Implementation and evaluation of data persistence tools for temporal versioned data models

Tor Knutsson

Reg Nr: LIU-IDA/LITH-EX-A-09/032-SE
Linköping 2009
Master’s Thesis

Implementation and evaluation of data persistence tools for temporal versioned data models

Tor Knutsson

Reg Nr: LIU-IDA/LITH-EX-A–09/032–SE
Linköping 2009

Supervisor: Pascal Muller
Wealth CCC S.A

Examiner: Lena Strömbäck
IDA, Linköping University
The purpose of this thesis was to investigate different concepts and tools which could support the development of a middleware which persists a temporal and versioned relational data model in an enterprise environment. Further requirements for the target application was that changes to the data model had to be facilitated, so that a small change to the model would not result in changes in several files and application layers. Other requirements include permissioning and audit tracing. In the thesis the reader is presented with a comparison of a set of tools for enterprise development and object/relational mapping. One of the tools, a code generator, is chosen as a good candidate to match the requirements of the project. An implementation is presented, where the chosen tool is used. An XML-based language which is used to define a data model and to provide input data for the tool is presented. Other concepts concerning the implementation is then described in detail. Finally, the author discusses alternative solutions and future improvements.
Abstract

The purpose of this thesis was to investigate different concepts and tools which could support the development of a middleware which persists a temporal and versioned relational data model in an enterprise environment. Further requirements for the target application was that changes to the data model had to be facilitated, so that a small change to the model would not result in changes in several files and application layers. Other requirements include permissioning and audit tracing. In the thesis the reader is presented with a comparison of a set of tools for enterprise development and object/relational mapping. One of the tools, a code generator, is chosen as a good candidate to match the requirements of the project. An implementation is presented, where the chosen tool is used. An XML-based language which is used to define a data model and to provide input data for the tool is presented. Other concepts concerning the implementation is then described in detail. Finally, the author discusses alternative solutions and future improvements.

Sammanfattning

Acknowledgments

Many thanks to Dominique Portal for giving me the opportunity to come to Switzerland and carry out this thesis project. I would also like to thank Pascal Muller for taking the time and showing a great interest in the project and for the many discussions about everything relating to it. Special thanks also goes to to my colleagues Guillaume and Laurent for helping me out with tricky technicalities, and to my friend Marcus Frödin for his support and motivation. Thank you to my examiner Lena Strömbäck for showing patience and providing a lot of feedback when writing this report. Finally, a big thank you to my parents for pushing me and a huge eulogy to Albane for showing patience and motivating me during late nights and weekends.

Genève, May 2009
Tor Knutsson
# Contents

1 Introduction .................................................. 1  
1.1 Prerequisites ................................................. 1  
1.2 Problem statement ........................................... 1  
1.3 Overview .................................................... 2  
1.4 Method ...................................................... 3  
1.5 Glossary .................................................... 3  

2 Literature Study .................................................. 5  
2.1 Overview .................................................... 5  
2.2 Problem description ......................................... 6  
2.2.1 Scope .................................................... 6  
2.3 Software patterns ............................................ 7  
2.4 Object/Relational Mapping ................................. 8  
2.5 Database efforts ............................................. 10  
2.5.1 Object-relational databases ............................. 10  
2.5.2 Object-oriented databases ............................... 11  
2.5.3 Reflections ............................................... 12  
2.6 Temporality .................................................. 12  
2.6.1 Thoughts on revision control ......................... 14  

3 Requirements ................................................... 15  
3.1 Functional requirements ..................................... 16  
3.1.1 Object/Relational Mapping to Temporal and Versioned entities ............... 16  
3.1.2 Permissioning ........................................... 16  
3.1.3 Audit Tracing ............................................ 17  
3.1.4 Adjacent database identifiers .......................... 17  
3.1.5 Variable Cache ......................................... 17  
3.1.6 Compile-time error prevention .......................... 18  
3.1.7 Environment ............................................ 18  
3.2 Non-Functional Requirements .............................. 18
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>The Temporal and Versioned Properties</td>
<td>19</td>
</tr>
<tr>
<td>4.1</td>
<td>Temporal entity</td>
<td>19</td>
</tr>
<tr>
<td>4.2</td>
<td>Versioned entity</td>
<td>20</td>
</tr>
<tr>
<td>4.3</td>
<td>Temporal Versioned entity</td>
<td>20</td>
</tr>
<tr>
<td>4.4</td>
<td>Comparing the definitions with literature</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>Available Tools</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Model Driven Architecture</td>
<td>23</td>
</tr>
<tr>
<td>5.2</td>
<td>Availability</td>
<td>25</td>
</tr>
<tr>
<td>5.3</td>
<td>Coverage, compliance, artifacts</td>
<td>26</td>
</tr>
<tr>
<td>5.4</td>
<td>Discussion</td>
<td>26</td>
</tr>
<tr>
<td>5.5</td>
<td>Conclusion</td>
<td>29</td>
</tr>
<tr>
<td>5.6</td>
<td>Criticism</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>Architecture Proposals</td>
<td>31</td>
</tr>
<tr>
<td>6.1</td>
<td>Layers and Tiers</td>
<td>31</td>
</tr>
<tr>
<td>6.2</td>
<td>Direct Stakeholders</td>
<td>31</td>
</tr>
<tr>
<td>6.3</td>
<td>Database Representation</td>
<td>32</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Linked lists approach</td>
<td>32</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Property separation</td>
<td>33</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Summary</td>
<td>34</td>
</tr>
<tr>
<td>7</td>
<td>Implementation</td>
<td>35</td>
</tr>
<tr>
<td>7.1</td>
<td>Creation process</td>
<td>35</td>
</tr>
<tr>
<td>7.1.1</td>
<td>The modeling language</td>
<td>36</td>
</tr>
<tr>
<td>7.2</td>
<td>Overview</td>
<td>37</td>
</tr>
<tr>
<td>7.3</td>
<td>Code generation</td>
<td>38</td>
</tr>
<tr>
<td>7.4</td>
<td>The mockup model</td>
<td>39</td>
</tr>
<tr>
<td>7.4.1</td>
<td>Database Tier</td>
<td>39</td>
</tr>
<tr>
<td>7.4.2</td>
<td>Database Access Layer</td>
<td>45</td>
</tr>
<tr>
<td>7.4.3</td>
<td>Communications and Helper layer</td>
<td>45</td>
</tr>
<tr>
<td>7.4.4</td>
<td>Persistence Layer</td>
<td>46</td>
</tr>
<tr>
<td>7.4.5</td>
<td>Domain Layer</td>
<td>47</td>
</tr>
<tr>
<td>8</td>
<td>Performance Test</td>
<td>51</td>
</tr>
<tr>
<td>8.1</td>
<td>Environment</td>
<td>52</td>
</tr>
<tr>
<td>8.2</td>
<td>Test data</td>
<td>52</td>
</tr>
<tr>
<td>8.3</td>
<td>Test results</td>
<td>53</td>
</tr>
<tr>
<td>8.4</td>
<td>Criticism</td>
<td>54</td>
</tr>
<tr>
<td>9</td>
<td>Discussion</td>
<td>55</td>
</tr>
<tr>
<td>9.1</td>
<td>Alternative solutions</td>
<td>55</td>
</tr>
<tr>
<td>9.2</td>
<td>Future Work</td>
<td>56</td>
</tr>
<tr>
<td>9.2.1</td>
<td>Dynamic Queries</td>
<td>56</td>
</tr>
<tr>
<td>9.2.2</td>
<td>Extended Properties</td>
<td>56</td>
</tr>
<tr>
<td>9.2.3</td>
<td>Scalability and distribution</td>
<td>56</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

WealthCCC implements and maintains a large financial consolidation system called Atrium. Atrium is a strongly data-driven multi-tiered application which uses relationally stored data from multiple sources, to generate reports, analysis and projections. This thesis investigates and reflects over persistence tools and their different properties, to find out if or how each tool can be of use when creating a middleware solution that can be used by WealthCCC and other enterprise application developers with similar requirements.

1.1 Prerequisites

The reader of this rapport should have a good knowledge of object-oriented programming, database programming, and the concept of relational databases and the relational model. Furthermore, basic knowledge of XML, XML-related technologies and the programming language Visual Basic.NET may facilitate the reading experience.

1.2 Problem statement

The purpose of this thesis is to investigate object/relational mapping concepts, tools, and patterns in order to create the foundation of a generic middleware layer for a relational database that handles persistent data connections. Enterprise applications tend to grow large due to their complexity. To enhance readability and maintainability, the application can be divided into layers. Each layer handles specific tasks. By dividing the tasks this way, the layers become more interchangeable [39]. Secondly, enterprise applications can be divided into tiers. The thought application is divided in at least three tiers. The data model in the scope of this thesis has specific characteristics, such as high data contention, versioned storage and values which vary over time. The need for an automatized tool that handles data persistence arises from maintainability, something which is increasingly complex with each factor mentioned above. A small change to the data model may
result in a large number of changes throughout several layers of an application [16].

The problem statement is as follows:

“What are the requirements for a generic middleware layer providing support for temporal and versioned relational data, and how can it be implemented in a way that facilitates change?”

1.3 Overview

Chapter 1: Introduction

In this chapter, the reader is presented with a quick description of the project as well as the problem statement.

Chapter 2: Literature Study

This chapter consists in a summary of literature relevant to the thesis subject, and serves as an overview as well as supporting arguments for some statements made throughout the thesis.

Chapter 3: Requirements

The requirements chapter is an elaboration of the requirements that was stated by the project initiator at WealthCCC. Many of the choices made in implementation and the comparison is based on the paragraphs in this chapter.

Chapter 4: The Temporal and Versioned Properties

This chapter is a short introduction to the temporal and versioned properties, and also serves as a definition of the concepts.

Chapter 5: Available tools

This is a comparison of a set of Computer-Aided Software Engineering and Object/Relational mapping tools which can aid the development of the product described in the requirements, as well as a conclusion and motivation of which ones that can be useful.

Chapter 6: Architecture Proposals

In this chapter, a few proposals are discussed as both a basis and a comparison to the final implementation.
1.4 Method

This thesis project has been completed through a set of steps. Initially, the requirements of the project has been further evolved, and general proposals for the project architecture has been made. Next, a survey of available tools and libraries has been carried out. Together with a literature study, this material provided a foundation to create a prototype middle ware application, using concepts from different tools. The prototype is built in two phases. The first phase has been to create a mockup middleware application, and in the second step the mockup serve as a specification for an automatized prototype. In the scope of this thesis, the first phase was completed, and the mockup was tested with a set of data from a real-world application.

1.5 Glossary

<table>
<thead>
<tr>
<th>English</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temporal Entity</td>
<td>The property value set of a temporal entity is referring to a period of time, for which the values are valid.</td>
</tr>
<tr>
<td>Versioned Entity</td>
<td>A versioned entity keeps all the past values of properties.</td>
</tr>
<tr>
<td>Black box</td>
<td>A system or an object analyzed by its input and output signals rather than its internals.</td>
</tr>
<tr>
<td>Atomic</td>
<td>A piece of data that cannot be broken into smaller parts.</td>
</tr>
</tbody>
</table>
### Table 1.2. Chapter 5 Glossary

<table>
<thead>
<tr>
<th>English</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQL</td>
<td>Structured Query Language: A language designed to retrieve, store, modify and manage data in relational database management systems</td>
</tr>
</tbody>
</table>

### Table 1.3. Chapter 6 Glossary

<table>
<thead>
<tr>
<th>English</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clustering</td>
<td>The concept of grouping computers or servers together in a way that they form a single computer or server in the perspective of other application components.</td>
</tr>
<tr>
<td>ascii</td>
<td>American Standard Code for Information Interchange: A standard which contains the Americans alphabet and other important characters for human-readable communications and data.</td>
</tr>
</tbody>
</table>

### Table 1.4. Chapter 7 Glossary

<table>
<thead>
<tr>
<th>English</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML</td>
<td>Extensible Markup Language: a text format specification designed to represent, store and transport data.</td>
</tr>
<tr>
<td>Migration</td>
<td>The process of moving from one set of data structures to a new or modified set of data structures.</td>
</tr>
<tr>
<td>SQL Injection Attack</td>
<td>A way of attacking SQL-driven applications by exploiting the fact that both data and instructions is represented as strings and concatenated when communicating with the database from the application.</td>
</tr>
<tr>
<td>normalization</td>
<td>A systematic method of designing databases so that the table structure is free of properties which can lead to data corruption or a loss of completeness and integrity when modifying the dataset.</td>
</tr>
<tr>
<td>serialization</td>
<td>A way of representing data as a binary or textual sequence rather than the object-oriented representation, for storage or transportation.</td>
</tr>
</tbody>
</table>
Chapter 2

Literature Study

This chapter consists in a summary of literature relevant to the thesis subject, which has been studied in order to complete the project. It serves as a broad-perspective overview as well as supporting arguments for some statements made in the thesis.

2.1 Overview

This chapter begins with an elaboration of the problem statement which was presented in the introduction, and also a comment on the scope of the thesis.

In section 2.2, the author presents a couple of software patterns which are relevant to the subject as they relate to persistence and enterprise applications. This section is motivated not only by the ambition to create high-quality software in the implementation, but also to find ways of describing and comparing different tools and techniques. The patterns which are discussed are a collection of patterns which are found in sources [16], [4], [22], [18] which relate to enterprise application development and persistence.

In section 2.3 the author presents the concept of object/relational mapping and a bit of background to why this problem needs to be solved. This is relevant to the thesis because it describes one of the problems that needs to be solved and how some other solutions address it. Hibernate is chosen as an example as it is popular (statistics show that the hibernate package is being downloaded between 80-100 thousand times per month from sourceforge.net since 2005 [36]) and was also found as a basis for comparison in other literature [42].

Section 2.4 describes how database management systems has evolved to address some of the issues mentioned in section 2.4. A couple of examples of databases and what efforts are made to address the object-relational mismatch issue are presented. In object-relational databases the author exemplifies with the Oracle database, because it is the most widely used DBMS. SQL Server is also brought up because it is the DBMS which will be used when the thesis project is implemented (it is the DBMS which is used where the thesis project is carried out). Informix is also described as it is an alleged object-relational database management system.
In object-oriented database management systems db4o and Intersystems Caché is exemplified. db4o diversifies itself by being an open source database which does not use SQL or an SQL-like language to form queries, which makes it good for comparison and it is also mentioned in literature [42]. Caché is a distributed database system which is interesting as it is an object-oriented database which is queryable using an SQL-like language.

Section 2.6 investigates temporality and how this subject is described by authors. A couple of models to represent temporal data which are suggested in literature are presented. These models are included to provide a comparison with the work done in this thesis project.

2.2 Problem description

As stated in the introduction, this thesis aims to investigate object/relational mapping concepts, tools, and patterns in order to create the foundation of a middleware layer for a relational database. Creating such a layer addresses the area of enterprise application development. Enterprise applications is a wide definition but distinguishes itself from other application types in that its major purpose is to process (often large amounts of) data. Examples of other application types which are not enterprise applications are microcontrollers, operating systems or web browsers [16].

The data which is processed by an enterprise application is often based on a model of the business that the application addresses. This model aims to represent the different parts of the business and how these parts interrelate. As the model is the reason for the application to exist, the entire application is influenced by the characteristics of the model. Business requirements frequently change. This results in that the model has to be changed just as frequently to reflect and support the business. Large but also small changes to the data model can influence all parts of the application, from how data is internally represented and interrelated to how a graphical interface displays data [5, 16].

In addition to this rather general description of the problem, the business that is addressed at WealthCCC where this thesis is carried out poses some additional specific problems. The data in this specific business has characteristics which makes it difficult to model; these are so-called temporal and versioned aspects. The data is sensitive in that any specific part of it should not be able to be accessed by anyone who is not permitted to do so. The data is subject to high contention: several instances may access part of the data at a given time - both to read and to write the data. Furthermore, the application should be able to scale gracefully. Finally, there is the general issue of persisting application data in a relational database, and this is influenced by all the other issues mentioned above.

2.2.1 Scope

In this thesis, the author investigates how the impact of these problems can be diminished, by using a bottom-up approach when looking at the structure of enterprise applications. One of the most central parts is persistence, and an effort
is made to investigate how different tools deal with structuring the persistence mechanisms in enterprise applications. Some discussions regarding approaches to scalability was investigated in the beginning of the project, but this was not pursued as it was considered as out of the scope of the thesis. Also, this chapter presents some information on alternative persistence technology such as object-oriented databases, but using such a persistence solution in the implementation was not in the scope of this thesis either.

2.3 Software patterns

A software pattern or a design pattern is a general solution to a problem that re-occur in software engineering. Patterns are general in the way that they does not specify exactly how the pattern should be transformed into code - it is a high-level conceptual solution, to a conceptual problem. This generality leads to the fact that patterns normally has to be re-implemented, as the conceptual solution has a practical solution which differs on a case-by-case basis. Software patterns are regarded as a factor that increases software quality by anticipating problems before they occur, and increasing the maintainability of software. Another advantage is that they provide a vocabulary for concepts of implementation [16]. This section gives an quick overview of design patterns commonly used in the context of persistence, caching and enterprise development. Other high-level pattern approaches are discussed in chapter 6.

Gateway

The gateway pattern is one of the simplest persistence-related patterns. Basically, it consist in the principle of encapsulating the persistence logic from the rest of the application. When using a relational database management system (RDBMS) to persist data, this could be an encapsulation of the database-specific code, such as SQL strings. Using this pattern simplifies a change of persistence method, since all the code related to the specific persistence implementation is encapsulated here.

A sub-pattern of the gateway pattern is the Table Gateway Pattern, which is specific to implementations that build on relational databases. This pattern abstracts access and modification code to a table in a relational database. [4, 16]

Object Mapper/Data Mapper

The pattern consists in implementing an independent mediator between the object in the application and the corresponding persistent representation. The object is independent and isolated of it’s own persistence as the mediating object, the “mapper”, takes care of updating, deleting and inserting data.[16]

Unit of work

The unit of work is a concept pattern which keeps track of changes made to the current dataset. Rather than committing data each time changes are made (to
stay consistent with the database), the unit of work is committed when all changes are done. This reduces the amount of connections made to the database, and may also relieve the programmer of keeping track of changes him/herself.

The unit of work can be implemented by letting the caller explicitly register objects that should be part of the unit of work. Another method is to “hide” registration inside object setter methods, to transparently register affected objects.[4, 16, 22]

Identity Map

Identity Map is a way of keeping track of which objects which has been loaded into memory. A map has to be kept for each set of identities in the application. This prevents objects from being loaded several times, which would not only decrease performance but could also be a potential error source for keeping objects consistent with the database.[16, 22]

Lazy Load

A lazy loaded object appears to contain the data but does not actually contain the data until it is requested. This pattern is a potential memory saver for heavily branched code where the data might not be used at all. The pattern can be implemented in several ways, where the simplest approach is just checking if the value equals null and if so fetch the value. Other, more complex solutions include the virtual proxy, which is the concept of an empty object which looks just the object that should be returned, but any property access triggers loading of the value at hand. [18, 16]

Domain model

Using the domain model pattern, the domain data is modeled together with its behavior in the application. Any object which is significant to the business that the application is conducting is represented as an object and connected to the other objects in the domain. The domain forms a graph of connected objects - for small applications, all these objects may be loaded into memory. For enterprise applications, this is usually not feasible as the object graph would be too large, so the data has to be pushed to and pulled from memory. The memory management is not part of the pattern per se, it is done by lower layers, using tools or patterns mentioned above to perform Object/relational mapping, or using Object-oriented databases [18, 16].

2.4 Object/Relational Mapping

Many applications handle data - enterprise applications do it more or less by definition. Data which is processed by the application resides in the random access memory (RAM). When the application is terminated (weather it is on purpose or not), this data falls out of memory and is overwritten by other applications.
To survive the timespan of the execution of an application, the data has to be *persisted*: it has to be stored somewhere else than in the RAM. Traditionally, this is done using a hard drive or any medium capable of keeping the data without the need for power (the data should also be able to survive power outages or maintenance). The data can be stored directly as a file, using the file system of the operating system, or by a remote operating system on a server dedicated for file. Simple mechanisms for carrying out these operations are provided by most operating systems. However, when data grows complex and large, this solution is inefficient in that it is unstructured and slow. It is difficult to save or retrieve just a specific part of the data.

A solution to this problem is to use a database management system (DBMS) to organize and structure the data of an application. A Relational Database Management System (RDBMS) is such a system which is based on the relational model. The relational model was presented by E.F Codd of the as early as 1970, and is still widely used in application development [33]. As mentioned in the introduction, the reader is assumed to have knowledge of the relational model so this will not be developed any further.

Object/relational mapping (O/RM) is the concept of solving the mismatch between relational data and object-oriented data. The difference in representation of data depends on the complexity of each side of a metaphoric representation wall. Relational data in a relational database resides on one side of the wall. Pure relational data is represented as a set of tables. Each table contains one or more fields. Each field can contains data of a type that is supported by the relational database management system. A set of such data forms a row in the table.

On the other side of the wall we find an object-oriented programming language. Objects can have fields, which hold data of the types that the programming language supports, or types that can be created in the language environment. Objects can inherit behavior, methods and fields from one or several other objects. Objects can also contain methods, a piece of code that can perform any task the programming environment allows it to. Objects can also be manipulated by other objects.

The problem of Object-Relational mapping is addressed by a large number of tools - and the problem is solved in a large number of ways. One way of solving the problem is to avoid it by storing the objects as they are - more about this in section 2.5.2. The reason for the problem to originate in the first place is that the two technologies are based on different paradigms: The object-oriented programming environment origins from principles of software engineering, while relational database technology is based on principles of mathematics [1].

The implementation and complexity of each persistence tool differs greatly depending on what kind of application the tool or framework is targeting. A couple of persistence tools are discussed in detail in [4] (Barcia). Barcia evaluates persistence tools from a manager’s perspective, in a java environment. The tools that are discussed are JDBC, iBatis, Hibernate, OpenJPA and pureQuery. JDBC is provided mostly as a reference; as it not really an object/relational mapper, but simply a way of abstracting the database connectivity in Java. iBatis and pureQuery are described as a simple light-weight approach to object-relational
mapping, as they implement the table gateway pattern; they lack direct support for more complex object models where each object does not directly correspond to a table. These O/RM framework targets smaller applications that has no need for complex object structures. Hibernate and OpenJPA are described as "full object-relational mapping frameworks", as they fully implement the object mapper pattern.

**Hibernate**

Hibernate is an example of an O/RM framework which implements the object mapper pattern to performs it’s task. Hibernate is originally a Java open-source project, but exists as a .NET port called NHibernate since 2005. It aims to provide so-called *transparent persistance*: The classes that the programmer wants to persist does not have to be written in a certain way, implement interfaces or inherit any base class [25]. The object mapper pattern is implemented by letting the programmer specify a mapping file. The mapping file is an xml-based specification for each class that needs to be persisted: each property and relation that should be persisted is specified using an application-domain name and a relational-domain name. At runtime, the mapping file is parsed by Hibernate, which uses this information to persist and fetch objects from the relational database. [25, 22]

The database can be queried using HQL: Hibernate Query language. This is an SQL-like string-based query language which is used to specify the fetching of objects from the relational database. HQL is used together with the mapping specification to generate SQL at runtime, which in the end is what is sent in the query to the relational database server. The mapping file is quite complex and can be used to manipulate the way hibernate generates SQL, and also enables the use of stored procedures [25, 22].

### 2.5 Database efforts

There is an effort to to partially or completely avoid the mismatch between relational data and object-oriented which is mentioned in the previous section. By storing the data as objects, or at least providing an object interface, the application can map directly to the objects. There are quite a few different approaches to this, but the top categories are object-relational/extended relational databases and object-oriented databases.

#### 2.5.1 Object-relational databases

The relational databases that support an object-oriented database model but also works as relational databases are referred to as object-relational databases or extended relational databases. Note that the latter name is also used in some literature to describe object-oriented databases. This functionality is commonly realized by extending the standard datatypes that adhere from the SQL standard.

---

1 More about stored procedures in section 7.4.1
with complex types, but in general the relational way of accessing data would be preserved [33].

IBM Informix is an object-relational database management system. It started out as a relational database application suite in the early 1980s but evolved to an object-relational database in the 1990s when the product was integrated with the technology of a product called Illustra. The informix system enables object-relational functionality, or extended functionality by enabling the developer to describe entities and relations as objects [9]. Paul Brown of IBM describes the development with an Object-relational database such as Informix as a process where the developer can think of things such as state and behavior already in the database layer [7].

The Oracle Database is relational database management system which supports an object-oriented database model. As of 2007, Oracle held the worlds largest market share of relational database management systems[29]. Oracle realizes object-relational features by providing the user with a set of extensions of the SQL standard. The database engine provides the api user with object types: A user can define new types which can be inherited and relate to each other. Oracle also contains direct approaches to O/RM by providing Object views, a virtual object view of relational tables. Other object-relational features include list types and nested fields and tables [33, 12].

The Microsoft SQL Server database also realizes object-relational features by extending the SQL Standard. The database user can create custom data types, not only by using T-SQL (the name of Microsoft’s extended SQL Language); user-defined types can be defined directly in an object-oriented programming language by using CLR integration. However, the Microsoft states that this feature should be use with care as it may have a negative impact on performance [11].

2.5.2 Object-oriented databases

Object-oriented database management systems (OODBMS) go a step further by representing objects in an object-oriented application more or less "as they are". No mapping to relational structure is done, which fully eliminates the object-relational mismatch and any overhead associated with it. Object-oriented databases has not had the same market success as relational or object-relational databases.

db4o is a fully object-oriented database. This means that the application programming interface is always object-oriented - there is no such thing as an object-relation mismatch. This database is primarily used in embedded appliances or as an embedded framework in the application itself. Working with db4o is largely different from using a relational database management system.

Querying the database for objects is done using three different querying systems: Query By Example, Native Queries and SODA. Using Query by Example, the programmer passes an instance of the sought class to the query function, and objects which match the fields that are instantiated are returned. The SODA querying system is represented as a tree which holds the object graph of all stored objects. The user starts with the root and limits the branches of the tree using constraining functions to return objects of certain types and/or objects with members
which holds certain values. Finally, Native Queries is implemented by creating a boolean function of the sought type, which is passed to the query function [13].

Intersysystems Caché is a database management system which provides both a traditional relational api and an object-oriented one. Intersysystems describes Caché as the world’s fastest object database. Caché is primarily used in the healthcare industry, but its usage is also increasing in financial and telecommunication industries [30]. Database queries are done using an object-extended SQL query language, where complex query mechanisms familiar to an SQL programmer such as joins can be replaced by a C-like pointer syntax. Business object that needs to be persisted can be defined directly in the preferred application programming environment. This requires some glue code to persist the objects, but the extent of this depends on what environment is used. A tool called Jalapenjo (Java Language persistence with no mapping) can automate this process when using the Java environment. There are other, semi-automatic tools provided for other programming environments. Objects can also be defined in a special environment provided by the OODBMS. [10].

2.5.3 Reflections

The mismatch which O/RM tools attempts to over bridge is eliminated by products such as db4o and Caché. In caché, an object-to-object mapping is used instead, however this can be done automatically and with substantially with less effort. Some research also points out that significantly better performance can be achieved for some scenarios with object-oriented databases, compared to using a relational database with an O/RM Tool [42].

The mismatch can be partially eliminated using the extensions of object-relational databases. Extensions can also be used to facilitate the mapping process when using an O/RM tool. An example of this is that Hibernate can be configured to use certain stored procedures to persist and query certain objects. [25]

2.6 Temporality

There is a substantial amount of research done in how temporality is best represented in relational databases. This section summarizes a few of the different approaches to representing temporal and versioned data.

In reference [14] (Darwen et al.), the concept temporal [database] is defined loosely as something that "contains historical data instead of or in addition to current data" (p53). Darwen et al. takes small steps to gradually represent data temporally. The first step introduces an entity attribute called since, which holds the time from when an entry is valid. This is later extended to a model where since becomes from and the attribute to is added which represents the time when the entry stops being valid. Next, this is generalized into a new type interval to introduce more details regarding open and closed intervals (if to include the time constraint in the interval or not) and more practical details such as non-arbitrary primary key selection.
Richard Snodgrass has published several papers on the subject of temporal databases. In reference [34] Snodgrass et al. categorizes theoretical databases in snapshot databases, rollback databases, historical databases and temporal databases, and uses the terms valid time and transaction time to distinguish them. The snapshot database would be a plain database with only current data, a rollback database one where previous state can be queried (transaction time), and a historical one which contains data which is valid for different intervals (valid time). The temporal database concept is what evolves from the two latter concepts. Snodgrass introduces a query language for temporal databases, called TQuel (Temporal QUery Language) which should facilitate querying in temporal databases. The language is SQL-like with where-clauses but also contains temporal-specific clauses such as "as of <date>" and "when <object1> overlaps begin of <object2>".

In later research [35] Snodgrass et al. presents a model where each temporal entity is accompanied by both an interval of transaction time and an interval of valid time. Much like in [14], a notion of open or closed intervals is introduced, at least "upwards", with the concepts UC (until changed, for transactional time) and NOW (the current time, for valid time). The difficulties of querying temporal data using a conventional (non-temporal) database management system is also mentioned: using an example of a videotape and a customer which has the relation checked out to, Snodgrass et al. claims that specifying the valid time requires four select statements unioned together. Putting constraints that verifies the integrity of the temporal properties would make things even worse in terms of complexity, especially when the relation is more complex. Snodgrass et al. expresses a frustration that the research area is not taken more seriously, and that much research is based on assumption that application would be built with not-yet-existing future databases with a new temporal model. Furthermore; two approaches is identified to solving the problem practically: an integrated approach (modify the underlying DBMS) and a stratum approach (build a temporal-managing layer or models on top of a non-temporal DBMS).

In [38], Tansel develops the temporal model further using primary keys of temporal relations and other constraints to form integrity. The focus in this article is on what is referred to as valid time above. An interesting part of this article is the two concepts Attribute Time Stamping and Tuple Time stamping. Attribute Time Stamping assumes that the developer has the ability to express her/himself in terms of interval-marked values when creating the data model. This would apply for a primary key field as well a field in a row: The value is attached to an interval, using a similar notation as the one mentioned in [14]. Using this concept, each field in a row could have multiple values, each paired with an interval. To complement this, Tansel presents the same data using Tuple Time Stamping, which represents the data by adding a Start and an End field to each temporal property. This shows that one row using Attribute Time Stamping with \( n \) values, each separate in their validity of time, would result in \( n \) tables when represented in Tuple Time Stamping. Worth noting is that Tuple Time Stamping follows the stratum approach mentioned in [35], while Attribute Time Stamping rather follows the integrated approach.
2.6.1 Thoughts on revision control

At a first glance the versioned\(^2\) concept might imply that a revision control system could be used to achieve a notion of version. The author has not considered this category of tools or concepts in this thesis. A revision control system manages different revisions or versions of data and code. Snodgrass mentions similar functionality in the database itself, when referring to the *rollback database* concept [34]. Both of these approaches practically treats former values of data as information which is not to be regarded anymore: it does not allow present and former data to coexist. This means, for example, that creating a query that compares current values with former ones would be very impractical.

\(^2\)The versioned concept is more explicitly defined in chapter 4, however it is similar to *transaction time* which is discussed in this section.
Chapter 3
Requirements

In this chapter, the requirements of this project are described. The requirements is an elaboration on high-level requirements stated by the project initiator. The first formulation of ideas from the project initiator was to specify, design and implement a software layer in .NET to define persistent objects on top of relational data structures. The following specific requirements were also stated in short:

- Application Level Security
- Caching of objects
- Transaction Management
- Temporal properties management
- Versioned properties management
- The adding of dynamic properties

These requirements were elaborated and developed, resulting in the requirements in this chapter. The dynamic properties clause was decided that it was out of the scope of the project. In section 3.1.1 the problem of object/relational mapping with temporal and versioned entities is described, and what considerations which should be done when constructing a software layer such as this. The following sections 3.1.2 and 3.1.3 discusses security and auditing consideration that has to be made, as the intended data domains often contains sensitive data. The author affirms that it is important to consider who can access the data, who accessed it in retrospective, and in what way. Section 3.1.4 is a further elaboration on the same subject, to ensure that not even application bugs may give inconsistent auditing meta-data.

Cache considerations is presented in 3.1.5: to preempt possible performance issues, a configurable cache layer should be implemented. Section 3.1.6 describes the risks of allowing loosely typed querying, and that this should be avoided if possible. The constraints on the software environment is described in section 3.1.7. In the final section 3.2 the author presents a few visionary key points that should be thought of in this implementation, such as maintainability and preventing repetitive, error-prone software development.
3.1 Functional requirements

As stated in in the introduction, the purpose of this project is to create a middleware solution for persisting data with temporal and versioned characteristics. The application which it is intended for is foremost a multi threaded application server, serving multiple clients with data and calculations. Other fields of usage is not unthinkable, such as a client-side persistence provider.

3.1.1 Object/Relational Mapping to Temporal and Versioned entities

Most enterprise applications today uses a relational database to persist data [29]. This introduces an paradigm mismatch between relational representation and object representation. An object in the application can be represented in many ways in the database depending on the objects complexity, but also on the database. Over bridging the paradigm mismatch is the most important task of the middleware. It relieves a domain programmer\(^1\) of writing repetitive persistence code, so that focus can be on solving business-domain problems instead[16, 4].

An alternative to developing an Object/Relational Mapping tool is to use a different database solution, like object-oriented databases, which are discussed in section 2.5.2. Using anything other than a relational database is out of the scope of this thesis, however it should be considered when constructing a mapping solution, for future work, that the underlying database management system cannot be assumed to be relational.

Databases in general does not natively provide support for versioned and temporal properties. They provide a general interface for storing any kind of data. For a lot of applications the notion of version and time is important, this applies to any system which relies on data that describes real-world entities, since all real world entities vary by time more or less. Examples of applications or data domains that need access to both current and historical data are administrative applications, security systems, medical journal systems, statistical applications, and financial applications[37]. The see Chapter 4 for a definition of the concepts Temporal and Versioned.

3.1.2 Permissioning

The data domains which this project aims to support can be sensitive in the aspect that from a juridical perspective some modifications of data may be inconsistent, misleading or even illegal. Permissioning should be implemented to prevent breach of security and the impacts security breach activities may have on a system. Another reason is for the sake of privacy. For instance, medical and financial applications store information which should not be revealed to anyone but the person who actually has use for the information - information such as a medical journal or a persons stock possession.

\(^1\)The domain programmer role is explained further in chapter 6
3.1 Functional requirements

3.1.3 Audit Tracing

Auditing functionality is best motivated by two factors: security and anomaly identification.

The security factor is that the data domains mentioned above are often sensitive, as mentioned in section 3.1.2. All events that occur should be attachable to a physical person if a person was in fact responsible for the modification (there are scenarios when this is not possible, such as automated maintenance).

Tracking anomalies in a complex application is difficult, as the anomaly can be a consequence of environmental nature (such as a hardware failure), operating system malfunction, application malfunction, application design and application misuse. The audit trace can provide information that isolates an anomaly further.

Automating the audit tracing functionality is motivated by the fact that this kind of functionality is not different depending on the structure of an entity, as the concept simply consists in relating every modification of the data domain to a user. Also, permissioning itself is not enough to protect a database-driven system from intrusion and misuse, it has to be completed with an audit process[31].

3.1.4 Adjacent database identifiers

Every entity should have a unique numeric identifier. This identifier should increase with one for each row that is added. This requirement relates to the previous section audit tracing when discussing anomalies. With a database representation which allows row identifiers which are not adjacent, understanding the order and consequence of events is difficult. There is no way of telling what happened when two non-adjacent identifiers are found. If adjacent entity identifiers is enforced, there can be only one interpretation of the order of which events has occurred. A structure which presents the identifier in a simple and easy-readable way should be implemented.

This requirement differs from the others in that it is very specific. It has a tight relation with an implementation which builds upon a relational database. Actually using a relational database is not a specific requirement for this project, however practically it is the most feasible alternative as it is the current persistence solution of the applications currently developed at WealthCCC, and also the most common way in general to persist data (this is discussed briefly in section 3.1.1).

3.1.5 Variable Cache

A large part of the overhead in multi tier applications reside from the communication between database and application server. This overhead can be avoided or decreased by implementing a database cache: a local copy of parts of the database which is stored in the application server. Implementing a cache will also increase consistency control when implementing O/RM (see section 3.1.1). Using a cache may also decrease the risk of integrity loss when managing multi threaded access to data, by for example using an identity map (see section 2.3).
3.1.6 Compile-time error prevention

Errors must be discovered as fast as possible to avoid production downtime, which has great costs. A great deal of errors can be discovered at runtime by the compiler, by giving the programmer a warning. The more of these errors which can be discovered early, the better. To ensure this, code which is not type-safe should be avoided to the largest extent possible. Examples of non-type-safe code is run-time casted types and string-encapsulated code such as SQL.

3.1.7 Environment

For rational and practical reasons, the middleware layer should be as compliant as possible with the .NET application platform, Visual Studio, and preferably VB.NET. WealthCCC has a large VB.NET code base and breaking this standard is unnecessary if it can be avoided. The development team is familiar with the Visual Studio integrated development environment and tools that are similar to it or integrated into it is preferred.

3.2 Non-Functional Requirements

As stated in section 1.2, the project aims to increase maintainability and facilitate change. These are requirements which are hard to measure but still plays a significant role in the development process. A small change to the data model may result in a numerous changes throughout several layers of an application, and finding a way to avoid this kind of error-prone work is a high priority requirement. An approach to measure this requirement is that a small change to the data model should be sufficient, necessary changes to the code should propagate throughout all affected parts of the middleware.
Chapter 4

The Temporal and Versioned Properties

The concepts temporal and versioned are very central to the discussions in this report, from database architecture to user interface. To provide the reader with a more hands-on understanding of the concepts versioned and temporal, the concepts will be defined and exemplified in this chapter. The definitions constitutes a foundation for discussion on how data that satisfies the definitions can be represented. The examples are written for this sole purpose and should not be interpreted as an implementation or proposals of such, this is rather found in chapters 6 and 7.

Another important aspect of this short chapter is that the words used in literature in this area have several and sometimes contradictory definitions. The definitions made in this thesis differs somewhat from common definitions, mainly due to the jargon at WealthCCC where the thesis project was carried out.

4.1 Temporal entity

A temporal entity is defined as an entity which has one or several properties each referring to a period of time. Consider two arbitrary, separate moments in time, time$_A$ and time$_B$. The values of a temporal entity are considered atomic in the sense that if we behold the entity properties at a time$_A$, we perceive each property as having either no value or one value. Looking at the properties at time$_B$, one or several properties may be different. The property values may however be identical at time$_A$ and time$_B$. An entity with these characteristics is said to belong to a temporal serie.

An example of such an entity could be a description of a person: Let one of the properties be the age of the person, and a second property the job the person is holding (see table 4.1). The age property would have a different value depending on what time the beholder chooses to look at the properties of the person entity.

Notice that an analysis of a set of data represented as a temporal series will yield a result even if the sought time does not equal to an entry in the time field.
Consider the example in table 4.1, and consider a time $t_1$. Where $t_1 \geq t_{\text{time}_2}$ and $t_1 < t_{\text{time}_3}$, the data states that the person is holding the PaperBoy job, and is of the age 15. This is the best analysis possible with this set of data. It is not necessarily accurate, especially considering the age field which is only accurate between $0 - 10\%$ of the values of $t_1$, depending on the values of $t_{\text{time}_2}$ and $t_{\text{time}_3}$.

### 4.2 Versioned entity

A versioned entity is defined as an entity with any kind of structure, but if the entity itself or any of its properties were to change, the old value is retained rather than changed. To distinguish entities or property values from each other, each set of values has a notion of version.

Again, consider a person entity, and consider the entity as part of a government register (see table 4.2). Let one of the properties be the birthplace of the person. Let another property be the version, and let it be denoted by the numbers 1 to $n$. Consider that the register states that the person was born in Linköping, and that the record has version 1. At a tax inspection, it turns out that the person was actually born in Stockholm, but the records were wrong due to a mistake. As the entity is versioned, a new copy of the person record is created. The new record contains the same properties as the last one, except for the birthplace which is now Stockholm, and the version which has now increased to 2.

Assume that the version is represented by an integer as in the above example. This provides a notion of a state of the dataset. The state can be defined as the data the dataset contained when the version counter had a certain value. In the above example, the data can be referred to as of state 1 or state 2.

### 4.3 Temporal Versioned entity

The temporal versioned entity satisfies both of the above definitions. The temporal entity and versioned entity concepts may seem to have very similar properties at a first glance. When beholding each concept separately as a black box, both can be seen as key-value tables, where the time and the version respectively represent the key, and any other properties combined represents a composite value. This fact changes slightly when considering the Temporal Versioned entity.
Consider a door in a building, which is controlled by a security system. The system logs when the door is opened, and when it is closed. Each log entry also has a version field, which states when the log was registered, using a time stamp. Consider the following events on the first of January: The door is opened at 08.00, closed at 08.01, and opened at 10.01, and closed at 10.03. Due to a malfunction of the door sensor, the log states that the door was opened at 08.00, closed at 08.01, opened at 10.02 and closed at 10.03. The day after, at 15.00, the log is verified with the aid of a security camera. The faulty record of 10.02 is discovered, and the record is changed to the correct value 10.01.

After the record has been corrected, the combined data can no longer be correctly interpreted using only the time property. Consider the example in table 4.3, and consider a time parameter $t_2$. Where $t_2 \geq 1 \text{ Jan 10.02}$ and $t_2 < 1 \text{ Jan 10.03}$, an analysis would encounter multiple records, resulting in ambiguous results, breaking the definition in section 4.1. A version parameter must be introduced to get an unambiguous result. Practically, further mechanisms must be introduced to avoid ambiguous results, more about this in section 6.3.
4.4 Comparing the definitions with literature

The temporal property, as defined here, is described in several pieces of literature as valid time or stated time. There are several suggestions of how to represent temporal property. In [14], Darwen et. al. presents several models: a simpler one where a temporal property would be accompanied by a time stamp called since, and the more specific one with the from and to fields. Finally, the concept of an interval with open or closed ends is presented. All the examples in this chapter illustrates something similarly to using a since field to describe a temporal entity, to be as clear as possible, but the concept which finally evolves to an interval in [14] is consistent with the authors definition of a temporal entity. The main difference between the definition in this thesis and the many definitions by Darwen et. al. is that the definition in this thesis does not specify exactly how to implement temporal properties, whereas many concepts in [14] are close to be defined by an implementation. However, Darwen et. al. provides a mathematical definition for stated time as follows: “The stated time for a proposition $p$ is the set of times $t$ such that, according to what the database currently states( which is to say, according to our current believes), $p$ is, was or will be true at time $t$.”

The versioned property has been described in other literature as transaction time or logged time, and is similar to the definition stated in this chapter. In [14], Darwen et. al. states that it is important to stress that the versioned property (transaction time) cannot be changed as opposed to the temporal property (valid time), this is not something that is stated explicitly in the definition which is made in this chapter. Similarly with the temporal entity, it is the authors opinion that this is specific to an implementation rather than the definition. However, the way that the versioned property is defined, the fact that it should not be changed is close to implicit. An example of a situation where a version could be changed is if the version is represented by a time stamp and that the application server was set to the wrong time when records were changed. Similarly to the temporal concept, Darwen et. al presents a mathematical definition of logged time: “The logged time for a proposition $q$ is the set of times $t$ such that according to what the database states or stated at time $t$, $q$ is, was, or will be true.”

Publications by Snodgrass and Tansel ([34, 35, 37, 38]) does not go to deep on the definition of the temporal or versioned (valid time/transaction time) concepts, but rather discusses models and languages to interact with such a property, more about this in section 6.3.
Chapter 5

Available Tools

There are many tools that aim to implement Object/Relational Mapping (O/RM) and related functionality. The tools that have been investigated in this thesis can be divided in three groups: Model driven application frameworks, code generators, and O/RM technologies. The discrimination factor is complexity, output, and control over details. This is illustrated in figure 5.1.

The Model driven application frameworks aim to implement as much as possible from a model - database structure and functionality, O/RM, business object layer, and even interfaces such as web pages or windows forms. This is the category with the highest representation in this survey, because of the non-functional requirements in section 3.2. These requirements are partially fulfilled by the goals of Model Driven Architecture (MDA) [20], which these tools are based on [26, 3, 28, 32, 27, 19]. In this survey Acceleo, AndroMDA, ArcStyler, CodeFluent, Eco and Tangible Architect is looked at.

Code Generators have a scalable approach to what they produce - possibly everything that an application framework does, but fully configurable in terms of language, platform, or technology. In perspective to the application frameworks they can be seen as tools to build them. In this survey CodeSmith and GenWise is looked at.

The O/RM technologies are simplistic, and very general, and therefore hard to compare with Model driven application frameworks and generators. The O/RM technologies provides important insights on best practices on many of the implementation details and patterns used when constructing a middleware layer. Also, both the model driven application frameworks and the generators can take advantage of the O/RM Tools. In this short survey only NHibernate is included as a stand-alone O/RM solution.

5.1 Model Driven Architecture

The requirement discussed in section 3.2 put emphasis on that the middleware should have the ability to change base on a model: It should be model-driven. An effort to this is the MDA software design initiative by Object Model Group
Figure 5.1. Overview of artifacts produced by the three groups

Figure 5.2. Overview of the MDA Process

(OMG), which started in 2001. The initiative consists in a set of guidelines (and future standards) to build software based on model, to separate business logic from platform. The methodology is based on a set of artifacts, some of these are the Platform Independent Model (PIM), the Platform Specific Model (PSM) and the Meta-Object Facility (MOF). The basic concept of MDA is shown in figure 5.2: The PIM is a high-level description of the software, which is created from the requirements. With the aid of a transformation tool, the PIM is turned into a PSM, which, as the name implies, is specific for each platform. This means several PSM’s if there are several platforms. The final transformation is from the PSM to code.

MDA puts high pressure on the PIM to be expressive enough for a transformation tool to produce a PSM based on it. The actual transformations is supposed to be implemented by a tool: They need only to be developed once, and ideally looks
the same in every development cycle. The MOF is a standard that defines the way the models are defined: it can even be used to define itself. It also defines how transformations can be made between different modeling languages. A practical approach to this is the XML Meta data Interchange (XMI) standard, which is also made by OMG. [5, 20]

5.2 Availability

The availability of a tool is a wide concept but there are a few characteristics which are significant: Pricing, licensing, and run-time dependency. The pricing for obvious rational reasons; a product which has a high cost must provide high production gain to be a defensible investment. Licensing may prohibit resale or deployment of a product which takes advantage of the product code. Run-time dependency is a long-term property which is interesting in several aspects: a closed-source third-party runtime dependency is hard to investigate in case of application failure. Also, there is the question of general openness towards law enforcement agency inspections. Table 5.1 shows a short summary of these characteristics for a selection of tools. The deviating contenders are ArcStyler with the high price and CodeFluent and Tangible Architect which does not expose the internals of runtime dependencies. The licenses EPL, BSD and LGPL does not prohibit use in proprietary software [21, 15, 17]. However, the templates or modules that drive Acceleo and CodeSmith have individual licensing.

Note that the run-time dependency field in table 5.1 refers to generated artifacts, not to the generator itself. The prices are as of September 2008.

<table>
<thead>
<tr>
<th>Name</th>
<th>Cost</th>
<th>License</th>
<th>Run-time dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceleo</td>
<td>-</td>
<td>EPL</td>
<td>No, but depends on module</td>
</tr>
<tr>
<td>AndroMDA</td>
<td>-</td>
<td>BSD</td>
<td>Yes, but open source.</td>
</tr>
<tr>
<td>ArcStyler</td>
<td>5000-10000 €</td>
<td>Com.</td>
<td>Yes, source N/A</td>
</tr>
<tr>
<td>Eco</td>
<td>700-10000 €</td>
<td>Com.</td>
<td>yes, but open source</td>
</tr>
<tr>
<td>Tangible Architect</td>
<td>1180-9990 €</td>
<td>Com.</td>
<td>Yes, source N/A</td>
</tr>
<tr>
<td>CodeSmith</td>
<td>360 €</td>
<td>Com.</td>
<td>No</td>
</tr>
<tr>
<td>Genwise</td>
<td>500 €</td>
<td>Com.</td>
<td>No</td>
</tr>
<tr>
<td>NHibernate</td>
<td>-</td>
<td>LGPL</td>
<td>Yes, but open source</td>
</tr>
</tbody>
</table>
5.3 Coverage, compliance, artifacts

The tools offer a differentiated set of produced artifacts. Some produce end-to-end solutions, some specialize in specific parts and some are general enough to do basically anything. An apparent trend is that the more artifacts the tool produces natively, the less broad is the language and application support. To provide an overview of the compliance the tools provide and a base for comparison, a set of abbreviations has been created, presented in table 5.2. Some of these abbreviations are briefly explained in a reference model. The reference model builds upon the relaxed three-layered architecture presented in [39] and on the requirements presented in chapter 3.

The field Native Features in table 5.3 is defined as features which are provided directly using declarative programming either by using a modeling tool or a Domain Specific Language (DSL). This means that a developer can start focusing on the business related problem rather than the programming problem. The contrary would be features that has to be implemented before they can be used in practice, by developing a cartridge or a template, by rewriting generated code, or by writing tool-specific wrapper code.

ArcStyler, CodeFluent, Eco and Tangible Architect aims to offer end-to-end solutions: They implement all or most of the functionality in all the layers of an application; from database schemas to end-user interface. Acceleo, AndroMDA, CodeSmith and Genwise focus on specific parts of the implementation. However, Acceleo, CodeSmith and Genwise are built for and supports any component in any language which is represented by ascii character code.

The most popular target source code language is C#. The compiled output of C# is equivalent of VB.NET [24], so using a tool that generates C# source code does not have to be a problem. However the languages are syntactically different, and readability would decrease if both languages were to be mixed in an implementation in the same layer.

5.4 Discussion

Table 5.3 gives an overview of what support the tools can provide when creating the middleware - but gives a vague decision basis even when combined with table 5.1. Looking at quantity of features, CodeFluent and Eco are two candidates which is conceived by the author as mature and compatible, CodeFluent has the edge of natively supporting VB.NET while Eco has a more competitive price, considering the available run-time source code. Acceleo and AndroMDA are less interesting as they lack support for the preferred environment, discussed in section 3.1.7. ArcStyler is highly priced, but provides a full IDE and the developers of ArcStyler has had a strong belief in the OMG’s MDA philosophy [41], [20]. Tangible Architect provides further interface generation, such as ASP.NET controls. A large benefit of the MDA frameworks is that the code which is generated is well-tested and proven to work, compared to fully-configurable templates used by Code Generators. [26, 3, 28, 32, 27, 19].
### Table 5.2. Abbreviations

<table>
<thead>
<tr>
<th>Abbrev.</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAR</td>
<td>Cartridge: The tool uses a tool-specific code generation specification</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment: The tool is bundled with such an application</td>
</tr>
<tr>
<td>SQL</td>
<td>Sql Server: The tool can integrate with SQL Server</td>
</tr>
<tr>
<td>TPL</td>
<td>Template: Cartridges are built from a set of template specifications</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language: A general-purpose, graphically annotated modeling language</td>
</tr>
<tr>
<td>VS</td>
<td>Visual Studio: The tool is bundled with Visual Studio integration tools</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Markup Language: A general-purpose markup language specification</td>
</tr>
</tbody>
</table>

### Features

<table>
<thead>
<tr>
<th>Abbrev.</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AU</td>
<td>Auditing: The tool has support for an audit trail</td>
</tr>
<tr>
<td>CA</td>
<td>Caching: The tool has support for generating cache functionality</td>
</tr>
<tr>
<td>DBA</td>
<td>Database Access: The tool creates a database access layer</td>
</tr>
<tr>
<td>DOM</td>
<td>Domain Layer: The tool creates a domain layer</td>
</tr>
<tr>
<td>DIS</td>
<td>Distribution: The tool has support for scalability</td>
</tr>
<tr>
<td>NHI</td>
<td>NHibernate: The tool uses NHibernate for O/RM</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface: The tool creates end-user interface artifacts</td>
</tr>
<tr>
<td>PM</td>
<td>Permissioning: The tool generates permission mechanisms</td>
</tr>
<tr>
<td>SVC</td>
<td>Services: The tool generates services interfaces</td>
</tr>
</tbody>
</table>
### Table 5.3. Tool compliance [26, 3, 28, 32, 27, 19, 23, 8, 25]

<table>
<thead>
<tr>
<th>Name</th>
<th>Target Lang.</th>
<th>Native Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceleo</td>
<td>C#</td>
<td>CAR, DBA, NHI, SQL, UML</td>
</tr>
<tr>
<td>AndroMDA</td>
<td>C#</td>
<td>CAR, DBA, UML</td>
</tr>
<tr>
<td>ArcStyler</td>
<td>C#</td>
<td>CAR, DOM, DBA, SQL, UML, IDE</td>
</tr>
<tr>
<td>CodeFluent</td>
<td>VB.NET</td>
<td>CA, DBA, DOM, XML, GUI, IDE, SQL, SVC, VS</td>
</tr>
<tr>
<td>Eco</td>
<td>C#</td>
<td>CA, DBA, DOM, GUI, IDE, SQL, SVC, UML, VS</td>
</tr>
<tr>
<td>Tangible Architect</td>
<td>C#</td>
<td>DOM, DBA, GUI, SQL, UML, VS</td>
</tr>
<tr>
<td>CodeSmith</td>
<td>Any</td>
<td>CAR, DOM, NHI, IDE, SQL, VS</td>
</tr>
<tr>
<td>GenWise</td>
<td>Any</td>
<td>CAR, DOM, NHI, IDE, SQL</td>
</tr>
<tr>
<td>NHibernate</td>
<td>VB.NET</td>
<td>CA, XML</td>
</tr>
</tbody>
</table>

Considering the last requirement stated in section 3.2, the author perceives that the tools Arcstyler, CodeFluent and Eco are the tools that natively fulfills the sense of not having to duplicate data modeling code anywhere - there is one, central place for creating and maintaining the data model. This is especially clear when CodeFluent, which does not provide a graphical interface for creating the model. The only actual input to the tool other than a few configuration options and output choices is the model definition, which is specified in an XML file. Arcstyler offers a complex IDE for defining models using UML, while Eco offers a graphical modeling tool as an add-in to Visual Studio. However, the fact that CodeFluent has a simplistic interface does not mean that Arcstyler nor Eco should be thought less of in this perspective.

When putting the MDA frameworks above in perspective to the latter template-driven generators CodeSmith and GenWise, it is clear that they actually provide for more than stated in the project requirements in chapter 3, and at the same time there is little support for requirements which are not standard procedure. A lot of the requirements are met, but the full focus has not been put on the requirements in this project. There is little or no support for auditing, permissioning and scalability as described in sections 3.1.3, 3.1.2 and 9.2.3. These requirements are too detailed to be met by the MDA frameworks without workarounds.

Another disadvantage with the MDA products is the lack of maturity. Even though some tools generate all the artifacts described in figure, there is still need for manual edition and development of the artifacts produced by the tools [41, 5, 40].

The code generators CodeSmith and GenWise takes a step back from the implementation specifics, and provides aids to create artifacts that produce similar results as the MDA frameworks. What is lost when using code generators instead
of MDA frameworks is the great amount of pre-written, pre-connected cartridges and standardized templates. A part of this loss is regained when using templates which are provided by the generator communities [23, 8].

NHibernate is very lightweight and meets a few of the requirements such as the O/RM and the Caching functionality, as mentioned in section 3.1.5. Despite the light weight, NHibernate is rather complex and is dependent on run-time resolution of types and database connectivity. NHibernate also uses a string-based query language called HQL. These properties are in violation to the requirement to finding as many errors as possible at compile-time (see section 3.1.6). These drawbacks can be encapsulated - alternatively, due to the openness of NHibernate, the implementation ideas and patterns can be re-used in a project that mimics parts of NHibernate, without actually using NHibernate [25, 16]. This is discussed further in chapter 3.

5.5 Conclusion

The code generator is the rational choice, as it does not imply any significant new way of implementation, all the requirements can be met, and the control of what is actually generated is maximized. There are no run-time dependencies other than the ones that are implemented, and large amounts of existing in-project code can be used to create templates. It is not a big difference between the two code-generators, but the author has perceived that there is a larger support community and a larger set of templates available for CodeSmith, which favors this product over GenWise. CodeSmith also have the advantage of a built-in XML parser, which is of great aid when driving a template with data.

Using CodeSmith to generate code in a methodologic perspective to the MDA frameworks is agile: it is an aid which is just barely good enough to complete the task. If more than just barely good enough is produced, then the process is not effective enough. This agile approach argumentation to modeling is presented in the Agile Model Driven Development (AMDDA) approach, as described by Scott Ambler [41, 2].

5.6 Criticism

This survey was done in part by reading about tools from their respective web pages and a few other sources. This means that much of the information may be somewhat biased, which may affect the conclusion of this survey. However, all tools in the survey except Acceleo was installed and tested on a developer machine, but as the tools is severely differentiating in usability, working examples was only produced with a few of these tools. As an example, the author did not manage to make AndroMDA produce anything, while a working example was up and running with ECO in just a few minutes.
Chapter 6

Architecture Proposals

In this chapter, implementation proposals for the prototype are presented. As set of solutions and models has been considered for implementation, and they are collected here as a reference for comparison with the resulting implementation. Also, future work may benefit from seeing alternative implementation suggestions.

6.1 Layers and Tiers

Dividing applications into layers is a common practice that is used in a variety of application fields. It is particularly important in enterprise applications due to their size and complexity. In general, these layers can be equal to or be grouped by three principal layers: The presentation layer, the domain layer and the data access layer [16, 39].

Enterprise applications are also commonly divided into tiers, which are close to or purely physical limits between the integral parts of the application. A common architecture is the three-tiered distribution pattern [39], which consist in a data tier, an application tier and a client tier. Notice that this pattern does not prohibit more than one physical unit per tier; on the contrary clustering and distribution is encouraged.

Figure 6.1 shows an high-level example of how the concepts of layers and tiers can be combined to create an enterprise application where the concerns has been separated.

6.2 Direct Stakeholders

The project has two direct stakeholders: The end-user and the domain programmer. The domain programmer has knowledge of a business; he or she implements business-specific logic, such as mathematical functions or data analysis. The domain programmer implements functionality in the business or service layer. He or she should only need to have knowledge of the interfaces of the layer which is directly under: the domain layer. The domain programmer is also responsible
6.3 Database Representation

Databases in general do not natively provide support for versioned and temporal properties. They provide a general interface for storing any kind of relational data. For a lot of applications, the notion of version and time is very important, this applies to any system which relies on data that describes real-world entities, since all real-world entities vary by time more or less. Examples of applications that need access to both current and historical data are administrative applications, security systems, statistical applications, and financial applications.

In this section, suggestions to standardize the relational table structure in the context of entities with temporal and versioned properties are proposed.

6.3.1 Linked lists approach

This design proposal uses a simple, intuitive approach to representing a versioned temporal entity: All data relating to the entity is stored in the same database table. Table 6.1 demonstrates the table fields and example data. To link entries in the temporal series, a header field ('hdr' in the example in table 6.1) is introduced. 0 would represent a row as a header record that other records could point to. This implies that it is not possible to represent an empty series with this design, which is not necessarily a problem. The version field consists in record entry date. The temporal fields ('start' and 'end' in the example in table 6.1) are redundant - the 'end' field is the same as the 'start' field of the next record in the series. The MadeBy And MadeWhy fields contain the auditing information.

In the example in table 6.1, two separate series are represented. In this example, an entity with three properties data 1, data 2 and data 3 is represented. Notice
that the user Bob has corrected the data in the record with id 2, resulting in the
record with id 3. Also notice that Alice has updated Bob’s entry with id 5. The
latest version of the example in table 6.1 (>= 09.00) consists in two temporal
series, each with two temporal entries spanning from time 1 to 2 and from 2 to 3.

This design breaches the requirement for a consistent and easily-readable series
of row ids (as presented in section 3.1.4), as it is not immediately clear which rows
that other entities may relate to, and every entity does not have an id which has
an adjacent value with the previous entry. The greatest advantage with this design
its simplicity, each entity will always be represented with just one table.

The example also demonstrates a light approach to audit tracing which is
presented in 3.1.3. In practice, the names Alice and Bob would rather be pointers
to a user table.

This one-table solution or something similar is not found in the literature that
has been studied. The models that Tansel et. al. presents in [37] and [38] is
either dependent on a way of expressing several, timestamped values in the same
row for one value, or that several tables is needed to achieve the same thing. The
models which are presented by Snodgrass et. al. in [35] and [34] are based on the
integrated approach described in section 2.6 - and is therefore nothing like linked
list approach.

<table>
<thead>
<tr>
<th>Table 6.1. Linked list approach for arbitrary objects</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arbitrary Object</strong></td>
</tr>
<tr>
<td>id</td>
</tr>
<tr>
<td>----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

6.3.2 Property separation

This design proposes a more complex structure to represent entities. Each entity is
represented by a header table, to satisfy the requirement for a consistent and easily-
readable series of row ids (see figure 6.2). Properties which are not historically
interesting or temporal is placed directly in the header (in figure 6.2 these are
called *Rigid* properties). A temporal header table points to the object header, this
header table contains information about the temporal aspects, such as start time
and end time. Each property is then represented in a separate table, where each
entry is marked with a version.

Using this design would increase the number of joins that has to be done
when querying entity data. For large entities (with a great number of properties)
it decreases direct readability. The design excludes versioning of the temporal
entries. However, each dimension is clearly separated, so any change to an entity property results in normalized storage - for entities with large property fields this would greatly decrease the storage size. For readability, the properties that support audit tracing has been left out of figure 6.2, but these properties would reside on property level.

This solution is similar to the concept Tuple Time Stamping proposed by Tansel et. al. in [38], discussed in section 2.6. The main difference compared with Tuple Time Stamping is that properties can share time intervals using the temporal header. However, the solutions proposed by Tansel et. al. all relate to integrity constraints that is less considered in the property separation model.

### 6.3.3 Summary

The proposed solutions show how the meta-data that needs to relate to the data model with temporal and versioned properties and an audit approach can be represented in different ways. Using an approach with just one table per entity as in the linked list approach would provide a very readable model in a relational database, at least on an overview level. However, it is questionable if the linked list approach is more readable on the detail level since there is a lot of meta data on every row which has to be interpreted, and that a lot of data will be duplicated. The property separation model provides avoids duplication and it is easy to see how each individual property has changed over time. Also, this model complies better with the requirements stated in section 3.1.4. On the other hand, it might not be very important to distinguish when individual properties changed, it is rather important to be able to form one consistent picture of how the entity looks or looked at a point in time. To achieve this, concepts from both of these models should be used in a final implementation.
Chapter 7

Implementation

To concretize the discussions and conclusions from the requirements, the data domain, and architectural proposals, a prototype has been created. This chapter describes how the prototype was created and its characteristics.

7.1 Creation process

The author perceived that it was very difficult to create code generation templates when there was no well-defined example of the output of the generation process. As a result of this, the prototype was built in two phases. The first phase has been to create a mockup middleware application which contains as many variations as possible to cover equally many model use-cases. In this phase, no generator is used; the generation is done manually based on the model. In the second step, this mockup serve as a specification for how to implement the templates of the generator. The process is visualized in figure 7.1. Templates for database tables, about 90% of the stored procedures, data access layer connectors for these stored procedures, and a part of the interface classes were developed before phase one was completed. This was a part of the tool evaluation process. It was also a result of not having realized that the process should look as is does in figure 7.1. Not following this process in the beginning of the project resulted in that the templates became inaccurate quickly.

To create the model a model specification was needed: a standardized way of expressing to a generator what the model consisted in. There are several approaches to what type of model specification to use. A few of the MDA tools that was mentioned in chapter 5 (ArcStyler and Eco) uses graphical UML interface to create the model. CodeFluent uses an Extensible Markup Language (XML) interface to define the model, and an XML Schema Definition (XSD) is used to describe how the XML can be used. [28, 32, 27]

Both methods constitute a way of producing the first artifact which is described by the MDA process: the Platform Independent Model (PIM). In the scope of this project, the modeling language of the PIM should be extensive enough to describe all the artifacts that should be produced[5]. If the MDA concepts is simplified
in the case of CodeFluent, the XML represents the PIM and the XSD represents the MOF. The same method was chosen to define the model in this project. The effort of implementing a graphical tool to define the model would have pushed other aspects of this project aside. As the UML can be represented in XML using XMI to define the limitations of the XML, the practical difference is very small when the model is to be consumed.

7.1.1 The modeling language

The XSD that defines the language to describe the model in this project is practically a modified subset of CodeFluent. It contains the basic elements that is needed to describe a set of entities, their properties and how they relate to each other. Figure 7.2 shows the language definition. The attributes of each artifact enables further customization. Examples of entity attributes include the permisioned property that marks an entity as secure and the temporal property which marks the entity as temporal. More examples can be found in appendix A and B, where the mockup model and the model language are defined.

The implemented language is limited to describing the properties of entities and how they relate to each other. It does not describe any business conducted by the entities. This limitation reflects the requirements of this project as described in chapter 3.

The language of the modeling language (the XSD) is insufficient for completely verifying a model. A small model verification program has been built to validate references to relations and entities not in the model. This program can be extended to provide the model creator with information about the model, such as circular relations.
7.2 Overview

The SQL which is generated for creating tables and stored procedures also contains logic which examines the current content of the database during the migration process. Read more about this in section 7.4.1.2.

7.2 Overview

The prototype implements a subset (figure 7.3) of the enterprise application model described in section 6. This limitation is in based on the limitations of the expressibility of the modeling language, which reflects the requirements of the project. Figure 7.4 shows a more detailed description of the implementation.

The bottom layer is the database tables, each entity is represented as two tables: a header table and a value table. Adding, modifying and reading data from the tables is done by stored procedures in the stored procedures layer. The data access layer provides the application tier with a connection to the database.
The persistence layer is mediating and optimizing the data access layer usage, by caching data and queries. The domain layer provides a programming interface for the domain programmer to read, modify and manage data. The domain programmer does not need to and should not be allowed to modify or take advantage of layer lower than the domain layer.

Knowledge and communications is always done downwards; lower layers does not know about layers which are higher. This makes the layers exchangeable at least from the bottom and up. However, there are some cases where the lower layers need to communicate upwards (see section 7.4.4.2). This is realized using the event model, as described in the observer pattern [18]: As lower layer has no knowledge of upper layer structures, they cannot make method calls. Instead they raise an event, which upper layer can choose to capture or listen to.

7.3 Code generation

As mentioned in chapter 5, the author came to the conclusion that CodeSmith would be a suiting tool aid the development of the middleware layer.

CodeSmith uses templates to drive the code generation. The template language is based on ASP.NET, to provide a familiar interface to web programmers. The approach is in fact very similar to web programming, as the output of a web application is actually code that will be interpreted by an interpreter (in the web case, the browser). Basically, this approach consists in writing the code in the target language in a document, and interrupt the text flow with so-called tags. The tag is a programmatic input, usually for outputting text but also containing control structures to loop a piece of output. In listing 7.1, a part of the code for
The mockup model

A model was made based on real-world entities so that the mockup prototype could be used to produce similar results to a real-world application. This property was used to be able to compare some results to the application after the mockup was completed, more about this in section 8. Figure 7.5 presents the classes in the entity layer which is directly associable with an entity, created using Visual Studio’s Class Diagram tool. The relations between the classes which are defined in the model (see Appendix A) are simply represented as properties in this view.

7.4.1 Database Tier

The database tier consists in tables and stored procedures. Each entity in the model render two tables, which is shown in figure 7.6: A header table and a value table. The header table contains one field which is actual data: The reference id of an instance entry. The other fields are meta-data: who created it, when it was created, if it is deleted and if so, when it was deleted.
Figure 7.5. Entity Layer Class Diagram of Mockup model
The value table contains all the values of the value-type properties each entity has, each represented as a field. It also contains fields that represent the eventual relations that the entity is involved in. If the relation is a one-to-one relation or many-to-one, the field contains the reference id of the other instance entity, that is the reference stored in the header table of the other entity. If it is a many-to-many relation, the relation is represented in a join-table instead: a dedicated table just to represent the relation. However, such a table looks just like any other entity, with the difference that it is not represented in either the XML model or to the domain programmer.

An entity can have several relations to another entity with different cardinality. The other fields are meta-data, describing the status of the entity (actual, updated or deleted), who created it, when it was created, a transaction identifier (see section 7.4.1.3) and fields for manually marking entries as deleted, for maintenance reasons. Temporal entities also have the field "valid from" and "outdated" which refers to real-world time, as described in section 4.1.

The stored procedure is a database-level subroutine. All access to the tables is conducted through a set of stored procedures. Using stored procedures centralizes all database-specific code in the database itself, in contrast to the possibility of writing the same code in the application tier. Besides centralizing the database logic it also improves security slightly by avoiding some common sql injection attacks. Centralizing database logic this way is described in the Table Data Gateway pattern [6, 16].

Listing 7.2. Stored Procedure Body for Querying "Security" Entity

```sql
BEGIN

DECLARE @table_id as int
SELECT @table_id = [dbo]. uf_SecurityHeadersTableId()
SELECT DISTINCT tblh.id as id from SecurityHeaders as tblh
JOIN SecurityValues as tbl on (tblh.__header_reference = tblh.id)
JOIN OWPermissions as p ON (tblh.id = p.[row_id])
JOIN OWPermissionGroupsUsers as pgu ON (pgu.[group] = p.[group_id])

-- Relations
LEFT JOIN CurrencyValues as __Currency ON (_Currency.id = tbl.Currency)
LEFT JOIN PriceValues as __Price ON (_Price.id = tbl.Price)

-- Transaction intervals
LEFT JOIN OWTransactionReference as owtr ON (owtr.header_id = tblh.id)
and owtr.updated <= @simulationDate
and owtr.table_id = @table_id
WHERE pgu.[user] = @user_id
and p.table_id = @table_id
and p.permission >= 0
and (tblh.[__status] = 0 or tblh.[__deleted_date] > @simulationDate)
and (@NameLike is NULL) or (tbl.[Name] = @NameExact)
and (@NameLike is NULL) or (tbl.[Name] LIKE @NameLike)

-- Currency relation
and (@CurrencyId is NULL) or (_Currency.[id] = @CurrencyId)
and (@CurrencyId is NULL) or (_Currency.[id] = @CurrencyId)
and (@CurrencyIsoCodeExact is NULL) or (_Currency.[IsoCode] = @CurrencyIsoCodeExact)
and (@CurrencyIsoCodeLike is NULL) or (_Currency.[IsoCode] = @CurrencyIsoCodeLike)
and (@CurrencyNameExact is NULL) or (_Currency.[Name] = @CurrencyNameExact)
and (@CurrencyNameLike is NULL) or (_Currency.[Name] LIKE @CurrencyNameLike)
and (@CurrencyExchangeRateIsNULL) or (_Currency.[ExchangeRate] = @CurrencyExchangeRate)
and (_Currency.__updated <= @simulationDate) and (_Currency.__outdated is NULL) or ...

-- Price relation
and (@PriceId is NULL) or (_Price.[id] = @PriceId)
and (@PriceId is NULL) or (_Price.[id] = @PriceId)
and (@PriceValueMax is NULL) or (_Price.[PriceValue] >= @PriceValueMax)
and (@PriceValueMin is NULL) or (_Price.[PriceValue] <= @PriceValueMin)
and (@PriceValueMax is NULL) or (_Price.[PriceValue] = @PriceValueMax)
and (@PriceValueMin is NULL) or (_Price.[PriceValue] = @PriceValueMin)
and (_Price.__updated <= @simulationDate) and (_Price.__outdated is NULL) or ...

and (@table_id) as int
and _table_id = @table_id
and tblh.[__updated] <= @simulationDate
and (@table_id <= @simulationDate)
and (_Price.__outdated is NULL) or (_Price.__outdated is NULL)
and tbl.[__outdated] <= @simulationDate
```
Figure 7.6. Implemented Data Model
7.4 The mockup model

```
-- Removes records which are in the TransactionReference for transactional
-- queries
and owtr_id IS NULL
END
```

The stored procedure set bestows in the stored procedures Create, Update, Delete, LoadHeader, LoadHeaders, LoadValue, LoadValues, GetAllHeaderIds, GetHeaderIdsByProperty, GetAllValueIds, GetValueIdByHeader, GetValueIdsByHeader. The Update and Delete functions differs slightly from what one might expect; as both functions actually just adds new records and updates the meta-data in the target record. The Load procedures transfers actual field data to the application. The get functions only returns the identifiers of each table. This avoids unnecessary data transport.

The GetAll procedures are used by the cache mechanism, read more about this in section 7.4.4.

The most complicated stored procedure is the GetHeaderIdsByProperty. This procedure enables the domain programmer to query the database based on relations, and is the stored procedure that is used the most. The complexity is dependent on a depth attribute which is set for the relationproperty on an entity in the model. Depth zero will yield a procedure where the domain programmer can filter an entity based on its properties only. Depth one will yield a procedure where filtering can be made on the entity properties and also on the properties of the related entities. Depth two also enables filtering of the relatives of the related entities, and so on. Listing 7.2 shows an example of how this is realized in case of the entity named security, which has the standard depth of one set on both relations. The variable declaration is left out in this example to keep the example short, but all input variables except the @userid and the @simulationdate are optional.

### 7.4.1.1 Breaking normalization

Filtering the right state of the database (see section 4.2 for a more detailed explanation) is done using lines 43 and 44 in the example in Listing 7.2, as well as lines 29 and 40 for the two relations. The mandatory variable @simulationdate is used to specify out the point in time that represents the state. Notice that the __outdated field overlaps the __updated field of the following value entry, if the value has been updated. This means that the database is not normalized as a redundancy is created. The alternative would be to calculate the right state using a leaf-finding algorithm, by building the value table like a linked list. Test were conducted using an algorithm which would first reduce the dataset to the entries which were older than the simulationdate, and then only return rows which did not have any children rows (rows which were linking to them) in the set which was older than the simulation date (figure 7.7 illustrates this). The algorithm proved to have a significant negative performance impact when using datasets with 100000 rows or more, especially when using many of the optional variables. This motivated the breach of normalization in favor of performance. The decision was also supported by the fact that stored procedures control all modifications of the data, which avoids pitfalls which can possibly compromise data integrity.
7.4.1.2 Meta tables

The meta tables can be seen in the lower part of figure 7.6. Entities which are marked as permissioned in the model are filtered against the @userid variable, see rows 6, 7 and 15 in the example in listing 7.2. Entries are created automatically in the permissions meta table when permissioned entities are created or modified. The permissions meta table is joined with the global groups meta table. By default, the entries are only visible to the user that created them, unless specific grants are given to other users or groups. Each user is a member of a user-group containing only that user, this is the group which is given the permission by default.

The permissions meta table is aided by the structure meta table. The structure meta table contains information about how the database looks: what fields the tables have, and what types the fields have. The structure meta table enables tables to be referred to using a reference id rather than the name. It also enables a way of tracking modifications to the database structure, which simplifies automated migrations to new database structures. The tables are assigned new identifiers each time they change, and the information is saved and related to a specific structure version.

The audit trace meta table keeps a separate log of all modifications to the database. This table contains table reference, action time, the action itself (Create, Update or Delete), reason for update and delete (actually updated values or erroneous entry), the user who made the change, the resulting row and a possible source row. Using the information in the audit log, the reason for anomalies can be easily be traced. As the audit trace meta table contains a table reference, data which is no longer compliant with the current data structure can still be audited, as the information can be retrieved using the structure meta table.

7.4.1.3 Transactions

All data modification is done using transactions. A transaction is a concept which means that a set of modifications will all be carried out or none of them will. The general purpose of transactions is to keep the data set in a consistent state. Transactions are tracked using two transaction meta tables, the transaction table and the header table, which has a column that refers to each ongoing transaction in the transaction table for this purpose.

Only one transaction per table is allowed at one time. This avoids most possible reasons for compromised data integrity due to multiple threads. This means that
concurrent writers are blocked and have to wait until the previous writer has carried out its task. Reading is always allowed - but readers are only allowed to read data which is created at the last time a transaction finished.

Entries which are deleted in header tables during the transaction are stored as references in the header pointer table during the duration of the transaction, as the header table content may still be visible to reader during the transaction.

A lot the transaction-related logic is done in the application tier, read more about this in section 7.4.4.

It is worth noting that in this implementation, the process containing the software presented in this thesis assumes to be the only process which is accessing the underlying database. The introduction of further processes which modifies the same database would require interprocedural communication when it comes to organizing the transaction model as described above. On the contrary, further processes which is limited to read-only access could be introduced without interprocedural communication.

### 7.4.2 Database Access Layer

The database access layer provides higher layers with an interface to the stored procedures. This requires a lot of code, but is not particularly complicated. The layer is necessary to keep higher layers independent of the storage type, this increases maintainability and portability. The database access layer is a typical example of code which is ideal for code generation, as creating it manually is very repetitive and error-prone.

### 7.4.3 Communications and Helper layer

The communications and helper layer facilitates communications between the layers in the application by providing classes which can be used by all layers. This layer is not generated, as it is not specific to what the model looks like. The layer is divided in two parts; one which is known to all layers, and one part which is only known to the internal layers.

The classes basically consist in different extensions of existing primitives in the .NET environment, such as decimals, strings, integers, and lists. Other than facilitating communications it also limits the way the domain programmer can use different types. An example is the database identifier class. The internals of the class is a wrapper around the integer primitive. Methods that require a domain identifier as an argument does not accept an integer: they only accept this class. This forces the origin of the identifier to be an actual identifier and not just a number.

Using a layer which is known by all other layers decreases maintainability somewhat as all layers are dependent on communicating with the artifacts in this layer, which makes layers less exchangeable. It does however increase readability as data structures does not have to be *serialized* when passing data between layers.
7.4.4 Persistence Layer

The persistence layer intercepts data storage and retrieval calls to the database access layer to realize possible optimizations. This is done by storing data in a cache. The persistence layer also operates transaction management and non-generated data management such as user and management.

7.4.4.1 Cache implementation

If and how data should be cached is determined from the model, where three cache-modes are available: always, never and on-demand. Specifying "always" as the cache-mode for an entity results in that the persistence layer always keeps entity instances in memory. This is suitable for high-contention entities which does not constitute an overwhelmingly big memory impact. Defining what is overwhelmingly big is up to the model developer, as this varies greatly depending on the business domain and hardware. The set of entity instances is loaded from the database upon initialization and stays there until the application is shut down. The exception from this rule is deleted entities, which is not kept after they are deleted, to save some memory.

Specifying "never" as the cache-mode means that the entities are never loaded in to the persistence memory, all retrieval operations are passed through directly to the database. This is suitable for entities with low contention and large memory impact, examples includes binary storage of files such as images or documents.

Specifying 'on-demand' means that the entities are loaded into memory first when they are requested. The time that objects stay in memory depends on restraints set at runtime, such as maximum number of entities in memory or a certain time that they are kept without being accessed. This cache-mode is suitable for entities with characteristics which fall in the category between the two above. As it is the most general case, it is the default behavior.

Cached entities are stored in a dictionary structure; a list where a key is used to access items. Each entity-table in the database is represented by such a list in the persistence layer. Each entity which is marked with cache-mode 'always' or 'on-demand' is represented by two lists: one for the header table and one for the values table. The key consists in the database identifier for each record. If the entity is marked 'on-demand', a lookup of entity data is always done on the dictionary first. If the data is in the dictionary, this data is returned. If not, the data is fetched from the database. This implementation is described in the Identity Map pattern [16]. The identity map pattern is also used by NHibernate to cache entities [25].

The domain programmer uses the Find function to find entities with specific characteristics and relations (more about this in section 7.4.5). The answer from the database is a set of database identifiers. With each such method-call there are several optional parameters but the simulationdate parameter is always passed. This parameter defines a certain state of the database. This results in that the answer to a query containing the simulationdate parameter always render the same answer if the parameters contain the same data. This effect is exploited by the query-cache, which stores the result of such a query in a query-cache. The
query-cache consists in a dictionary similarly to the normal cache, but uses the serialized parameters as the key. Each time a query is performed, a lookup is first done in the query cache to see if the result is there.

As both the query parameters and the database answer is relatively small, the query cache can contain a very large number of cached queries without causing a severe memory impact.

7.4.4.2 Transactions

To store data, the domain programmer has to initiate a transaction. The transaction is attached to the thread that initiates it, and to one or several entities. Each retrieval and storage action is controlled by the persistence layer for the presence of an ongoing transaction. If the thread is not bound to a transaction, storing data is not allowed and reading is limited to the last time something was stored in the database. If the thread is attached to a transaction, storing of the attached entities can be done, and reading is not limited to on the attached entities.

Several transactions can commence at the same time, but never on the same entities. This is controlled using locking: an operating-system lock which prevents several threads to execute code which is protected by such a lock.

Transactions can fail, due to conditions set by the domain programmer or due to exceptions. Changes done to the entities during a transaction is stored in a transaction log structure. In case of a transaction failure, the eventual changes done by the transaction is undone, this is also called a \textit{rollback}. A successfully executed transaction is called a \textit{commit}. The persistence layer executes the necessary operations on the cache and the database after a transaction has succeeded or fail. A rollback or a commit of an entity is communicated to upper layers by raising an \textit{event}, as the persistence layer is not aware of upper layers.

7.4.5 Domain Layer

The domain layer is the application program interface that the domain programmer uses to store and retrieve information from the database. Each such operation has to be initiated by creating a \textit{domain object}. The domain object is specific to each database that the application is connected to. Several databases can be used at the same time, but objects cannot relate across databases. This way, two or more instances of the same data structure on separate databases can coexist in an application, for example holding the data for two different companies.

Related objects can be found simply by using the generated method for the specific relation. However, if not all the related objects are desired, a query has to be done. The query is done using the domain object. Both these ways of finding objects are shown in listing 7.3, where the market value of a set of portfolios is calculated. On Line 11 an example is shown on how a string argument can be used: the \texttt{Portfolio\_ShortnameLike} argument contains the SQL wild card character \% to find the desired related portfolios.

As all operations needs to be traced to a person, but also to enable permissioning, all queries are done using a reference to a user. Automated operations can be
done using a special system user. Specific permissioning implementation such as authentication or password policies etc is left to the domain programmer to implement. The prototype contains a helper class for integrating with a Lightweight Directory Access Protocol (LDAP) catalog, which can be used to retrieve user and group information from the Microsoft Windows operating system among others.

Each entity defined in the model is represented by a class in the domain layer. These classes does not have public constructors. New entities are created by using a create function on the domain object - this way there is no doubt what domain the domain programmer is working with - an important distinction if an application is working with several domains. An entity can be marked as a propertyentity. Propertyentities exist only as a property of one or several other entities, as they do not make sense by themselves. This distinction is only made at domain level, where the domain programmer prohibited of creating such an entity - in lower layers, these entities are treated just like any other entity. Propertyentities are created automatically when creating other entities which has a property of the concerned propertyentity type.

```
Listing 7.3. Simple retrieval operations using domain layer artifacts

' Get a domain object
Dim AcmeDomain As Domain = New Domain(New Server('sqlserver.mycompany.com'), "ACMEDatabase")

' Get a user
Dim AliceUser As OwUser = AcmeDomain.OWUsers.FindUserByName("Alice")

' Set the domain standard version mark
AcmeDomain.SetMark()

' Find all the positions of portfolios that alice has access to,
' that begins with Customers
Find(User:=AliceUser, Portfolio_ShortnameLike:="Customer%", )

' Calculate the market value for all positions
Dim AlicesValues As Decimal = 0D
For Each p As Position In AliceCustomerPositions
    For Each t As TransactionCollection In p.Transactions
        For Each t As Transaction In t
        Next
    Next
Next

System.Console.WriteLine("Market value of customer portfolios managed by {0}, as of 1 december 2008: {1} USD", AliceUser.Name, AlicesValues)
```

### 7.4.5.1 Transactions

To store data, the domain programmer has to initiate a transaction. The transaction is attached to the thread that initiates it, and to one or several entities. All the entities that are affected must be attached - an exception to this rule is entities which are marked as propertyentities. The propertyentity is automatically locked when a class with such a property is locked for a transaction.

The domain programmer is responsible for rolling back and committing transactions. A transaction always has to be finished with either a commit or a rollback. Both the commit and the rollback calls result in a raised event by the persistence layer. All objects created or modified during a transaction listen to this event. Affected instances can roll back to the state they were in before the transaction started. As this is a very expensive operation for both memory and processing power, this feature has to be explicitly expressed by the domain programmer.
7.4 The mockup model 49

Listing 7.4. Saving entities using a transaction

```vbnet
' Get a user
Dim AliceUser As OwUser = AcmeDomain.OWUsers.FindUserByName("Alice")
' Start a transaction
Using tx As OwTransactionManager.LockedTransaction = AcmeDomain.StartTransaction(New Entities() {Entities.Portfolios, _
  Entities.Clients}, AliceUser)
Try
  Dim usd As Currency = AcmeDomain.Currencies.Find(user:=AliceUser, IsoCodeExact:="USD").First
  Dim clientBob As Client = AcmeDomain.Clients.Create(BillingState:=0, _
    City:="Stockholm", _
    ContractsDate:=DateTime.Now, _
    DefaultCurrency:=usd, _
    ShortName:="Bob")
  Dim bobsFirstPortfolio As Portfolio = AcmeDomain.Portfolios.Create(Client:=clientBob, _
    Custodian:="BobsCustodian", _
    DefaultCurrency:=usd, _
    EndExistingDate:=DateTime.MaxValue, _
    ShortName:="BobsPortfolio", _
    StartExistingDate:=DateTime.Now, _
    Type:=PortfolioType.Type.Advisory)
  bobsFirstPortfolio.Save(AliceUser)
  tx.Commit()
Catch ex As Exception
  tx.Rollback()
End Try
End Using
```

Listing 7.5. Example of the validation interface

```vbnet
Partial Class ExchangeRate
  Partial Class Validation
    #Region "Field-wide validation"
    Public Sub ValidateRate(ByVal Rate As OWTypes.Nullables.OwDecimal, _
      ByVal valType As ValidationType) Implements IExchangeRateValidation.ValidateRate
    End Sub
    Public Sub ValidateValidFrom(ByVal validFrom As OWTypes.Nullables.OwDateTime, _
      ByVal valType As ValidationType) Implements IExchangeRateValidation.ValidateValidFrom
    End Sub
    #End Region
    #Region "Domain-wide validation"
    Public Sub ValidateEnvironment(ByVal instance As ExchangeRate, _
      ByVal domain As Domain, _
      ByVal valType As ValidationType) Implements IExchangeRateValidation.ValidateEnvironment
    End Sub
    #End Region
  #End Region
End Class
```

7.4.5.2 Entity-level cache

All of the public properties of an entity class instance is undefined at class instantiation - the only known data is the relationship to the domain, and the entity identifier. As long as the entity is not changed, neither is this fact; calls to read the properties simply redirects the request to the persistence layer, through the domain. This means that as long as no properties change, entity instances have a very small memory impact.

When a property of an entity is changed, the change resides in the class until it is saved - this is the small entity-level cache. There is one variable for each
property that can be changed (see listing 7.6). The entity level cache is actually not a cache by definition. It is not persistent, and only exists during the execution of the thread (the scope of a thread spans a function call, and is available only for one single client). It does, however act somewhat like a query cache when it comes to one-to-many relations. In this case, the collection of foreign keys which are returned when an entity resolves a relational property, are saved in an object for future reference (see listing 7.7). Of course, a new entity instance class is instantiated and kept for one-to-one relations as well, but in this case there is no need for a database query.

Listing 7.6. The Type property of the Price entity

```vbnet
Public Property Type() As PriceType
    Get
        If Saved Or (_type Is Nothing And Persisted) Then
            Return _storage.GetValueRow(Id).Type
        End If
        Return _type
    End Get
    Set(ByVal value As PriceType)
        SaveState() ' Save the state for an eventual rollback.
        If Persisted AndAlso Status <> OWTypes.Enumerations.RowStatus.Actual Then
            Throw New OWTypes.UpdateNotAllowedException
        End If
        _type = value
        Saved = False
    End Set
End Property
```

Listing 7.7. The Transactions property of the Position entity

```vbnet
Public Property Transactions() As TransactionCollection
    Get
        If _Transactions Is Nothing And Persisted Then
            _Transactions = New TransactionCollection(_domain, Me, _simulationDate)
        End If
        Return _Transactions
    End Get
    Set(ByVal value As TransactionCollection)
        SaveState() ' Save the state for an eventual rollback.
        If Persisted AndAlso Status <> OWTypes.Enumerations.RowStatus.Actual Then
            Throw New OWTypes.UpdateNotAllowedException
        End If
        _Transactions = value
        Saved = False
    End Set
End Property
```

7.4.5.3 Validation

Simple validations of entities can be defined in the model and be generated but more complex ones require manual programming. The interface for creating validations is a partial class with empty function bodies, which implements an interface. Using an interface forces the class to contain the functions for validation or a compile-time error will be generated. The manually created validation code is clearly separated from the generated entity code this way, but has more access to its internals as it actually resides in the domain layer. Validation is divided in three parts, where each part has different levels of knowledge of its surroundings, an example of this is shown in listing 7.5. The validation type can be either of delete, save or update, the functions in the validation interface are called by the entity class each time such an event happens.
Chapter 8

Performance Test

To test the calculation performance of the prototype, an implementation of a real-world problem was made. The market value of a client portfolio was calculated using Atrium, and the same data was loaded into the prototype in an attempt to make the same calculation. The market value is calculated from all the transactions\(^1\) which make up a position. A position is the concept of possessing a part of a security. This calculation is rather straight forward, but a portfolio with a large number of transaction was chosen, so that thousands of transactions had an impact on the final result.

The test which was implemented is displayed in 8.1. An iteration is made over each position with the security type equity and the portfolio client id (the query at rows 14-16) of the client which was fetched at at row 4-5. The query to find the positions is carried out using parameters with \textit{indirect} relations with the position: \textit{SecurityType} is a property of the \textit{Security} which relates to the Position, and \textit{Client} is an entity which relates to \textit{Portfolio} which relates to the position. The query mechanism is described in more detail in section 7.4.1, and the mockup model is described in figure 7.5 and Appendix A.

Rows 23-26 can be perceived as somewhat strange. This is due to an optimization: In this solution, an iteration over the available prices for a security is done to find the appropriate date we are looking to calculate the market value for. This could also be done by querying the database for this price, using the date as a query parameter using the \textit{Find} function (much like in rows 14-16). The in-memory sorting and comparison was chosen because it improved the speed compared to querying the database for the price index in each iteration.

Finally, the quantity which has been bought or sold (and a few other possible actions yielding positive or negative quantity) is accumulated by iterating through all the transactions of the position. This quantity is multiplied with the price which was found earlier. Finally, the result from the current position is accumulated with the totals of all the previously calculated positions.

\(^1\)The transaction \textit{entity} in the mockup model, not to be confused with the \textit{concept} of transactions when saving data using the prototype
8.1 Environment

It should be taken into consideration that this performance test was for one user, during low database load, without concurrent writing threads, all which are things that could give a false positive effect on the results. On the other hand, the tests were run using a debug environment without compiler optimizations and with IDE debug bindings, which could possibly yield false negative effects.

8.2 Test data

The test data was selected using a client portfolio in Atrium, the financial consolidation system developed by WealthCCC. The entries involved in calculating this number resides in three tables in the mockup model: Positions (about 1.400 entries), the securities (about 9.800 entries), the transactions (about 2400 entries) and prices (about 175.000 entries). The transaction entity describe events in time, such as purchase, a sell, or a split, and sum up to a quantity. If this quantity is multiplied with a price, the market value for the position is calculated.

The data used in this experiment was extracted from a live database, using a script to "translate" the data to the mockup model (the mockup model is a simplified version of the one used by Atrium\(^2\)). This means that both the data

\(^2\)Atrium is a financial consolidation system developed by WealthCCC.
content and the size is realistic in relation to the live application. Also, this type of calculation is carried out frequently in Atrium. Each quantitative value in table 8.1 is based on an average of 5 test runs, however there were very little or no anomalies in the tests.

<table>
<thead>
<tr>
<th>Transaction Entity Configuration</th>
<th>Total execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>First run</td>
</tr>
<tr>
<td><strong>Cache Mode</strong></td>
<td><strong>Query cache</strong></td>
</tr>
<tr>
<td>Never</td>
<td>Off</td>
</tr>
<tr>
<td>On-demand</td>
<td>Off</td>
</tr>
<tr>
<td>Always</td>
<td>Off</td>
</tr>
<tr>
<td>Never</td>
<td>On</td>
</tr>
<tr>
<td>On-demand</td>
<td>On</td>
</tr>
<tr>
<td>Always</td>
<td>On</td>
</tr>
</tbody>
</table>

8.3 Test results

The test results are shown in table 8.1. Each value in the table is the medium value from ten test runs, however the results were very similar, differing only in milliseconds. The main performance impact was how the cache of the entity transaction was configured, and this is why these values are displayed in the first two columns.

In the first case, the cache mode was set to never and the query cache was not used. This means that each time a transaction entity is requested from the database, the entity data will be fetched but only this once, and the next time we request it, another round-trip is made to fetch the data again.

In the second case a slight improvement can be seen in the second run as on-demand cache mode is used. This means that objects reside in cache after the first time they have been requested.

In the third case, the always cache mode is used for transaction entities, and the speed improvement can be seen directly. This mode means that data is loaded into the cache when the application instantiates. However, the second run does not show any signs of improvement.

In the fourth case the cache mode is set back to never, and the query cache is turned on. With the query cache turned on, database queries which return indicies (entity identifiers) are never asked twice to the database (unless the cache limit is reached). In the second run, this setting has a big impact on the result.

In the fifth case, the cache mode is set to on-demand. Not unexpectedly, this does not render anything specific. The second run does however yield the best result so far.

In the sixth case all the caching mechanisms which is at the model designers disposal is used, and this renders good results in both cases.
The query cache had the most significant impact on the performance. The significant impact was due to the fact that transaction has a many-to-one relation with positions, which in the generated results in a foreign key in the transactions table. This means that $p\text{.Transactions}$ (row 15 in listing 7.3, which is a simplified version of the calculation algorithm) cannot use the normal cache directly to resolve, a database lookup is needed. Compare with the case one-to-one relation, where $p\text{.Transaction}$ would have resolved directly from the cache, as the foreign key can reside in the positions table.

The performance test shows that at least this part of the mockup model is sufficiently fast to use in a production environment: the speed which was measured in this test is comparable with the speed of the application which is used today at WealthCCC.

8.4 Criticism

It should be noted that the results yielded in the second run for each test is not very useful when using the query cache. In the first run, repeatedly asked questions to resolve entity relations can be reused during the calculation, and the query cache shows its strength. But actually making a second run on the exact same data (rendering the exact same questions for the query cache to take advantage of) is not very realistic - thus, the rapid result in the fourth but even more in the fifth and the sixth case on the second run is not unrealistic - but it is irrelevant.
Chapter 9

Discussion

Creating a middleware layer for temporal and versioned data types is a big project, which has only been started with the scope of this thesis. The code that has been produced, either manually or by the generator, is built from concepts founded both in literature and in products with similar purposes.

The added complexity from the versioned data model gives this prototype a strong advantage in concurrent scenarios: reading is always allowed, even during writes. This also means that writes do not have to wait for readers to finish before the writing can start. The only scenario where a thread is blocked is where two or more threads wants to modify the same entity. During high contention scenarios, this will greatly increase the responsiveness of an application.

The temporal property turned out to be a detail in this project. When developing the entity model in the domain layer it was discovered that temporal entities were likely to be so called propertyentities, as discussed in section 7.4.5. The temporal entity is still important, as it facilitates the model by decreasing one level of relational complexity, and that the entity type is common in the application field. As the temporal properties are 'hidden' in a simple model attribute, the model is easier to understand.

The development of the mockup model has shown that it is possible to centralize the specifications of a data model into one single file. The performance test has showed that it is possible to create a large and model in relatively short time, which functions with performance comparable to production applications. The use of a code generator is crucial not to "drown" in the complexity of a data model when setting up an environment such as the one in the mockup model.

9.1 Alternative solutions

In section 7.1.1 it is mentioned that the SQL which is generated to create tables also contains logic for examining the current database structure. The information which is obtained by the generated SQL in the migration stage could possibly be mined before the actual migration process. This would vastly decrease the amount of SQL needed to create or migrate the database - and also make the script more
readable. It would however require a wider perspective; the generated SQL script can process the current database column by column.

In chapter 6 a few proposals to how the relational data should be represented is discussed. The data model which was implemented was presented in 7.6. The final implementation suggests that two tables is used for each entity; in chapter 6 the proposals has either just one table or several, depending on the number of properties that needs to be persisted. However, the final implementation is adapted not only to entities with temporal and versioned properties, but also to permissions to keep the data secure and transactions to keep it uncorrupted.

In the scope of this thesis, the focus has been on temporal representation in a relational database and on tools that can aid the development: this is where comparisons has been made. Other areas which are still crucial to the implementation has been investigated less throughly, an example of this is the transactional model, which could be improved in terms of adapting better to the data model and integrating better into the application environment. Also, as stated in chapter 2 there are other approaches to persisting data. Using a object database such as Intersystems caché to compare with the implementation in this thesis could be interesting, at the very least from a performance perspective.

9.2 Future Work

This section lists a few requirements which are required for a mature middleware. Due to the small time frame of this thesis project, all desirable features of the middleware cannot be fully investigated and implemented. However, glancing at future requirements at an early stage changes the way that the other requirements are interpreted and implemented.

9.2.1 Dynamic Queries

The way of querying the database in the prototype created in this project is useful but somewhat limited. With the arrival of the Language Integrated Query (LINQ) on the .NET platform, more powerful queries could be made by implementing a run-time interpreter for a subset or all of the LINQ query-tree.

9.2.2 Extended Properties

Extended properties makes an application more dynamic. The concept consists in that properties of an entity which was not known or not implemented at compile time (not the property value, but the property itself), can be added at run-time. This is used to personalize parts of the data structure or to make fast run-time additions to the data structure.

9.2.3 Scalability and distribution

The kind of applications that the prototype developed in this thesis project targets has to be able to scale easily. A distribution layer could be introduced in the
developed application model to facilitate database or cache distribution. This could increase the efficiency of data usage which originates from external sources; as this kind of data is generally read-only it could easily be distributed over several application instances.
Bibliography


Appendix A

The Mockup Model

```xml
<?xml version='1.0' encoding='utf-8'?>
<wf:project xmlns:wf="http://tempuri.org/2008/WealthCCCFluent" model_version='0.4'
project_name='Mockup3' project_version='0.2'>
  <!-- entities -->
  <wf:entity name='Client' persistent='true' permissioned='true' cacheMode='Always'
temporal='false' versioned='true'>
    <! [CDATA[
      A client, which is served by this consolidation system
    ]]> 
    <wf:documentation>
    </wf:documentation>
    <wf:property name='Shortname' type='varchar(100)'/>
    <wf:property name='ContractDate' type='datetime'/>
    <wf:property name='City' type='varchar(100)'/>
    <wf:property name='Billingstate' type='decimal(25,10)'/>
    <wf:relationproperty name='DefaultCurrency' relation='ClientCurrency'/>
  </wf:entity>

  <wf:entity name='Portfolio' persistent='true' permissioned='true' cacheMode='Never'
temporal='false' versioned='true'>
    <! [CDATA[
      A portfolio, holding several securities for a client
    ]]> 
    <wf:documentation>
    </wf:documentation>
    <wf:property name='ShortName' type='varchar(100)'/>
    <wf:property name='StartExistingdate' type='datetime'/>
    <wf:property name='EndExistingDate' type='datetime'/>
    <wf:property name='Custodian' type='varchar(100)'/>
    <wf:property name='Type' type='PortfolioType'/>
    <wf:relationproperty name='DefaultCurrency' relation='PortfolioCurrency'/>
    <wf:relationproperty name='Client' relation='PortfolioClient'/>
  </wf:entity>

  <wf:entity name='Position' persistent='true' permissioned='true' temporal='false'
versioned='true'>
    <! [CDATA[
      A position holds a complex link between Portfolios and securities
    ]]> 
    <wf:documentation>
    </wf:documentation>
    <wf:relationproperty name='Portfolio' relation='PositionsPortfolio'/>
    <wf:relationproperty name='Security' relation='PositionsSecurity'/>
    <wf:relationproperty name='Collateral' relation='PositionCollateral'/>
    <wf:relationproperty name='Transactions' relation='PositionTransactions'/>
  </wf:entity>

  <wf:entity name='Transaction' permissioned='true' persistent='true' cacheMode='Never'
temporal='false' versioned='true'>
    <! [CDATA[
      The transaction holds information on the creation and modification of a
      position, and most of it's properties
    ]]> 
    <wf:documentation>
    </wf:documentation>
    <wf:property name='Date' type='datetime'/>
    <wf:property name='Quantity' type='integer'/>
    <wf:property name='Ammount' type='integer'/>
    <wf:relationproperty name='Position' relation='PositionTransactions'/>
  </wf:entity>
</wf:project>
```
The Mockup Model

<wf:entity name="Security" permissioned="true" persistent="true" cacheMode="Always" temporal="false" versioned="true">
  <wf:documentation><![CDATA[
Holds information about an asset type
]]></wf:documentation>
  <wf:property name="Name" type="varchar(100)"/>
  <wf:property name="SecurityType" type="SecurityType"/>
  <wf:relationproperty name="Currency" relation="SecurityCurrency"/>
  <wf:relationproperty name="Price" relation="Price"/>
</wf:entity>

<wf:entity name="Price" permissioned="true" persistent="true" cacheMode="OnDemand" temporal="true" versioned="true" propertyentity="true">
  <wf:documentation><![CDATA[
Holds the price of a security over time
]]></wf:documentation>
  <wf:property name="Type" type="PriceType"/>
  <wf:property name="PriceValue" type="decimal(25,10)"/>
</wf:entity>

<wf:entity name="Currency" permissioned="true" persistent="true" cacheMode="Never" temporal="false" versioned="true">
  <wf:documentation><![CDATA[
Represents the world currencies used in trading
]]></wf:documentation>
  <wf:property name="IsoCode" type="varchar(100)"/>
  <wf:property name="Name" type="varchar(100)"/>
  <wf:relationproperty name="ExchangeRate" relation="CurrencyExchangeRate"/>
</wf:entity>

<wf:entity name="ExchangeRate" permissioned="false" persistent="true" cacheMode="OnDemand" temporal="true" versioned="true" propertyentity="true">
  <wf:documentation><![CDATA[
Represents the exchange rate of a Currency
]]></wf:documentation>
  <wf:property name="Rate" type="decimal(25,10)"/>
</wf:entity>

<wf:entity name="PriceType" enum="true" permissioned="false">
  <wf:documentation><![CDATA[
The types a price can have
]]></wf:documentation>
  <wf:constant name="Manual"/>
  <wf:constant name="BloomBerg"/>
</wf:entity>

<wf:entity name="SecurityType" enum="true" permissioned="false">
  <wf:documentation><![CDATA[
The types a security can have
]]></wf:documentation>
  <wf:constant name="Equity"/>
  <wf:constant name="Bond"/>
  <wf:constant name="Cash"/>
</wf:entity>

<wf:entity name="PortfolioType" enum="true" permissioned="false">
  <wf:documentation><![CDATA[
The types a Portfolio can have
]]></wf:documentation>
  <wf:constant name="Managed"/>
  <wf:constant name="Custodized"/>
  <wf:constant name="Advisory"/>
</wf:entity>

<wf:relation name="ClientCurrency">
  <wf:FirstEntity name="Client" min="1" max="n"/>
  <wf:SecondEntity name="Currency" min="1" max="1"/>
</wf:relation>

<wf:relation name="PortfolioCurrency">
  <wf:FirstEntity name="Portfolio" min="1" max="n"/>
  <wf:SecondEntity name="Currency" min="1" max="1"/>
<wf:relation name="PortfolioClient">
  <wf:FirstEntity name="Portfolio" min="1" max="n"/>
  <wf:SecondEntity name="Client" min="1" max="1"/>
</wf:relation>

<wf:relation name="PositionsPortfolio">
  <wf:FirstEntity name="Position" min="1" max="n"/>
  <wf:SecondEntity name="Portfolio" min="1" max="1"/>
</wf:relation>

<wf:relation name="PositionsSecurity">
  <wf:FirstEntity name="Position" min="1" max="n"/>
  <wf:SecondEntity name="Security" min="1" max="1"/>
</wf:relation>

<wf:relation name="PositionsCollateral">
  <wf:FirstEntity name="Position" min="1" max="n"/>
  <wf:SecondEntity name="Position" min="1" max="1"/>
</wf:relation>

<wf:relation name="PositionTransactions">
  <wf:FirstEntity name="Position" min="1" max="1"/>
  <wf:SecondEntity name="Transactions" min="1" max="n"/>
</wf:relation>

<wf:relation name="SecurityPrice">
  <wf:FirstEntity name="Security" min="1" max="1"/>
  <wf:SecondEntity name="Price" min="1" max="1"/>
</wf:relation>

<wf:relation name="SecurityCurrency">
  <wf:FirstEntity name="Security" min="1" max="n"/>
  <wf:SecondEntity name="Currency" min="1" max="1"/>
</wf:relation>

<wf:relation name="CurrencyExchangeRate">
  <wf:FirstEntity name="Currency" min="1" max="1"/>
  <wf:SecondEntity name="ExchangeRate" min="1" max="1"/>
</wf:relation>

</wf:project>
Appendix B

The Model Language

<?xml version="1.0" encoding="utf-8"?>
<xs:schema id="project"
targetNamespace="http://tempuri.org/2008/WealthCCCFluent"
xmlns:mstns="http://tempuri.org/2008/WealthCCCFluent"
xmlns="http://tempuri.org/2008/WealthCCCFluent"
xmlns:xsi="http://www.w3.org/2001/XMLSchema"
attributeFormDefault="qualified">
<xs:annotation>
<xs:documentation>
<! [CDATA[
Version 0.4 27.11.2008
]]>
</xs:documentation>
</xs:annotation>

<xs:element name="documentation" type="xs:string">
<xs:annotation>
<xs:documentation>
<! [CDATA[
The documentation element is a simple property which just contains text
to document objects in the model specification. ]]>
</xs:documentation>
</xs:annotation>
</xs:element>

<xs:element name="code" type="xs:string">
<xs:annotation>
<xs:documentation>
<! [CDATA[
The code element is used for code snippets which can be inserted into the model
for simple operations. ]]>
</xs:documentation>
</xs:annotation>
</xs:element>

<xs:element name="constant">
<xs:annotation>
<xs:documentation>
<! [CDATA[
The constant type is used to define enumerable types, and also to enable
localized properties on objects. ]]>
</xs:documentation>
</xs:annotation>
<xs:complexType>
<xs:sequence>
<xs:element ref="documentation" minOccurs="0" />
<xs:element ref="localization" minOccurs="0" maxOccurs="unbounded" />
</xs:sequence>
<xs:attribute name="name" type="xs:Name" />
<xs:attribute name="versioned" type="xs:boolean" />
<xs:attribute name="type" type="xs:string" />
<xs:attribute name="value" type="xs:string" />
</xs:complexType>
</xs:element>

<xs:element name="localization">
<xs:annotation>
</xs:annotation>
</xs:element>
</xs:schema>
The localization type is used to set several names on an object or a property depending on culture.

The snippet type is used to insert custom code into properties and validations.

The validation type is used to set rules on an object property for validation.

Property of an object. Can be set to a relation or a custom function with the aid of a snippet.

The entity type is used to create entity objects.
The name of the accessor of this relation on this entity

The name of the relation

How deep the query recursion will go

The first entity in the relationship

The first entity in the relationship
<xs:element name="project">
  <xs:annotation>
    <xs:documentation>
      The project type is the base element. It also holds project-wide attributes.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="constant" minOccurs="0" maxOccurs="unbounded" />
      <xs:element ref="entity" minOccurs="0" maxOccurs="unbounded" />
      <xs:element ref="relation" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
    <xs:attribute name="model_version" type="xs:string" use="required" />
    <xs:attribute name="project_name" type="xs:string" use="required" />
    <xs:attribute name="project_version" type="xs:string" use="required" />
  </xs:complexType>
</xs:element>

<xsl:annotation>
  <xs:documentation>
    Supporting SimpleTypes
  </xs:documentation>
</xs:annotation>

<xsl:simpleType name="cascadeReaction">
  <xs:annotation>
    <xs:documentation>
      Reaction to an event, such as delete or update
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="before" />
    <xs:enumeration value="after" />
    <xs:enumeration value="no" />
  </xs:restriction>
</xs:simpleType>

<xsl:simpleType name="cacheMode">
  <xs:annotation>
    <xs:documentation>
      The type of cache pattern that an entity should use.
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="Always"/>
    <xs:enumeration value="Never"/>
    <xs:enumeration value="OnDemand"/>
  </xs:restriction>
</xs:simpleType>
This datatype is used to define an upperbound of a relation, which can be from 1 to any positive number. If there is no limit, the user may provide 'n'.