it is not required to conduct a robust and thorough test procedure to remove every possible bug in the system. On the other hand, not even a prototype will be functional without any form of test.

Therefore, the scope of the test is two fold as stated in the requirements. During development, a functional test is conducted to ensure that the behaviour of the system is in accordance with the Contracts defined in Appendix C. After the development phase, an application test is performed to ensure that the architecture and design meet the conditions set out in the requirements. To do this, an existing application is redesigned to use the Object / Data Source Mapping Layer as a mean of accessing data.

7.3.1 Demo Data Sources

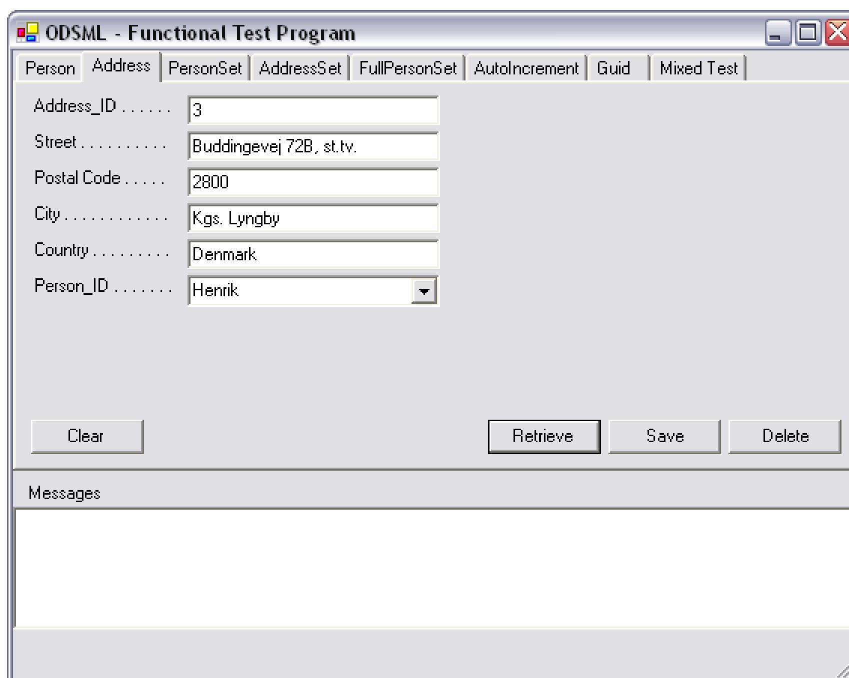
In this test phase, two data sources are used. The first one is the database used for functional testing. As you may recall, I went through the implemented data schema in section 7.2.1. This data schema was implemented on two different databases, namely the MySQL and MS SQL databases. This is to prove that data schemas can be moved between different types of data source without changing the application source code. Since file and semi-structured data sources are of a lower priority, testing has been concentrated around data sources that implement a data server with query capabilities. The data schema can be found in Appendix E.1.

For the application test, I have used the DAISY database. Currently, the data schema is implemented and in service on an MS SQL Server installation. It contains data about companies, contact persons, MBD employees, addresses, contact information and more. The data model is rather extensive today, but in the near future is will be connected to the financial system SAP which will take over ownership of the data. The next version will then function as a slave database, with a more simple data model.

7.3.2 Functional Testing

The functional test application is made for testing all exposed interfaces. This is to ensure that the exposed objects behave in accordance with the contracts defined in Appendix C. The actual test is conducted through an application with a Windows GUI.

For the two relations called Person and Address, there are four tabs on the Windows form. For each relation there is one tab for the single-tuple object and one for the multiple-tuple object. This is for testing both types of entity objects.



The fifth tab is called “FullPersonSet”. This tab represents a view that spans over multiple relations; in this case, the Person and Address relations. The two relations have a relationship that can be seen on the data model in Appendix E.1. This relationship is used to construct a new relation that includes all attributes from both the Person and the Address relations.

The next two tabs, number six and seven, cover two kinds of auto-generated key. The AutoIncrement tab is based on the relation called “Ai” in the test data schema. When a new tuple is inserted, a list of all data contained in the relation will be shown. This verifies that the object behaves correctly. The GUID tab works much like the AutoIncrement tab, except that the primary key now consists of a GUID. The Object / Data Source Mapping Layer ensure

that the GUID is auto-generated which can be verified on the list that can be viewed on the tab. The last tab, Mixed Test, is for ad hoc testing.

7.3.3 Application Testing

Currently, the DAISY application runs as a .NET Web Application using Web Forms. On the intranet it offers a way of searching through the various data stored in the DAISY database. The two main entities on which the application offers search capabilities is Person or Company. All other data stored in the data source is related to these. Person covers both contact persons inside and outside the MBD company.

The DAISY database is currently running on a MS SQL Server, which means that the application test is conducted on that data source only. It has no impact on the test quality that

the data source is located on only one kind of database. The purpose of the test is to prove the usability of the framework, not to do functional testing.

The criteria that is set for a search is entered into a custom control. This is a custom-made control on top of the abstract UI control class hierarchy in the .NET framework. This particular control is made in three different versions. The three versions are: Textbox, Dropdown box and Checkbox of which the latter is not used as this control handles Boolean values that are not present in the DAISY data model. The textbox allows users to specify which type of search criteria the control must represent.

First Name Begins with
Country Cameroon, United Republic

The six different types of criteria were defined in chapter 5.3.6. When the necessary criteria are entered, a search is executed by pressing the "Search" button. This will result in a selection list, which is actually a selection of Entity objects in an EntitySet object behind the curtain. A set of details can be shown for each pf the elements in the result list. The details page will emerge when clicking on the hyperlink representing an element in the result list.

Due to the use of this framework, the development of the new DAISY application went rather fast. It took one working day to design the Web Forms, add the view controls on the right spots and resize everything to the correct size. Having done this, it took only one more working day to extract the data schema using the Management Console utility and implement the objects in the application.

7.4 Summary

The key and general requirements are now implemented in the framework. Only non-critical and nice-to-have's are left undone. The framework has passed a functional test and more than three application tests by three different developers. This means that a fully functional framework is ready as a beta release. Therefore, it will be pre-released in the IT-department to gain more usage experience.

The framework has already shown good signs for the rapid implementation of a data application. The development of the new DAISY application took only two working days. It could be an indication of the framework supporting rapid application development.

Of course, it must be taken into consideration that a requirement specification for the DAISY application was already made. Furthermore, the development of both the framework and the application was conducted by me which means that I had an extended knowledge about the framework prior to the development. A new test needs to be conducted on either the same or a similar application prior to the decision about whether to continue developing this framework or to buy an existing framework from another provider.

CHAPTER 8

Conclusion

8 CONCLUSION 3

8.1 PURPOSE 3
8.2 GOALS 3
8.3 SUMMING UP 4
8.4 EVALUATION OF ESSENTIAL RESULTS 5
8.5 SUMMARY OF CONTRIBUTIONS 6
8.6 FUTURE IMPROVEMENTS 7

8 Conclusion

8.1 Purpose

The purpose of this project is to establish the strategy for a common data access layer for the software development departments at MAN B&W Diesel A/S. The strategy was founded on a requirement study and has resulted in the design of a data access architecture. On the basis of this architecture, a prototype has been built.

The choice of strategy and design of architecture has been inspired primarily by the work of Scott W. Ambler. His papers deal with the principle of data source access strategies and architecture, in particular with the mapping of non-object-oriented data into an object structure.

Since the IT departments at MAN B&W Diesel A/S have a platform strategy specifying the use of Microsoft Windows, the implementation basis of the project is the Microsoft .NET platform. This platform is the future platform architecture in the product line of Microsoft and therefore it seemed only natural to explore its capabilities in this thesis.

8.2 Goals

There are two main goals with this project. The first goal was to establish the requirements for a programmatic data access layer. A requirement analysis has been conducted, and the gathered requirements reflect the organisational needs. It includes an analysis to find the best-suited strategy for fulfilling these requirements. Still, it must be emphasised that no decision about whether to continue development or buy an existing product has been made. This decision is to be made by the IT management subsequently to this project.

The second goal was to design a prototype that implements the strategy chosen in the analysis. The prototype respects the requirements of mapping non-object-oriented data into an object model. Furthermore, it offers a graphically based utility that initialises the generic prototype automatically and sets up connections to data sources.

8.3 Summing Up

The requirements for a data access framework in the software development department are gathered and rated according to their importance to the project. This requirement study can be used as a basis for choosing a future strategy for data source implementation in the software development organisation.

The .NET platform offers a simple way of accessing data in data sources, however not in an object-oriented fashion. Since the rest of the .NET framework is purely object-oriented, the standard .NET data access approach suffers from the *Impedance Mismatch*. In this framework extension, data are now stored in an object structure and mapped into non-object data sources.

In contrast to other object encapsulating frameworks on the market, this framework explicitly reflects the data schemas in the framework. This enables software developers to access attributes in data relations without having to worry about names and definitions. It is all visible during implementation inside Visual Studio .NET

The data object is independent of the type of data source used. To the software developer, the actual data sources are not visible but managed inside the framework. The only thing that needs to be handled is the data schema through the object layer.

All selection of data through queries can now happen without any knowledge about query languages. Admittedly, this framework does not support all advanced kinds of query types, instead it makes it very easy to build simple and semi-advanced data applications.

8.4 Evaluation of Essential Results

In my opinion, the prototype works like a charm. During this project, I have stopped using SQL directly as a query language and instead based my applications on the DataLib.dll. Furthermore, two developers at MAN B&W Diesel A/S have actually used the framework as the basis in their own applications. Now, the application for managing members of the Art Club in MAN B&W Diesel A/S runs on top of the framework – as does the DAISY demo application.

The two developers have expressed that they found the framework easy to use as data schemas are explicitly reflected in the class definitions in the object layer. Furthermore, the reflections are automatically generated through a utility, a feature that vouches for fast implementation of data sources in applications. The whole access layer is ready even when you plan your application.

This was demonstrated during the development of the DAISY demo application. The requirements had been defined earlier as the application already exists in the organisation whereas the data model had been designed on the basis of an old mainframe application. With these two preconditions, the application development consisted of auto-generating the reflection of the data schema in class files through the use of the Management Console, and connecting a front-end Web GUI to the data objects. All search capabilities and the persistence behaviour was right at hand from the very beginning of the implementation.

As for now, the framework is adequate to handle applications with a controlled user environment for update of data, and it will require only the implementation of transaction management in order to support larger multi-user applications. The specifications and design are ready, so this feature is depending on the necessary resources for implementation and test.

I therefore recommend that we continue development of this framework. I realise that essentially it is about money but it is my opinion that the last few features could be implemented and the framework thoroughly tested in a matter of months. Although it has not been officially decided yet, I am under the impression that the framework will be allowed further resources for development in the department.

8.5 Summary of Contributions

This thesis applies the conceptual model proposed by Scott W. Ambler on the object-oriented .NET framework recently released by Microsoft. Along with the .NET framework, Microsoft launched a new version of ADO.NET, which encapsulates data access behind a façade and wraps relational data in a single common data object. ADO.NET, however, still relies on relational principles that do not solve the impedance mismatch between relational data and object-oriented programming structures. Furthermore, data schemas are only internally represented in the single object structure.

The major contribution from this framework is that data schemas are directly reflected in the framework in an object-oriented way. It utilises the fact that business logic requires knowledge about the data structures provided by a data schemas whether the data access framework is generically implemented or not. Therefore, the object layer can be used to implement data schemas through class definitions generated on the basis of data schema definition and through inheritance of behaviour from a generic class structure. This approach is facilitated in this framework.

The principal contributions of this thesis are:

- **Reflection of data schemas in a data access framework** – the definitions contained in data schemas are reflected in the class structure of the data access layer thereby enabling developers to gain a good grasp of the details in data structures inside a persistence infrastructure when using the Visual Studio .NET environment.

- **Object-oriented mapping of non-object data** – the framework handles the transformation of non-object-oriented data structures into object-oriented principles and vice versa with a full integration between various types of data sources and the .NET framework including types and collections.

- **Easy-to-use query behaviour** – all queries are handled by the framework through a simple interface leaving it to the developer to decide only on the type of query and its value limitations.

8.6 Future Improvements

There are two kinds of future improvements: The improvements that I have included in the analysis and design, but left out of the implementation – and improvements that I have not thought of yet. The framework implements the façade strategy in the architecture which makes it possible to extend interfaces to implement future features.

The future improvements of the framework are:

- **Transaction Manager** – the implementation of a transaction manager will make it possible to control the update of data in multi-user environments. The architecture and behaviour is already specified, and the only thing missing is the implementation.

- **Cursor** – like the Transaction Manager, the architecture and behaviour for this feature is already specified. Implementing the cursor will enable developers to traverse data in data sources sequentially without making queries.

- **Cache of data objects** – implementing the cache pattern to store data objects can be maintained inside the persistence mediator and will disable constant communication with data sources. The data object is required to have identities in order to manage objects in the cache storage. These identities are implemented on the generic data objects as a combination of data source and tuple identity to achieve global uniqueness.

- **Support for advanced queries** – the query builders in the framework can be extended with additional features like phonetic conditions or behaviour to combine conditions in a more advanced manner on a single attribute.

- **Utility management of views** – the Management Console can be extended to create and alter view classes, a task that is performed manually today.

- **Delete rules** – to enable the framework to execute bulk deletions, delete rules can be added. It is a trigger that enables cascading deletion of data combined with relationships according to explicit rules.

Bibliography

Bibliography

[1] C# Design Patterns - A Tutorial
James W. Cooper
Addison-Wesley
ISBN: 0-201-84453-2

[2] The Fundamentals of Mapping Objects to Relational Databases
Scott W. Ambler
http://www.agiledata.org/essays/mappingObjects.html
Copyright 2003

[3] Encapsulating Database Access
Scott W. Ambler
http://www.agiledata.org/essays/implementationStrategies.html
Copyright 2003

[4] Applied Microsoft .NET Framework Programming
Jeffrey Richter
Microsoft Press
ISBN: 0-7356-1422-9

[5] The Rational Unified Process – An Introduction (Second Edition)
Philippe Kruchten
Addison-Wesley
ISBN: 0-201-70710-1

[6] Use Case Driven Object Modelling With UML
Doug Rosenberg with Kendall Scott
Addison-Wesley
ISBN: 0-201-43289-7

[7] Object/Relational Access Layer
A roadmap, Missing Links and More Patterns
Wolfgang Keller
http://www.objectarchitects.de/ObjectArchitects/orpatterns/

[8] Document Definitions for LIME Project
QMS Document
Jan Sørensen, MAN B&W Diesel A/S

[9] Data on the Web
From Relations to Semistructured Data and XML
Serge Abiteboul, Peter Buneman and Dan Suciu
Morgan Kaufmann, 2000
ISBN: 1-55860-622—X

[10] The Joy of Legacy Data
Scott W. Ambler
http://www.agiledata.org/essays/legacyDatabases.html
Copyright 2002-2003

[11] The Design of a Robust Persistence Layer For Relational Databases
Scott W. Ambler
www.ambysoft.com/persistenceLayer.html
Copyright 1997-2000

[12] Foundations of Object Relational Mapping v0.2
Mark L. Fussell
http://www.chimu.com/publications/objectRelational/objectRelational.pdf

[13] Applying UML and Patterns
An introduction to Object-Oriented Analysis
And Design and the Unified Process
Craig Larman
Prentice Hall, Inc.
ISBN: 0-13-092569-1

[14] Design Patterns
Elements of Reusable Object-Oriented Software
Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides
Addison-Wesley 1995
ISBN: 0-201-63361-2

[15] Microsoft .NET
Perspektiver, muligheder og teknik
Morten Strunge Nielsen
Gordion 2003
ISBN: 87-91301-00-9

[16] MSDN
Microsoft online documentation
http://msdn.microsoft.com

Appendix

APPENDIX .... **II**

APPENDIX A – CLASS DIAGRAMS III

Appendix A – Conceptual Class Diagram .... *III*

Appendix A.2 – Full Class Diagram .... *IV*

APPENDIX B – ENTITY SEQUENCE DIAGRAMS V

Appendix B.1 – Entity:Retrieve .... *V*

Appendix B.2 – Entity:Save (New Tuple) .... *VI*

Appendix B.3 – Entity:Save (Existing Tuple) .... *VII*

Appendix B.4 – Entity:Delete .... *VIII*

Appendix B.5 – EntitySet:Retrieve .... *IX*

Appendix B.6 – EntitySet:Delete .... *X*

APPENDIX C – INTERFACE CONTRACTS XI

Appendix C.1 - Entity Behaviour .... *XI*

Contract Entity:Retrieve XI

Contract Entity:Save (New Tuple) XI

Contract Entity:Save (Existing Tuple) XII

Contract Entity:Delete XII

Appendix C.2 - EntitySet Behaviour .... *XIII*

Contract EntitySet:Retrieve XIII

Contract EntitySet:Save XIII

Contract EntitySet:Delete XIV

Appendix C.3 – EntityCursor Behaviour .... *XV*

Contract EntityCursor:First XV

Contract EntityCursor:Last XV

Contract EntityCursor:Next XVI

Contract EntityCursor:Previous XVI

Appendix C.4 - EntityTransaction Behaviour .... *XVII*

Contract EntityTransaction:Commit XVII

Contract EntityTransaction:Retry XVII

Contract EntityTransaction:Rollback XVII

Contract EntityTransaction:Attempt XVIII

APPENDIX D – INITIALISING METADATA IN THE FRAMEWORK XIX

Appendix D.1 - Data Schema Reflection in Object Layer .... *XIX*

Appendix D.2 – XML Schema for Configuration File .... *XX*

APPENDIX E – DATA SCHEMAS XXI

Appendix E.1 - Functional Test Database .... *XXI*

Appendix E.2 – DAISY Data Model .... *XXII*

Appendix A – Class Diagrams

Appendix A – Conceptual Class Diagram

PersistenceLayer::*QueryBuilder*
+BuildSelectCommand(in _entity : Entity) : string
+BuildInsertCommand(in _entity : Entity) : string
+BuildDeleteCommand(in _entity : Entity) : string
+BuildUpdateCommand(in _entity : Entity) : string
+BuildSelectCommand(in _entitySet : EntitySet) : string

1..1
1..1
1..1
1..1
1..1

Entities::*EntityCursor*
-Size : int
+Next() : Entity
+Previous() : Entity
+First() : Entity
+Last() : Entity
+BOF() : bool
+EOF() : bool

Entities::*EntitySet*
+Retrieve()
+Save()
+Delete()

1..1
1..1
1 ..1
1..1

Entities::*Entity*
+Retrieve()
+Save()
+Delete()

EntityBroker
+RetrieveEntity(in _entity : Entity)
+RetrieveEntity(in _entitySet : EntitySet)
+SaveEntity(in _entity : Entity)
+SaveEntity(in _entitySet : EntitySet)
+DeleteEntity(in _entity : Entity)
+DeleteEntity(in _entitySet : EntitySet)

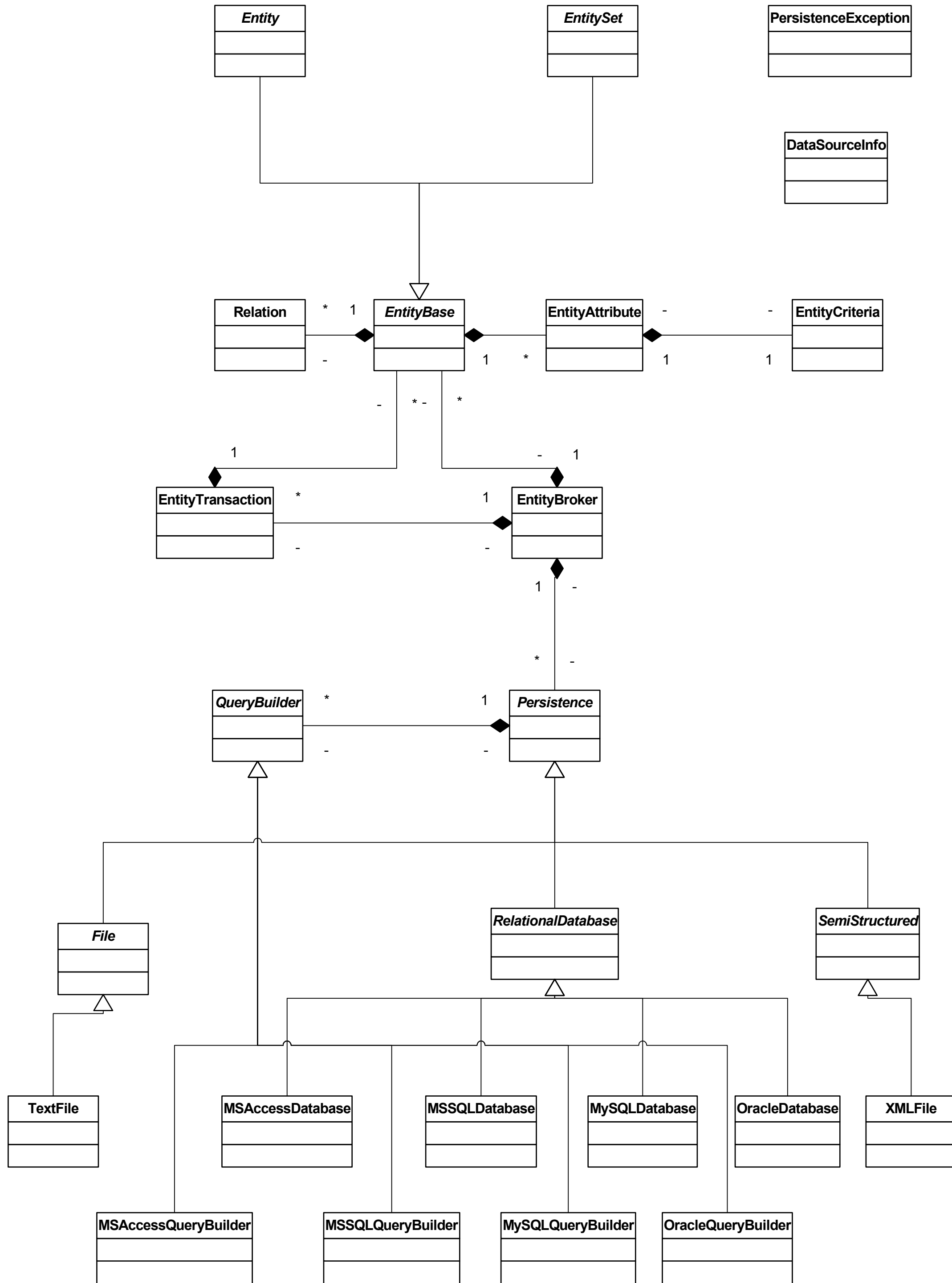
0..n
1..1
1..1
0..n

EntityTransaction
+Commit()
+Retry()
+Rollback()
+Attempt()
+AddRetrieve(in _entityTask : Entity)
+AddSave(in _entityTask : Entity)
+AddDelete(in _entityTask : Entity)
+AddTransaction(in _transactionTask : EntityTransaction)

0..n

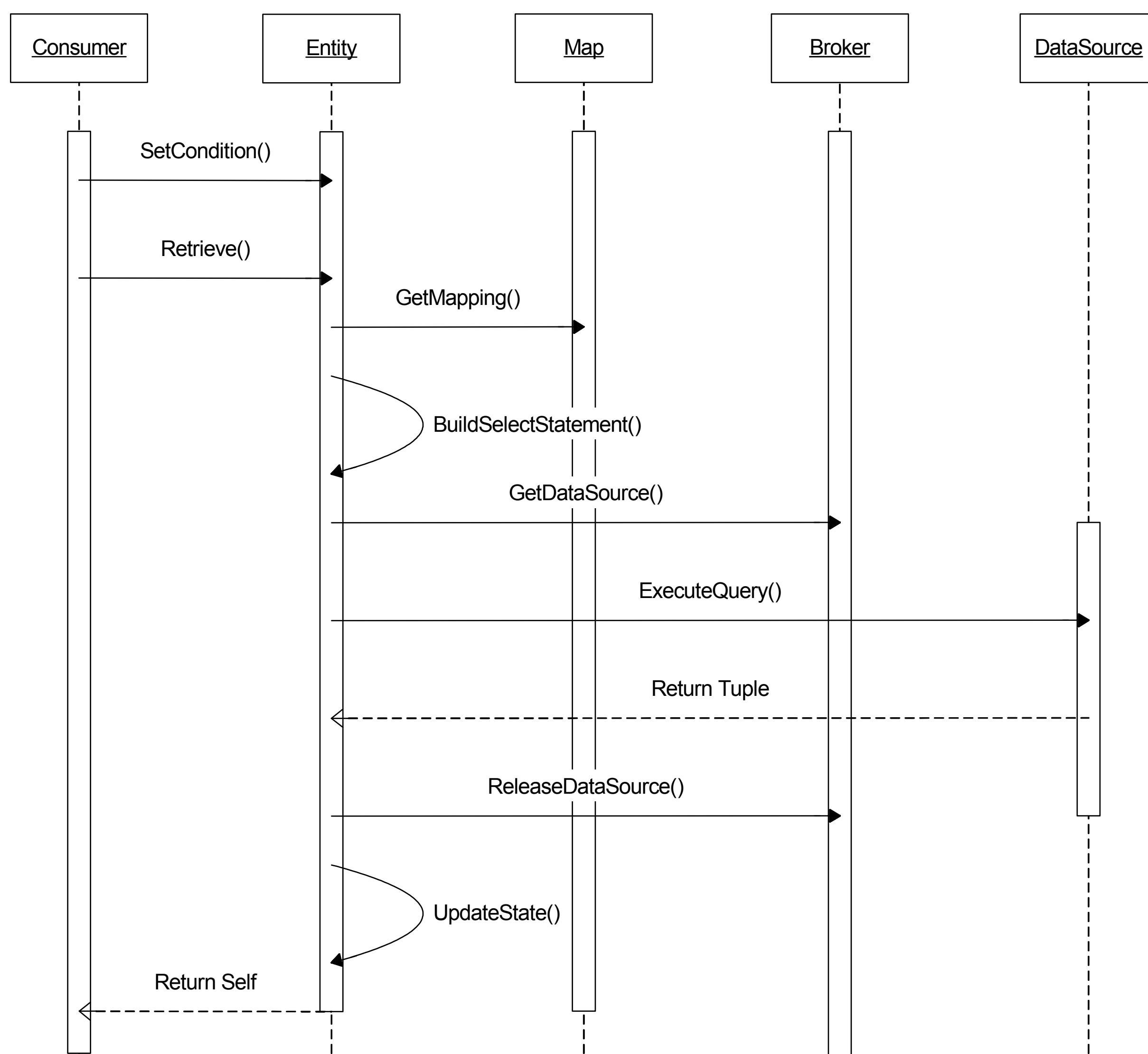
PersistenceLayer::*Persistence*
+SetConnection(in _connectionString : string)
+Execute(in _query : string) : DataSet
+ExecuteNonQuery(in _query : string) : int
#QueryBuilder() : QueryBuilder

Appendix A.2 – Full Class Diagram

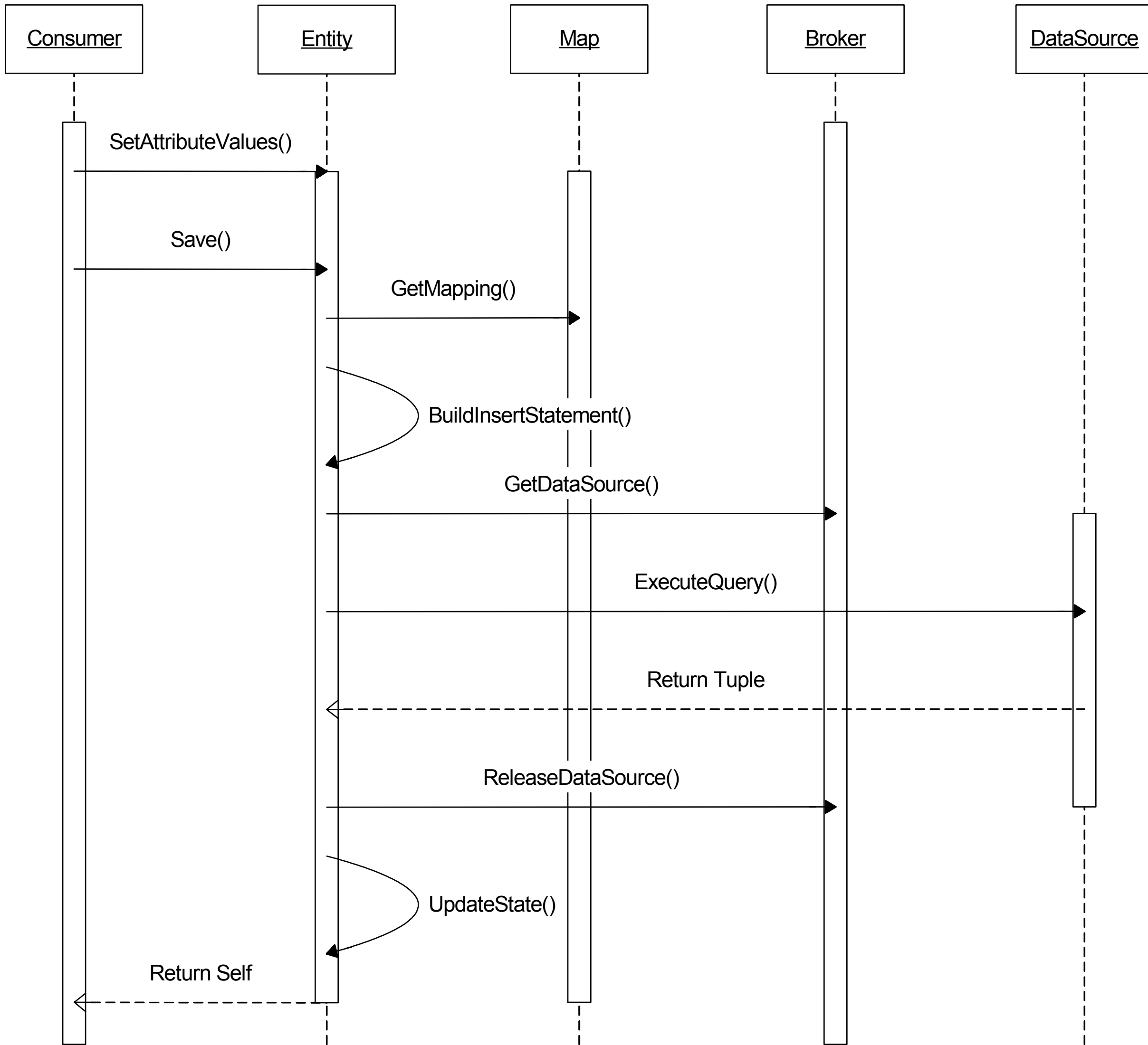


Appendix B – Entity Sequence Diagrams

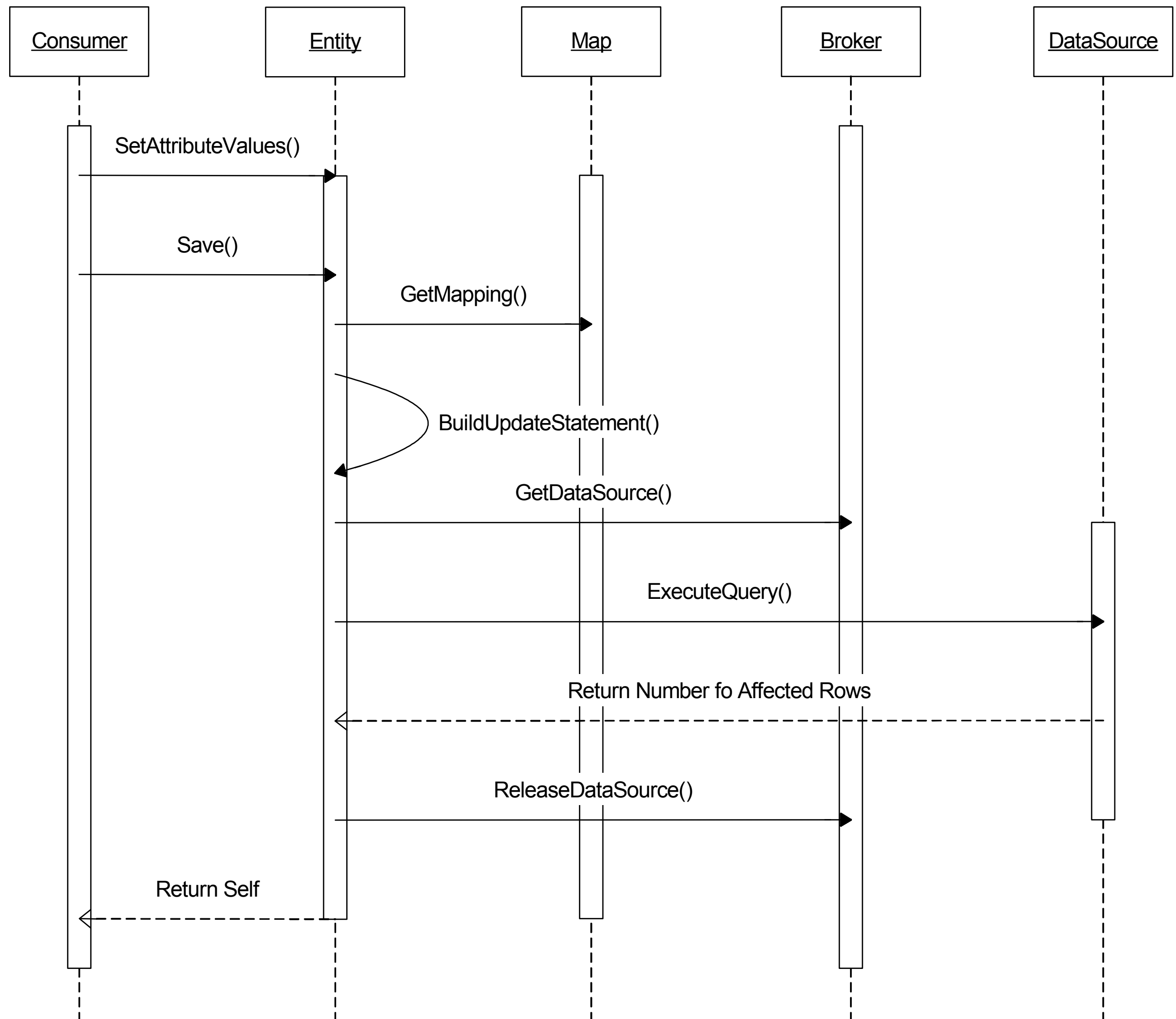
Appendix B.1 – Entity:Retrieve



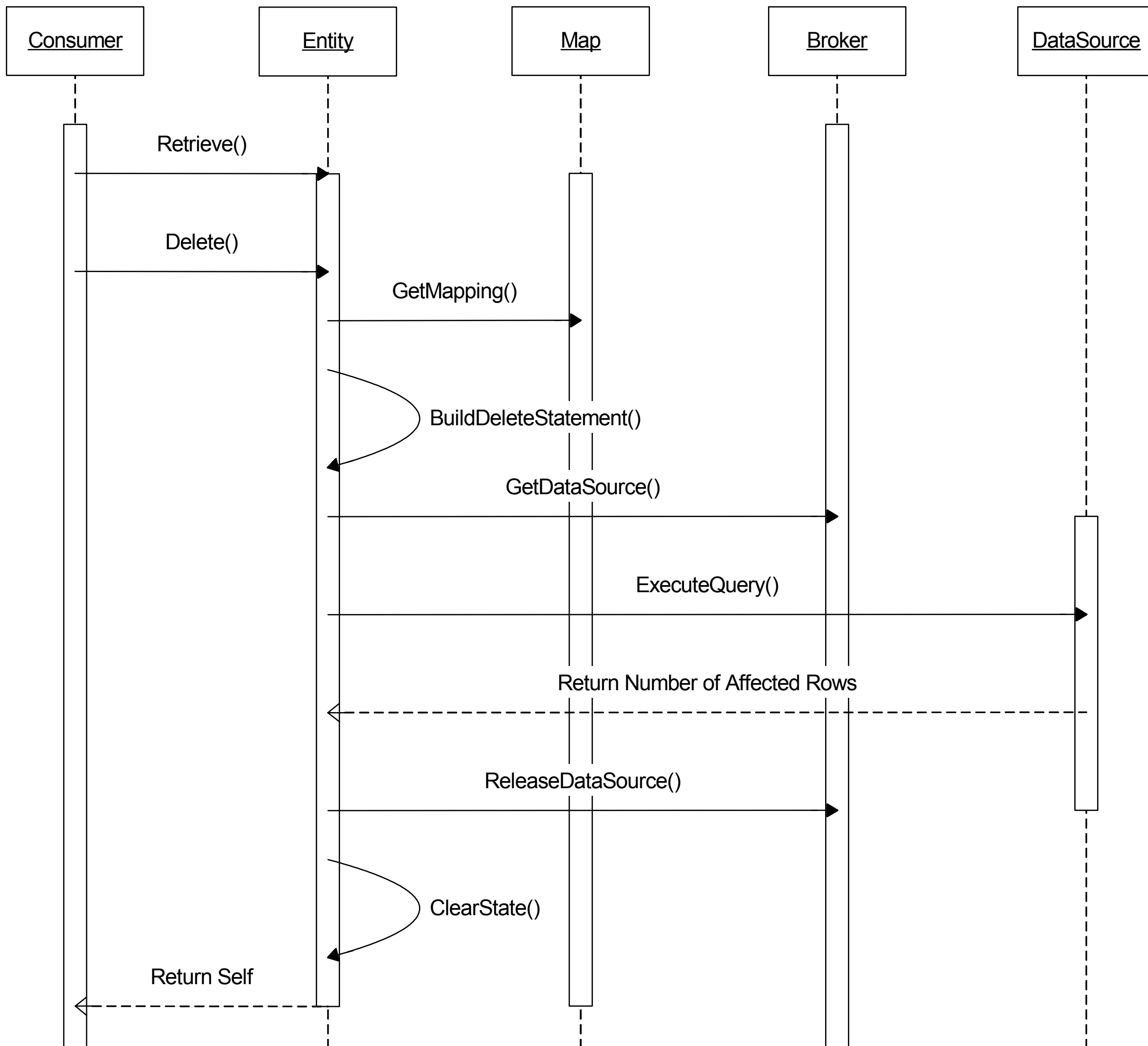
Appendix B.2 – Entity:Save (New Tuple)



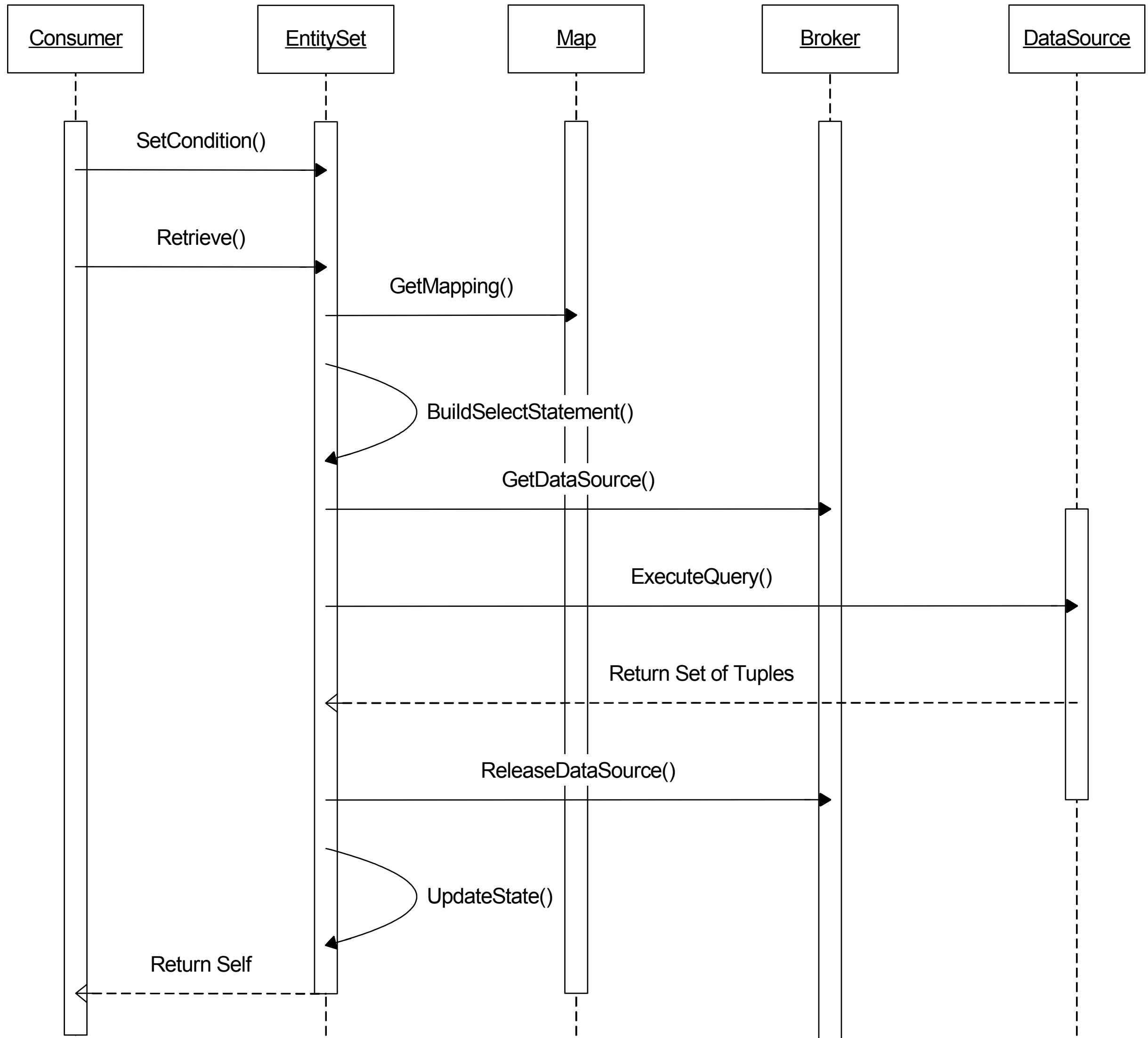
Appendix B.3 – Entity:Save (Existing Tuple)



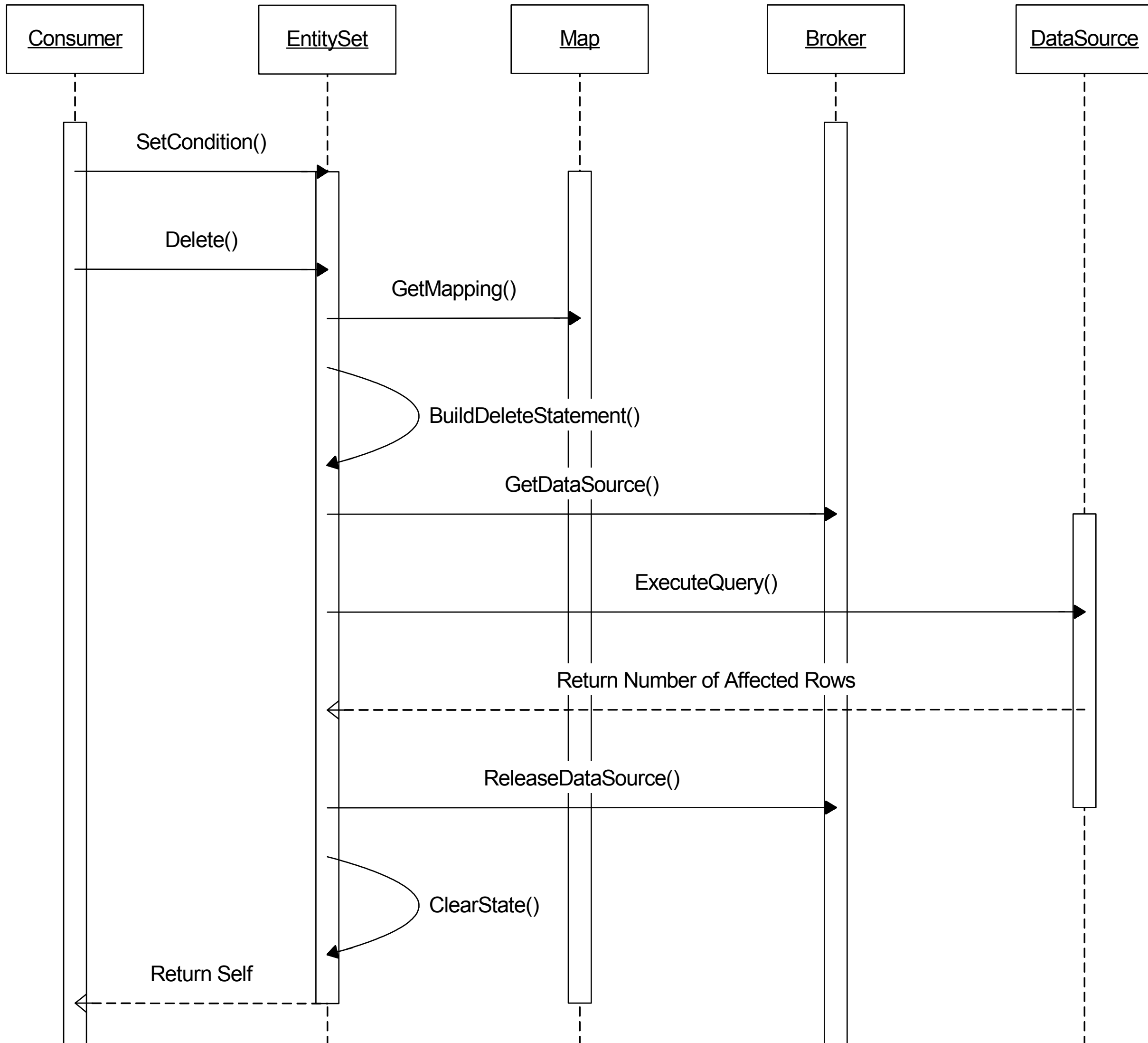
Appendix B.4 – Entity:Delete



Appendix B.5 – EntitySet:Retrieve



Appendix B.6 – EntitySet:Delete



Appendix C – Interface Contracts

Appendix C.1 - Entity Behaviour

Contract Entity:Retrieve

Operation:	Retrieve()
Responsibilities:	
Cross Reference:	None
Exceptions:	If no conditions was set, or data source could not be contacted an exception was thrown.
Preconditions:	Conditions were set for one or more attributes and the connection string to the data source was loaded.
Postconditions:	Attribute values became attribute values from the retrieved tuple and key attribute values were set as read-only.

Contract Entity:Save (New Tuple)

Operation:	Save()
Responsibilities:	
Cross Reference:	Save – update existing tuple
Exceptions:	If key attribute values was not set and the key is not auto-generated, or data source could not be contacted an exception was thrown.
Preconditions:	A value was set for one or more attributes values and the connection string to the data source was loaded.
Postconditions:	The tuple was made persistence in the appropriate data source and key attribute values were updated with the generated key, if the key is set to be auto-generated. Key attribute values were set to read-only.

Contract Entity:Save (Existing Tuple)

Operation:	Save()
Responsibilities:	
Cross Reference:	Save – new tuple, Retrieve
Exceptions:	If the data source could not be contacted an exception was thrown.
Preconditions:	One or more attribute values were changes sine last retrieval. Since the entity was retrieved before this update, it could be assumed that the connection string was set correct.
Postconditions:	The entity was retrieved and attribute values became attribute values from the updated tuple.

Contract Entity:Delete

Operation:	Delete()
Responsibilities:	
Cross Reference:	Retrieve
Exceptions:	If the entity did not represent a tuple or the data source could not be contacted an exception was thrown.
Preconditions:	The entity was retrieved at some point and represented a tuple. The connection string to the data source was loaded.
Postconditions:	The tuple was deleted from the data source, and the attribute values e.g. object state was cleared.

Appendix C.2 - EntitySet Behaviour

Contract EntitySet:Retrieve

Operation:	Retrieve()
Responsibilities:	
Cross Reference:	None
Exceptions:	If the data source could not be contacted an exception was thrown.
Preconditions:	Conditions were set for none or more attributes and the connection string to the data source was loaded.
Postconditions:	A set of tuples was loaded from the data source.

Contract EntitySet:Save

Operation:	Save()
Responsibilities:	
Cross Reference:	Retrieve
Exceptions:	If the data source could not be contacted, an exception was thrown.
Preconditions:	A selection was retrieved and one or more attribute values were set. Since the entity was retrieved before this update, it could be assumed that the connection string was set correct.
Postconditions:	The selection of entities was updated and the attribute values were set for each entity to contain the attribute values from the updated tuple.

Contract EntitySet:Delete

Operation:	Delete()
Responsibilities:	
Cross Reference:	None
Exceptions:	If the data source could not be contacted an exception was thrown.
Preconditions:	Conditions were set for none or more attributes and the connection string to the data source was loaded.
Postconditions:	A set of tuples was deleted from the data source.

Appendix C.3 – EntityCursor Behaviour

Contract EntityCursor:First

Operation:	First()
Responsibilities:	
Cross Reference:	None
Exceptions:	If the data source could not be contacted an exception was thrown.
Preconditions:	The connection string to the data source was loaded.
Postconditions:	An Entity containing the first tuple from the mapped relation was returned. The state was updated with a reference to this tuple.

Contract EntityCursor:Last

Operation:	Last()
Responsibilities:	
Cross Reference:	None
Exceptions:	If the data source could not be contacted an exception was thrown.
Preconditions:	The connection string to the data source was loaded.
Postconditions:	An Entity containing the last tuple from the mapped relation was returned. The state was updated with a reference to this tuple.

Contract EntityCursor:Next

Operation:	Next()
Responsibilities:	
Cross Reference:	None
Exceptions:	If the data source could not be contacted an exception was thrown.
Preconditions:	The connection string to the data source was loaded.
Postconditions:	An Entity containing the tuple indexed next to the previous referenced tuple in the mapped relation was returned. The state was updated with a reference to this tuple.

Contract EntityCursor:Previous

Operation:	Previous()
Responsibilities:	
Cross Reference:	None
Exceptions:	If the data source could not be contacted an exception was thrown.
Preconditions:	The connection string to the data source was loaded.
Postconditions:	An Entity containing the tuple indexed previous to the previous referenced tuple in the mapped relation was returned. The state was updated with a reference to this tuple.

Appendix C.4 - EntityTransaction Behaviour

Contract EntityTransaction:Commit

Operation:	**Commit**()
Responsibilities:	
Cross Reference:	Entity
Exceptions:	If a problem was reported by an added entity an exception was thrown
Preconditions:	One or more entities were added to the transaction.
Postconditions:	The added actions were completed.

Contract EntityTransaction:Retry

Operation:	Retry()
Responsibilities:	
Cross Reference:	None
Exceptions:	If a problem was reported by an added entity an exception was thrown
Preconditions:	A Commit() was performed with one or more errors.
Postconditions:	The added actions were completed.

Contract EntityTransaction:Rollback

Operation:	Rollback()
Responsibilities:	
Cross Reference:	None
Exceptions:	If a problem was reported by the data source during the rollback an exception was thrown
Preconditions:	A Commit() was performed with one or more errors.
Postconditions:	The actions performed by Commit() on the data source were reset back to its original state.

Contract EntityTransaction:Attempt

Operation:	Attempt()
Responsibilities:	
Cross Reference:	None
Exceptions:	If a problem was reported by an added entity, an exception was thrown.
Preconditions:	One or more entities were added to the transaction.
Postconditions:	The actions were attempted on the data source without changing the state of any tuples.

Appendix D – Initialising Metadata in the Framework

Appendix D.1 - Data Schema Reflection in Object Layer

The structure of the automatically generated classes:

```
using System;
using System.Data;
using MBD.DataLib.Entities;

namespace MBD.DataAccess.ODSML
{
   /// <summary>
   /// Summary description for Person.
   ///
   /// Information: This file is autogenerated by DataLib Management Console.
   ///
   /// Copyright (c)2003 Uffe Dejligbjerg, All rights reserved.
   /// </summary>
   public class Person : Entity
   {
      public readonly EntityAttribute Person_ID;
      public readonly EntityAttribute FirstName;
      public readonly EntityAttribute MiddleName;
      public readonly EntityAttribute LastName;

      public Person()
      {
         persistenceName = "ODSML";
         entityName = "Person";
         isReadOnly = false;
         primaryKeyType = PKTypes.Combined;

         InitialiseField(Person_ID =
                  new EntityAttribute("Person.Person_ID", "", "Person", this));
         InitialiseField(FirstName =
                  new EntityAttribute("Person.FirstName", "", "Person", this));
         InitialiseField(MiddleName =
                  new EntityAttribute("Person.MiddleName", "", "Person", this));
         InitialiseField(LastName =
                  new EntityAttribute("Person.LastName", "", "Person", this));

         SetPrimaryKey(Person_ID);
      }
   }
}
```

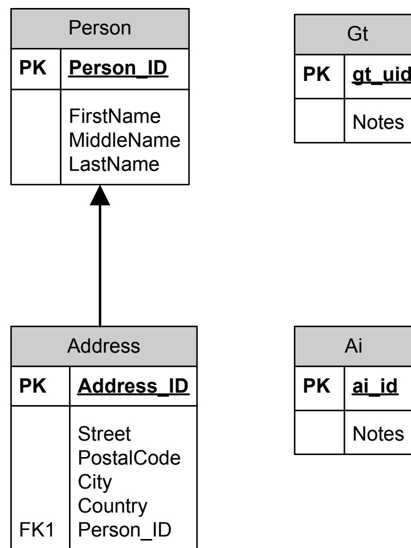
Appendix D.2 – XML Schema for Configuration File

The XML Schema of the DBConfig.xml file is:

```
<?xml version="1.0" ?>
<xs:schema id="Configuration" targetNamespace=”http://tempuri.org/DBConfig.xsd”
    xmlns:mstns="http://tempuri.org/DBConfig.xsd" xmlns=”http://tempuri.org/DBConfig.xsd”
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
    attributeFormDefault="qualified" elementFormDefault="qualified">
   <xs:element name="Configuration" msdata:IsDataSet="true" msdata:Locale="da-DK"
     msdata:EnforceConstraints="False">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="DataSources">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="DataSource" minOccurs="0" maxOccurs="unbounded">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="ConnectionName" type="xs:string" minOccurs="0"
                      msdata:Ordinal="0" />
                    <xs:element name="ConnectionString" type="xs:string" minOccurs="0"
                      msdata:Ordinal="1" />
                    <xs:element name="Namespace" type="xs:string" minOccurs="0"
                     msdata:Ordinal="2" />
                  </xs:sequence>
                  <xs:attribute name="type" form="unqualified" type="xs:string" />
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Appendix E – Data Schemas

Appendix E.1 - Functional Test Database



Appendix E.2 – DAISY Data Model