Universidade do Minho
Escola de Engenharia

Miguel Esteves

CazDataProvider: A solution to the object-relational mismatch

Outubro de 2012

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Miguel Esteves

CazDataProvider: A solution to the object-relational mismatch

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor José Creissac Campos

Outubro de 2012

To my parents...

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Christopher Alexander

Abstract

Today, most software applications require mechanisms to store information persistently. For decades, Relational Database Management Systems (RDBMSs) have been the most common technology to provide efficient and reliable persistence. Due to the object-relational paradigm mismatch, object oriented applications that store data in relational databases have to deal with Object Relational Mapping (ORM) problems. Since the emerging of new ORM frameworks, there has been an attempt to lure developers for a radical paradigm shift. However, they still often have troubles finding the best persistence mechanism for their applications, especially when they have to bear with legacy database systems.
The aim of this dissertation is to discuss the persistence problem on object oriented applications and find the best solutions. The main focus lies on the ORM limitations, patterns, technologies and alternatives.
The project supporting this dissertation was implemented at Cachapuz under the Project Global Weighting Solutions (GWS). Essentially, the objectives of GWS were centred on finding the optimal persistence layer for CazFramework, mostly providing database interoperability with close-to-Structured Query Language (SQL) querying.
Therefore, this work provides analyses on ORM patterns, frameworks, alternatives to ORM like Object-Oriented Database Management Systems (OODBMSs). It also describes the implementation of CazDataProvider, a .NET library tool providing database interoperability and dynamic query features. In the end, there is a performance comparison of all the technologies debated in this dissertation.
The result of this dissertation provides guidance for adopting the best persistence technology or implement the most suitable ORM architectures.

Key Words: *ORM, SQL, RDBMS, Domain Model, ADO.NET, NHibernate, Entity Framework (EF).*

Resumo

Hoje, a maioria dos aplicações requerem mecanismos para armazenar informação persistentemente. Durante décadas, as RDBMSs têm sido a tecnologia mais comum para fornecer persistência eficiente e confiável. Devido à incompatibilidade dos paradigmas objetos-relacional, as aplicações orientadas a objetos que armazenam dados em bases de dados relacionais têm de lidar com os problemas do ORM.
Desde o surgimento de novas frameworks ORM, houve uma tentativa de atrair programadores para uma mudança radical de paradigmas. No entanto, eles ainda têm muitas vezes dificuldade em encontrar o melhor mecanismo de persistência para as suas aplicações, especialmente quando eles têm de lidar com bases de dados legadss.
O objetivo deste trabalho é discutir o problema de persistência em aplicações orientadas a objetos e encontrar as melhores soluções. O foco principal está nas limitações, padrões e tecnologias do ORM bem como suas alternativas.
O projeto de apoio a esta dissertação foi implementado na Cachapuz no âmbito do Projeto GWS. Essencialmente, os objetivos do GWS foram centrados em encontrar a camada de persistência ideal para a CazFramework, principalmente fornecendo interoperabilidade de base de dados e consultas em SQL.
Portanto, este trabalho fornece análises sobre padrões, frameworks e alternativas ao ORM como OODBMS. Além disso descreve a implementação do CazDataProvider, uma biblioteca .NET que fornece interoperabilidade de bases de dados e consultas dinâmicas. No final, há uma comparação de desempenho de todas as tecnologias discutidas nesta dissertação.
O resultado deste trabalho fornece orientação para adotar a melhor tecnologia de persistência ou implementar as arquiteturas ORM mais adequadas.

Key Words: *ORM, SQL, RDBMS, Domain Model, ADO.NET, NHibernate, EF.*

Acknowledgements

It is with immense gratitude that I acknowledge the support and help of Prof Dr. José Creissac Campos who managed to carefully review my dissertation even on a tight schedule. I also thank Prof Dr. António Nestor Ribeiro for some solid and experienced advice.

To my parents and girlfriend who have been very patient with me.

I would like to thank Cachapuz for the scholarship and especially Eduardo Pereira for considering and helping the development of my work.

Finally, I thank my friends particularly my colleague Ricardo Santos for all the discussions and ideas we debated together at Cachapuz.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today, most software applications require mechanisms to store information persistently. A good example of that lies on enterprise applications, in which, email services store mail, banks keep records of the transaction operations on every account and hospitals hold the medical files of its patients. Persistence is the method used by a program to create non volatile data capable of outlasting software or even hardware systems of which it depends on.

For decades, Relational Database Management Systems (RDBMSs) have been the most common technology to provide efficient and reliable persistence in enterprise software. Therefore, the also successful object oriented languages pushed forward the development of new methods and technologies, for communicating with such databases, like Java Database Connectivity (JDBC), Open Database Connectivity (ODBC), ADO.NET, object serialization.

However, due to the object-relational paradigm mismatch, most of the object oriented applications that store data in relational databases have to deal with Object Relational Mapping (ORM) problems. In seeking to resolve these problems, ORM patterns were documented and frameworks have been developed, such as Enterprise JavaBeans (EJB) **Entity Beans**, Hibernate, NHibernate and Entity Framework (EF). All these methods and technologies share the same goal: to efficiently persist objects independently from the underlying database system. Although these technologies improved considerably the development of applications with complex domain models, ORM problems remain, especially in database centric applications.

In fact, it is common for software developers to have trouble finding the best persistence mechanism for their applications. Also the Object-Oriented Database Management System (OODBMS) brought about curiosity for being able to bypass these ORM problems.

This dissertation is about identifying the ORM problems, patterns and solutions in ORM frameworks as well as seeking alternatives to the object-relational mismatch.

The following sections of this chapter define the context of work for this dissertation, the problem of persistence, the objectives, some relevant concepts and the structure of this document.

1.1 Context of Work

This dissertation was developed in Cachapuz, within the Global Weighting Solutions (GWS) Project. Cachapuz Weighting Solutions is based in Braga, Portugal, and belongs to the Bilanciai Group, the major manufacturer of industrial weighting solutions in Europe, with eight production units and presence in thirty countries. In recognition of Cachapuz technological know-how, Bilanciai Group decided to gather in Portugal its Technological Competences Centre, with the goal of designing and implementing software automated solutions that complement and raise value to the equipments, manufactured and commercialized in the remaining members. The mission held by this Technological Competences Centre drives the development of high level software for the next generation of weighting readers of the Bilanciai Group.

The GWS Project focuses on various improvements on practices of software development in order to make several enhancements to the CazFramework, which is currently the most crucial software artefact in use at Cachapuz, acting both as a framework for developers and a solution for customers. The core of this framework consists on a fat client application, developed with the platform .NET, and implements all the required business logic and Graphical User Interfaces (GUIs). There is also a core database that provides critical data to the framework. When submitted to a process of configuration and development of external modules or add-ins, this core becomes a solution for a particular customer and able to interact with other data sources. This development process occurs in an agile feature oriented process in order to give rapid responses after a few customer interactions. The deployment is done at the customer through the installation of a pre-configured CazFramework with extensions, and a supporting core database (in SQL Server) together with other data sources as needed. Thus CazFramework is delivered to the customers as a bundle product composed by a fat client, databases and technical support.

The functions of the core CazFramework can be divided into management features, mainly for administrators or developers, and data query features for common users. The business model contains entities that support location or session context (`Area`) and security control (`Functionality`) over a single `User`. A `Functionality` represents permissions for actions or behaviours that `Users` are allowed to perform. The principal features of the framework provide query and management of `Lists`, `Forms` and `Menus`. These three primary business entities support a generic and dynamic mechanism for the creation of data query and management GUI components. `List` is the nuclear entity of the framework and consists on a rich view over a database in the form of a data grid or table, and further it can be enhanced with filters or custom styles. A `List` can be specified dynamically in runtime with a query in Structured Query Language (SQL). `Menus` can be summoned from a selected row of a `List` data grid to transfer its context either to a new `List` or `Form`, thus enabling new query and editing capabilities with fresh business operations. Hence, the most valuable characteristic of CazFramework is the dynamism with which one is able to create new listings.

In terms of practices of software development, CazFramework is supported by a bottom-up process, i.e. from a relational model or database schema, the Data Access Layer (DAL) code is generated via ClassBuilder, which is an ORM tool for generating DAL code in VB.NET, built and maintained by Cachapuz developers. The generated code is object oriented and carries both an entity class and a Create, Read, Update, Delete (CRUD) class per table of the database schema. The CRUD code uses a generic Object Linking and Embedding, Database (OLEDB) API to connect the database, striving for a level of abstraction that allows compatibility with multiple OLEDB providers and various RDBMSs. The DAL generated code is further assembled together with GUIs and business logic operations via a Visual Studio library project. The compiled Dynamically Linked Library (DLL) of this assemble, or simply add-in, is thereafter, pluggable to the CazFramework itself, thus providing control of new resources such as Lists, Forms, Functionalities, etc.

With the increasing size and complexity of CazFramework, together with a pursuit for expanding and exporting itself to other Cachapuz partners, so do the technological and methodological commitments strengthen. Therefore, GWS Project arises to endorse the best decisions on conceiving this framework through a process of evaluating the most modern technologies and methods possible, so it can avoid critical problems in the future. The result is expected to capitalize the investment in short term, with major increments on efficiency and speed of development of solutions at Cachapuz.

Due to the fact that customers often need to operate with different brands and versions of RDBMSs, GWS Project plans for CazFramework, the pursuit for database interoperability. Therefore the framework needs an abstraction layer to the data sources so it can connect a wide range of RDBMSs through an underlying system that can unify SQL dialects, particularities of different providers and synchronize the business model with the data model to solve database incompatibilities and updates. Since these are conventional requirements in the software development, they have been formerly discussed within the ORM subject. Thus, it is essential to perform an analysis of the viability to embark on ORM technologies.

1.2 Persistence Problem

RDBMS have been around since the 1970s and established a basis for most software applications existing today. Over the years, relational databases have become a solid solution for data keeping.

Meanwhile, it was in the 1990s that the object-oriented paradigm gained wide acceptance as the solution for modelling and programming complex business logic. As a result, the merging of both object and relational paradigms into the same application forced the need to use abstraction layers between business logic and data. The result could not avoid the so called impedance mismatch, which is a term borrowed from the electronics to emphasise the gain of resistance both paradigms have to each other, thus diminishing their true capabilities.

It was not long before a new paradigm was born by the name of Object Relational Mapping (ORM)[1] to attempt a resolution of object-relational impedance problems. As a result, the ORM paradigm gained valuable contributions, especially on patterns documentation, such as Fussel's [Fus97], Beck's [Bec97], Alur's [AMC01], Fowler's [Fow02] and even from Gang of Four (GoF) book [GHJV95], which clarified and found ways to minimize most of the ORM limitations.

The stage was set so the technological advances would take place bringing upfront the establishment of ORM frameworks such as Hibernate, which is currently the earliest and most mature. Additionally, the ORM put some effort into minimizing other problems like the continuous loss of an SQL standard due to the isolated progress of the various RDBMS. As a result, the database interoperability support was implemented, thus taking better advantage of the extra abstraction layer and liberating the users from the idiosyncrasies of RDBMS.

The persistence problem is achieving an increasingly wide range of solutions and RDBMSs are no longer the answer for all the businesses. While the vast and rigid legacy world of current RDBMSs is attempting to match the object world, major organizations

[1]Refer to section 2.3 in chapter 2 for details on ORM as a paradigm.

are abandoning SQL. The Not only SQL (NoSQL) movement has strengthened in the last few years mostly due to the appearance of Facebook's Cassandra, Google's BigTable and Amazon's Dynamo which are scalable and flexible, unlike RDBMS. Also the increasing popularity of OODBMS, is enabling a wide variety of alternatives to bypass the impedance mismatch.

Still, the motivation for this dissertation lies on the demand for ideal technologies and methodologies to build a DAL that fits into CazFramework and can improve the production of new code that can run, through less effort.

1.3 Objectives

This dissertation centres itself upon the persistence problem and attempts to demonstrate the best approaches to solve it, under an enterprise environment. Most objectives of this dissertation are shared within the goals of GWS Project, which include:

- The identification of the ORM problem;
- Alternatives to the ORM problem;
- A research and discussion on the paradigms, architectures and patterns, to set a ground for ORM technology;
- A confrontation of ORM theory at the current .NET ORM frameworks: NHibernate and EF;
- An analysis of ClassBuilder, to find the system design constraints and thus build a consistent DAL;
- The study of SQL dialects and database connectivity idiosyncrasies in order to implement database interoperability for the DAL;
- The design and implementation of a database agnostic DAL by the name of CazDataProvider;
- Confrontation and performance measurement of the discussed technologies altogether.

In the end, the final objective is to reach a conclusion on what methods and technologies to use for data storage and access under different circumstances so it can provide guidance for software developers in the future.

1.4 Structure of Dissertation

This chapter ends with the structure of this dissertation. The chapter 2 defines the in-depth theory behind the ORM, delineated by both the explanation of relational and object paradigms, the ORM problems and limitations and the alternatives to ORM (e.g. OODBMSs). The chapter 3 provides an analysis and description of the most relevant design patterns used in ORM solutions mostly from the Fowler's "Patterns of Enterprise Application Architecture" [Fow02] with the support of Beck's [Bec97] and GoF's [GHJV95]. In finding solutions for the persistence problem, the chapter 4 matches the theory and the patterns explained before against two popular ORM frameworks: NHibernate and EF. The chapter 5 describes the various steps of research (analysis to ClassBuilder ORM tool and ADO.NET data providers), design and implementation of CazDataProvider and presents an alternative to the typical ORM frameworks and the object-relational mismatch. The chapter 6 draws an overall performance benchmark to compare all the approaches discussed in this dissertation, including the ORM frameworks NHibernate and EF, ClassBuilder with CazDataProvider an two OODBMSs. The chapter 7 enclosures this dissertation with a summary of the results and achievements of this dissertation.

Chapter 2

Object Relational Mapping Theory

Object Relational Mapping (ORM) frameworks are emerging in the software engineering of today and gaining more and more followers. From very early on did software enterprises realise the importance of using these frameworks. ORM frameworks generate repetitive and critical code to better encapsulate Data Access Layer (DAL) implementations. The management of this code requires less effort from the development teams, thus resulting in increased productivity.

Platforms like System, Applications and Products in Data Processing (SAP), Oracle, Java Platform, Enterprise Edition (J2EE) and .NET have been pursuing this goal with a fair amount o success.

However, the currently available frameworks tend to be generic and, therefore, do not always meet the specific needs of every project. Hence, many development teams decide to invest on producing internal frameworks to accelerate specific development processes. Building such frameworks can an ambitious task for they often raise problems of complexity and maintenance.

In the case of Cachapuz a DAL code generator (ClassBuilder) was developed that handles a mapping between database tables and records into collections and objects. Accordingly, an API is provided to easily manage these objects, so called entities, throughout the business logic.

Relational Database Management System (RDBMS) are established today at most enterprises and so the relational paradigm became prevalent among software developers. That being the case in Cachapuz, the Global Weighting Solutions (GWS) project was driven into a research that found, at some point, the following options:

- Adopt one of the ORM frameworks analysed in chapter 4 and thus strive for a radical change of paradigms from relational to object oriented. This presents a number of challenges. The uncertainty of the learning curve and dealing with human and technological resistance alone are tough barriers. This change would also imply the disposal of the ClassBuilder code generator, followed by a possible chain reaction among all the dependent components of the prior established DAL mechanism. Although, this scenario could be less tragic with an additional effort to enforce a migration process for DAL replacement;

- Improve the current ClassBuilder. This is done by not fully engaging or committing to the object paradigm at first. Postponing this decision makes it easier to revert to the relational paradigm if needed. The improvements can either be superficial and easier to implement or profound but evoking a similar chain reaction to the adoption of an ORM framework. The paradigm and technological change is less obvious in this approach because the process is a progressive evolution of the current development environment. However, successive and profound improvements may, in the long term, approximate ClassBuilder to the level of any of the ORM frameworks available.

- Establish a commitment with the current methods and paradigms surrounding the relational model. This can either be done discarding changes to the framework completely or adapt the ORM patterns and practices to the established paradigms and technologies. The latter may increase productivity in CazFramework in the short term and with less change effort.

All three options have their own pitfalls. Therefore, before making a decision whether to embark or not into the ORM paradigm, many variables should be taken into account, such as the technical know-how of the developers, the legacy constraints regarding RDBMSs or Structured Query Language (SQL), the progress of the domain model complexity and the frameworks used.

It is not new that object oriented technology is the right technology to encapsulate business logic and relational technology has long been used to store and query large amounts of data effectively [Nur05].

Often enterprise applications reach complexity levels that affect directly the business logic. Object oriented is the paradigm that stands out to deal with that kind of problems, mostly because of its high flexibility for modelling business. Java and `C#` have been the most popular object oriented languages in the last decade and are known to have greatly increased productivity among software developers. However, it is only natural for most applications that persistence will be required eventually and especially in enterprise applications.

Once again the RDBMSs stand out among the other databases as the most efficient and widely used storage or persistence mechanisms. That is essentially due to the following factors:

- Relational algebra, a solid mathematical foundation;
- Decades of improvements, optimizations and increments;
- SQL a widely accepted standard for all RDBMSs;
- Legacy database systems, as a consequence of widespread use and the fixed schema defined in the early specifications;
- Wide application integration with relational databases, such as reporting tools.

Oracle and SQL Server are very successful RDBMSs in enterprise software development.

In the end, the problems raised by the need to couple the object oriented and the relational technologies, originated the need of creating a middleware layer, commonly designated as the ORM layer today.

This chapter is further divided into 6 sections. First it introduces the object oriented paradigm and the relational paradigm separately. Then it introduces to the concept of ORM paradigm. The largest and most important is the section 2.4 which introduces to the commitment of ORM, together with the limitations and awarenesses to be taken into account before embarking onto this paradigm, or in other words the object-relational impedance mismatch. The following section presents alternative database paradigms to the common RDBMSs in order to overcome its disadvantages towards ORM. At the end, a conclusion is taken upon possible solutions to the ORM commitment.

2.1 Object Paradigm

Programming languages like Java became popular in the late 90s which propelled the promotion of the object oriented paradigm. Ever since, many new methodologies, patterns and even paradigms where found to support it. Thence even the RDBMSs, along with its standards, attempted and at some point found a way to adopt the object paradigm.

At this stage, it is relevant to explain the concepts that surround the object paradigm. It is acknowledged today that the object paradigm stands for the concepts of [Fus97]:

- *Identity* distinguishes every object from each other even the ones with an equal state or value. In object oriented languages such as `C++`, `C#` and Java it is common to raise discussions about equivalence and strict equality where there is a difference between `a==b` (`a` and `b` are the same object) and `a.equals(b)` (`a` and `b` are equal). Also the existence of an identity allows the use of pointers and object references in object oriented systems;

- *Behaviour*. As opposed to passive data structures in procedural programming languages like C, objects do provide an abstraction or interface of communication so the object caller can access the object itself. This is given by a collection of operations within the object itself. All these operations are part of the object's behaviour as are the responses it gives to the caller and the changes made to the object's state through its operations;

- *State* or the so called current object value associated with his unique identity. Any changes to the state of an object are caused by its behaviour. Also, because objects are encapsulated (see below), the result of a state change and the actual value of an object are only visible through its behaviour as well;

- *Encapsulation* or the abstraction that hides object's behaviour implementation and state in a black box preventing the client to know what is inside. Thus, everything from an object's behaviour implementation to a static structure it may have inside, can easily be extended or changed without affecting its clients.

From the above, more familiar concepts to the object world were derived [Fus97]:

- *Type* is a specification of an interface. An object implements one or more types when it provides implementations to the relevant interfaces;

- *Association* of types through navigable links, which enables the object graph;

- *Class* is a kind of object that defines its implemented types, the behaviour and the state variables required. Every object of the same class shares the same implemented types and behaviour but they all contain a different state and identity;

- *Inheritance* can be applied to both types and classes. If `type A` inherits from `type B`, any object that implements `type A` will be of `type B` as well, because all the specifications in `type B` were inherited by `type A` in the first place. A class that inherits from another class, inherits all its behaviour implementations and is of the same type as the parent.

Also, through *encapsulation* the object world has synthesised and sold *polymorphism* as a trademark feature that too often caused trouble to the developers of strongly typed languages such as C. Although *parametric polymorphism* can be used in dynamic programming languages like Perl, the *subtype polymorphism* is characteristic of object oriented languages. Thus, different objects can be identified by the same common superclass and handled in common interfaces. In the end, type casting can bring back the original type from the abstract super-type.

2.2 Relational Paradigm

The relational paradigm was introduced in the early 70s by Edgar F. Codd. Since then, it has grown and given rise to the most popular databases of today, the RDBMSs. Examples of such are Oracle and SQL Server.

The relational model is supported by relational algebra and stands for *knowledge* and not for any of the object's concepts. A relational system allows the storage of facts through predicates or statements. The principal concepts of the relational terminology are described by [Fus97] and based on [Dat04] as the following:

- *Relation* is a truth predicate defining its meaning, often implicitly, through its attributes. For instance if a relation `Person` has the attributes of `Name` and `Age` it means that there is a person called `Name` and is `Age` years of old. In a RDBMS, a relation is usually described as a table definition except that tables are two-dimensional and relations are n-dimensional. Also the columns of a table are ordered and the rows as well, while the attributes and tuples of a relation are not.

- *Attribute* identifies the name of each meaningful aspect of the relation. An attribute also defines the domain of values it must contain. For instance, in the above relation, the attribute `Name` is limited to values of type `string` and the attribute `Age` must be a positive integer commonly identified of type `integer`. In RDBMS an attribute is defined by a table column.

- *Domain* is the data type specified to constrain the values of an attribute, limiting as well, the operations available to handle those values. In certain types such as `string` it also identifies the maximum size or length the value can have.

- *Tuple* is a truth statement based on a relation, sharing all its meaningful attributes, including the respective domains and *attribute values*. There is no order in the components of a tuple. In RDBMS it is identified as a table row with the exception that a table allows duplicate rows while a relation does not with tuples. *Relation value* is composed by the set of all tuples that satisfy a relation. In RDBMS it is a table representation with headers, columns and rows of data. *Relation variable* is represented by a name and contains the relation definition with its values at a given time. It is common to name such variables in plural, e.g. the relation variable `People` is named after its relation `Person`. In RDBMS a relation variable corresponds to the table structure and data. A database is the collection of relation variables. *Derived relation values* are calculated from other relation values known to a database. They are often the result of relational expressions and queries using a set of operators: selection, projection, Cartesian product, join, difference, union, intersection and division. In RDBMS the SQL is the universally accepted language to enable these operators.

Normalization has become a popular design technique for optimizing relational databases. It consists of removing redundant data to keep simple the data manipulation and consistency, free from abnormalities and saving disk space. On the other hand, the object paradigm does not allow redundancy due to the unique identity of each object. Still, the underlying persistence layer of an Object-Oriented Database Management System (OODBMS), may as well have to deal with flat files and tabular structures for disk input/output (I/O) operations, thus facing similar issues of the relational technology.

According to the relational model theory in [Cod70], a *nullable* attribute would be a violation of the first normal form. Also these columns have a typical problem of indexing. Null values are described as "missing information" by [Cod70] and thus, often RDBMS choose to ignore them on indexes.

The notion of *Transaction* is a fundamental concept in all persistence technology. When multiple users access a database, its data can easily become inconsistent. A transaction prevents this by allowing isolation, concurrency control, recovery from failures and restore of data consistency. The RDBMSs are known to implement the most reliable requirements known as the Atomicity, Consistency, Isolation and Durability (ACID) properties within a database transaction. Date [Dat04] classifies a transaction as a "logical unit of work" obeying the following principles:

- A transaction is an *atomic* operation which means its execution either fails entirely or succeeds completely. If anything fails during the process, the system must restore the previous state of consistency;

- Even in case of failure, a transaction always carries the database from one *consistent* state into another. Date [Dat07] changes his view from correctness to consistency for the "C" property in ACID . Integrity is responsibility of the Database Administrator (DBA). He ensures that the correct constraints, triggers and cascades are activated in order to permit only true propositions in the database. This is correctness and therefore cannot be enforced by the system. A correct state implies consistent and incorrect does not necessarily imply inconsistent but inconsistent implies incorrect. Therefore a transaction is only granted to be consistent in this principle.

- Transactions are *isolated* from each other so that the changes made by one transaction are made visible to the other concurrent transactions, only and when this first transaction commits successfully. Moreover, if it fails, no data is affected and thus nothing made visible to the other transactions since the first one rolled-back;
- A transaction is *durable* because if it *commits* successfully all the updates are ensured to be persistent. Even if the system fails, for instance when the changes are still in a buffer, the new data must be committed to the database. This can be implemented with transaction logs, backups and snapshots.

2.3 ORM as a Paradigm

Throughout this dissertation, the ORM is often referred to as a paradigm, despite this term being rarely used by other authors. A term that authors commonly use, however, is object/relational paradigm mismatch which views the ORM as a crisis rather than a role model to be followed. In the context of this dissertation, the ORM is often addressed as a paradigm. Therefore it is important to clarify the definition of scientific paradigm and confront it with the events that triggered the ORM.

The concept of paradigm by Thomas Kuhn in the *The Structure of Scientific Revolutions* [Kuh96], can be concisely defined as the practices that define a scientific discipline at a certain historical period, greatly influenced by socio-cultural factors. The following characteristics and exerts of the same book present some relevant factors of Kuhn's definition of the term paradigm:

1. There is no paradigm change without a crisis. Kuhn states that "they do not renounce the paradigm that has led them into crisis" [Kuh96];
2. Replacing an old paradigm for a new requires that both are compared together. Kuhn states that "the decision to reject one paradigm is always simultaneously the decision to accept another, and the judgement leading to that decision involves the comparison of both paradigms with nature and with each other" [Kuh96];

3. Paradigms change in a cumulative way and are built on top of older paradigms and knowledge. Sudden changes make incompatible paradigms. Accordingly, Kuhn asserts that "non-cumulative developmental episodes in which an older paradigm is replaced in whole or in part by an incompatible new one" [Kuh96];

4. The success of a paradigm is dictated by how well it is accepted by the community. Likewise, Kuhn foresees "the success of a paradigm is dictated by how well it is accepted by the community" [Kuh96];

5. "Research under a paradigm must be a particularly effective way of inducing paradigm change" [Kuh96];

6. "Incommensurability" of a paradigm makes the new and the old paradigms incompatible and not measurable against each other for old problems may become less trivial or other limitations rise [Kuh96]. Despite apparently the item 2 suggests the opposite, in here Kuhn affirms that paradigms should not be compared as the results can be misinterpreting when the end is to rate one as better than the other. That is because it is likely that they are measured against incompatible problems from different times, areas or even disciplines;

7. Paradigms transform a scientist's view of the world for they produce expectations that can and often obscure perception. Kuhn states that "literally as well as metaphorically, the man accustomed to inverting lenses has undergone a revolutionary transformation of vision" [Kuh96], so it is valid to say a paradigm may skew observation;

8. Paradigms change the scientists' workbench, for as Kuhn recalls "until that (Galileu pendulum) scholastic paradigm was invented, there were no pendulums, but only swinging stones, for the scientists to see. Pendulums were brought into existence by something very like a paradigm-induced gestalt switch" [Kuh96];

9. Past data can predict future data and thus different paradigms become comparable as working them against the same data. This is because data can generate ambiguous impressions and thus the paradigms change while the data persists;

10. Tests or verifications are often the tools for competing paradigms to win or loose the acceptance of the scientific community. As Kuhn states, "testing occurs as part of the competition between two rival paradigms for the allegiance of the scientific community" and "verification is like natural selection: it picks out the most viable among the actual alternatives in a particular historical situation" [Kuh96];

11. According to Kuhn, to make a successful communication of the conversion to a new paradigm is to: claim to "solve the problems that have led the old one to a crisis"; claim to entirely new predictions and claim to be "'neater', 'more suitable' and 'simpler"' with better aesthetics than the old paradigm [Kuh96];

12. It is important for the success of a paradigm to make feel the "new proposal is on the right track" for a bright future. This can be sought through "personal and inarticulate aesthetic considerations" [Kuh96].

The factors that lead to the rise of ORM, can also easily fit Kuhn's definition of paradigm. The ORM paradigm emerged in a time of crisis, caused by the problematic practice of using both the object and the relational paradigms together in the same application (analogous to item 1). Rather than renouncing to either the relational paradigm or the object paradigm, the ORM focuses on coupling the two. Thus, the ORM paradigm attempts to please the most communities (analogous to item 4).

The ORM arose as a paradigm middleware, rather than a replacement of old paradigms. A different course was travelled by the object paradigm which replaced, at some point, the procedural programming. However, the object paradigm and the relational paradigm had to be compared with each other in order to give birth to the ORM, and so that all three paradigms can coexist. That is analogous to item 2 but not necessarily with a full replacement of paradigms. That is, the old practices use relational paradigm to develop most of the application whilst the new ORM provide a way of building more complex domain models and applications but still use the relational paradigm at the bottom layer with a certain abstraction level. So, it is valid to say that ORM is a cumulative paradigm change (analogous to item 3).

On the other hand, other approaches such as the OODBMSs do allow the applications to reject the relational paradigm and only use the object oriented paradigm. However, the lack of mathematical foundation and robust standards for object persistence from both OODBMSs and the object paradigm itself, constitute a strong argument that yet there is no replacement for the relational paradigm to provide efficient data persistence.

Also, to define an object persistence paradigm without borrowing ideas and experience from the relational paradigm would most likely be a huge mistake (analogous to item 3). In the end, such sudden change would pose a major expensive restructuring process of many applications built on top of relational tools and paradigms.

Intensive research on the relational paradigm (SQL standards, Oracle keep up with new features, etc.) and the object paradigm (Gang of Four (GoF) design patterns, Patterns of Enterprise Application Architecture (PoEAA) patterns, etc.) influenced many changes throughout the last couple of decades including the ORM (analogous to item 5).

Despite the object and the relational paradigms are not measurable against each other, it is possible to compare the two practices of:

- Predominantly using the relational paradigm for designing data and application code;
- Using the relational paradigm for the persistence, the object paradigm for application code and the ORM paradigm for mapping both.

Both methods may be comparable as more or less efficient against different problems which depends on a number of variables (e.g. the technologies, the software infrastructures, know-how, etc.). For instance, a data intensive application that uses mostly reads and writes to the database, as it implements a simple model with few business logic, will likely perform better with a relational paradigm. On the other hand, the object model is more efficient for business modelling of complex domain models, especially as they grow, which can make the second method a better choice for such cases. Time is also an important variable, because before Hibernate ORM framework, the second method was not very popular. Thus, as the two methods deal with different limitations, they cannot be measured in a single variable (analogous to item 6).

For some time, the two methods above, have been competing with each other by running various tests capable of measuring speed, complexity, flexibility among other variables. The winner often achieves the most acceptance from the scientific community (analogous to item 10).

Throughout decades of evolution, the relational paradigm became expert in solving persistence problems. Thus, possible future paradigms that deal with persistence will have much to learn from the past researches of the relational paradigm (analogous to item 3).

When the relational paradigm was chosen on the 70s among other paradigms (network and hierarchical) it was mostly because its simplicity was enough to solve the problems of that time. Today, most problems are complex enough to consider other paradigms like the object oriented and the ORM (analogous to item 9).

The popularization of the relational paradigm brought a new view of the world. Therefore, today many developers use a relational approach for every problems even those that are not well suitable for such paradigm (analogous to item 7). The same happens in the ORM paradigm: there are problems not complex enough to require an object oriented approach but most developers still decide to adopt an elaborate ORM framework as their one and only solution for all the situations (analogous to item 7).

On the other hand, such paradigms offer a new set of concepts, patterns and tools to orient and enhance developers work, that did not exist earlier. For instance, before the object oriented paradigm was invented, the ORM was not even a problem (analogous to item 8).

The items 11 and 12 are marketing strategies that much resemble Steve Jobs Apple presentations. The ORM researchers also claimed to solve the crisis of matching two very different paradigms. Thus they provided practices and tools to generate new code that could be simpler, better structured and easier to maintain than the old one. From very early, successful ORM frameworks such as Hibernate or Entity Framework (EF) made the developers believe of its future success.

Additionally, this dissertation describes a set of patterns and practices to strengthen the ORM theory and by all means, it suggests that ORM can be called a paradigm.

2.4 The ORM Commitment

In Software Engineering it is common for projects to reach a point where all the investment made by then can and often must be discarded. This can happen mainly due to the restless advances in frameworks and other tools or new patterns and methodologies that support software developers today. At the same time, also when the course of things tends to not provide the expected outcome. With all the effort and the work already paid off, a dilemma emerges: if it is worth to discard all the investment made so far and adopt new methods thus sacrificing short term gains for long term whose success is not yet ensured. This step is of major importance for it can be critical to the rest of the project and is often handled by high ranked developers. Therefore, it needs to be guided through wisdom, experience and sense of responsibility. At this point and as the long term gains are never obvious, the working teams tend to manifest resistance to both technological and methodological changes.

Ted Neward's "Object/relational mapping is the Vietnam of Computer Science" analogy [New06] is known to have risen perplexity, dispute and lack of confidence among the developers community. His article circles around the ORM commitment, often comparing it to the Vietnam War. Its subject-matter strives for the recognition that the decision to step onto the ORM path leads to a risky path where compromises will have to be made. A path that will always require investments of some degree and yet does not provide a "clear exit strategy" [New06].

This section describes a group of major ORM problems, or limitations, as Ted Neward commonly refers to. Some solutions to those problems are also introduced in here.

2.4.1 Inheritance

Despite Relational Databases (RDBs) not supporting inheritance, at first this mapping problem may seem of simple solution. Apparently every object maps to a table and its fields to columns, type `string` to `varchar` and so on. Yet complexity raises when the developer seeks for polymorphism in the object model. Thus, mapping inheritance into the relational world does not present one clear solution but three strategies for different situations: table-per-class, table-per-concrete-class and table-per-class-family.

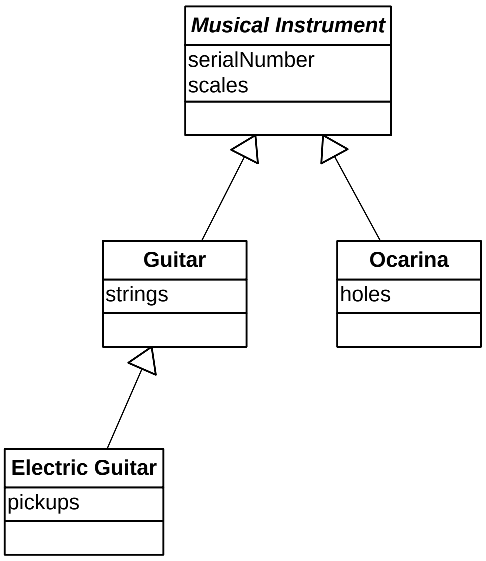


Figure 2.1: Hierarchy class diagram example

The class diagram of Figure 2.1 expresses the domain of a music store. Below the three different strategies are applied to map this object oriented model into a relational model of tables.

2.4.1.1 Table-per-class

The table-per-class (refer to class table inheritance in [Fow02]) strategy explores the proximity between objects and relations by mapping each table to a different class with all the columns mapped directly to fields. It is the less efficient inheritance mapping strategy for most cases and thus it is often avoided by the developers due to the following reasons [Fow02]:

- To retrieve a full object instance, it is required at least as many join operations as levels of the hierarchy and throughout as many tables;
- Refactoring of fields up or down the hierarchy results in costly database schema

changes (create X column in table A, transfer all data from column X of table B to table A and then delete column X of table B);

- It generates a bottleneck for the superclass table due to frequent accesses;
- More complex ad hoc queries.

On the other hand, this strategy provides a clearer table schema and a highly normalized form of data. Hence it wastes less disk memory and allows the DBA to better understand the domain model. Also it may be a more suitable solution if there is a high variability of object fields in the hierarchy.

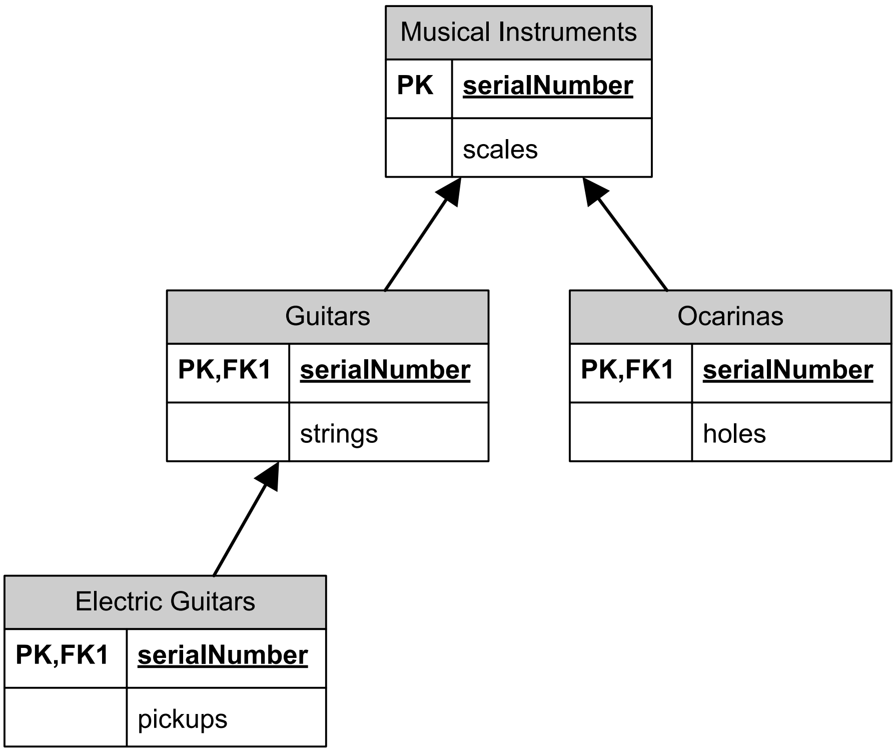


Figure 2.2: Relational model of table-per-class strategy example

The solution given by Figure 2.2 for the test case presented early, delivers a table per class mapping for the music store domain model. As expected, this relational model looks very alike the object model. It is common to name tables in plural and objects in singular forms as explained in the section 2.2. The arrows in Figure 2.2 do not represent inheritance like in the object model but rather the one and only kind of association possible among tables: the foreign key constraint (see the section 2.4.2 for mapping associations).

This strategy comes with two challenges: how to link the tables and how to query in the most efficient way possible. Figure 2.2 suggests all the tables share the same primary key `serialNumber`, so that the rows of different tables that belong to the same object can be identified. Thus, the `Musical Instrument` table contains a row for each row of the tables directly below.

For instance, a `Guitar` table row of `serialNumber` 100 has a foreign key constraint to the `Musical Instrument` table row of the same key. In the same way, the `Electric Guitar` table row has a constraint on the `serialNumber` to the `Guitar` table row. Hence, the same key lives in three different tables at the same time. Plus, to prevent inconsistencies, there is often the need for two foreign key constraints which slows down inserts and deletes. Therefore, to retrieve a full object instance of an `Electric Guitar`, at least two join operations are needed.

Making a call for each table is not efficient mostly due to the impact on start up and tear down of the SQL processing and network overheads. On the other hand, multiple joins can heavily slowdown the database, mostly restrained by the limited memory size and slow disk I/Os. The database optimizer often opts for multi-pass algorithms, in such cases, with high disk I/O cost (see appendix A for details).

For example, if a customer of the music store wants to browse the electric guitars, it is known which *subtable* to query. However, if he wants to browse all the musical instruments in the store that have a specific number of scales, the query must perform multiple outer joins (slower than inner joins) among all the *subtables* in the hierarchy, which can be very inefficient. An alternative is to only query the `Musical Instrument` table first and then, join the filtered rows with the relevant *subtables*, although that requires multiple queries.

However, the model in Figure 2.2 does not permit one row of `Musical Instrument` to know which is its relevant *subtable*. Thus, a solution to avoid performing outer joins with all the *subtables* is to add a discriminator column to the root table for identifying which *subtable* that row should join in the next step. While this increases the speed when loading a single object, if many rows are managed, the required multiple queries may prove slower than a single query with various outer joins as in the initial approach.

Loading objects in the mapping layer code, using this strategy, is not straightforward. In case the query loads objects at the bottom of the tree (e.g. `Electric Guitar`), it knows which class to instantiate as well as the relevant tables to join until the root (provided by the static mapping information). However, in the multi outer join scenario (e.g. loading all the musical instruments) with no discriminator, it is impossible to know which object to instantiate (either instantiate an electric guitar, a guitar or an ocarina) unless some logic is added to the SQL code providing such information.

2.4.1.2 Table-per-concrete-class

The table-per-concrete-class (refer to concrete table inheritance in [Fow02]) strategy is similar to the first, except that all superclasses in the hierarchy are not mapped into tables, only leafs of the tree are. Thus the number of query join operations required is reduced, but on the other hand it reduces data integrity. As the superclass is not mapped into a table, all its fields will be replicated for all the leaf tables, thus causing denormalization. Also, from the database schema, it is impossible to know which tables belong in the same hierarchy.

One issue about this strategy is that the keys among all the tables of the hierarchy must be unique. If two rows of different tables share the same primary key, there is a conflict of data integrity. This cannot be solved by the primary key uniqueness. Eventually, it gets worst when different systems use the same database. This can be solved either using triggers to compliment consistency rules (a slow option of last resort) or composite keys that contain an identification of the table.

Therefore this strategy presents the following advantages and disadvantages [Fow02]:

- Each table has no irrelevant fields;
- No joins are needed to retrieve a full object instance;
- A table is only accessed when the concrete class is accessed, thus preventing bottlenecks;
- The primary key uniqueness is difficult to implement;
- The implementation of a relationship in an abstract class must be mapped to all its leaf tables, which is not straightforward;
- To refactor fields up and down the hierarchy still poses a schema change even though with less modifications than the table-per-class strategy;
- Field updates on a superclass requires schema updates on all the relative leaf tables unlike in the table-per-class strategy;
- To query the superclass forces outer joins throughout all the tables which is expensive. The same occurs in the table-per-class strategy.

In table-per-concrete-class strategy, the music store object model would have the `Musical Instrument` and `Guitar` as abstract classes. This model is mapped in three leaf tables as presented in Figure 2.3. Any new implementation on the abstract classes would

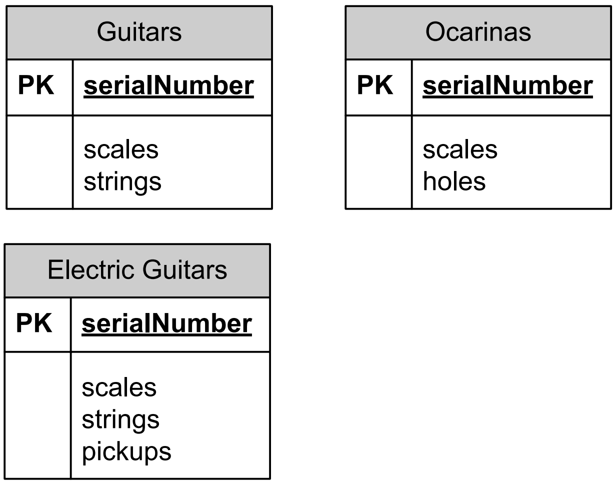


Figure 2.3: Relational model of table-per-concrete-class strategy example

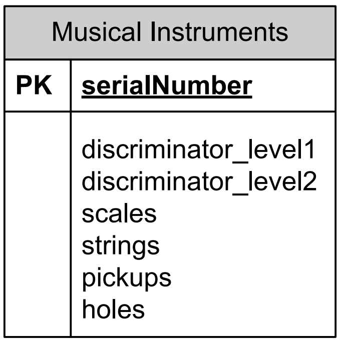
have to be mapped to these three leaf tables. For instance a relationship on `Musical Instrument` would probably require for all three leaf tables to acquire a foreign key constraint. For such cases, the table-per-class strategy would provide easier manageability.

2.4.1.3 Table-per-class-family

The easier strategy is table-per-class-family (refer to single table inheritance in [Fow02]) for it maps all classes in the hierarchy into a single table. There is a discriminator column per hierarchy level to identify which class maps that table row. This approach presents the following advantages and disadvantages [Fow02]:

- No joins are required to retrieve data, so, it is faster to load an object instance;
- The refactoring of fields up and down the hierarchy does not require database schema changes;
- Irrelevant columns are left empty which can be confusing to use ad hoc SQL queries, if there are frequent null values in the table;
- It can involve a large number of null values in the tables resulting in denormalization and constraint problems. Also indexes and `order by` clauses can get inefficient in columns filled by null values;
- The waste of space lead by unused columns can be a problem depending on the efficient compression features of RDBMSs;

- Greater table growth rate than the linear of the other strategies. Eventually the table size is too large to manage many indexes and frequent locking, thus recurring to other strategies for data partitioning and caching;
- The namespace is the same so the fields must be all have different names.



Musical Instruments	
PK	**serialNumber**
	discriminator_level1 discriminator_level2 scales strings pickups holes

Figure 2.4: Relational model of table-per-class-family strategy example

Figure 2.4 presents the single table structure that maps to the music store object model using the table-per-class-family strategy. There are two discriminators, one for each level of the hierarchy. Thus, if an electric guitar is loaded, both the discriminators are verified so that the correct class is instantiated. If the class name is embedded into the discriminator value, it can be used to instantiate the class. On the other hand it wastes more disk space and memory to load than using small codes.

Application developers often decide for the table-per-concrete-class or table-per-class-family strategies mainly due to performance considerations. Although, in some cases, the best solution may be to combine different strategies. On the other hand the table-per-class is what keeps the cleaner relational model and does not aggravate much the job of the DBA.

2.4.2 Associations

Mapping associations represents a further challenge even though both the object and the relational models support it at some level. However, business requirements tend to be more complex than that. Thus, domain models and Unified Modeling Language (UML) class diagrams describe the most detailed form of association with the three following variables:

- *Direction* of the association. It can be from A to B, B to A or bidirectional;
- *Cardinality* or multiplicity in both ends of an association;

- *Strength*, which defines simple associations, aggregations and compositions between objects from the weakest to the strongest.

To implement such variables and grant data integrity of the specified domain model, the object model requires code at business logic level. While objects are navigable through its associations, relations are not. They use SQL joins to associate with other relations. Also data integrity is granted at the level of foreign key constraints or with triggers that require additional programming code.

Objects only understand unidirectional associations in which the class that performs the association is the only one whom has knowledge or is aware of the association itself. One bidirectional association can only be achieved through two unidirectional associations. For instance, the object `A` has a reference to `B` and `B` has a reference to `A`. This is an example of two unidirectional associations with cardinality *one-to-one* or one bidirectional association.

In the relational model the notion of directional associations does not exist. That is because associations are established at the level of foreign key constraints over singular columns and not the relations as a whole.

Typically, in the object model, associations are categorized with three different kinds of cardinality: *one-to-one*, *one-to-many* and *many-to-many*.

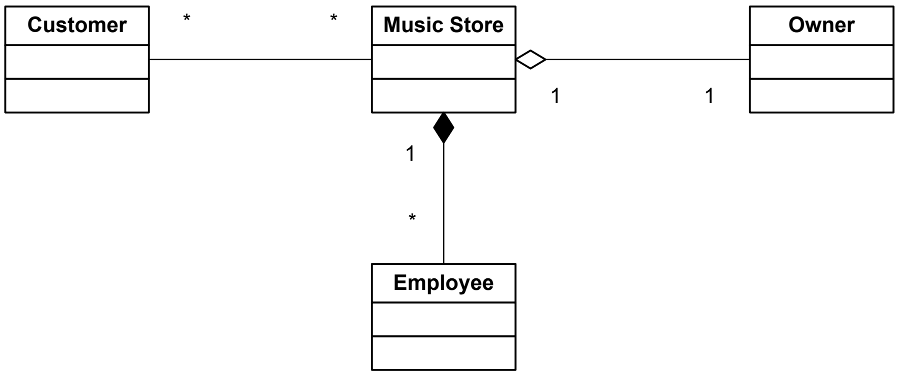


Figure 2.5: Class diagram of the object model for the music store with three kinds os associations

Figure 2.5 is the class diagram representation of an object model for the music store containing three kinds of bidirectional associations with different strengths. This diagram provides the following three associations:

- *One-to-one*. One `Owner` can only own one `Music Store` and one `Music Store` can only have one `Owner`. Also the owner is free to own different kinds of stores or other things outside the model. This association is of medium strength and is called an aggregation;

- *One-to-many* or *many-to-one*. One `Music Store` is composed of many `Employees`. This means that if the *music store* ceases to exist, the elements are no longer considered `Employees` as they become unemployed. Also one `Employee` can only be working in a single `Music Store` at once;

- *Many-to-many*. One `Music Store` can have many `Customers`. These `Customers` are shared throughout all the `Music Stores` which means one `Customer` can attend various `Music Stores` and as the association is weak, other things outside the model as well.

To implement the bidirectional one-to-one association (aggregation) between the `Music Store` and the `Owner` in an object oriented programming language, two unidirectional associations are required between both the classes. Thus, the `Music Store` class has a reference to a single `Owner` in the class attributes. The opposite had to be implemented as well. To instantiate such an association, both references have to be filled with the respective object addresses or pointers. Because every object has its own identity (address or pointer) independently from its fields, two objects with the same fields can coexist hence there is no exclusive data constraint in the association. Thus, the object model can easily fall within inconsistent data.

Due to a certain ambiguity on the strength of associations and the lack of object oriented mechanisms to enable data integrity, this has to be manually implemented in business logic code. For instance, when associating an `Owner` to a `Music Store`, the logic code has to check if the `Music Store` is already owned by some `Owner` and if that owner does not already exist in the data store. When deleting an `Owner`, the reference to its `Music Store` has to be eliminated as well.

The bidirectional *one-to-many* association between a `Music Store` and its `Employees` requires two unidirectional associations as well:

- One-to-many unidirectional association between `Music Store` and the `Employee`. This means the `Music Store` has a collection of `Employees` inside;

- One-to-one unidirectional association between the `Employee` and `Music Store`. Each `Employee` has a reference to its `Music Store`.

Data integrity ought to be here again checked with business logic code. If a `Music Store` is deleted, all its `Employees` should also be deleted.

The bidirectional many-to-many association between `Music Stores` and `Customers` is implemented with two unidirectional one-to-many associations. Thus, the `Customer` class has a collection of `Music Stores` and a `Music Store` has a collection of `Customers`. If one `Customer` is deleted, all its references should also be deleted.

The relational model works differently though. Data integrity can be ensured at some level. As the identity of a tuple is its primary key, which is unique, repeated tuples are not allowed. Also, the foreign key constraint associates two relations together and thus already prevents data inconsistency. The associations of the example above can be mapped to the relational model as demonstrated in Figure 2.6.

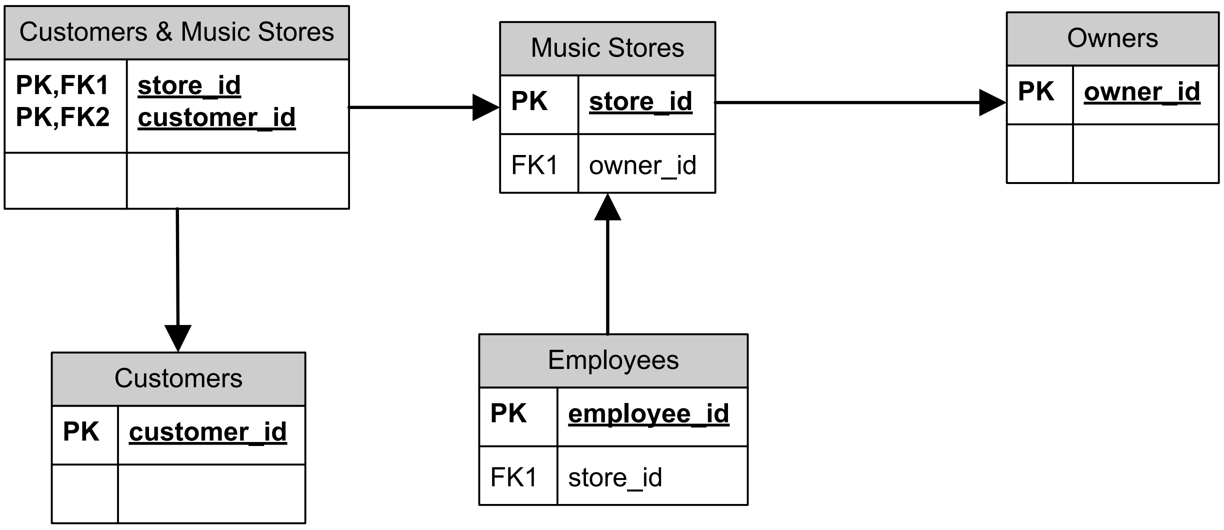


Figure 2.6: Relational model for the music store with three kinds of associations

The one-to-one association between `Music Store` and `Owner` can be mapped with a foreign key constraint as in Figure 2.6. Here, the relation `Music Stores` references the `owner_id`. This approach does not prevent one `Music Store` to have multiple `Owners`. Two crossed foreign key constraints could prevent such cases. Although that would require a drop of those constraints every time an `Owner` or a `Music Store` was created which could effect the performance of the RDBMS. Another alternative would be to have all the columns of the relation `Owners` in the `Music Stores` and thus abandon the relation `Owners`. This is the most correct approach if the association was a composition. Although it would hinder flexibility and extensibility to the model and thus, if one `Owner` was to own other things than `Music Stores`, that would be a problem. The one-to-one and many-to-one associations have the same implementation in Figure 2.6.

The one-to-many association is also implemented with a foreign key constraint as in the case above. Many tuples in relation `Employees` can reference to the same tuple in

`Music Stores` through the `store_id` primary key.

The many-to-many association is represented in the relational model with a new intermediary relation. In Figure 2.6, this new relation is `Customers & Music Stores` with a compound key containing two references, via foreign key constraints, to both the relations `Customers` and `Music Stores`. Thus a `Customer` can be associated with many `Music Stores` and vice versa.

Note that the multiplicity of the first edge of a unidirectional association in an UML class diagram is useless at the object implementation level.

Mapping object associations to and from the database can be implemented with the Fowler's patterns of Foreign Key Mapping (one-to-one or one-to-many), the Association Table Mapping (many-to-many) and the Dependent Mapping (composition) [Fow02]. The Lazy Load pattern is also common here to improve efficiency.

2.4.3 Schema complications

Another problem lies on the data schema itself. A primary goal that empowers an ORM tool to allure so many developers is persistence ignorance. These tools step on to database ground and soon take control of both the database schema and the application persistent object schema. This conflict is ascribed by Ted Neward in [New06] as the Dual-Schema, which raises a schema ownership problem.

Actual software development is deeply settled to work among relational databases and brought with it new roles such as DBA. The DBA takes responsibility for database performance and stability and for that he requires freedom to work the database schema whenever needed, e.g. do some refactoring and denormalization.

On the other hand it is frequent that developers step onto legacy applications and frameworks dependent on the same database schema that they want to deliver to an ORM tool. That can be either a very ambitious project or a disastrous one.

This is indeed a fact that tends to be neglected, for these tools are so keen to allure developers into model driven software, abstracting too far from the real world at times. Problems often happen when an application is built from scratch with an ORM tool and as the system grows, other applications can slip into using the same database but without the same abstraction.

It is important to acknowledge that any changes on one schema will reflect changes on the other. Here, the role of DBA must be reconsidered to progress on to ORM administration as well mostly because an ORM tool is not constrained to a one way mapping. It rather generates database updates at every variance it finds on schemas. If this work is neglected, it can compromise the whole application it supports.

However, it is no easy job to stand between two schemas and apply optimizations to both while enduring the impedance mismatch. Although, it is long gone for most software developers the time when designing data was a simple exercise of designing relations.

As ORM tools like Hibernate preserve a good separation of concerns between what is data domain and logic domain, it becomes easy for the DBA to work with the data domain and the business developer to work on the logic domain together. Thus, the DBA should be responsible for managing the mapping between objects and relations, write or optimize some more complex queries and perform his old job as well. This way, he can change the relational schema and the ORM layer to map the same object schema while not compromising the work of business developers.

In the end, ORM tools can be applied to legacy databases, which is frequently called reverse engineering. Although that is often a bad augury to architectural liability.

Regardless of these schema complications, it is all a matter of policies, planning decisions and impedances of settlements and paradigms.

2.4.4 OID (object identity)

Identity is another mismatch between relations and objects. It brings the issues of data exclusivity and integrity, already described above in regard of the associations debate.

Identity in objects is represented by a pointer or an object address, two different objects containing the same values are still different. Thus in Java or `C#` the operator `==` and the function `equals` coexist to solve that problem. The first compares object addresses and the latter the object attribute values.

In the relational model, the identity is represented by a primary key, which is a unique and immutable identifier for a table row. Also, it is important to acknowledge the following three characteristics [Fow02]:

- A key may be meaningful when it specifies something like a *Social Security* number or meaningless which is often a random number not intended for human use;

- A key may be simple or composite. The latter is represented by multiple fields and is commonly meaningful but requires special code handling such as an extra key class;

- There are database-unique keys and table-unique keys. The latter is the most used, however the first allows every row of the database to have a unique identifier. This can ease caching of different objects in the application.

The table-per-class-family poses some limitations with keys as it was explained in 2.4.1.1. Also, when multiple databases are used in the same application and object schema, it is likely to encounter key collisions if some preventive measures are not taken.

Common solutions for using meaningless keys lies on using auto-generated fields or database counters (sequences) that auto-increment the column value. Then have that column mapped to an identity attribute in the object itself.

However, one problem with auto-generated keys in the database, is that the primary key is only generated when the data is committed. That does not allow the application to know which key was generated in the first place. In most cases, the application runs a single transaction where it inserts a `person` in a table and its `profile` or `bank account` in other tables that have foreign keys pointing to the first one and only commit all changes in the end. Because there is no key before committing data to the database, it is impossible to code some logic with it before persisting and loading the persisted object.

Another solution consists of having separate key tables that contain the next keys available for each relevant table. Thus, the application has to read that separate key table before inserting a row in the relevant table. Concurrency problems may arise when two clients read the same next-key before using it to insert a row in the relevant table. This can be prevented by using a procedure that increments the next key value whenever it is read.

Keys can be generated within the application as well. For that it is common to use the Globally Unique Identifier (GUID) provided by the platform API which grants its uniqueness in all machines. Although, they are big, eventually they become hard to read or debug and slow in performance.

As the object paradigm does not provide data constraints like the relational paradigm, the developer has to implement some additional measures in the Domain Model. One of them is to make the object identity field immutable or read-only. This can be solved omitting the setter in Java or having a read-only property in .NET. Then, implement a constructor with the object identity as parameter and omit or make the empty constructor private.

These problems and solutions only take place in a Domain Model (see 3.1.3). More details on these can be found in Fowler's Identity Field pattern [Fow02].

One important feature provided by relational databases is the isolation levels and transactions. Once the entity is retrieved from a database row into an object, it no longer takes advantage of the ACID control. At this stage the same entity can be retrieved again and coexist in the same application as another object instance in different persistence context or session (e.g. Session or EntityManager). Whenever the entity object wants to return to the table row for an update, the same row could already have been updated. Thus, with an optimistic locking strategy, only the last update will take effect.

This commonly brings headaches form transaction control and locking policies, although it is not so much a particular ORM problem. Nevertheless, these ORM tools often provide plenty caching support to amplify this problem especially in clustered environments.

2.4.5 Data retrieval

Querying data from a database without using SQL may seem unreasonable for many developers, today. That is because many infrastructures are built on pillars of SQL, with SQL oriented business, reporting and frameworks. Also, throughout decades of SQL, many developers have been assimilating the relational paradigm for as long as they begun to use it for all purposes.

One problem of the ORM is make the object schema *queryable*, so that it can provide powerful queries enough to compete with the SQL features and still be mappable to SQL.

Frequently, ORM tools have more than one query mechanism, which provides different levels of flexibility, speed and simplicity. There are three known methods for querying objects in the ORM paradigm: Query-By-Example (QBE), Query-By-API (QBA) and Query-By-Language (QBL).

QBE consists on using an object as a template to retrieve other objects like that one. It is the simplest and most high level strategy for it adopts the philosophy of giving an example to demonstrate what the query result should look like. However it fails for more complex operations which require a larger number of QBE operation calls or queries to the database at once and thus sacrificing performance. This strategy can also pose a problem for the domain objects since, at some degree, there will be domain objects with null fields that, according to business logic, cannot be nullable.

The QBA consists on applying sub-queries to the raw result in an object API, thus filtering conditions in a criteria like way. Therefore this is the most verbose strategy and is easily better supported by any Integrated Development Environment (IDE) due to an object oriented standard API. Even though it can have some features similar to SQL, it still lies within the scope of object paradigm, granting a higher flexibility than the QBE strategy. However, join operations are difficult or in many cases impossible to achieve with QBA. Although, it is common to encounter some lack of compile validation support for this strategy. Hibernate Criteria API and Language Integrated Query (LINQ)-to-Entities (EF) are two examples of QBA.

QBL is the most flexible and lower level strategy to query the object schema. It is also known as SQL for objects. Two examples of such languages are Hibernate Query Language (HQL) (supported by Hibernate and NHibernate) and Entity Query Language (EQL) (provided by EF). QBL is an entirely new language with specific commands, in a way very similar to SQL, featuring only a subset of SQL itself. For that matter it is often questioned if all the trouble to change for a query language so close to SQL but with fewer features is worth the trouble. One advantage of QBL is that it provides smaller and simpler queries, allowing to navigate throughout the object schema without worrying about projections or join keys, unlike the SQL, which on the other hand, may originate two or three times larger and more complex queries.

SQL functions or stored procedures are not well embraced by ORM tools neither its community mostly due to separation of concerns. Furthermore, QBL is not commonly integrated with IDEs, and is used by developers without any syntax highlighting, compile validation or debug support, hence being more susceptible to bugs. Nevertheless even though joins are available in most QBL engines, they are a relational feature which is often not recommended to use. When join operations become a common necessity for querying an object schema, either the object model lacks relevant associations or those scenarios are better suited to a relational model.

2.4.6 Partial-Object dilemma and Load Time trap

It is common, especially in distributed environments (e.g. client-server web applications) to optimize bandwidth. Therefore the transferred data must be reduced to the minimum amount needed by the end point (e.g. client application that requests data often) and still be aware of the overhead each round-trip provokes.

Therefore, it is important to predict the average amount of data to be required from the end point. That can be calculated through statistical analysis and probabilities, in the same way Online Analytical Processing (OLAP) calculates data trends for databases. Note that there is no automatic analysis and optimization of object schemas and business logic code yet, which has to be a manual process of business modelling.

The real load time problem lies on the fact that, for instance, when retrieving an object from database that has various fields and associations with other objects and so on, the loading time of that object can be very inefficient, especially if the intention was to only use a small subset of the fetched data.

Thus, optimizations can be made so that objects are instantiated with only the basic field types (integers, strings or booleans) loaded eagerly and the associations (collections or single references to other objects) loaded lazily or on demand. That it, at first the object comes with only the relevant fields assigned and the others are filled as the user accesses them via a Proxy. This is commonly called the Lazy Load pattern (by Fowler [Fow02]) which can be implemented using other known patterns such as Lazy Initialization (by [BW99]) or Virtual Proxy (by GoF [GHJV95]).

The Partial-Object has lured many developers to falling into a trap. One way of slipping into the trap is to fall for the N+1 query problem. For instance, consider the example of Figure 2.5 and focus on the one-to-many association between a `Music Store` and its `Employees`. When a `Music Store` is partially loaded, each has a lazy collection of `Employees` that is not loaded at first. If all the `Employees` are accessed eventually, they will have to be loaded one by one. Thus 1 trip to the database for loading the `Music Store` and N trips for its `Employees`.

Hence if the intention was to use all the data in the first place, it is more efficient to spare round-trip overhead time and sacrifice bandwidth at the beginning of the operation or even use the mean of both through a batch size in some situations.

For basic type fields, there is the Ghost object [Fow02] which loads only the object id at first and when any of the other fields is accessed, all the fields are fetched (1+1 database calls).

ORM tools face this problem in two different policies: convention over configuration, configuration over convention and the various tools mostly decline for one side or the other. Whether to use one or another strategy and the appropriate ORM is up to the developer, even though he may encounter unsuitable use-cases for the chosen approach.

The other way of slipping into the Partial-Object trap is to fall for the detached objects problem. While an object is attached to a session of the ORM, it lazily accesses to data via a proxy which is managed by the same session. However in N-Tier architectures, these objects are often sent to remote tiers such as client Graphical User Interfaces (GUIs). Thus, they become detached objects in a foreign environment where the session is unreachable. Hence there is no proxy to perform lazy fetch operations. This problem is often solved by forcing the object to load every field before being sent, or using Data Transfer Objects (DTOs) for data transfer between application tiers, which takes a more serious separation of concerns but with annoying extra code to manage.

Another problem is handling cyclic references when loading objects. Assuming that the object `Parent` has a a set of references to its `children` and each `Child` has a reference back to its `Parent`. Loading this object graph can reach to the point where one `Parent` loads its `children` and each `Child` loads its `Parent` again and so on. In order to prevent this problem, loading a `Parent` can be performed with the following methods:

- Eager loading runs an SQL query with a join between the corresponding two tables. This query can either be manually defined or constructed via a graph algorithm that navigates through all the accessible objects to that `Parent`. In here, the object graph is loaded from the query results, in which the same `Parent` joins with many `Child` rows. Thus, that `Parent` has to be verified in order to prevent many object instances of it to coexist in the a single client session. This can be achieved with an Identity Map, caching the `Parent`'s `id` and have them checked on every `Parent` load call;

- Lazy loading fetches a `Parent` and only when one of its `Child` references is accessed it fetches all the `children`. Loading this `Child` implies checking its foreign key to verify if its `Parent` is already an object instance. This can as well be done with an Identity Map.

2.4.7 Transparent Persistence

ORM tools and any other systems that manage persistent objects (e.g. OODBMSs), strive for transparent persistence. This basically consists of managing persistent objects and transient objects in the same way. Thus, using object persistence becomes easier and closer to Platform Independent Model (PIM), so the technological details are kept invisible from the developers [VZ10]. However in ORM, objects behave differently depending on its state. Typically an ORM framework provides three object states [BK06]:

- *Transient*, when the object is instantiated with the constructor and does not exist in the database. It also does not have any persistence aspects or is out of persistence context. Its behaviour is like of any other objects as if there was no persistence mechanism at all;

- *Persistent*, if it has a representation in the database. It is within the persistence context and any changes made to it during this time, can be synchronized to the database. With Lazy Load it fetches data as the associations are navigated;

- *Detached*, when an object has been persistent before but lost the persistent context, for instance if it was serialized to be sent to a different application tier. It is not navigable when it has Lazy Load implemented on its associations and also there are no incremental changes.

Detached and transient objects often need to be made persistent in order to prevent unexpected behaviour. That is, an object behaves differently depending upon its state.

This hinders, to some extent, the support for full transparent persistence. Because of that, a business logic developer cannot entirely abstract from the ORM details.

2.5 Alternatives to ORM

In the last few decades, RDBMSs adopted the strategy of “One size fits all”, thus designing a generic solution to support all kinds of data-centric applications [Sto05]. For that reason, RDBMSs have been the conventional database applications elected for a wide variety of purposes. Recently though, as demonstrated in [Sto05], RDBMSs are a poor fit for many situations.

Scalability has grown an increasingly essential requirement for the Web industry, which forced companies like Amazon or Google to abandon the traditional RDBMSs for the Not only SQL (NoSQL) databases. Other solutions consist of distributing caching layers on top of RDBMSs or scalable Online Transaction Processing (OLTP) RDBMSs.

In fact there are two kinds of scalability: the vertical and the horizontal. The former means to add capacity to a single machine, often done by adding CPU cores and increasing memory. The latter means to add more machines. Every database product is vertically scalable. However, hardware upgrades on a single machine are limiting. So, to horizontally scale a RDBMS is something that can only be implemented on the client side and can be accomplished through sharding (dividing data into horizontal partitions) or data replication. Despite the many challenges, Facebook and Twitter have managed to do this by distributing MySQL database servers.

Moreover, to distribute a system in the network, it is likely that an asynchronous model will be required, where there are no clocks available. This allows the existence of stale data and thus, according to the article [GL02], it is not possible to simultaneously provide a system that ensures consistency, availability and partition-tolerance.

The NoSQL term can be deceiving for it categorizes databases that do not follow the relational model rather than having direct effect on the SQL itself. Moreover this kind of databases is usually known for the following characteristics [Cat11]:

- Horizontal scalability, data replication and distribution over many servers or nodes;
- Simple call API or protocol in exchange for SQL;
- Efficient use of indexes with scalability, via distributing and replicating the caching of objects over multiple nodes;

- Weaker concurrency model through eventual consistency and the abandonment of ACID transactions;

- Dynamic or no-predefined schema with no downtime for schema updates, thus improving extensibility;

- High availability through transparent failover and recovery.

NoSQL products like Cassandra and Voldemort enable the automatic partition of data by adding more machines, thus taking care of rebalancing the cluster. Also, Google's BigTable and Amazon's Dynamo have been scaling to thousands of nodes efficiently.

On the other hand, there has been great effort to make RDBMSs scalable. Some examples include MySQL Cluster and Oracle RAC. The latter is officially stated to scale up to 100 nodes. These are categorized as the new RDBMSs and consist of an extra layer that provides the clustering and high availability environments to the traditional RDBMSs. Nevertheless, this approach is not new. For a decade, systems like TerraCotta or GemStone have been already providing a distributed caching layer replacing some or all RDBMS operations [Cat11]. These systems still support ACID properties, unlike the NoSQL. Therefore they are not expected to scale well if both the SQL operations and the transactions span many nodes [Cat11].

Despite the extent of capabilities provided by the many different kinds of systems above, very few present any support to rival the popular ORM frameworks:

- Some RDBMSs such as Oracle have extended themselves to incorporate ORM features so that data can be provided via language specific object APIs;

- Application frameworks like **GemStone** do provide an object oriented front-end, although it still manages an ORM layer for keeping object schemas and relational databases synchronized;

- The main focus for most NoSQL databases (*key-value*, *document* and *extensible record stores*) is to provide high performance, availability and scalability through simple, often dynamic data structures. Therefore, complex object models always require a mapping layer similar to the ORM;

- The NoSQL *graph database systems* such as **OrientDB**, save objects in XML like structures, provide referencing of nodes throughout the graph and perform a simple mapping to plain objects (e.g Plain Old Java Objects (POJOs) in Java). This presents a poor alternative to ORM and proves inefficient for complex object models;

- OODBMS like DB4O are sometimes excluded from the NoSQL category due to some contrasting characteristics. Although, they truly represent an alternative to the popular ORM frameworks in some cases. More details are presented below.

These systems can and often are pluggable to each other which enables the design of confusing architectures. Therefore, at this stage it is important to understand where each system fits into the whole.

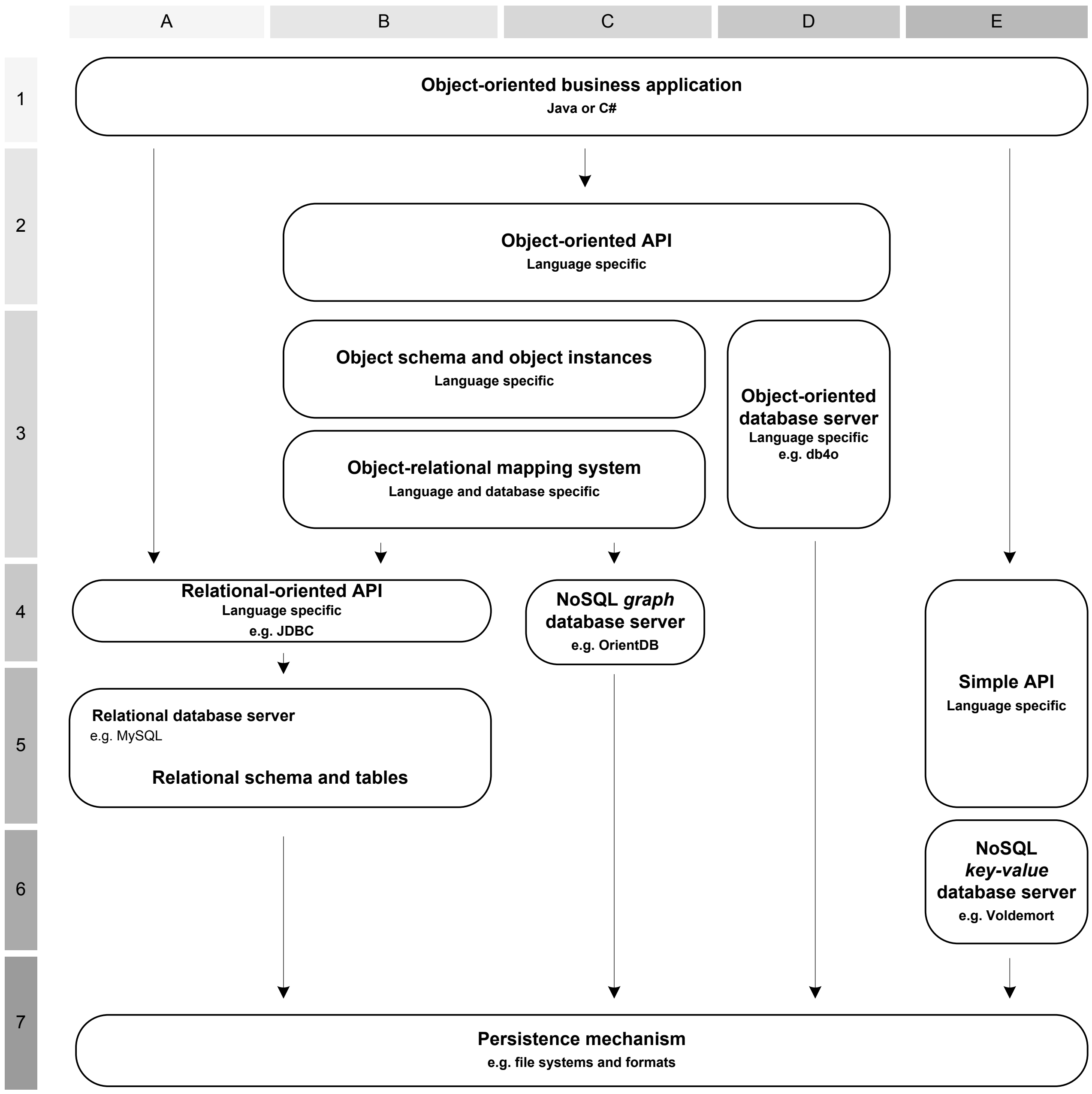

A
B
C
D
E
1
2
3
4
5
6
7
Object-oriented business application
Java or C#
Object-oriented API
Language specific
Object schema and object instances
Language specific
Object-relational mapping system
Language and database specific
Object-oriented database server
Language specific
e.g. db4o
Relational-oriented API
Language specific
e.g. JDBC
NoSQL *graph* database server
e.g. OrientDB
Simple API
Language specific
Relational database server
e.g. MySQL
Relational schema and tables
NoSQL *key-value* database server
e.g. Voldemort
Persistence mechanism
e.g. file systems and formats

Figure 2.7: Diagram of the different architectural combinations of databases

Figure 2.7 focuses on N-Tier applications that use object oriented programming languages to implement business logic and especially demonstrates where the ORM and the OODBMSs fit among the other databases. To avoid confusion the distributed features are ignored in this example where the NoSQL databases are single instance components.

The diagram of Figure 2.7 delineates five parallel architectures in different columns:

- Column `A` defines an object oriented application that connects to a RDBMS directly (e.g.Java Database Connectivity (JDBC)). This approach leads to great programming effort when complex object models are required;
- Column `B` uses the same components as `A` plus an ORM framework which is still the most popular and mature approach of ORM paradigm;
- Column `C` demonstrates that NoSQL *graph* databases such as **OrientDB** can be highly flexible in the sense that they can provide ORM at some level and can use any other database as a data store;
- Column `D` proves the simplicity of using OODBMSs which enables a simpler architecture without change of paradigms along the way. It provides the easiest to use object API and proves flexible for being able to use other databases as data stores;
- Column `E` displays a NoSQL *key-value* database system and a very simple API. It is likely the faster solution and the most scalable.

Figure 2.7 displays different levels on the left, which means the components of different columns can be combined or exchanged if they do not overlap levels. For instance, a RDBMS such as MySQL can have custom pluggable storage engines other than **MyISAM** or **InnoDB**. Interfaces can and have been implemented to support schemaless databases like NoSQL *key-value* stores. This approach was used to implement MySQL Cluster so that it could accomplish scalability. In the same line of thought, the following rather inefficient example consists of having `E6` (Voldemort) plugged into `A5` (MySQL) which then can be used as a data store for `C4` (OrientDB) and again serves data to an OODBMS (`D3`).

Ultimately, the OODBMSs can be a good alternative to ORM frameworks and RDBMSs usually when applications must perform extensive reference-following as its data does not overcome memory size [Cat11]. Some OODBMSs support ACID transactions unlike the other NoSQL databases although not yet as efficient as the RDBMSs. Horizontal scaling is also common in these systems which enables distributed reference-following and query decomposition [Cat11]. Withal, the OODBMSs present the following advantages towards the RDBMSs:

- OODBMSs can store highly complex object models including whole collections in different structures;
- Code and data can be stored and managed together as most object models and design patterns require the behaviour and state to stay in the same object;
- Object oriented programming languages such as Java or `C#` can be used to manipulate and query data. Thus there is no need to learn a new language like SQL;
- The data model in OODBMSs often specifies a dynamic schema which eases the extensibility and management of new features;
- There is only one data model,it can be defined in object oriented programming languages and there is no mapping between different paradigms like in ORM. This results in more productivity and avoids the impedance mismatch.
- Objects and the references to other objects are stored together through pointer swizzling techniques (see below for details about pointer swizzling). This improves search or traversal time as no join operations are required unlike in the relational model.

In turn, the OODBMSs present the following disadvantages:

- They still have performance drawbacks and consistency issues towards RDBMSs which, at the time, makes the OODBMSs a poor choice for data intensive and multi user applications;
- SQL is widely used for generating reports of relational databases. Learning Object Query Language (OQL) would be required for object databases. Plus there are very few graphical reporting applications for OODBMSs;
- Lack of solid mathematical foundation unlike relational databases which are set by relational algebra;
- Lack of standards. Despite the Object Data Management Group (ODMG) and Java Data Objects (JDO) standards, they were not adopted by many vendors. Also, the Java Persistence API (JPA) standard does not mention object databases.

In the end OODBMSs can perform well in mobile applications, prototypes and academic works or even serving as a caching layer to optimize speed of other databases.

The concepts behind an OODBMS are not any different from those of the object oriented paradigm. In the early 90s, Kim [Kim90] stated that an object oriented database is a collection of objects whose behaviour, state and relationships are defined accordingly to an object oriented data model. Soon did the OODBMS gain a standard, the ODMG [CB00], now disbanded. Even though that did not affect OODBMSs progress. The functionality provided by the various OODBMSs is commonly given by the following items [VZ10]:

- *Persistence.* Objects can be made persistent in a number of ways: *post-processing* (the *bytecode* of the class is modified), *pre-processing* (the source code is modified before compilation), via a modified virtual machine or by reachability (when a root object is made persistent, all the objects referenced by it must be made persistent). Also an object can be in the *persistent state*, if its data is already stored in the database or *transient state* if otherwise. Finally, the concept of *transparent persistence* consists of hiding persistent code from the application, thus providing a better separation of concerns for the developers, as it is given by *post-processing*;
- *Transaction management.* The same ACID properties used in the relational theory are once again an important feature for granting data integrity;
- *Recovery and backup management*;
- *Concurrency control*;
- *Version control and schema management.* A schema manager allows the object database to load specific versions of the object schema as any schema change makes it evolve or increment its version;
- *Indexing.* It increases the speed of queries when filtering is applied to iterate over an indexed field. Thus, a search key is generated for that field and stored in an auxiliary structure such as B-tree or hashing table because they are faster to iterate;
- *Query processing, optimization and OQLs.* The query processing has two steps: compilation and execution. The ODMG standard defines Object Definition Language (ODL) (a Data Definition Language (DDL) to define the object schema and thus increase portability among OODBMSs), OQL (similar to SQL except object oriented) and Object Manipulation Language (OML) (the API for data manipulation);
- *Pointer swizzling*, lazy and eager loading. There is a mapping between the in-memory object and the disk format object. Thus, swizzling is a technique that converts the references among the objects from the in-memory absolute address

(also known as pointers) to the disk relative address. Among the various methods for pointer swizzling, the most popular are eager and lazy loading. With eager loading, the object is loaded as well as all the objects referenced by it. The lazy loading loads only the object itself, leaving its referenced objects to be loaded only when actually needed;

- *Clustering of objects* groups the objects physically (in-memory or disk) so that they can be loaded faster;
- *Optimization features, performance management and monitoring*;
- *Cache management*. Providing a buffer to cache frequently accessed objects increases their access speed due to avoiding disk access;
- *Visual interfaces*;
- *Security, authorization and authentication.*

Although some of these features are already in use for a long time in RDBMSs, there are some new patterns and techniques that do not only prove effective for OODBMSs but also for ORM tools and frameworks.

2.6 Conclusions

ORM is not just a fancy strategy but indeed a real necessity in the enterprise world. Most applications that follow the object paradigm and do not use any ORM technology to access relational databases, will still have their custom ORM mechanisms prepared for specific needs. Even though adopting an ORM tool seems the right approach, the fact is that there is a risk it might fail.

Furthermore, these tools are not limited by just the basic mapping strategies. They also strive to enhance performance, scalability, portability and separation of concerns by providing features such as lazy proxy initialization, Unit of Work, database interoperability, version control, caching and N-Tier support.

At this point, it is clear the impedance mismatch and the ORM technologies learnt to live together, which may incite into a fair amount of doubt when the time comes to find the most suitable option for persisting objects. Among the various paths to take, the developer is often recommended to decide for one and only one of the following seven alternatives in order to better suit his own situation [New06]:

1. Abandon objects totally and avoid the impedance mismatch mainly because ORMs in some cases may raise more overhead than they save. Use the relational model both in the client and the server in which the client's GUI is organized around rows and tables. Current GUI frameworks fairly ease this practice;

2. Give a 360 degree turn, abandoning RDBMSs completely and instead use OODBMSs. Thus again avoiding the impedance mismatch, Dual-Schema problem and accept implications like the disappearance of DBAs and logic in database;

3. Accept the ORM paradigm and manually or by code generation create the mappings while considering that all the ORM limitations do not pose a problem;

4. Accept ORM for some cases and complement its limitations through direct access to the Database Management System (DBMS) via JDBC or ADO.NET and still be aware of the Partial-Object, schema ownership, Dual-Schema and other problems of ORM;

5. Integrating relational concepts the object oriented programming language, typically with scripting languages rather than strongly typed. SQLJ for Java and LINQ for .NET are two examples of this strategy;

6. Integrating relational concepts into objects changing at some level the perspective of the object oriented paradigm using more relational friendly models with low impedance mismatch. This may seem a radical approach still it is currently the most used.

Each of these techniques impacts differently on performance, scalability, cost, applications behaviour, development time and maintenance and none is best for all criteria. While some applications work well solely with the relational model, others will grow in size and complexity becoming easier to manage with the object model.

Chapter 3

Design Patterns for ORM

In the context of Software Engineering, the use of *Design Patterns* was initially proposed by Gamma et al. [GHJV95], and has become a common practice in modelling software architectures. The authors proposed a pattern collection which came to be commonly known as the Gang of Four (GoF) Patterns. Each GoF pattern is a granular design solution, described in a generic way so it alone, can serve as a guide to implement many various solutions to fit different problems within the scope of object oriented design. It is common for those who have not heard about the GoF patterns, to already have experienced situations when they had to use some of them.

Despite the fact that the GoF patterns are today considered good practices by the community, each problem is unique and thus may find its best solution outside the scope of any documented patterns. Sometimes a good solution can even go against a documented pattern, although it is not very common.

Since the disclosure of the GoF design patterns, other publications arose describing new design patterns, mostly object oriented. Examples of such are pattern collections by Beck [Bec97], Alur et al. [AMC01] and Fowler [Fow02] (the PoEAA patterns from the title of Fowler's book *Patterns of Enterprise Application Architecture (PoEAA)*).

The patterns presented in PoEAA are less granular than the GoF patterns in the sense that they are more specific for N-Tier architectures and higher level design. Also PoEAA patterns such as Lazy Load, Unit of Work, Query Object and Active Record do address the Object Relational Mapping (ORM) problems. Thus, this book is an important reference among the ORM community developers that use and implement ORM tools and frameworks. Therefore, the PoEAA book is mentioned many times within this section.

After a brief analysis of the PoEAA patterns it is evident that Fowler was influenced by the early GoF design patterns. He created a contextualization with some low level patterns called the base patterns which in turn do not differ much from the GoF patterns. For instance the Plugin (PoEAA base pattern) is a customized Factory Method (GoF pattern). Also the Gateway (PoEAA base pattern) resembles the Facade and the Adapter (GoF patterns). The Gateway provides an API that commonly wraps code for accessing external resources. Thus, in a client-server scenario it has to be written by the client of the service. On the other hand, the Facade, encapsulates an internal API and thus it is usually part of the service itself. Adapter often implies the existence of a previous API that faced some incompatibilities with its clients unlike the Gateway. Adapter can still be used to map some implementation to the Gateway interface [Fow02].

This section presents an analysis of the most relevant ORM patterns of the PoEAA book from chapter 9 to 13.

3.1 Domain Logic Patterns

The three patterns described here provide different ways of organizing the domain logic and the data access layer (Data Access Layer (DAL)) thus better supporting both the business logic and the presentation layers. The most relevant PoEAA patterns introduced here are: Transaction Script, Table Module, Domain Model, Table Data Gateway, Gateway, Data Transfer Object (DTO), Money, Value Object and Record Set.

3.1.1 Transaction Script

Transaction Script is a very efficient method to build domain logic and DAL for it requires very little overhead, therefore providing high performance. Also, its simplicity allows the code to be easily perceivable, which makes it natural to organize the logic code in applications that require it in less quantity and complexity.

This pattern organizes logic as a set of single procedures making direct calls to a database or through a thin DAL usually using the Table Data Gateway pattern. In the first case, the class that implements the Transaction Scripts has its domain logic methods to deal directly with Structured Query Language (SQL) scripts and database connections. The second case can be described in a generic way by Figure 3.1.

A Table Data Gateway much resembles the Data Access Object (DAO) pattern [AMC01] except it focuses on a simple Create, Read, Update, Delete (CRUD) API that is exclusively relational oriented. Also, usually its methods return row sets although it can use other structures like DTOs. On the other, the DAO pattern is a Facade CRUD for any data source and is more common to manage DTOs.

The Table Data Gateway in Figure 3.1 demonstrates its proximity to SQL API. Sim-

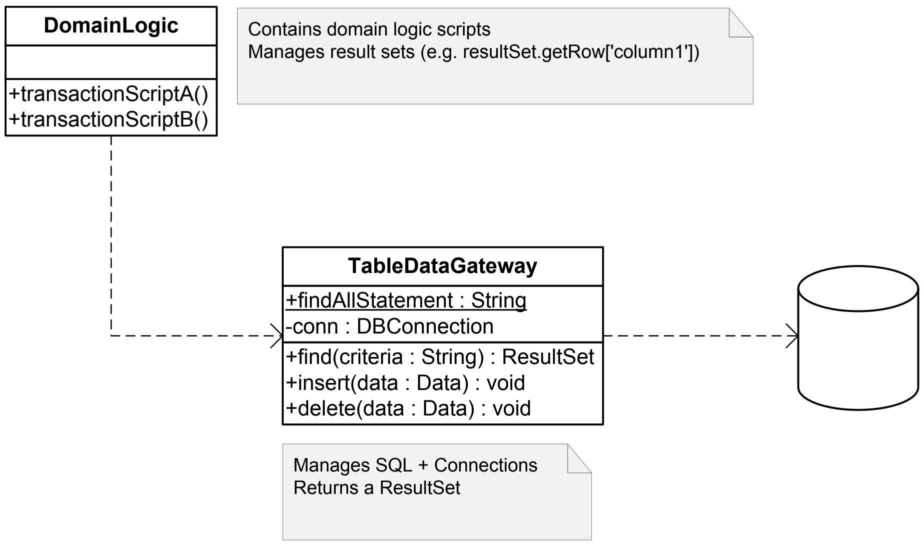

DomainLogic
+transactionScriptA()
+transactionScriptB()
Contains domain logic scripts
Manages result sets (e.g. resultSet.getRow['column1'])
TableDataGateway
+findAllStatement : String
-conn : DBConnection
+find(criteria : String) : ResultSet
+insert(data : Data) : void
+delete(data : Data) : void
Manages SQL + Connections
Returns a ResultSet

Figure 3.1: Class diagram of the Transaction Script pattern

ilarly to the DAO object, it also defines that it ought to exist one table per class as demonstrated in Figure 3.2.

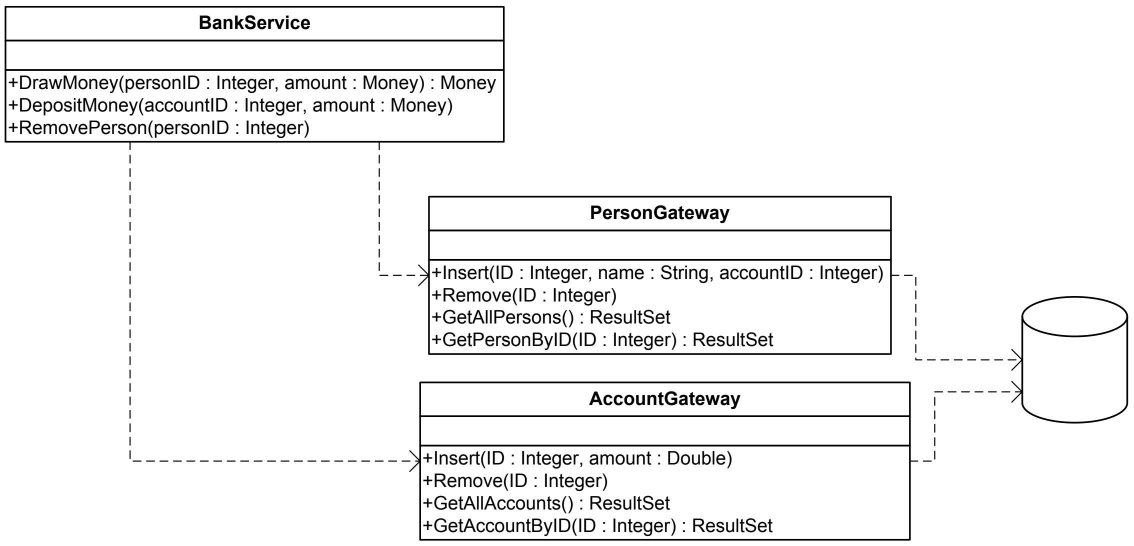

BankService
+DrawMoney(personID : Integer, amount : Money) : Money
+DepositMoney(accountID : Integer, amount : Money)
+RemovePerson(personID : Integer)
PersonGateway
+Insert(ID : Integer, name : String, accountID : Integer)
+Remove(ID : Integer)
+GetAllPersons() : ResultSet
+GetPersonByID(ID : Integer) : ResultSet
AccountGateway
+Insert(ID : Integer, amount : Double)
+Remove(ID : Integer)
+GetAllAccounts() : ResultSet
+GetAccountByID(ID : Integer) : ResultSet

Figure 3.2: Class diagram of a Transaction Script example

Figure 3.2 presents an implementation example of the model in Figure 3.1. The thin database access layer is given by the classes `PersonGateway` and `AccountGateway` which, using the Table Data Gateway pattern, define the methods for direct access to the database providers (e.g. ADO.NET). These classes encapsulate the data providers as well as the SQL statements.

The Transaction Script itself is implemented by the `BankService` methods which contain all the business logic, including information of the relationships between tables.

For instance, in the example of Figure 3.2 consider there is a relation between the table `People` and `Accounts`. This information has to be implemented in the `RemovePerson` method so that once a person is removed, all his accounts are removed as well.

It is common in business logic development to encounter situations that require data encapsulation like money or dates for they are very susceptible variables that can be relative to the client's location and system configurations. For instance, in Figure 3.2, the object `Money` encapsulates a `Double` and implements Money pattern (see PoEAA for more details on this pattern).

In a Transaction Script especially using a Table Data Gateway, when a query is sent to the database, it returns a single or multiple rows in a generic structure like an object array containing all the rows of a query result and the column definitions. These structures keep a connection alive and do some caching as they fetch data from a database. Examples of such APIs are the ResultSet for Java (the one used in Figure 3.1 and Figure 3.2) and DataReader for .NET. Hence, the business logic method has to know, a priori what to expect from that dynamic structure (ResultSet). This forces painful logic maintenance whenever changes occur in the database schema.

Fowler [Fow02] gave the name Transaction Script to this pattern because most of the time there will be one Transaction Script (business procedure) per transaction of the database. For that reason the use of a Table Data Gateway may raise some doubt on where to draw the frontier between the database provider API and the business logic code. That is, in the case of Figure 3.2, the method `RemovePerson` must use a database transaction through the provider API to run various operations to the database at once. If it fails it can *rollback* and thus take advantage of the isolation levels and locking mechanisms provided by the database to prevent concurrency failures and thus grant the Atomicity, Consistency, Isolation and Durability (ACID) properties. Therefore, in order so solve this problem it is appropriate to:

- Encapsulate all the relevant services of the provider API in the Gateway class, including transactions;
- Have direct access to the provider API in the Transaction Script methods without a Gateway;
- Allow direct access to the provider API and encapsulate some of its methods in the Gateway as well.

3.1.2 Table Module

This pattern organizes domain logic in a database table (or view) per Table Module class containing an in-memory table structure and the domain logic operations that manage this structure. For instance, the `People` object instance that manages every `Person` row is a Table Module. Typically, there are as many Table Modules as tables of the database.

The Table Module does not profit from the object paradigm features like associations, polymorphism, generalization, etc. On the other hand, it better suits applications that follow a table oriented structure. Also, this pattern is much simpler than the Domain Model as it does not require the complex mapping of two paradigms. Thus, implementation speed rises, especially if database interoperability features are involved.

The key component of Table Module is an in-memory table structure representing the resulting row set of an SQL query run to the database. This structure follows the pattern of Record Set (a base pattern defined by Fowler in PoEAA [Fow02]) which contains all the rows and column definitions of a query result, typically a whole table. It is common for a Record Set to provide locking mechanisms (multi-user environments) and allow incremental updates so that the same structure can be committed to the database efficiently (similar to the Unit of Work pattern applied to the Domain Model).

Record Sets are often supplied by the software platform and a good example is the DataSet API of ADO.NET. The following examples and figures use the term DataSet rather than Record Set so they can be more practical.

One way of implementing a Table Module is have it manage domain logic, Record Sets as well as connections and SQL statements in the same class. Here, a set of Factory Methods will suffice to instantiate the Record Set within the Table Module. Another way is having an extra class, a Table Data Gateway that manages SQL and database connections as demonstrated in Figure 3.3. As the data source or the SQL changes, this model becomes easier to maintain. Note that this implies one Table Module and one Table Data Gateway per table or view of the database.

Figure 3.3 represents this second strategy. Here, there is no direct access from the Table Module to the database. The Table Module provides all the CRUD methods for managing the DataSet and implements more complex domain logic methods as well. The Table Data Gateway contains the simple SQL statements and the connection settings to the database as well as the methods for retrieving and committing a DataSet to the database. In the committing process it implements the logic that converts the changes made to the DataSet into SQL statements for requesting updates to the database.

The DataSet in Figure 3.3 is requested first by the presentation to the Table Data

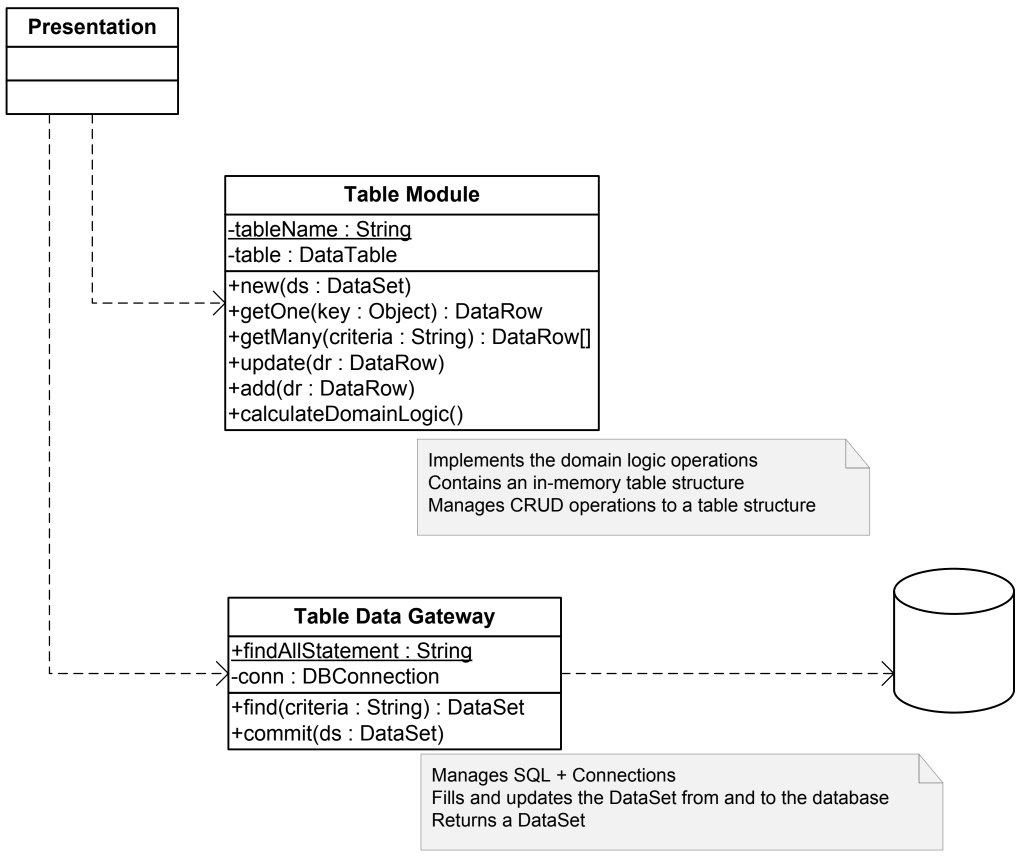

Presentation
Table Module
-tableName : String
-table : DataTable
+new(ds : DataSet)
+getOne(key : Object) : DataRow
+getMany(criteria : String) : DataRow[]
+update(dr : DataRow)
+add(dr : DataRow)
+calculateDomainLogic()
Implements the domain logic operations
Contains an in-memory table structure
Manages CRUD operations to a table structure
Table Data Gateway
+findAllStatement : String
-conn : DBConnection
+find(criteria : String) : DataSet
+commit(ds : DataSet)
Manages SQL + Connections
Fills and updates the DataSet from and to the database
Returns a DataSet

Figure 3.3: Class diagram of a typical Table Module with Table Data Gateway

Gateway and then used to instantiate the Table Module. Because a DataSet contains a set of results for cases that allow multiple querying at once, it is composed of a set of DataTables. Each DataTable represents the actual in-memory table and thus can better suit Fowler's definition of Record Set than DataSet.

The typical sequence of interactions among the above classes, for this strategy, is given by Figure 3.4. It is important to have the DataSet changes validated before committing to the database. That is to ensure data consistency and integrity when:

- The DataSet is shared through many users (dataset locking);
- The database is shared with other applications (database locking);
- There are constraints in the DataSet (mapped by the database schema) to be met.

In PoEAA [Fow02], Fowler gives an example of a useful modelling strategy for Table Modules that uses DataSets. That is demonstrated by Figure 3.5 and consists of using a generic class for instantiating any specific Table Module and its corresponding DataTable.

Figure 3.5 demonstrates a typical case study where people have bank accounts. If, in the database schema, there is a relation between both the correspondent tables, the

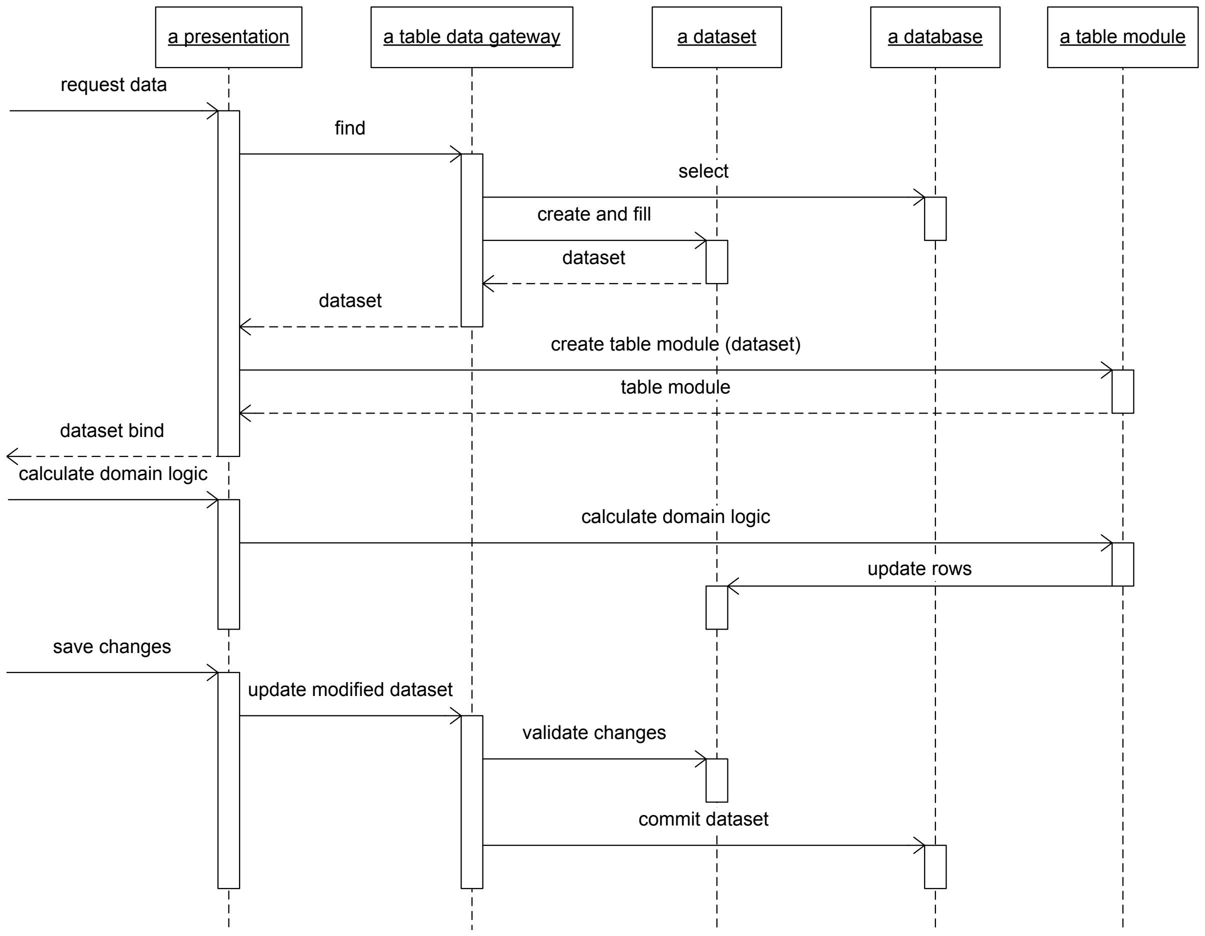


Figure 3.4: Sequence diagram of a typical Table Module with Table Data Gateway

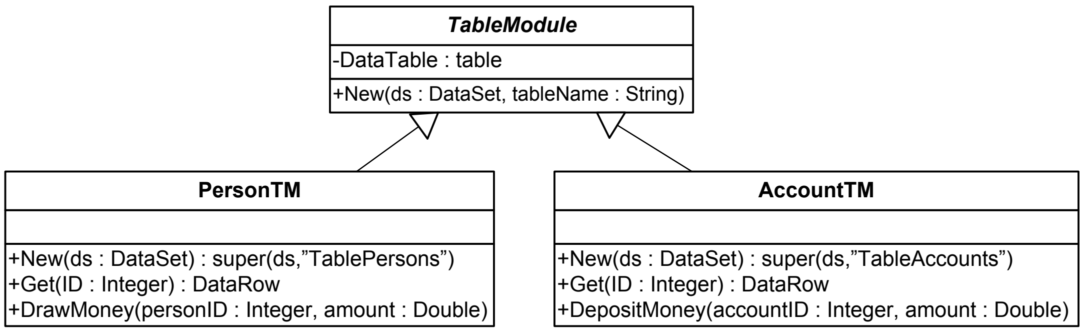

TableModule
-DataTable : table
+New(ds : DataSet, tableName : String)
PersonTM
+New(ds : DataSet) : super(ds,"TablePersons")
+Get(ID : Integer) : DataRow
+DrawMoney(personID : Integer, amount : Double)
AccountTM
+New(ds : DataSet) : super(ds,"TableAccounts")
+Get(ID : Integer) : DataRow
+DepositMoney(accountID : Integer, amount : Double)

Figure 3.5: Class diagram of a Table Module example

loaded DataSet will map this constraint as well. Thus, when both the Table Modules are loaded into memory, the dataset contains two related DataTables. Although, this relationship has to be manually coded in `PeopleTM` and `AccountsTM` logic. If a person is removed, for instance, the cascade delete constraint will imply that its accounts are deleted as well. This behaviour occurs during the DataSet validation.

In .NET, it is easy and common to implement the Table Module pattern via the

DataSet API, providing the following features:

- A DataSet manages all its modifications whilst building a *changeset* that further sends these changes to the database, rather than the whole thing. Thus, it manages the row states of `modified`, `added`, `removed` and `unchanged`;
- It manages the relations and constraints among tables and validates the integrity of data when the method `AcceptChanges` (available in DataSet or DataTable) is called;
- It provides optimistic locking concurrency control (as Fowler suggests for the Record Set pattern) for multi-user environments. Each row contains a version code to validate the updates;
- It performs in-memory ordering and filtering without database calls;
- As the DataSet is a dynamic structure, updates to the database schema may require less code changes, such as in the SQL statements in Table Data Gateway. Also, its relaxed type strategy delivers dynamic field accesses to the DataRows of a DataTable, in the Table Module (e.g. `personDataRow.item("Name")`), and thus the CRUD methods and the domain logic may need to be modified as well. This dynamism makes it easier for applications to use SQL statements in the domain logic or presentation layers;
- Presentation layer *two-way* data bindings, which is frequently used for grid Graphical User Interface (GUI) components, for instance in Windows Forms or ASP.NET;
- A dataset can be used as a DTO, which is good for distributed architectures as it has very few database calls, thus minimizing network overhead.

Nevertheless, the DataSet as the Table Module, can not connect to the database on their own. For that, a DataAdapter is required to manage the connections, fill and commit DataSet changes to the database. That is a similar behaviour to the Table Data Gateway.

On the other hand there are some drawbacks to using the DataSet API as the Table Module pattern:

- As databases tend to grow, it becomes infeasible to have the whole database structure or even a single table in-memory, thus resulting in great loss of performance;
- If other applications change de database when the dataset is still being modified, then it can fail to commit this data as concurrency issues may rise;

- Serializing DataSets is not straightforward and in the common distributed architectures of today (e.g. in Windows Communication Foundation (WCF)), it has lost some support for the ORM frameworks and object oriented schemas.

Of all the problems, the one about performance can be very restraining. Even so, the use of a DataSet can be tuned to handle less quantity of data while being more selective. One way of doing this is by SQL field projection, although that it may imply less flexibility for managing more complex or transversal domain logic. Another way to enhance performance is by applying pattern Lazy Initialization and handling SQL filters. This can be exemplified by Figure 3.6 which presents a particular implementation of the Table Module pattern. Here, the `DBContext` is more than a simple table data gateway for it keeps the whole dataset and the `PeopleTM` has knowledge of `DBContext` unlike in the other examples.

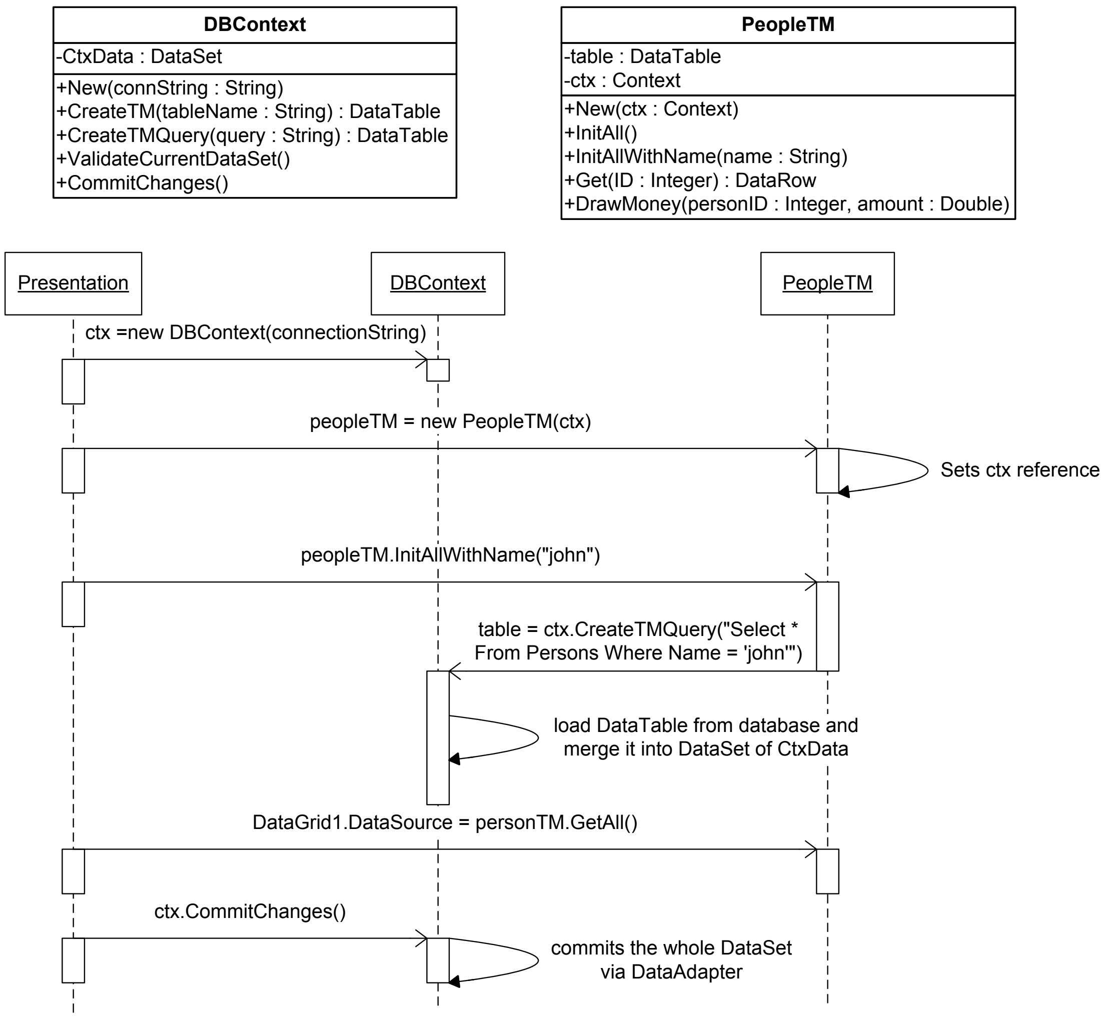

DBContext
-CtxData : DataSet
+New(connString : String)
+CreateTM(tableName : String) : DataTable
+CreateTMQuery(query : String) : DataTable
+ValidateCurrentDataSet()
+CommitChanges()
PeopleTM
-table : DataTable
-ctx : Context
+New(ctx : Context)
+InitAll()
+InitAllWithName(name : String)
+Get(ID : Integer) : DataRow
+DrawMoney(personID : Integer, amount : Double)
Presentation
DBContext
PeopleTM
ctx =new DBContext(connectionString)
peopleTM = new PeopleTM(ctx)
Sets ctx reference
peopleTM.InitAllWithName("john")
table = ctx.CreateTMQuery("Select * From Persons Where Name = 'john'")
load DataTable from database and merge it into DataSet of CtxData
DataGrid1.DataSource = personTM.GetAll()
ctx.CommitChanges()
commits the whole DataSet via DataAdapter

Figure 3.6: Class and sequence diagram of an example of a tuned Table Module

Additionally, the Virtual Proxy [GHJV95] pattern could be applied to this example so that when the `PeopleTM` DataTable was first accessed, it would implicitly load the data from the database.

Contrasting with the DataSet, ADO.NET provides the DataReader API which is similar to Java's ResultSet whilst using a more simplistic approach for data access. A DataReader iterates throughout the table rows of the database as it caches pages of results. Any change to the in-memory data is made via SQL updates directly to the database. As this does not provide the DataSet features, it allocates less memory. Therefore it can be regarded as a lower level API than the DataSet. The DataReader is often more efficient and less resourceful and thus may be a better solution than DataSet when queries return a lot of data.

In ADO.NET there is a DAL framework that uses DataSet API to manage the whole table schema and generate strongly typed DAL code accordingly. This is provided by the *Typed DataSets*. Figure 3.7 represents a diagram with the most relevant classes of a generated code by this *Typed DataSet* feature. It was generated from the database schema containing the tables `People` and `Accounts` of the examples above. Additionally, it presents a Table Module implementation.

The `PeopleTableAdapter` is a Table Data Gateway as it manages transactions, connections, internal SQL and also implements the basic CRUD methods for database direct access, like filling and committing a DataSet or a DataTable.

The `DataSet1` is a specific DataSet for this example as it contains all the tables and relations of the schema. Also, it manages serialization methods and contains event change handlers for the objects it is composed of. Note that `PeopleTableAdapter` is aware of both `DataSet1` and `PeopleDataTable` unlike in the example of Figure 3.3.

Both `PeopleDataTable` and `AccountsDataTable` are part of the `DataSet1` and they represent the Table Module classes. `PeopleDataTable` is a typed DataTable and it contains some generated CRUD logic for accessing the in-memory structure. Plus, it handles events for capturing changes on `PeopleRows`.

A `PeopleDataTable` is composed of `PeopleRows`. Each `PeopleRow` is a typed DataRow and follows a DTO like structure with getters and setters. Also it has a method for retrieving a person's accounts (using Lazy Initialization) as if it was a class association, which can be useful while coding complex domain logic. Another interesting feature is that a `PeopleRow` has encapsulated methods to check if database fields are null. That is, `DBNulls` are a common problem in the ORM, because database engines manage null values differently.

The typed dataset tools also provide a mechanism for updating this model from the

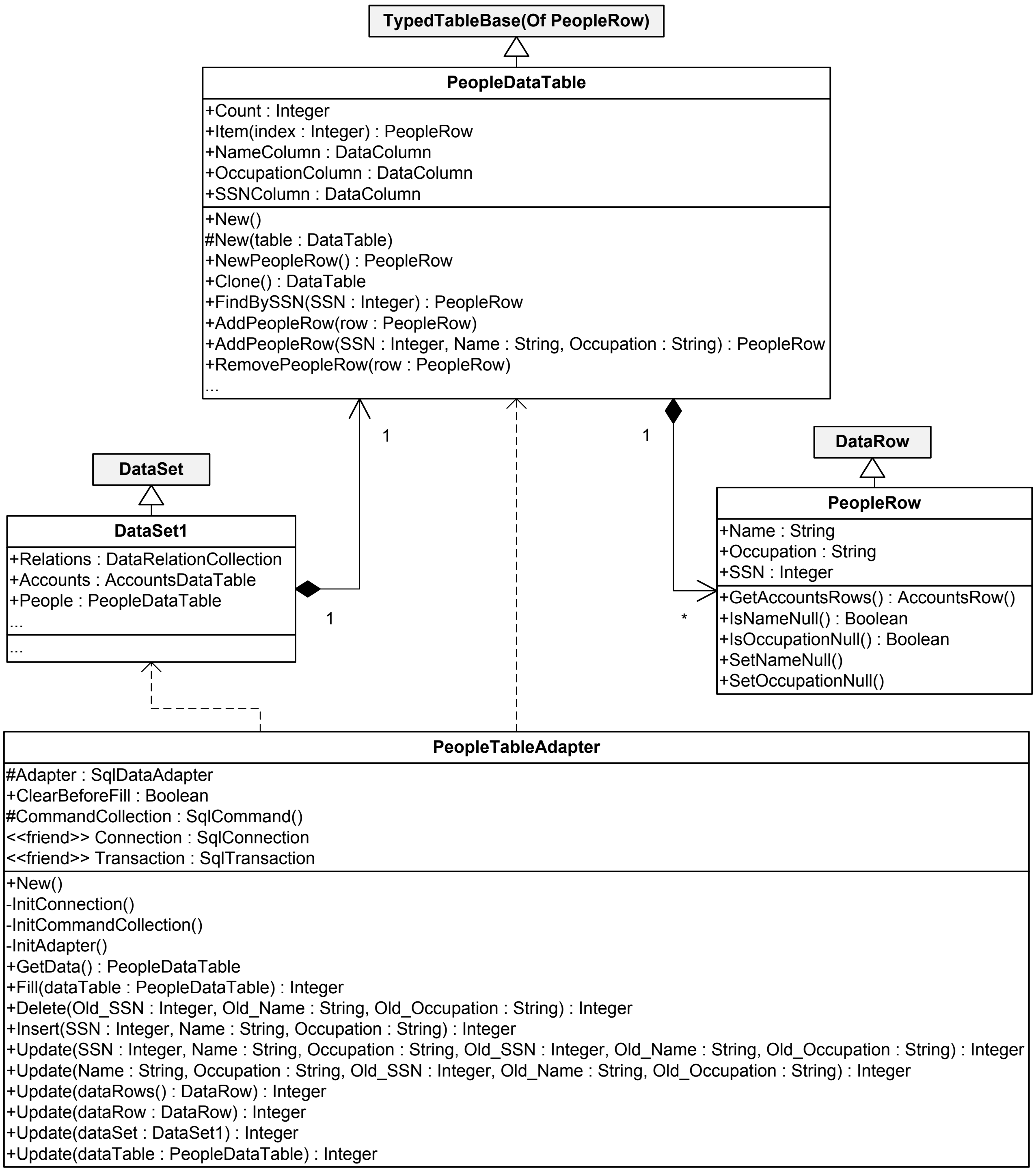

TypedTableBase(Of PeopleRow)
PeopleDataTable
+Count : Integer
+Item(index : Integer) : PeopleRow
+NameColumn : DataColumn
+OccupationColumn : DataColumn
+SSNColumn : DataColumn
+New()
#New(table : DataTable)
+NewPeopleRow() : PeopleRow
+Clone() : DataTable
+FindBySSN(SSN : Integer) : PeopleRow
+AddPeopleRow(row : PeopleRow)
+AddPeopleRow(SSN : Integer, Name : String, Occupation : String) : PeopleRow
+RemovePeopleRow(row : PeopleRow)
...
1
1
DataSet
DataSet1
+Relations : DataRelationCollection
+Accounts : AccountsDataTable
+People : PeopleDataTable
...
...
1
DataRow
PeopleRow
+Name : String
+Occupation : String
+SSN : Integer
+GetAccountsRows() : AccountsRow()
+IsNameNull() : Boolean
+IsOccupationNull() : Boolean
+SetNameNull()
+SetOccupationNull()
*
PeopleTableAdapter
#Adapter : SqlDataAdapter
+ClearBeforeFill : Boolean
#CommandCollection : SqlCommand()
<<friend>> Connection : SqlConnection
<<friend>> Transaction : SqlTransaction
+New()
-InitConnection()
-InitCommandCollection()
-InitAdapter()
+GetData() : PeopleDataTable
+Fill(dataTable : PeopleDataTable) : Integer
+Delete(Old_SSN : Integer, Old_Name : String, Old_Occupation : String) : Integer
+Insert(SSN : Integer, Name : String, Occupation : String) : Integer
+Update(SSN : Integer, Name : String, Occupation : String, Old_SSN : Integer, Old_Name : String, Old_Occupation : String) : Integer
+Update(Name : String, Occupation : String, Old_SSN : Integer, Old_Name : String, Old_Occupation : String) : Integer
+Update(dataRows() : DataRow) : Integer
+Update(dataRow : DataRow) : Integer
+Update(dataSet : DataSet1) : Integer
+Update(dataTable : PeopleDataTable) : Integer

Figure 3.7: Class diagram of an example using ADO.NET DataSet code generation

database. Due to these useful features ADO.NET DataSet tools are frequently used by developers today.

3.1.3 Domain Model

The most adopted domain logic pattern in ORM frameworks today is the Domain Model. It takes advantage of the flexibility provided by the object paradigm and supports more complex mapping than the above domain logic patterns. A Domain Model can be adopted at different levels of complexity and so Fowler defines two kinds of Domain Models [Fow02]:

- The *Simple Domain Models* define one class or entity per database table. The pattern Active Record is often enough to encapsulate the simpler mapping to the relational schema. At most, it implements unidirectional one-to-one and one-to-many associations. Other object oriented features like inheritance and the design patterns are avoided;

- The *Rich Domain Models*, on the other hand, diverge most from the relational model. They provide a flexible model that takes advantage of inheritance, associations, interfaces and more elaborate patterns like the GoF [GHJV95]. This model is more complex to match the database tables and thus, the Data Mapper pattern is usually required to encapsulate such mapping.

This pattern is often used in multilayer or N-Tier architectures. Since the Domain Model carries both data and business behaviour, it is subject to numerous changes of the logic domain, as it may provide DTOs to the upper tiers as well. Therefore, it is recommended that the Domain Model is isolated from the other layers. That also achieves better support for distributed objects and serialization, more independent builds and test development. Additionally, the Domain Model ought to be independent from the schema mappings to the database, otherwise any change of these mappings would affect the whole domain logic code.

Implementing a Domain Model incurs into an object loading problem, consisting of knowing which objects should and which should not be loaded to the in-memory graph. This incurs into the Partial-Object problem introduced by Neward [New06] as well. The Table Module sets a similar obstacle, although it has a relational structure which causes less trouble than the object graph structure.

For instance, a simple application of the Domain Model is to have a single-user application load the whole object graph into memory. Although this practice can be used in some desktop applications, it proves infeasible for data intensive multilayer applications with large schemas, because loading every object can take a lot of time and memory resources.

Object-Oriented Database Management Systems (OODBMSs) solve this problem implicitly as giving the impression that the schema is all in memory whilst, internally, disk to memory operations are performed. Some ORM frameworks allow implicit object schema navigation as well (via Lazy Load pattern).

Typically, a Domain Model requires a data context (like the Hibernate Session providing a Facade API encapsulation and some PoEAA patterns presented below) to communicate with the database efficiently. A data context is essentially a bundle of other ORM patterns. In general, it takes responsibility for: loading a specific object sub-graph querying object schema via Object Query Language (OQL) (Query Object pattern); caching results (using the Identity Map pattern); managing changes (Unit of Work pattern) and updating the database (Data Mapper).

ORM frameworks such as Hibernate and Entity Framework (EF), use a data context class for the same purposes.

Figure 3.8 displays an example of a Domain Model that takes advantage of object features like inheritance, unidirectional and bidirectional associations. The object schema is a navigable graph and, because it is a *Rich Domain Model*, each entity class does not implement behaviour.

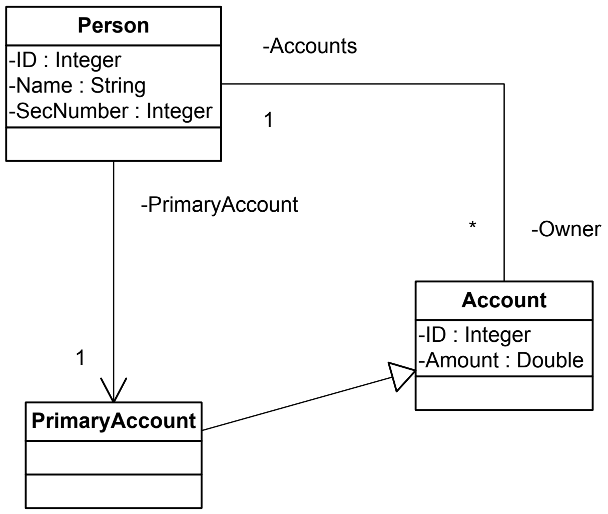


Figure 3.8: Class diagram example of a *rich Domain Model*

From the entity `Person` it is possible to access 5 fields of data, 2 of them through navigable associations in the graph. The business requirements state that one person has exactly one `PrimaryAccount` and can have zero or more `Accounts`. Thus, from a `Person` instance there is access to its `PrimaryAccount` as well as its `AccountsList`. Also, from any `Account` it is possible to access its `Owner`.

The data context will eventually be used to load the whole object graph or only a small portion of it, all according to the business requirements. If there is a trend for using the whole graph in the business logic or presentation calls within the same data context session, then it is more efficient to load it fully.

On the other hand, in Figure 3.8, if the business logic is operating only on `people`, then it is more efficient to only load `Person` data. Thus, each loaded `Person` instance has empty responsibilities or relationships with other entities which makes it a partial object. Graph navigation becomes impossible as well, unless there is some mechanism of Lazy Load that initializes the association on-demand. The Lazy Load pattern solves this for entity objects that are attached to a data context session. Although, for distributed environments, this implies that a data context session is continuous for remote calls, and thus has to provide *stateful* services, i.e. it needs to save the session state for each client. An alternative is to have the entire data context session transferred between the remote client and the server. These tend to be overcomplicated and inefficient solutions.

Thus, the data context is a server side component and is not designed to be transferred between tiers. Although, for instance, the Session is *serializable*, it triggers much overhead when transferred through the network. Also it loses some capabilities like the Lazy Load which requires an online connection to the database. In the same way, even if a DataSet is *serializable* and designed to work offline, there is still much overhead for transferring it through different tiers. Eventually, it has been discouraged by the community, to serialize and provide such structures remotely.

Withal, that limitation gives greater emphasis to the coupling between the 3-Tier architecture approach and the Domain Model pattern. That is, because a data context is exclusive to the business tier, it forces the developers to write all the relevant behaviour in there, hence having a duller presentation tier.

In Java Platform, Enterprise Edition (J2EE), Domain Models are typically implemented using the Enterprise JavaBeans (EJB) technology. EJBs provide a set of server side components for distributed architectures that help create modular enterprise applications. Most importantly, there are two kinds of EJBs: Session Beans and Entity Beans.

The Session Beans are more suitable for implementing *workflow*, which normally is part of the business logic code. They also manage logic operations on the business entities, keep *stateful* or *stateless* conversations with its clients but do not survive container crashes [AMC01].

Although Entity Beans were replaced by the Java Persistence API (JPA) entities in EJB 3.0, they consist of the first form of ORM framework provided by J2EE technologies. An Entity Bean defines a persistent domain object and were often used with Container

Managed Persistence (CMP). The CMP is an ORM framework that does not require the developer to code persistence mechanisms. Only the mapping definitions are required to set a domain model persistent to a database. Thus, an Entity Bean is a distributed, shared and transactional persistent object and it survives container crashes [AMC01].

A complex business layer, sometimes requires as much flexibility as the object paradigm can give. Entity Beans constrain this because they are Platform Specific Model (PSM) and so are bound to this technology. Plain Old Java Objects (POJOs), on the other hand, are considered Platform Independent Model (PIM) for they allow a clean separation between the Data Mappers and other ORM mechanisms from the business entities.

For that reason, EJB 3.0 developed a specification for abstracting the various ORM frameworks, whilst using entities as lightweight POJO classes. A common solution is to use the entity POJO classes and the XML mappings of an ORM framework such as Hibernate.

Opinions diverge towards the use of business logic inside the entities, which breaks the single responsibility principle [Mar03]. Complex domain models introduce entity to entity relationships which often makes it impossible to manage entity self-contained logic code and prevent dependencies to other entities [AMC01]. Also, entity serialization (e.g. via web service and XML or Remote Method Invocation (RMI)) often maps only the data contracts and not the behaviour of such entities. This compels to the use of DTOs, translation from business entities to DTOs and vice versa. DTOs allow the specification of safer security measures as well. However, with the DTO approach there is much more code to manage and it is likely that changes at the Domain Model will inflict changes on the DTOs and the presentation tier as well.

On the other hand, without additional DTOs, the domain objects have to be transferred to the presentation tier. Although some precautions must be acknowledged:

- Domain objects should not contain behaviour;
- A domain object typically contains more data, which may be in a rawer state than the one actually needed by the presentation;
- Lazy Load associations should be forced to load before the domain objects are transferred to a remote client or presentation tier, without a data context session;
- Editing a field value of a domain object in a remote client or presentation tier does not manage the *changeset* (no Unit of Work). The server rather thinks the whole object was changed.

Conceptually, the Domain Model pattern adopts a Domain Driven Design (DDD) approach that forces the development teams to set design practices and principles as tools for building core domain logic. This tends to work with Model-Driven Architectires (MDAs) as well, which focuses on modelling the domain before coding it. Thus, a Domain Model pattern is more suitable when applications begin being built from its domain rather than from the database.

However, many applications today, do operate on top of legacy databases or contain logic in the database. Typically, these applications do not respond well to the Domain Model approach as it tends to take the database schema from the Database Administrator (DBA) and give it away to the ORM framework. This incurs into the Dual-Schema problem [New06]. Hence, in a model where a legacy database is shared through various applications, the Domain Model approach may not be the fittest approach.

In the end, the Domain Model is useful because it focuses the developer to implement business logic code whilst, abstracted entirely (ideally) from the persistence mechanisms. That empowers the developer to use the best desired practices for modelling objects. Although ORM frameworks have given the misleading idea that this apparently flawless Domain Model is the answer for every problem. In fact, it triggers a number of problems that raise the complexity of applications and forces the development environment to fall within the ORM commitment (referred by Neward in [New06] as it is described in the section 2.4).

3.1.4 Making a Decision

In most cases, it is not obvious which domain logic pattern is best for fulfilling their own specific requirements and supporting the development teams. Some architectures may even adopt more than one single Fowler's domain logic pattern or none of the three.

There are a number of variables that limit the choice, such as the complexity of the domain logic involved, the tools and frameworks used and the know-how of the development team. As the complexity of the domain logic evolves, it becomes harder to maintain or change, as displayed in Figure 3.9. This figure does not provide a quantified axis as it is not based on any concrete data. It rather results from Fowler's perception of past experiences. Nevertheless it helps to visualize how the three patterns compare with each other.

Ideally one would find the best pattern by placing its application somewhere in the x-axis. Although, in practice there is no measure for this, which makes the best option to still rely on the advice of experienced people.

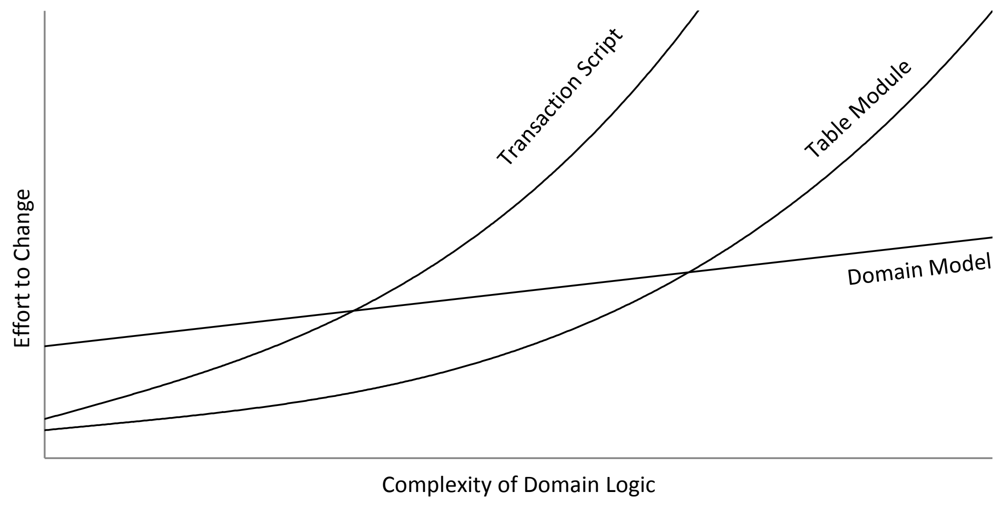


Figure 3.9: Sense of the effort changing rate as a domain logic complexity evolves for all three domain logic patterns (graphic replica from [Fow02])

Figure 3.9 emphasises the higher effort to start up a Domain Model that only begins to pay off the initial investment if the domain logic complexity increases substantially, whereas the other patterns start low but tend to raise exponentially.

The graphic in Figure 3.10 follows a similar concept to the above except it describes the learning curve involved with the three domain logic patterns.

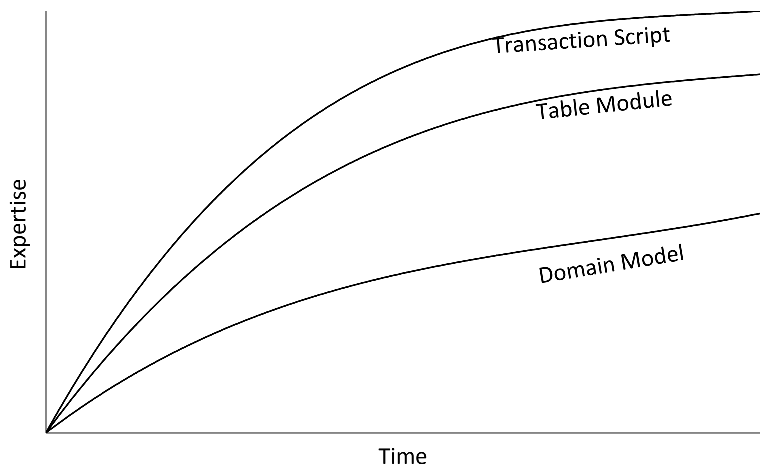


Figure 3.10: Sense of the learning curve taken among the three domain logic patterns

The Domain Model is more complicated to learn than the other patterns for it usually provides a number of mechanisms such as the Unit of Work, Lazy Load, caching, Query Object, Metadata Mapping, database interoperability and also allows the development of flexible and complex domain logic through the use of design patterns like the renown GoF patterns.

Moreover, the high initial cost of building or changing a Domain Model can be reduced or increased depending on the team experience and the helping tools. If a new ORM framework is being used, its learning curve will affect the starting efficiency of the team. An experienced team will lower the cost here. Without such tools, in a less automatic process, the initial cost raises substantially.

For instance, comparing NHibernate with EF it is likely that the initial cost will increase with the former but will evolve more rapidly with the latter. That is due to the higher complexity and versatility of NHibernate 2.1 towards EF 4.0 and also because the latter has better integration with Visual Studio. The learning curve will be slower with NHibernate as well.

Eventually, with a Domain Model, the domain logic grows easier to maintain mostly because it provides persistence ignorance whilst following a DDD approach.

The progress of the Table Module pattern depends on the tools provided by the development environment in the same way as the Domain Model. In Visual Studio .NET environment, the Table Module is supported by the DataSet API providing a set of useful features to communicate with the database. This increases productivity of the team, especially when the Domain Model is not an option (legacy databases) or the domain logic is not too complex for a relational model. The start up effort is small and the learning curve grows fast due to the simple DataSet API and the use of SQL. As SQL is common knowledge, it is assumed the developers feel comfortable from the start. Without such a mechanism the development team can either build it from scratch and face a very high start up time but with future gains or use a more manual process and still face a high start up time with increasingly more exponential effort as the complexity evolves. Whilst the former option is recommended when the domain model can not be used, the latter is not recommended at all.

The Transaction Script uses a simpler API than the other patterns, requires a more manual process and more SQL code as well. Thus it is easier to learn. On the other hand, with no automatic mechanisms, as the code grows, it becomes harder to manage or change. This pattern is the most used, typically in smaller applications, especially because it provides a fast start up time and every platform supports it. Also, the Transaction Script can be faster and lighter than the other patterns as it does not require any mechanism to work on top of the database provider API.

All in all, small applications typically use Transaction Script. In .NET, with an evolving domain logic and legacy databases, the Table Module is a good solution. When DDD is an option and the domain logic evolves to require complex modelling, the Domain Model is the right choice.

Changing from one domain logic pattern onto another often involves very few code reuse. Some applications adopt the Domain Model or Table Module for some business operations and the Transaction Script for critical functions that do not tend to change or must run faster.

3.2 Data Source Architectural Patterns

The patterns below constitute three different strategies for encapsulating the data source access. Thus, they provide a Facade API so that the domain logic can communicate with the database without having to write SQL or other data source specific code. Often, the domain logic pattern decided for an application, dictates which architectural pattern should be used.

3.2.1 Table Data Gateway

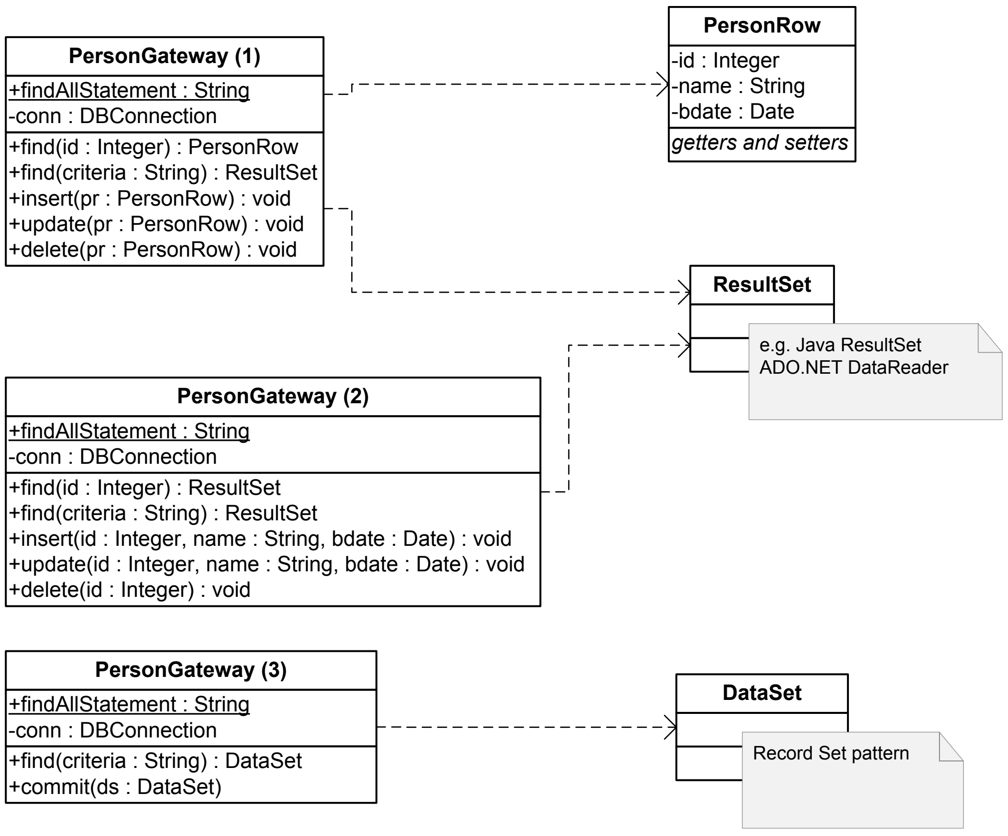
The Table Data Gateway is a specialization of the Gateway pattern that uses a table of the database. Thus, a Table Data Gateway instance represents the table itself as a data source which provides the methods: `insert`, `delete`, `find` and `update`.

Combining SQL with domain logic can cause some problems. That is, although many developers do not feel comfortable with SQL, others do, but may not write it in the most correct or efficient way. Also the DBAs needs to find the SQL quickly and easy within the application code as the database schema evolves or suffers some adjustments [Fow02].

Therefore, each method in the Table Data Gateway hides or encapsulates what is database specific like the provider APIs and the SQL code. A typical Table Data Gateway, for instance, a `PersonGateway (2)` or `(3)` (see Figure 3.11) returns to the domain logic, a dynamic structure (ResultSet or DataSet) that can represent any table.

Figure 3.11 provides three different applications of this pattern. The first two (`PersonGateway (1)` and `PersonGateway (2)`) are often used with Transaction Script. The third (`PersonGateway (3)`) is what connects the Table Module to the database itself.

With the Transaction Script often the Table Data Gateway manages simple row iterators (e.g. Java ResultSets or ADO.NET DataReader) for finder methods and primitive types for inserts, updates and deletes (`PersonGateway (2)`). Some cases opt for using DTOs as row data containers to call in the Table Data Gateway methods and thus increase portability whilst relieving the effort of refactoring the code when changes are needed (`PersonGateway (1)`). Table Data Gateway API is, this case, used directly in domain logic.



PersonGateway (1)
+findAllStatement : String
-conn : DBConnection
+find(id : Integer) : PersonRow
+find(criteria : String) : ResultSet
+insert(pr : PersonRow) : void
+update(pr : PersonRow) : void
+delete(pr : PersonRow) : void
PersonRow
-id : Integer
-name : String
-bdate : Date
getters and setters
ResultSet
e.g. Java ResultSet
ADO.NET DataReader
PersonGateway (2)
+findAllStatement : String
-conn : DBConnection
+find(id : Integer) : ResultSet
+find(criteria : String) : ResultSet
+insert(id : Integer, name : String, bdate : Date) : void
+update(id : Integer, name : String, bdate : Date) : void
+delete(id : Integer) : void
PersonGateway (3)
+findAllStatement : String
-conn : DBConnection
+find(criteria : String) : DataSet
+commit(ds : DataSet)
DataSet
Record Set pattern

Figure 3.11: Three ways of implementing the Table Data Gateway pattern

In the Table Module scenario, a Table Data Gateway manages Record Sets (e.g. .NET DataSets), and encapsulates all the behaviour for accessing the database. Nevertheless, it is the Record Set that actually provides a DAL API to be used in the domain logic, implementing the CRUD methods for managing row data.

For applications of Table Data Gateway in Transaction Script refer to 3.1.1. For Table Module and details on the DataSet structure or Record Set pattern, refer to 3.1.2. Note that ResultSet Java class is not the same as Record Set pattern, despite its similar naming.

In the end, the Table Data Gateway suits better the legacy database systems than a more refined approach like the Data Mapper because it allows the logic of querying data to be coded in plain SQL which is simpler and faster for most development teams to code.

3.2.2 Active Record

An Active Record is a domain object that carries both data and behaviour, i.e. it encapsulates one table row of a database, the data access code and domain logic applied to that data as well.

Normally, this pattern is used with a simple Domain Model, mapping one domain object per database table. There is a one-to-one mapping between the object fields and the table columns, with no data conversions. Typically it is necessary to apply the Identity Field pattern so that the unique object identifier (OID) matches with a primary key in the table. This key affects essentially the `find` and `insert` methods. For implementing associations and object graph navigation, the pattern Foreign Key Mapping can be used. Although, here, the complexity raises substantially, which deviates from the scope of the Active Record.

According to Fowler, the active record implements an API that provides the following operations [Fow02]:

- Build an instance of the Active Record from a record or row of a ResultSet;
- Build a new instance of the Active Record to insert into the database table;
- Static finder methods that encapsulate SQL queries and return Active Record objects;
- Update the database from the modifications made to the active record;
- Getters and setters;
- Implementation of some domain logic.

Figure 3.12 displays an example of an Active Record given by the class `Person`. It contains the person's fields, getters and setters, CRUD functions, static finders and some domain logic. It can also contain constants defining generic SQL statements so that they can be used in prepared statements.

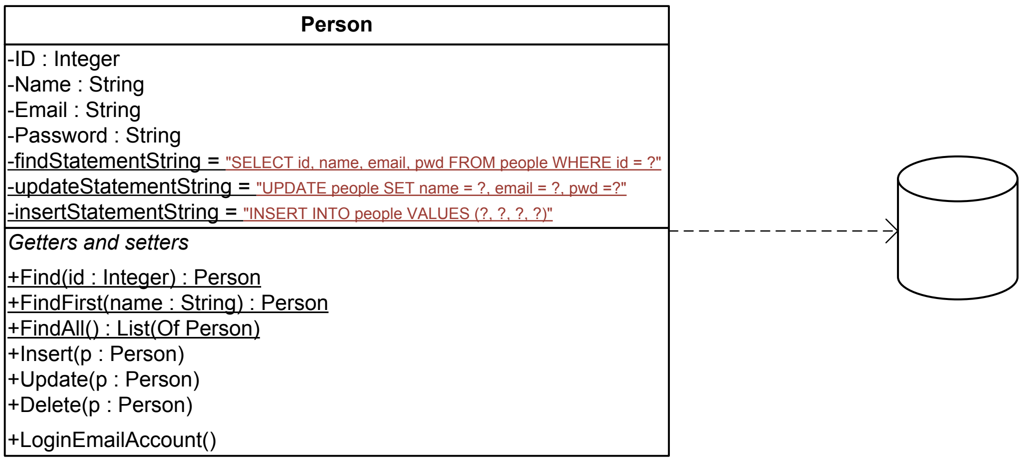

Person
-ID : Integer
-Name : String
-Email : String
-Password : String
-findStatementString = "SELECT id, name, email, pwd FROM people WHERE id = ?"
-updateStatementString = "UPDATE people SET name = ?, email = ?, pwd =?"
-insertStatementString = "INSERT INTO people VALUES (?, ?, ?, ?)"
Getters and setters
+Find(id : Integer) : Person
+FindFirst(name : String) : Person
+FindAll() : List(Of Person)
+Insert(p : Person)
+Update(p : Person)
+Delete(p : Person)
+LoginEmailAccount()

Figure 3.12: Example of an Active Record (class diagram)

Simple ORM database interoperability can also be achieved with Active Record and Figure 3.13 presents some code that features it.

```
private static Load(rs : ResultSet) : Person

static FindFirst(name : Strting) : Person
 If AppConfig.DataBase = "MySQL" OR "PostGreSQL"
     rs = database.ExecuteQuery("SELECT * FROM persons WHERE name = ? LIMIT 1",name)
     return load(rs)
 If AppConfig.DataBase = "Oracle"
     rs = database.ExecuteQuery("SELECT * FROM persons WHERE name = ? AND rownum < 2",name)
     return load(rs)
 If AppConfig.DataBase = "MSSQLServer"
     rs = database.ExecuteQuery("SELECT TOP 1 * FROM persons WHERE name = ?",name)
     return load(rs)

LoginEmailAccount()
    EmailServer.Login(This.Email,This.Password)
```

Figure 3.13: Code example in an Active Record

Active Record is useful in a Domain Model for its simplicity and speed of development towards Data Mapper. Also, the API it provides to higher levelled layers is more focused. However, it only responds well when objects match tables in a one-to-one simple mapping: isomorphic schema [Fow02]. In N-Tier applications, serializing an Active Record implies breaking its behaviour which makes it act as a DTO when working in a remote tier.

3.2.3 Data Mapper

Data Mapper is the most refined of the Fowler's data source architectural patterns and the most appropriate to work with complex domain logic. It is a specific form of Mapper which sets a communication between two systems that remain isolated from each other. The two systems are the object oriented domain logic and the database. The main responsibility of a Data Mapper is to transfer and convert data between these two systems.

A Domain Model that uses a high quantity and complexity of business logic has to be modelled with the flexibility provided in the object paradigm. Thus, supporting collections, inheritance, associations and a wide set of design patterns can be essential to achieve a well organized and more efficient business logic code. With the Data Mapper, since the two systems become isolated, there is no direct dependencies between them. That leads, however, to the Dual-Schema problem, because there are two schemas to manage: the Domain Model schema and the database schema. Generally, changes inflicted to one schema, lead to changes on the other as well.

Figure 3.14 presents a simple example using a Data Mapper that maps the domain object `Person` and the database. At first sight it appears to have some similarities with the Active Record. Although, the domain logic operations and CRUD methods suffer a slight organizational change towards better isolation, in the Data Mapper approach. It is often a good practice to support persistence ignorance among all the classes in the Domain Model. Thus, the domain objects are isolated from the Data Mappers in this approach.

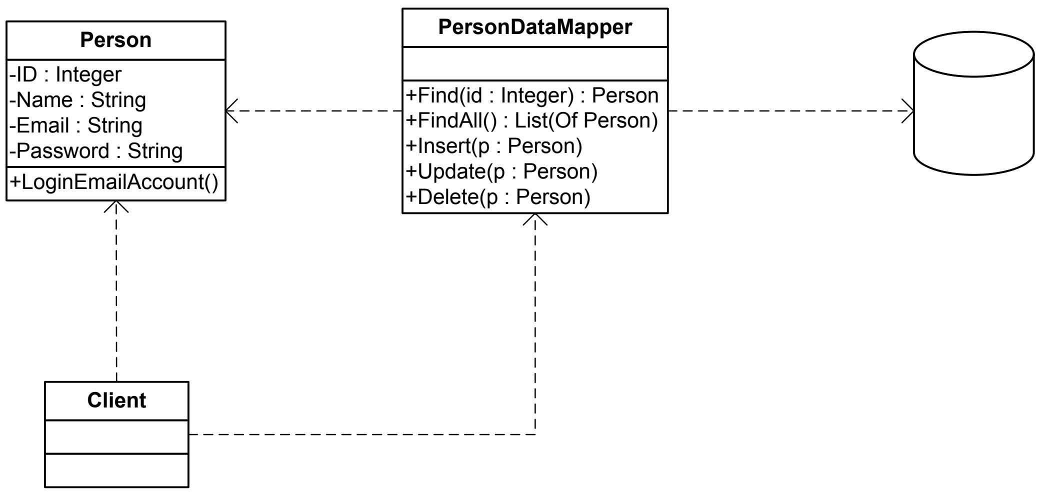


Figure 3.14: Data Mapper example (class diagram)

Figure 3.15 displays the sequence of operations carried out by the previous example when a client calls the `find` method of `PersonDataMapper`. Additionally, this example implements an Identity Map, useful for preventing bottlenecks, which caches loaded objects in-memory as they are fetched from the database. Here, the behaviour of converting a ResultSet into a Person instance is implemented in a similar way as in the Active Record.

The case of Figure 3.15 is a simple Data Mapper example and therefore does not manifest the real advantage of using the object's flexibility nor the negative effect of Dual-Schema.

One of the primary functions of a Data Mapper is to handle associations and inheritance between objects. The pattern Foreign Key Mapping is a sufficient solution for managing one-to-many associations. For many-to-many it requires the Association Table Mapping and a new table for the mappings. For composition where, for instance, a `Document` is composed of a `Collection` of `Lines`, it makes sense that one single Data Mapper manages the mapping for the `Document` and its `lines` as well. This can be done with Dependent Mapping pattern. More information on object relational association mapping can be found in 2.4.2. Inheritance can be mapped via the table-per-class, table-per-concrete-class, table-per-class-family. Details on object relational inheritance

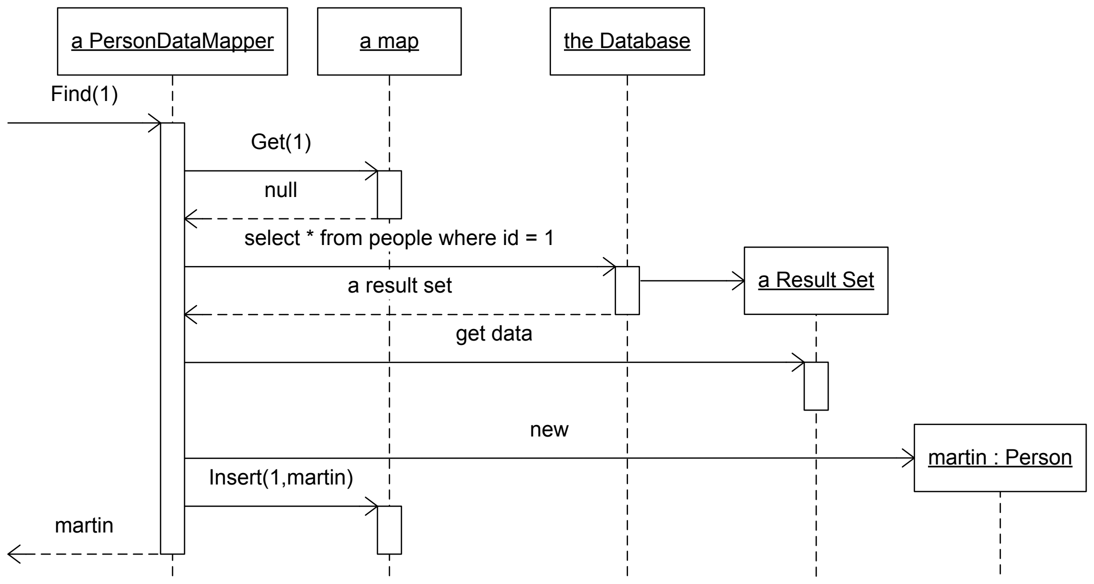


Figure 3.15: Data Mapper example (sequence diagram)

mapping are explained in 2.4.1.

It is a good practice to store static mapping information separately from the Data Mapper. This information mostly consists of the table and columns names and the corresponding class and fields names with the type of association and inheritance mapping toward other classes. One approach is to have these definitions on string constants in a specialized class (per domain object). Another strategy is to implement the Metadata Mapping pattern which stores all this information in a metadata file such as XML. This allows mapping details to be changed without recompiling the application as long as they do not force changes in the domain objects.

The more the Domain Model grows and evolves the more the need for transactional frameworks to manage changes in the objects and map them to the database. The Unit of Work pattern is a common solution for this problem.

Typically, a Domain Model is a graph of relations among objects. To load a `Person` instance may imply loading, for example, its `Accounts` as well. In such cases, the Data Mapper has to decide which SQL queries are faster and better suited to provide that kind of information. In the example, the SQL query shall involve a join between the `People` and `Accounts` tables.

With the increasing number of relations among objects of the graph, the eager loading proves inefficient because it tends to load the whole database in a single request, via the Data Mapper. Also it is likely that most fetched data will not be needed by the client. The Lazy Load pattern solves this, however making the domain objects not entirely persistence ignorant. This leads to the Partial-Object problem.

Another obstacle to the persistence ignorance is the typical need for a rich constructor, in a domain object, to set it to default values. For instance, assuming an entity has its id field mapped to an auto-generated key in the database. When creating a new transient object, it will not have an id until the object is made persistent, i.e. it is inserted in the database its key is generated. This may require the constructor to fetch the next generated key value and then instantiate the object with that id. An alternative to have this behaviour in the domain object itself is to put it in the Data Mapper. Thus, the Data Mapper becomes responsible for creating new domain objects via a Factory Method. Also it is important to make the constructor of the domain object invisible (protected) so that it is only accessible in the Data Mappers *namespace*.

In the various examples Fowler provides in [Fow02], he often implements Layer Supertypes for sharing behaviour of one class through all its subclasses. In this case he creates an `AbstractMapper` that generalizes all that can be shared among the different Data Mappers. SQL is often not generalizable though. Another example is to have all entities that have the same type of id to share the same *supertype*. In here, it makes sense to share the code that manages the id field in a *supertype* abstract Data Mapper.

In this pattern, there is no dependency from the Domain Model classes to the Data Mapper itself. Although, domain logic needs to find objects and change data. That kind of logic has to be dependent on the Data Mapper API. A common solution to isolate the domain logic from the Data Mappers is to build interfaces, implemented by the Data Mappers and use them in the domain logic classes instead. Note that, Factory Methods have to be implemented in order to create the specific Data Mapper.

Figure 3.16 shows how the Data Mapper relates to the other components and patterns in an ORM implementation. Generally a Data Mapper is implemented with the following guidelines:

- Without a Unit of Work, each Data Mapper contains its own Identity Map for a single domain object. Although, is some cases, the Identity Map can be shared among all the Data Mappers;

- There is typically one Data Mapper for each domain object, although it can comprehend more than one, normally in the case of hierarchies and compositions;

- In simple scenarios, one Data Mapper manages one table. When one domain object has data from two or more tables, the data mapper manages multiple tables. Hierarchies and compositions in the Domain Model can drive the Data Mapper to operate on multiple tables.

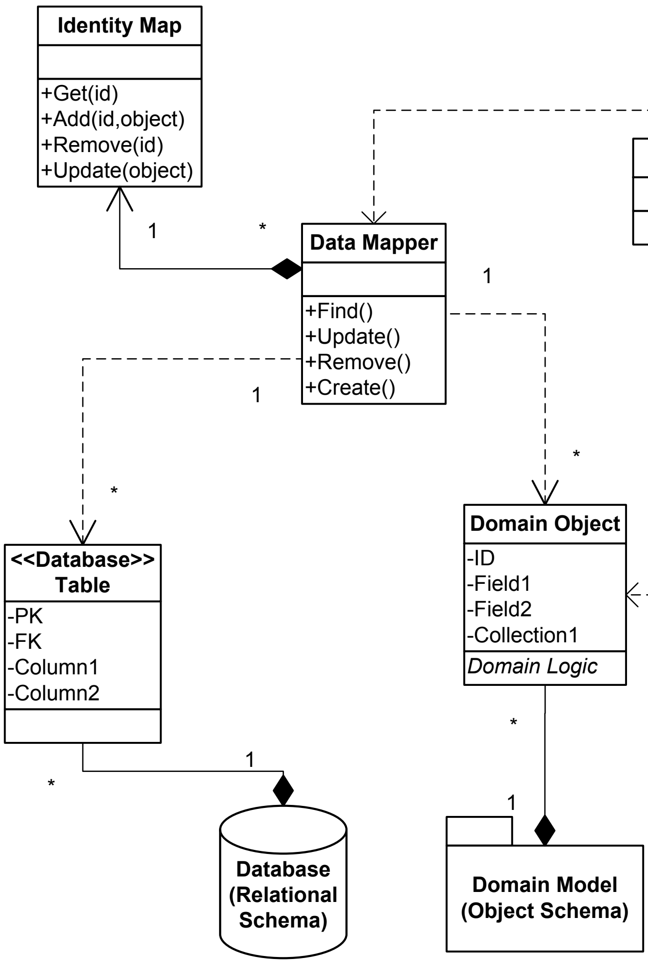


Figure 3.16: Data Mapper generic diagram

Additionally, one Data Mapper can be used per application or per domain object. If the same Data Mapper manages many domain objects, it tends to use reflection to make the code as generic as possible. This means it lacks compile-time checking, and therefore it becomes slower and harder to debug.

3.3 Object-Relational Behavioural Patterns

The following three patterns are common in ORM tools that use the Domain Model, providing strategies for managing the loading, in-memory keeping and updating domain objects. The implementation of these patterns enables a robust persistence layer and tends to enhance its performance substantially, which proves essential for rich Domain Models.

3.3.1 Unit of Work

The Unit of Work keeps a set of objects (usually in an Identity Map) affected by a business transaction, coordinates the updates or *changesets* of the objects towards the database and deals with concurrency issues.

Avoiding this pattern implies not having transactional support in the business layer. One of the implications of that is having to send the managed domain objects back to the database without verifying if they were even changed in the first place. The frequent but smaller calls to the database and the overhead provoked by the transfer of the whole objects rather than the *changesets*, can cause a substantial performance loss. Additionally, the database transaction has to be kept open during the entire interaction, including the user think time. In multi-user scenarios, this approach is exposed to concurrency problems such as the increased lock contention in the database. Thus, long database transactions prevent support for scalability and concurrent load [KBA+12].

Nested database transactions can constitute an attempt to manage *subtransactions* and solve these locking problems, however they lack Database Management Systems (DBMSs) support as they are not standard SQL. On the other, *savepoints* (an SQL standard) are a more encouraged mechanism, for the same purpose, to implement checkpoints during a transaction, although not solving the locking issues as well.

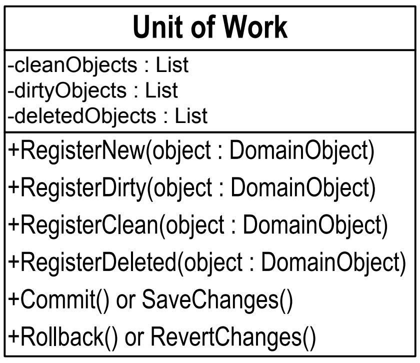

Unit of Work
-cleanObjects : List
-dirtyObjects : List
-deletedObjects : List
+RegisterNew(object : DomainObject)
+RegisterDirty(object : DomainObject)
+RegisterClean(object : DomainObject)
+RegisterDeleted(object : DomainObject)
+Commit() or SaveChanges()
+Rollback() or RevertChanges()

Figure 3.17: Unit of Work pattern

Figure 3.17 presents a simple API of a Unit of Work, to be used in the Domain Model for keeping track of changes during a business transaction and submitting updates to the database. The caller is responsible for manually registering the created new domain objects (`RegisterNew`), the changed objects (`RegisterDirty`), the loaded unchanged objects (`RegisterClean`) and the deleted objects (`RegisterDeleted`). Internally, the Unit of Work keeps the domain objects in the according lists (`cleanObjects`, `dirtyObjects` and `deletedObjects`) in order to be able to commit or revert changes, and caching objects as well. When the Unit of Work commits, it opens a database transaction, manages concurrency control using isolation levels and finally sends the *changeset* to the database. This *changeset* is often sent as a batch update, consisting of multiple SQL commands (updates or deletes), in a single call to the database, thus avoiding network overhead.

Preventing referential integrity conflicts in the database, is also a concern of the Unit of Work. It either writes the SQL updates in the correct order or has the database check referential integrity on commit only.

Typically, there is one Unit of Work for all the domain objects in the schema. For that, an abstract domain object or interface is needed to provide access to its object id. An abstract domain object can be implemented using Layer Supertype (see Data Mapper pattern in section 3.2.3 for a similar example). SQL and mapping definitions are often delegated to the Data Mapper and Metadata Mapping containers. Although simpler applications can build the SQL updates in the Unit of Work as well.

It is possible to have a single Unit of Work instance per application, per client or per request, although the third is the most common and encouraged in client-server applications. The first requires too much concurrency control code, thus being an overly complicated approach. Additionally there is the *unit of work per operation* often called an anti-pattern, as it causes too much overhead. See [KBA+12] for details.

There are three different strategies to implement a Unit of Work: **caller registration**, **object registration** and **unit of work controller**.

Caller registration has the client to manually register the object in the Unit of Work using the API in Figure 3.17. Figure 3.18 demonstrates the behaviour of this strategy when the client calls the method `RegisterDirty` in the Unit of Work, which may raise confusion and more concern for the developer. Nevertheless, this approach provides the advantage of leaving the domain objects free from any behaviour.

Furthermore in Figure 3.18, when the domain object is loaded, from the Data Mapper, it is registered in the `cleanObjects` list, but a clone of that object is returned to the client. Only when the client calls the `RegisterDirty` does the Unit of Work register this clone in the `dirtyObjects` list. If the property changes are passed in the method `RegisterDirty`, as it happens in Figure 3.19, the Unit of Work can start keeping track of changes in an additional list leaving less work for the committing process. Otherwise, on committing, it iterates the `dirtyObjects` list and for each dirty object, it grabs the corresponding clean object (in `cleanObjects`) and verifies, field by field, which has changed (like in the **unit of work controller** strategy). Note that it is more efficient to manage Identity Maps than lists, due to the frequent lookups performed.

On the other hand, the **object registration** has the object responsible for registering

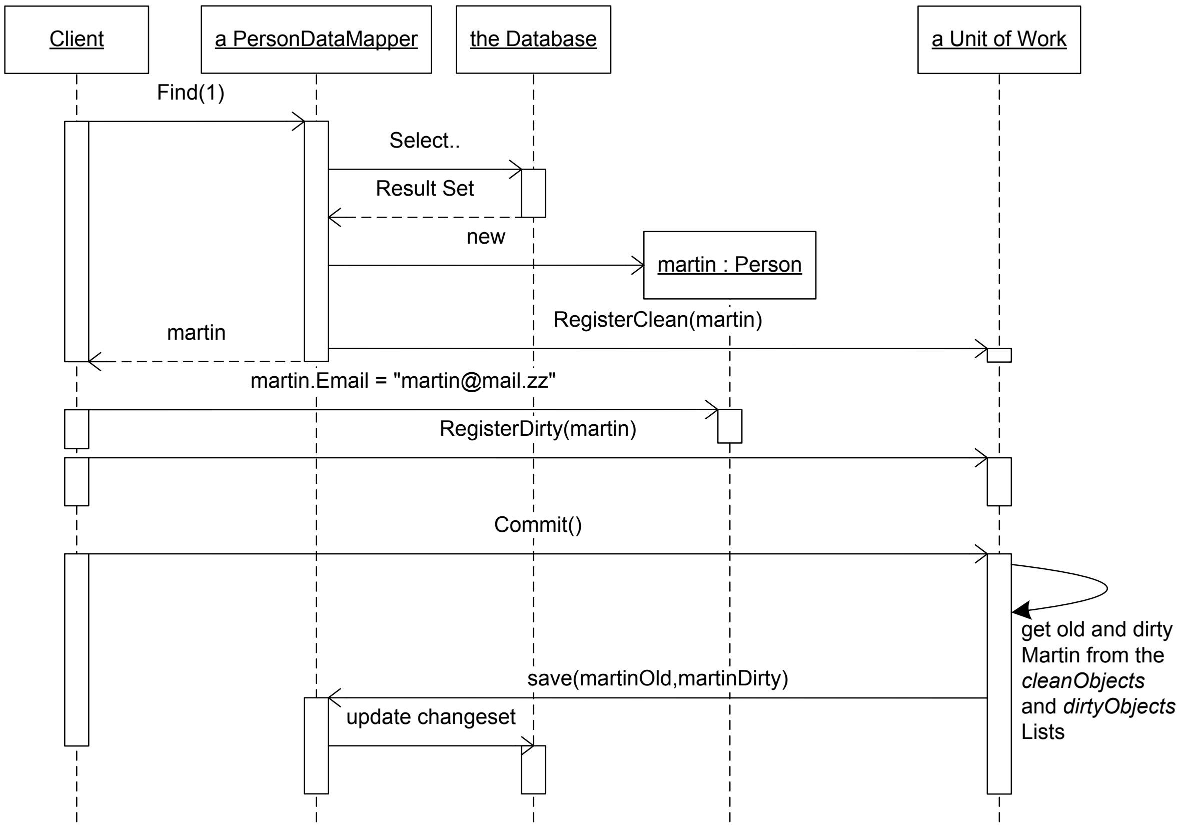


Figure 3.18: Unit of Work caller registration example

itself in the Unit of Work when it changes. Figure 3.19 sets an example of this behaviour. Here, the reference of the Unit of Work has to be passed on to the Data Mapper and to the domain object (`martin`) itself. The Unit of Work can have a Factory Method to create Data Mappers for that effect. Once the method `find` is called by the client, the Data Mapper loads the data and factories a domain object with a Unit of Work reference. Then, the object has its constructor call `RegisterClean`.

Typically the Unit of Work registers the whole object in the Identity Map as dirty when any property changes. Although, if the domain object has a lot of properties, it is convenient that each setter registers the property dirty as well. This can be done either using flags in the domain object for every property or implementing the setters to send the property name to the Unit of Work so it can keep track of any property change. Thus, the committing process is faster as opposed to the **unit of work controller**.

If both the setters and constructor, in the domain object, have Unit of Work behaviour, the Data Mapper will have to fill the objects using the constructor rather than the setters to avoid registering the object dirty. Other solutions consist on making the fields in the domain objects accessible (e.g. protected) to the Data Mappers only, or each object can

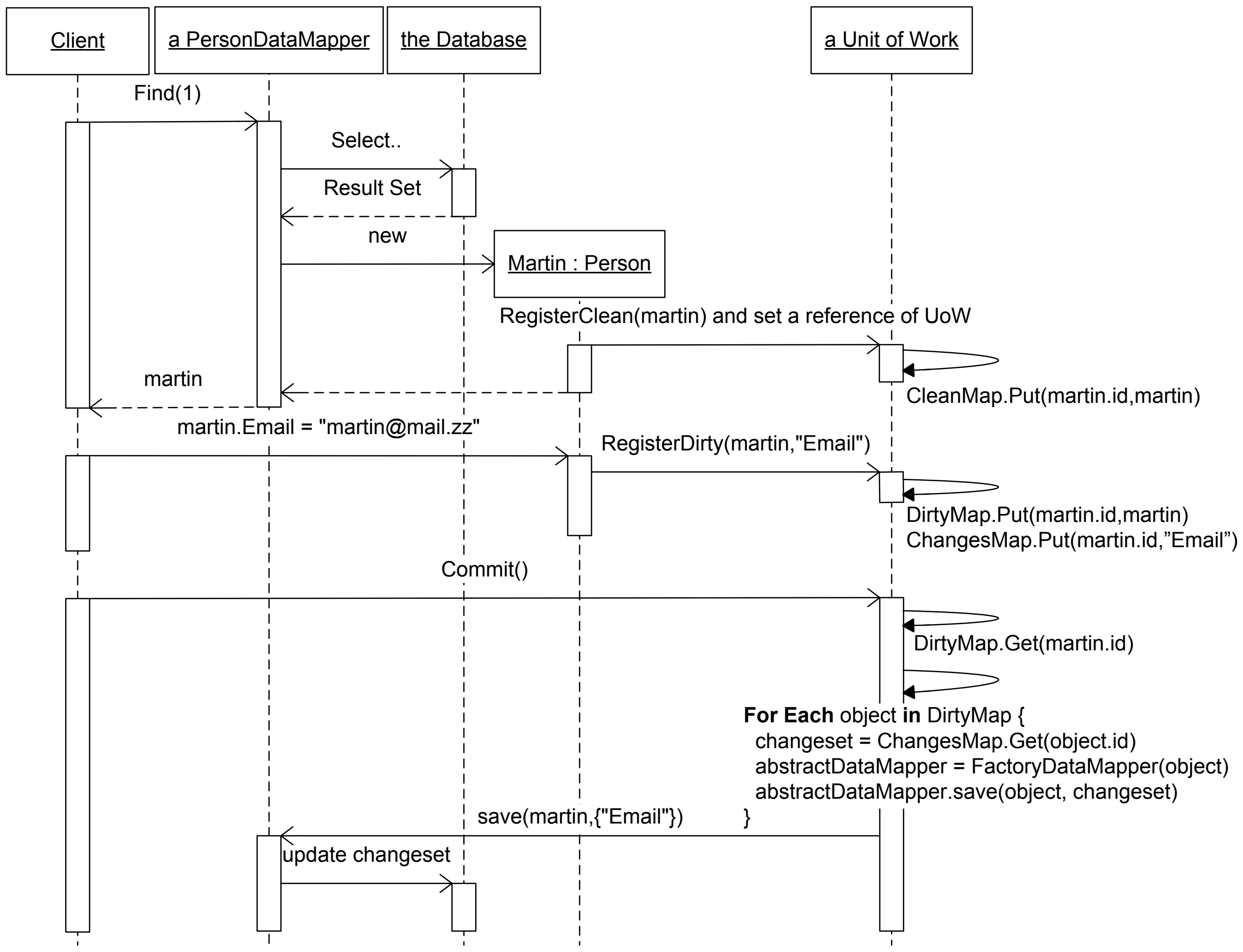

Client
a PersonDataMapper
the Database
a Unit of Work
Find(1)
Select..
Result Set
new
Martin : Person
RegisterClean(martin) and set a reference of UoW
martin
CleanMap.Put(martin.id,martin)
martin.Email = "martin@mail.zz"
RegisterDirty(martin,"Email")
DirtyMap.Put(martin.id,martin)
ChangesMap.Put(martin.id,"Email")
Commit()
DirtyMap.Get(martin.id)
For Each object in DirtyMap {
changeset = ChangesMap.Get(object.id)
abstractDataMapper = FactoryDataMapper(object)
abstractDataMapper.save(object, changeset)
}
save(martin,{"Email"})
update changeset

Figure 3.19: Unit of Work object registration example

have a protected flag to set on/off the Unit of Work context.

The disadvantage of **object registration** is that the domain object is filled with behaviour thus losing persistence ignorance. In turn it becomes easier for the client to make persistence.

In order to achieve better separation of concern and easier serialization, for instance, there can be a `Person` class without behaviour and a `PersonUoW` that inherits from Person, overriding its constructor and setters to register the object in the Unit of Work. Then the data mapper provides a Factory Method to return a `PersonUoW` encapsulated under `Person`.

One alternative to passing the Unit of Work as an argument from one object to the other is associate it to the current thread (Thread Local Storage (see specification in [SSRB00]) (TLS)). This is possible because each business transaction executes within a single thread [Fow02] (using `java.lang.ThreadLocal` in Java or `System.Threading.-ThreadLocal` in .NET).

The **object registration** fits the Aspect Oriented Programming (AOP) which can intercept the domain object setters or properties and inject code to send changes to the Unit of Work. This is useful because it keeps the domain objects persistence ignorant, i.e. unaware of Unit of Work behaviour and therefore PIM. This is a common practice in ORM frameworks like NHibernate (see more in section 4.2.1). Another approach is to implement the Observer pattern with one observer (Unit of Work) and a lot of subjects (domain objects). Any change in the properties of the subjects, notifies the observer.

A typical problem of **object registration** is that the Unit of Work is often not *serializable*. Hence, a domain object that is transferred to a different tier, becomes detached from the Unit of Work. From that point, the object loses its behaviour and becomes persistence ignorant, which makes the Unit of Work unable to track changes on it. Some ORM frameworks such as Hibernate allow the detached object to be merged back to the Unit of Work later, though that forces an additional verification process.

The **unit of work controller** acts as a mediator to the Data Mapper. Figure 3.20 presents the typical behaviour of this approach. Here, the Unit of Work plays the role of a Facade, encapsulating the Data Mapper methods (e.g. `finders`). In the same way as in **object registration**, there is no need to worry the client about registering the objects manually. Also, as everything is managed in the Unit of Work, the domain object does not need to implement behaviour, thus it can be persistence ignorant. When on commit, this strategy takes more time to iterate over the dirty objects and the clean objects in the Identity Maps, while checking the differences between them and then building the *changeset*. This approach is used in TopLink [Fow02].

Another concern of the Unit of Work is to handle concurrency. There are three ways of doing that:

- Pessimistic locking, which implies a lock on the database or entire tables. The multi-user sequential access to the database is likely to cause deadlocks, incurring into a loss of performance (see Pessimistic Offline Lock pattern in [Fow02] for more details);

- Use a level of caching that stores the retrieved objects from the database in a Unit of Work per application scenario. This requires the Unit of Work to implement concurrency control measures for multi-user environments, which is a lot of work. In the end it performs similarly to an OODBMS;

- Optimistic locking, which consists of using database fields like `timestamps` to control the different versions of the objects (see Optimistic Offline Lock in [Fow02] for more details).

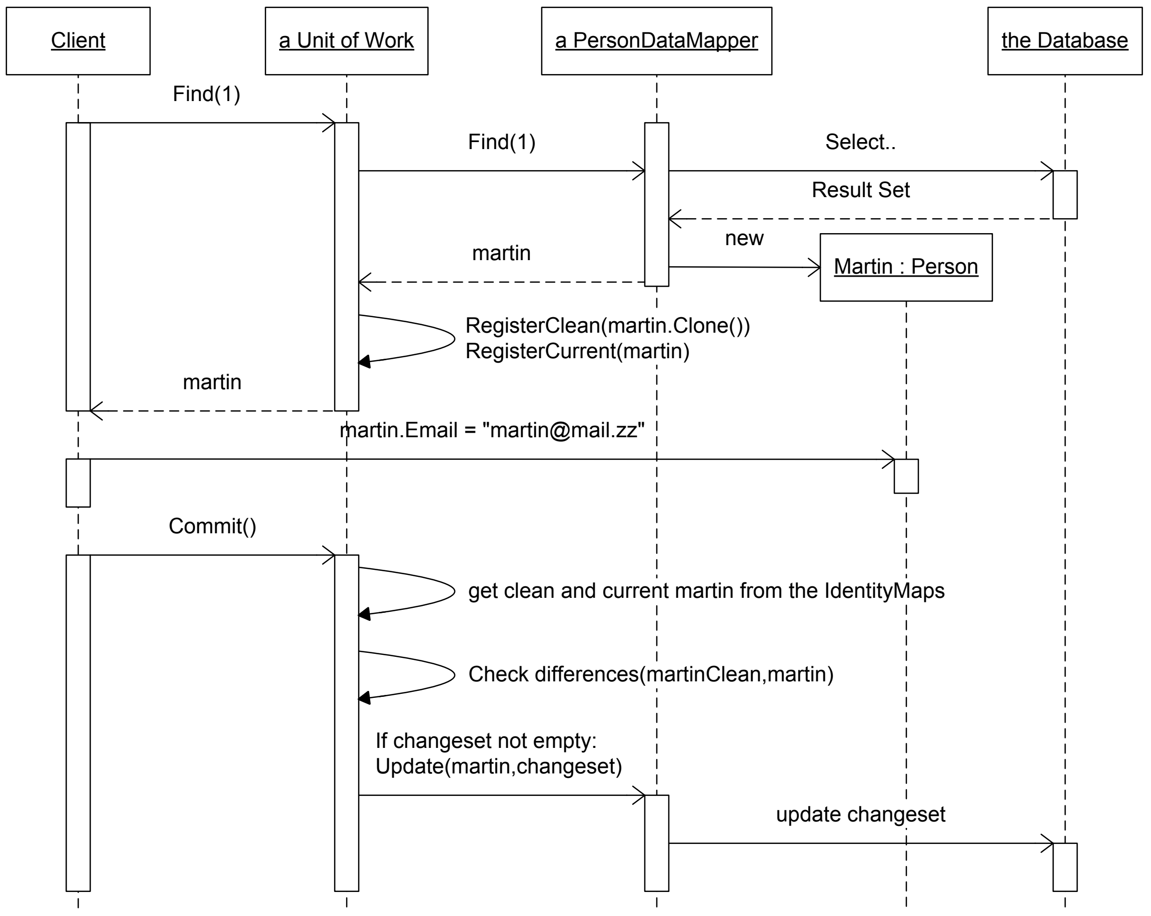


Figure 3.20: Unit of Work controller example

Some popular classes that implement Unit of Work in known ORM frameworks are:

- Session in NHibernate, which implements a **unit of work controller** in the sense it provides the finder and CRUD methods from a Data Mapper. It also performs **object registration** as, via Inversion of Control (IoC), it injects the Session in the domain objects for keeping track of property changes;

- ObjectContext in EF, working in a similar way as the Session;

- DataSet in .NET, implements a disconnected table structure that keeps itself track of changes on every row (using the row states of `modified`, `added`, `removed` and `unchanged`), which slightly differs from the three strategies presented in this section. See section 3.1.2 for details on DataSet implementation.

All in all a **unit of work controller** is an implementation of the data context introduced in section 3.1.3 (referent to Domain Model). This approach is used in EF and

Hibernate to encapsulate Data Mappers, Metadata Mappings, Query Object APIs, Identity Maps, Lazy Load and even Repositories.

3.3.2 Identity Map

An Identity Map caches each object loaded from the database in a map. Future finders check this in-memory map for the requested objects before accessing the database. This prevents bottlenecks to the database, thus enhancing performance.

The structure of an Identity Map consists of a map with key equal to the object id (table primary key) and value equal to the object reference [Fow02]. Every time a Data Mapper returns an object to the client, it ought to verify if that same object exists in the Identity Map already. Typically, any update or delete to that object should synchronize changes to the Identity Map as well.

One common problem of ORM occurs when the same database table row is loaded to two different object instances and conflicts arise when updates of both objects are submitted to the database. The identity map pattern ensures the same object is not fetched twice to the database. It solves cyclic references as well (see section 2.4.6 on the load time trap problem).

Typically, one Identity Map exists for each domain object and for each Data Mapper as well, with some exceptions like [Fow02]:

- In cases of Dependent Mapping, only the composite object requires an Identity Map as, for instance, it normally makes sense to fetch the whole `Car` with all its components rather than a single `Wheel` that belongs to a `Car` and is never fetched alone;

- In cases of inheritance only the table-per-concrete-class strategy may raise doubts. Any lookup to the abstract class will imply checking each Identity Map of every concrete class in the hierarchy. Therefore it can be simpler and faster to have a single Identity Map in the table-per-concrete-class hierarchy. The table-per-class-family, as it involves one table and one key, it only requires one Identity Map. Normally table-per-class has one identity map for each class.

Generally, one Identity Map exists for each domain object and for each Data Mapper as well. Although, it can be generic so the finder methods in different Data Mappers access the same Identity Map. The disadvantage of having a generic Identity Map is that the ids in all the domain objects have to be different but of the same type (database-unique keys). Also, less compiler-time checking makes the application slower and harder to debug.

The best place to have an Identity Map is in Unit of Work for it's responsibility is to keep track of domain objects [Fow02]. Otherwise it can be placed in a Data Mapper.

Most ORM tools provide a level one cache consisting of one Identity Map (generic or a set of maps for all data mappers) per Unit of Work session, which is often used per client request. This avoids multi-user concurrency and therefore the map can avoid locking mechanisms.

Queries that do not specify the requested object id, cannot cache objects in a normal Identity Map. That is because objects are stored in an identity map and accessed via its id (map key). Typically, ORM frameworks offer other levels of caching to store results of frequent queries. Some applications opt for using OODBMSs to build caching layers (shared among clients and providing transactional protection) to work side by side with ORM tools [Fow02].

Finally, one business transaction uses one Unit of Work session to cache loaded objects in an Identity Map. It is simpler to have one Identity Map per Data Mapper to avoid conflicts with ids. As one single generic Unit of Work is required in the Unit of Work session, it is recommended to put the Identity Map or maps inside it. Data Mapper methods implement caching in Identity Maps.

3.3.3 Lazy Load

Lazy Load is a mechanism for loading objects partially, i.e. some fields and associations, in the Domain Model or object graph are fetched from the database only when they are actually required. Without this pattern, fetching an object, in a complex Domain Model that has all its objects interlinked, may induce loading the entire database at once into memory. That is a waste of resources if most of the loaded objects are irrelevant for the operation.

Typically, Lazy Load implementations are transparent to the client. which makes the domain objects lose persistence ignorance, like in Unit of Work. As a result, the Partial-Object problem (see 2.4.6) emerges, i.e. when an object is partially loaded and then transferred (often serialized and sent remotely) out of the data context (here, data context is responsible for dealing with Lazy Load behaviour) to a client that will not be able to load the full object neither navigate through its associations. Thus, once out of the context, the Lazy Load is lost, in the same way as out of a business transaction the objects lose track of their *changeset*. For that reason, the Unit of Work is often a good place to keep track of the Lazy Load behaviour as well.

Figure 3.21 demonstrates the typical behaviour of Lazy Load applied to the one-to-many association `Accounts()`. At first, `martin` is filled with its properties or primitive type fields. Only when the `Accounts` getter is called it will actually load `martin`'s `accounts` from the database.

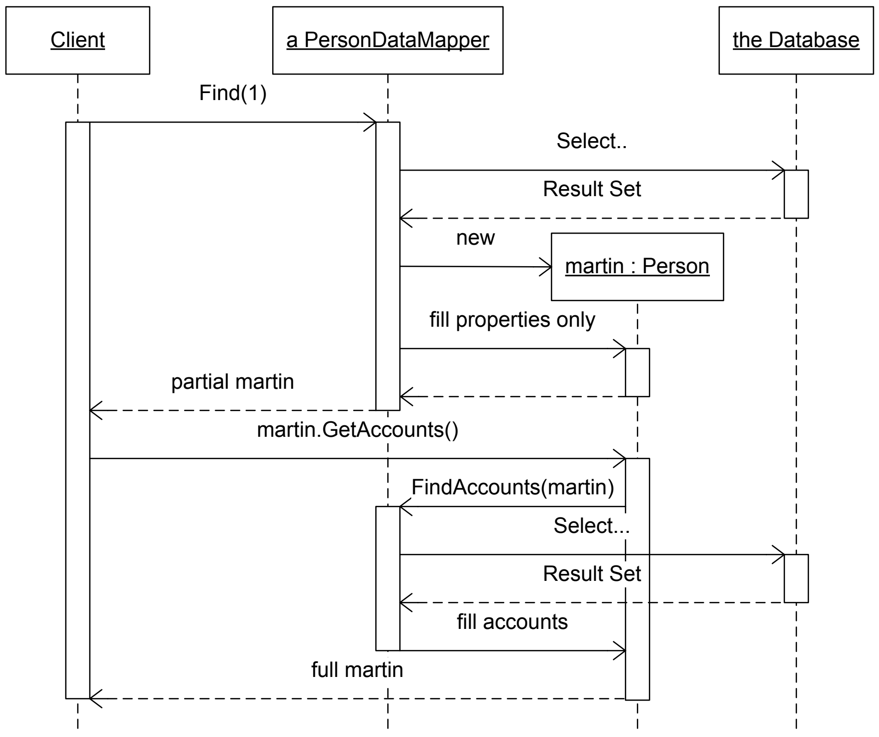


Figure 3.21: Lazy Load example

The Lazy Load strategies presented further in this section are: Explicit Initialize, Lazy Initialization, Virtual Proxy, Value Holder and Ghost.

Loading objects on demand has been a way of optimizing performance in data intensive applications since before the Lazy Load was documented. As a starting point to Lazy Load, the pattern Explicit Initialize, by Beck [Bec97], was early identified to implement on demand object load. It is controlled by the client via the method `initialize`, responsible for filling the partial object. Thus, this pattern incurs in a better code readability, although losing flexibility and transparency in the business logic.

The example of Figure 3.21 suggests that the data context or `PersonDataMapper` is passed in the constructor of `Person` and is used in the `Accounts` getter method, like in a typical implementation of the Lazy Initialization pattern.

Lazy Initialization, by [Bec97], is the simplest way of implementing implicit Lazy Load. It basically consists of defining behaviour for loading data in the properties or getters of the domain object fields, that were not loaded at first. Also, it is convenient to use a special character or code for verifying if the variable is filled or not, other than the `NULL` value. That is, a variable may already have bean loaded and its actual value be `NULL`. Although, Lazy Initialization is simple to implement, as it is based on a dependency between the domain object and the data access behaviour, it is more Active Record oriented. Data Mappers are based upon persistence ignorant domain objects, which is not the case here.

In some cases that require the domain objects to be shared throughout multiple users, Lazy Initialization has also been optimized to behave as thread-safe. This often consists on implementing locking strategies like the *double-check locking*[1] pattern. Only in rare scenarios that require a Unit of Work per application approach, is the thread-safe Lazy Initialization a relevant matter of debate, and therefore is not further discussed here.

Virtual Proxy, by [GHJV95], is the most popular pattern for Lazy Load which is basically a Proxy implementation for loading expensive objects on demand. Thus, it provides an indirection layer between the domain objects and the data access logic, which responds well to the Data Mapper approach, unlike the Lazy Initialization. A Virtual Proxy is an object that looks like the real object although it does not contain any data when initialized. Only during a getter invocation does the Proxy initialize the real object and then calls the respective getter property.

Figure 3.22 presents an example of this implementation using a lazy property. Thus, the `ProxyPerson` is hidden from the client behind the persistence ignorant `Person` domain object. The Data Mapper implements a Factory Method to create a `ProxyPerson` to carry out the data context and become encapsulated by `Person`. The getter property, in `ProxyPerson`, overrides the same getter in `Person` and has the same implementation that is expected in Lazy Initialization.

However, using Virtual Proxy poses a number of weaknesses. In some cases it involves using a Proxy for each domain class, which accentuates identity and inheritance problems. For instance, there can be more than one Proxy encapsulating the same domain object and all of the Proxies have their own identity. Thus, it is convenient to implement and use the proper equality methods. As for the inheritance problems, for instance lets assume the `Worker` class inherits from `Person` and the hierarchy is mapped as a table-per-class-family. When a `Person` is loaded with lazy properties, a `ProxyPerson` is instantiated and delivered to the client with no data on it and without any trip to the database (the discriminator column in the database table was not read at this point). From here, the

[1] *Double-check locking* pattern (see specification in [SSRB00]) consists on verifying if the field was not loaded, begin locked code section, verify again if the field was not loaded and then load it. Although, in theory *double-check locking* works, due to specificities of Java Virtual Machine (JVM), it may not be enough to prevent concurrency problems in practice.

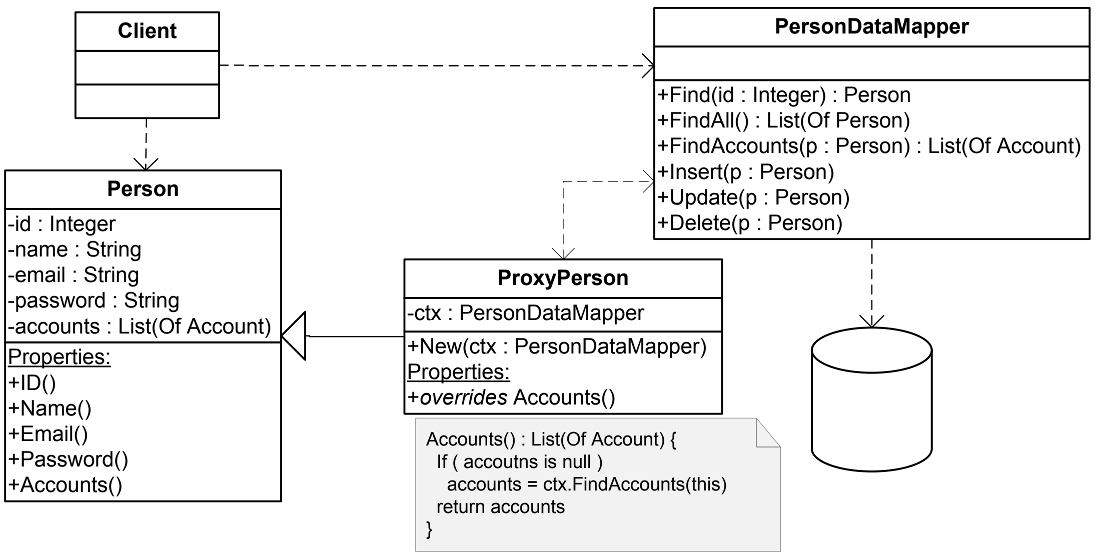


Figure 3.22: Lazy Load Virtual Proxy example

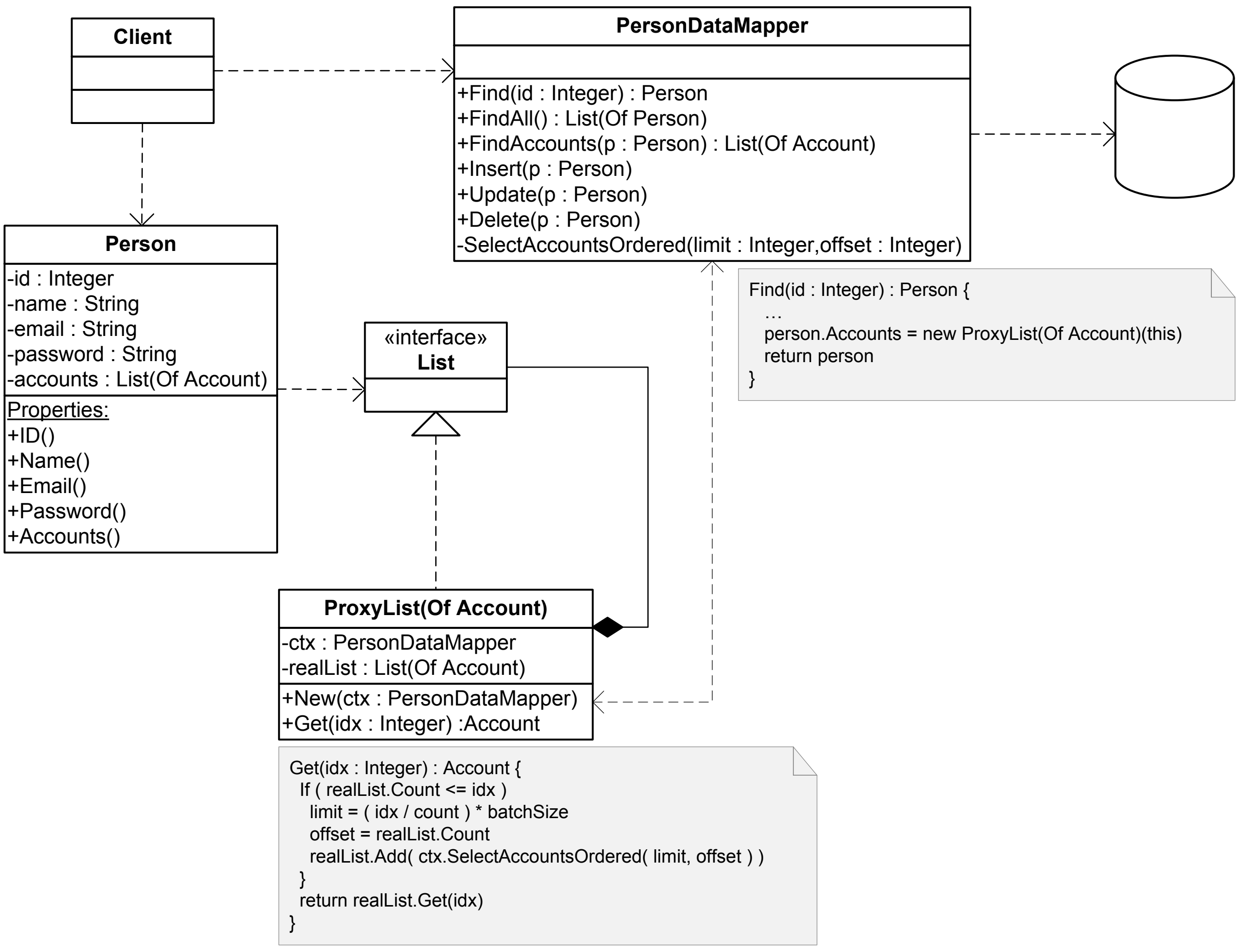
client will not be able to cast his `Person` to `Worker`. These problems break at some point, the object paradigm flexibility and therefore Virtual Proxies have to be used with caution.

Nevertheless, for one-to-many associations it is convenient to use a lazy collection instead of a proxy domain object. That consists of having, for instance, a Proxy lazy list that implements all the methods a list does plus the Lazy Load. This avoids identity and inheritance problems without having to write a Proxy class for each domain object in the list.

Figure 3.23 presents the upper example but with a lazy collection implementation via Virtual Proxy. Here, a `ProxyList` implements the Decorator pattern for decorating lazy features in a `List`, during runtime without the client knowing. Thus, the `PersonDataMapper` factories a persistence ignorant `Person` but containing an encapsulated persistence aware lazy `List` of accounts.

The lazy behaviour to be expected by a lazy collection much depends on which conceptual data structure it is implementing. For instance if the client wants to add an `Account` to a `Person`'s accounts, it is not clear if it is to be expected of a `ProxyList` to load all its accounts before adding a new one. That would be the case if the `ProxyList` managed an `ordered list` or a `set` of distinct objects. Although, conceptually, with a `map` or a `bag`, the client would execute `accounts.add(someAccount)` without having to fetch the other accounts from the database. Such nuances, when neglected, can be enough for causing unnecessary bottlenecks.

Value Holder poses an alternative to Virtual Proxy pattern for implementing lazy


Client
PersonDataMapper
+Find(id : Integer) : Person
+FindAll() : List(Of Person)
+FindAccounts(p : Person) : List(Of Account)
+Insert(p : Person)
+Update(p : Person)
+Delete(p : Person)
-SelectAccountsOrdered(limit : Integer,offset : Integer)
Person
-id : Integer
-name : String
-email : String
-password : String
-accounts : List(Of Account)
Properties:
+ID()
+Name()
+Email()
+Password()
+Accounts()
«interface»
List
ProxyList(Of Account)
-ctx : PersonDataMapper
-realList : List(Of Account)
+New(ctx : PersonDataMapper)
+Get(idx : Integer) :Account

```
Find(id : Integer) : Person {
   …
   person.Accounts = new ProxyList(Of Account)(this)
   return person
}
```

```
Get(idx : Integer) : Account {
  If ( realList.Count <= idx )
    limit = ( idx / count ) * batchSize
    offset = realList.Count
    realList.Add( ctx.SelectAccountsOrdered( limit, offset ) )
  }
  return realList.Get(idx)
}
```

Figure 3.23: Lazy Load lazy collections example (Virtual Proxy)

properties. A Value Holder is an object that encapsulates the real object. Usually, the real object is made accessible through methods like `GetValue`. Only when this method is called, is the real object effectively loaded. It is also common for a Value Holder to manage a value loader, which implements behaviour for loading data of a specific domain object. Value loader is a typical implementation of the Strategy pattern. Thus, the Value Holder can be generic and be used to encapsulate all the domain objects, although a value loader is still needed for each, as well as the according Factory Method in the Data Mapper. This strategy keeps better persistence ignorance in domain objects, avoids the troubles of inheritance (the real object is only instantiated when it is loaded entirely which avoids the casting issues) but keeps the same identity issues of Virtual Proxy as well as the remote out of the data context problem. A great disadvantage of Value Holder, though, is the loss of explicitness and strong typing [Fow02].

A Ghost, also known as light object, is a real object but with partial data. When returned by the Data Mapper, it normally contains only its id field with the rest of the fields being loaded as the other properties are accessed, recurring to the Lazy Initialization pattern. A Ghost can be kept in an Identity Map, which solves identity problems. Inheritance, on the other hand, has the same issues of the Virtual Proxy. As the ghost object uses Lazy Initialization, it loses persistence ignorance and thus better suits the Active Record approach.

Hence, in ORM, the Lazy Load can be employed in three different ways:

- Lazy collections, managing a persistent aware iterator in one-to-many associations, for avoiding expensive database calls. It is a popular form of Lazy Load, can greatly affect performance and is often implemented in ORM frameworks;
- Lazy properties, normally for primitive type fields, overriding or intercepting the getters, in the domain object. It often uses Virtual Proxy and can be useful in some cases where the object has a lot of columns, which can be a sign of poor Domain Model and ORM implementation or legacy;
- Lazy objects, for one-to-one and many-to-one associations, typically implemented using Virtual Proxy to encapsulate a Ghost object.

The Partial-Object problem is common in all the Lazy Load strategies. When, out of the persistence context, the domain object may be serialized but its underlying behaviour is lost, which causes the access to properties, not loaded, to fail. For that reason, the business layer should avoid sending Partial-Objects to other tiers. A common solution is to use DTOs for that matter.

Another problem of Lazy Load is its misuse, commonly result of a bad prediction from scenarios or use cases, leading to an excessive number of database calls and provoking unnecessary overhead. This problem is documented as Ripple Loading, in [AMC01]. This occurs, for instance, when a collection of partial objects is instantiated and the operation will iterate over all its elements, causing each object in the collection to load its data from the database for each iteration. The same happens with a lazy collection, except it does not instantiate the objects before its data is fetched from the database. Not using Lazy Load, in this scenario, turns out to be faster, with only one database call and less overhead. In some cases, loading the whole collection at once may be too expensive and cause out of memory failures. For that it is convenient to retrieve the collection by chunks. For example, in Figure 3.23, `ProxyList` has a batch size for both preventing Ripple Loading. Out of memory failures can also be avoided if the `ProxyList` is implemented as a linked list that only iterates forward. It specifies a buffer size, normally the same as batch size and has to make sure the past objects are destroyed by the Garbage Collector (GC). Such algorithms are similar to the two-pass algorithms implemented in the database (see the join algorithms in appendix A.4).

In the end, in order to keep persistence ignorant domain objects, the implementation of implicit Lazy Load, requires the use of IoC (also discussed in section 3.3.1 referent to Unit of Work, which provides an abstraction from the Lazy Load behaviour to the domain developers (PIM).

Lazy Load increases the complexity of a Domain Model and should be avoided unless it brings performance enhancements. In a great number of cases, lazy collections is the best and most effective approach to increase performance, avoiding some ORM problems as well.

3.4 Object-Relational Metadata Mapping Patterns

The three patterns described here, support mechanisms that manage the repetitive code in ORM and favour the legibility and flexibility of data querying.

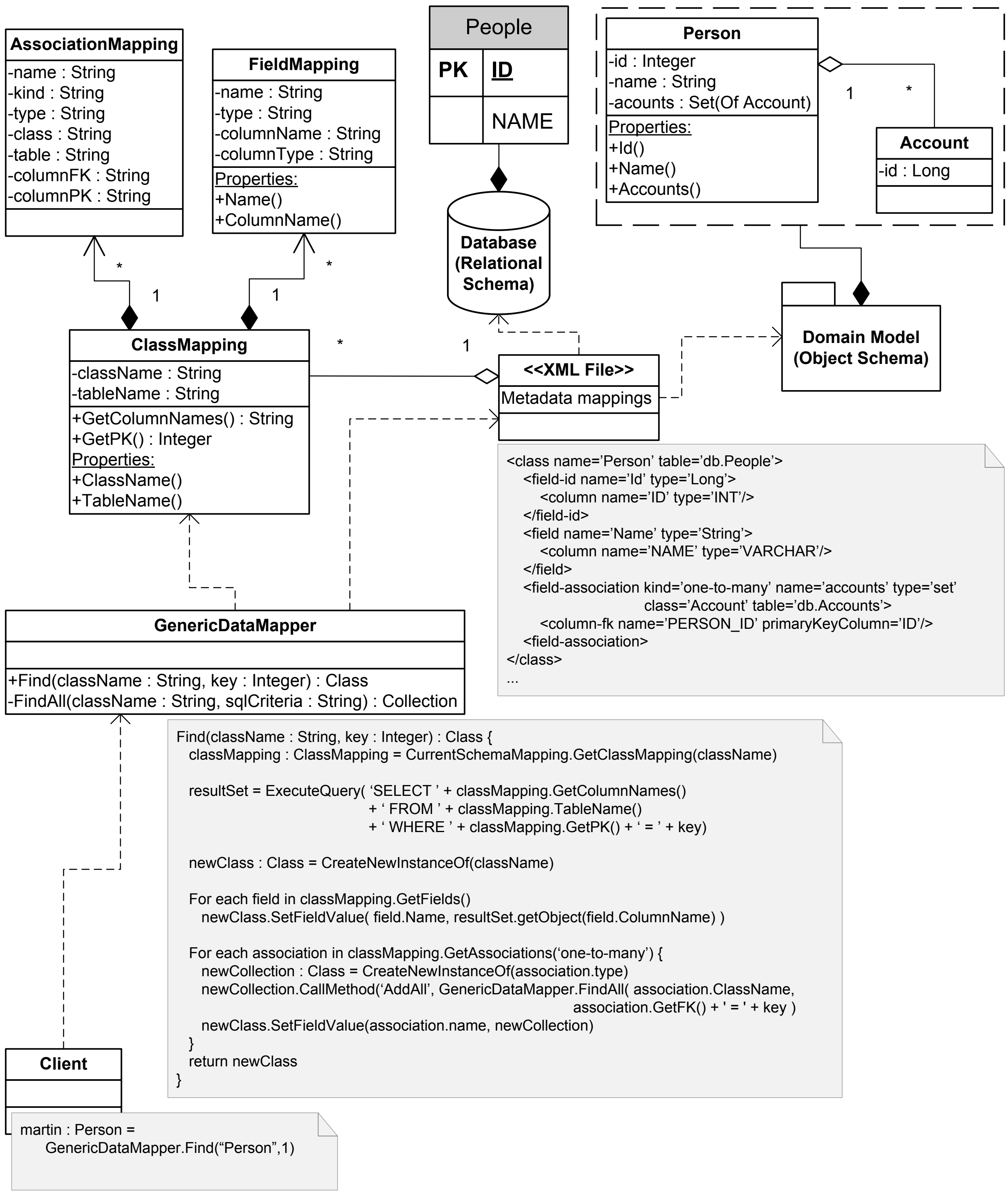
3.4.1 Metadata Mapping

Metadata Mapping is the code that defines the mappings between database tables and in-memory objects. It is often written in a repetitive and dull metadata description language, for instance using XML.

This code can be managed as an input for source code generation of mapping classes, i.e. Data Mappers, domain objects and any other classes that depend on both the mapped relational or the object schema. The generated code provides strong type checking, runs fast and is easy to debug. Although, when the Metadata Mapping code changes, this approach requires source code regeneration, recompiling and redeployment.

Another way to manage this Metadata Mapping code is keeping it in a file (e.g. XML) or even in the database. Then, use reflection to implement a generic Data Mapper, like in Figure 3.24. This approach is more dynamic than code generation, however, reflective code is slower and harder to debug.

With code generation, if the database changes, Metadata Mapping definitions require changing, which results in a regeneration of domain objects and Data Mappers code. If, for example a domain object requires a new field, table changes must be enforced (manually or via some synchronization mechanism), Metadata Mapping definitions have to be rewritten and mapping code has to be regenerated.



```
<class name='Person' table='db.People'>
    <field-id name='Id' type='Long'>
        <column name='ID' type='INT'/>
    </field-id>
    <field name='Name' type='String'>
        <column name='NAME' type='VARCHAR'/>
    </field>
    <field-association kind='one-to-many' name='accounts' type='set'
                            class='Account' table='db.Accounts'>
        <column-fk name='PERSON_ID' primaryKeyColumn='ID'/>
    <field-association>
</class>
...
```

```
Find(className : String, key : Integer) : Class {
   classMapping : ClassMapping = CurrentSchemaMapping.GetClassMapping(className)

   resultSet = ExecuteQuery( 'SELECT ' + classMapping.GetColumnNames()
                                  + ' FROM ' + classMapping.TableName()
                                  + ' WHERE ' + classMapping.GetPK() + ' = ' + key)

   newClass : Class = CreateNewInstanceOf(className)

   For each field in classMapping.GetFields()
      newClass.SetFieldValue( field.Name, resultSet.getObject(field.ColumnName) )

   For each association in classMapping.GetAssociations('one-to-many') {
      newCollection : Class = CreateNewInstanceOf(association.type)
      newCollection.CallMethod('AddAll', GenericDataMapper.FindAll( association.ClassName,
                                                             association.GetFK() + ' = ' + key )
      newClass.SetFieldValue(association.name, newCollection)
   }
   return newClass
}
```

```
martin : Person =
      GenericDataMapper.Find("Person",1)
```

Figure 3.24: Metadata Mapping example using the reflective approach

Using reflection, if the database changes, Metadata Mapping definitions have to be changed and possibly the domain objects as well, which ends in source code recompilation, unless the domain objects are created in runtime with *bytecode* instrumentation. In a domain driven approach, if the object schema changes, Metadata Mapping definitions have to be changed as well, which implies altering the database schema too.

With bottom-up or database centric architectures, code generators are normally the best solution. That is, typically these scenarios do not allow the application to alter the database schema. They rather have the Metadata Mapping generated from the database which results in source code changes and recompilation.

Top-down or DDD methodologies tend to undervalue the database schema, whose changes are often dictated by domain model updates. Thus, and depending on the requirements, it may be convenient to have a more dynamic application. Therefore using a reflective Metadata Mapping can save development time and accelerate the build of new features.

Metadata Mapping is also responsible for synchronizing Data Definition Language (DDL) with Object Definition Language (ODL). For that, it manages SQL `create`, `update` and `alter` commands to reflect domain model changes to the database at runtime. Some ORM frameworks provide tools to synchronize changes at development time and others can be configured to enforce changes to the database when the application starts.

Figure 3.24 presents an example of the reflective approach and an example of an XML file holding mapping definitions. At application startup, this XML file is parsed, validated (e.g. checking if mapped types are compatible or if both the domain model and the database schema are valid to the mapping definitions) and loaded into memory for an object structure like the one given by `ClassMapping`, `AssociationMapping` and `FieldMapping`. Then, the `GenericDataMapper` handles all the SQL, and reflective code to build the objects according to the mapping definitions.

Some strong typing can also be enforced to this reflective approach using generic programming, which is currently supported by both Java and .NET platforms.

Regarding to ORM frameworks, Metadata Mappings is used by both Hibernate and EF via XML files. The former uses Hibernate Mapping (HBM) and the latter Entity Data Model XML (EDMX) XML schemas.

3.4.2 Query Object

The goal of Query Object is to encapsulate SQL in a language that provides querying features on top of the object graph or Domain Model. Thus, Query Object abstracts from the SQL dialects and database schema specificities, which is often used to implement some degree of database interoperability if it is to support more than one Relational Database Management Systems (RDBMSs). Thus, changes on the relational model and mappings may not provoke changes on the domain model. If it does change, however, the object based queries will likely require modifications as well.

It is common to define query able interfaces as Repositories or Data Mapper methods. For example, have a lazy like collection structure implement a group of methods to build a query expression in an object oriented API. Only when the collection is iterated the actual query expression is converted to SQL and sent to the database, thus retrieving the actual results. This is known as Query-By-API (QBA) which provides strong type checking, is easier to debug but lacks flexibility for complex querying, especially when it involves join conditions. An example of QBA is the .NET Language Integrated Query (LINQ).

One of the first implementations in .NET, of Query Object QBA approach to SQL was the now obsolete Linq-to-SQL, followed by EF which supports such features as well. For that, it implements the LINQ interfaces using some reflection and generic programming to infer specific rules as parsing expressions and converting them into SQL.

A more flexible way of querying objects is using Query-By-Language (QBL), which is very similar to SQL except it supports the object paradigm rather than the relational. OQL standard, by Object Data Management Group (ODMG), was initially intended for OODBMSs, although works for ORM implementations as well, except some features might be more complicated to implement due to the object-relational impedance mismatch. Once again, LINQ implements QBL as well, for it adds native querying to .NET languages. However, it lacks much flexibility of other QBL implementations.

The concepts of QBL and QBA have been already introduced in section 2.4.5 to discuss the problem of data retrieval in ORM. Some practical applications of Query Object QBA and QBL, are provided by Hibernate and EF. See sections 4.1 and 4.2 for details.

For implementing query expressions using an object oriented API (QBA), the Interpreter pattern and .NET LINQ interfaces can be a good starting point (see section 5.3.6 for an example of an SQL Interpreter). Also, see in section 3.4.3 a simple application of QBA to provide a query able Repository.

However, for more complex querying it is convenient to use the QBL approach. Implementing a QBL normally uses parser generators for building the OQL grammar and infer the rules for translating it into SQL. One example of such tool is ANother Tool for Language Recognition (ANTLR) which supports a large set of languages including Java and `C#`. In such cases, because compiling a lot of language expressions in runtime can sacrifice performance, it is important to make a few optimizations. For instance, Hibernate pre-compiles, at application startup, Hibernate Query Language (HQL) queries that are specified in Metadata Mapping files, i.e. it translates them into SQL often with parameters. When the OQL query is called and the parameters are set, Hibernate only maps the parameters to SQL and adds them in the SQL pre-compiled statement. Hibernate can also be configured to cache the most used queries.

All in all, implementing a Query Object is essential for complex Domain Models, for they will likely require complex querying. This tends to work well when querying objects is expected to return objects. However, some cases may require unpredictable field projection or joins among entities that do not have any association in the Domain Model. Even though QBL implementations may solve this and return dynamic structures or anonymous types, having a dynamic data model is often a bad requirement for using Domain Model. For that, SQL and database providers or DataSets do the job more efficiently.

3.4.3 Repository

Repository is a mediator between the domain logic and the Data Mappers for it raises the abstraction to the data access by providing a collection like API. Thus, this strategy offers a more object oriented approach to the data source access.

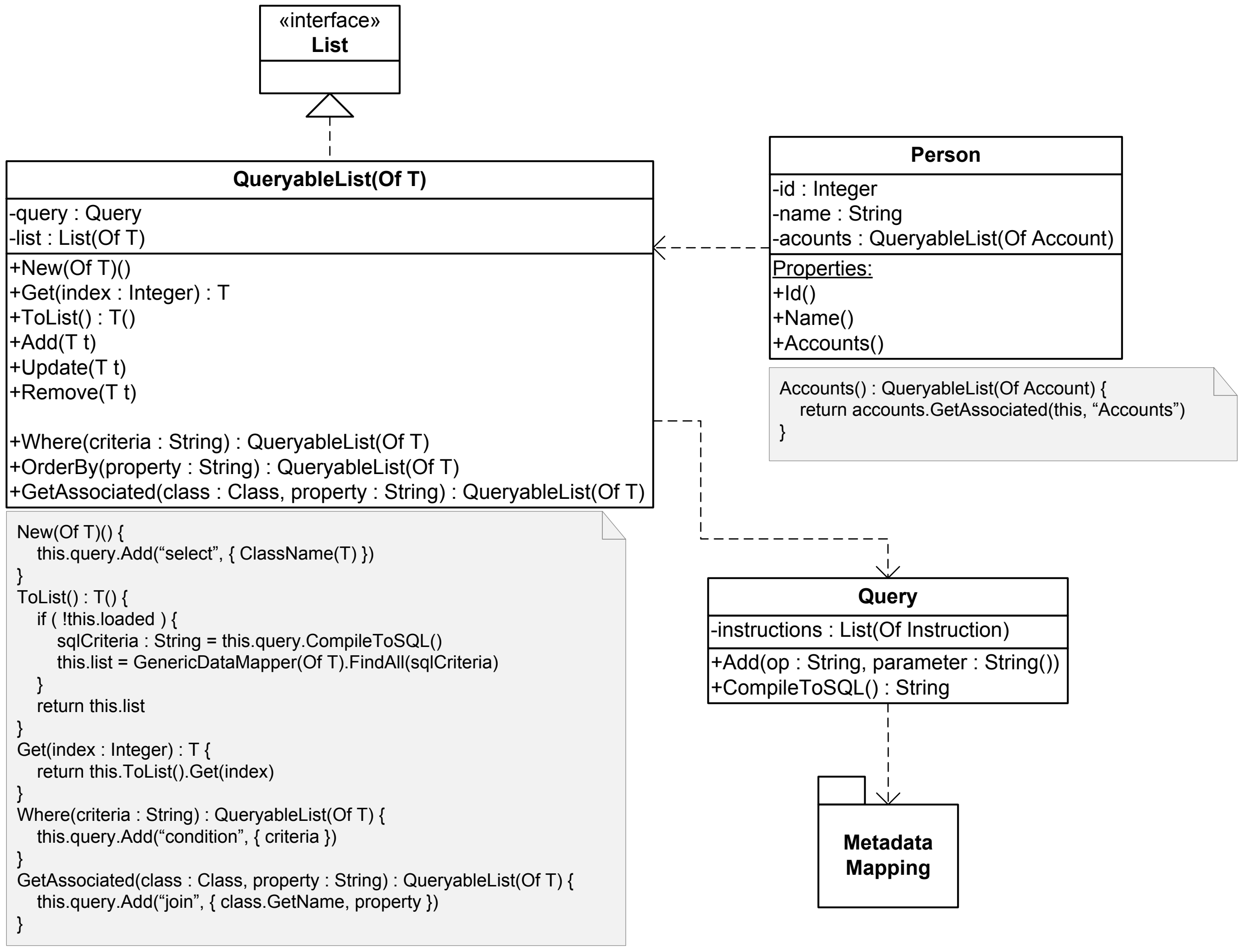
Complex and extensive Domain Models with intensive querying can find it useful having an abstraction layer that encapsulates all these queries, avoiding code duplication as well.

A Repository manages a collection of objects as if they were all in memory. As objects are added or removed from the collection, internally these operations follow to the Data Mappers and further to the database.

A Repository makes the ORM framework look like an OODBMS with a simple API, hiding any persistence details. Commonly, for simple querying, a Repository also implements an Query Object QBA. Thus, the collection can be iterated according to a given criteria, which is first translated to SQL and then fetches the filtered results from the database.

Figure 3.25 presents the behaviour of Lazy Initialization using a Repository and Query Object criteria for iterating over a `Person`'s `Profiles`.

The `QueryableList` acts as a Repository implementing a simple Query Object QBA by build a `Query`. Only when this `QueryableList` is iterated, i.e. the methods `Get` or `ToList` in Figure 3.25 are accessed, does the `Query` translate its instructions into an SQL criteria using Metadata Mapping definitions. Then the Data Mapper takes this SQL, sends it to the database and fills the objects with the according data using Metadata Mappings definition as well.


«interface»
List
QueryableList(Of T)
-query : Query
-list : List(Of T)
+New(Of T)()
+Get(index : Integer) : T
+ToList() : T()
+Add(T t)
+Update(T t)
+Remove(T t)
+Where(criteria : String) : QueryableList(Of T)
+OrderBy(property : String) : QueryableList(Of T)
+GetAssociated(class : Class, property : String) : QueryableList(Of T)
Person
-id : Integer
-name : String
-acounts : QueryableList(Of Account)
Properties:
+Id()
+Name()
+Accounts()
Query
-instructions : List(Of Instruction)
+Add(op : String, parameter : String())
+CompileToSQL() : String
Metadata Mapping

```
Accounts() : QueryableList(Of Account) {
   return accounts.GetAssociated(this, “Accounts”)
}
```

```
New(Of T)() {
   this.query.Add(“select”, { ClassName(T) })
}
ToList() : T() {
   if ( !this.loaded ) {
      sqlCriteria : String = this.query.CompileToSQL()
      this.list = GenericDataMapper(Of T).FindAll(sqlCriteria)
   }
   return this.list
}
Get(index : Integer) : T {
   return this.ToList().Get(index)
}
Where(criteria : String) : QueryableList(Of T) {
   this.query.Add(“condition”, { criteria })
}
GetAssociated(class : Class, property : String) : QueryableList(Of T) {
   this.query.Add(“join”, { class.GetName, property })
}
```

Figure 3.25: Query able Repository example using with Lazy Initialization

The `Accounts` property in `Person` class returns a `QueryableList` which is a lazy Repository. Also it provides specific queries like getting the associated `Profiles` to that `Person` via the property in question. For that, accessing Metadata Mapping definitions is required.

Note that, from the example of Metadata Mapping with reflection in section 3.4.1, the same approach is used here but with generic programming. Other less generic approaches can be used as well in Repository pattern.

3.5 Conclusions

This chapter demonstrated the most important design patterns in PoEAA [Fow02], relevant to the ORM. Section 3.1 presented three different strategies for modelling domain logic, with Transaction Script being the most simple, Table Module (containing an analysis on DataSet API) the one in the middle and Domain Model the most complex. Section 3.2 provided three architectural patterns used to support different domain logic approaches from the previous section, with focus on Data Mapper. Section 3.3 described three behavioural patterns (Unit of Work, Identity Map and Lazy Load) very common on ORM frameworks for managing transactions, caching, performing efficient loading and updating of objects. Three metadata mapping patterns were explained in section 3.4, very common on ORM frameworks as well, for managing static mapping definitions and complex querying of object model efficiently.

The patterns presented here set a basis for ORM and further proved essential for analysing ORM frameworks in chapter 4 and finding the best architectural approaches for ClassBuilder enhancements, including CazDataProvider implementation, in chapter 5.

Chapter 4

Object Relational Mapping Frameworks

Based on the ORM theory described in chapter 3, it was not obvious for Cachapuz which domain logic approach was best for their development environment. With the emergence of ORM frameworks for .NET and the possibility of discarding the custom built Class-Builder ORM tool, doubts arose on deciding which path to take.

Therefore, Cachapuz requested an analysis of the most relevant ORM frameworks available in .NET. For that, a set of specific requirements were defined to be fulfilled by such tools:

- Small learning curve;
- Simplicity of configuration;
- Efficient loading and persistence of data;
- Good integration with Visual Studio 2008 or 2010;
- Customizable code generation for logging, auditing and optimistic locking. The ORM framework has to integrate with a legacy database in which the tables contain the following columns for logging and concurrency control: `DataEdicao`, `DataCriacao`, `UserEdicao` and `UserCriacao`. Additionally, to generate logs, it must be possible to edit events such as *oninsert*, *onupdate* and *ondelete*;
- Able to deal well with legacy databases that use composite primary keys and no foreign key constraints;
- Able to work with different databases;

- Support for dynamic querying with projection features, allowing a loose coupling to the relational schema, so that written queries can endure certain changes made to the tables and columns of the database. For that, the ORM framework must provide a query language similar to SQL, avoid Metadata Mapping and the Dual-Schema problem by returning a dynamic structure similar to a DataSet to the presentation layer;

- Support for web and N-Tier architectures with efficient and portable serialization of domain objects between the client and the server. For that, domain objects have to be provided as persistence ignorant Plain Old CLR Objects (POCOs).

The ORM frameworks analysed in this chapter are EF and NHibernate, which are currently the most popular in .NET as in the enterprise software field.

Hence, this chapter consolidates the ORM theory (from chapter 3) by identifying some common patterns in the frameworks presented here. Also it confronts each ORM framework with the needs of Cachapuz, which in turn leads to a more clear decision whether to adopt a new ORM framework (with some degree of paradigm change) or keep enhancing the ClassBuilder tool to generate DAL code for their applications.

4.1 Entity Framework

EF is an ORM tool developed by Microsoft first released in Visual Studio 2008 with .NET 3.5, and came to expand the Linq-to-SQL features.

Due to early system specific requirements of Cachapuz, the analysed ORM tools were constrained to Visual Studio 2008 and .NET 3.5. Therefore, most of the examples in this section are of EF 1.0. The following version jumps directly to EF 4.0 and some of its new features are analysed in this section as well, because eventually Cachapuz begun migrating to .NET 4.0.

Essentially, EF supports the following features:

- Domain Model pattern;

- Database-first development with code generation and one-way schema updates;

- Model first development supporting two-way schema updates (from EF 4.0);

- Persistence ignorance and POCOs (from EF 4.0);

- Unit of Work, Identity Map and optimistic locking (control fields);

- Lazy Load (from EF 4.0);
- Reflective Metadata Mapping with generic Data Mappers;
- The Query Object and Repository patterns with Entity Query Language (EQL) and Linq-to-Entities;
- Composite keys;
- One-to-one, one-to-many and many-to-many associations;
- Table-per-class-family, table-per-class and table-per-concrete-class hierarchy strategies;
- Database interoperability via dotConnect data providers. Thus, ADO.NET specifies a data provider API to be implemented by third parties so a specific database can be accessed using EF;
- Visual Studio integration with GUI widgets and entity designer;
- Text Template Transformation Toolkit (T4) templating and event handling for code customization and audit logging;
- N-Tier support with WCF data services (from EF 4.0), thus remotely exposing a Linq-to-Entities *queryable* API which makes the query itself *serializable*.

The reminder of this section explores some of the more relevant features of EF 1.0 according to the Cachapuz requirements presented at the beginning of this chapter.

4.1.1 Unit of Work

In EF, the ObjectContext class is the implementation of the Unit of Work pattern, and constitutes the central object for accessing and persisting domain objects. It not only supports transaction management, Identity Map and optimistic locking but also provides Data Mapper and Repository APIs.

In ObjectContext, the method `SaveChanges` is most important for it commits the current business transaction, i.e. once it is called, all the changes made to the in-memory domain objects within a single transaction scope, are persisted to the database. In this last step, the ObjectContext accesses its `ObjectStateManager` (L1 cache or Identity Map) while checking for any `ObjectStateEntry` (encapsulates the value, key and state of an entity or domain object) whose `State` is not `Unchanged`. After checking changed objects in the Identity Map, it will generate the SQL batch inserts, updates and deletes [Ler09]. In the end, only the changed domain objects and its changed properties are sent to the database.

Essentially, the `ObjectStateEntry` is a Layer Supertype for all the domain objects, except it uses aggregation and interface implementation rather than inheritance as suggested by Fowler in [Fow02]. That is, the `ObjectStateEntry` is like an adaptor (Adapter pattern) which makes the domain object POCOs implement the `IEntityChangeTracker` interface so that different domain objects can be used in a single generic Identity Map.

Thus, according to the Unit of Work strategies presented by Fowler, the ObjectContext implements the **object registration** approach for keeping track of changes. It also uses the **unit of work controller** strategy in the sense that the ObjectContext itself provides the Data Mapper and Repository APIs.

As for **object registration**, the generated code in EF 1.0 avoids Proxies and code injection by implementing persistence aware entities (rather than POCOs). Thus, the setter properties of a given entity use events and *onchange* methods to keep track of property changes. These *onchange* methods are `partial` and `private`, which means the generated code can be extended as long as it defines another segment of the same generated entity class. It cannot be extended with inheritance though.

On the other hand, EF 4.0 uses AOP to avoid persistence aware entities, better supporting N-Tier architectures and data serialization. Hence, the EF uses dynamic Proxies to extend functionality of the POCO entities, at runtime, which enables both Lazy Load by Virtual Proxy (not available in EF 1.0) and dynamic change tracking (Unit of Work **object registration**). If dynamic change tracking is disabled, the ObjectContext must take a snapshot of the loaded entities, and once `SaveChanges` is called, it will check for variations between the dirty entities and the clean entities (in the snapshot) and then produce an efficient *changeset*. Fowler calls this the **unit of work controller** and can be significantly less efficient than **object registration**. See section 3.3.1 for more details on the Unit of Work pattern definition and section 4.1.4 to better understand how POCO entities work in EF.

Additionally, the EF ObjectContext acts as a Repository of entity collections, providing methods for adding and deleting objects from those collections. Also, these collections implement the `IQueryable` interface, which means Linq-to-Entities can be written on top of them (see the Query Object pattern in section 3.4.2 for more details).

Unlike relational databases, the ObjectContext cannot *rollback* to a previous state, thus keeping all the changes made to the entities during that transaction. That is due to the fact that the ObjectContext is itself considered to be used as a single transaction.

That follows the Unit of Work per request approach, which is the best approach for most cases (see section 3.3.1 for details). For instance, in web architectures, it is widely encouraged to have a single ObjectContext per request. Nevertheless, special cases may require more complex transactional features in the business layer:

- Using `ObjectConext.Refresh` to reset an entity to its original value (might have to iterate throughout all the entities loaded) or have the ObjectContext disposed and again reload data from database [Ler09];

- Using a `TransactionScope` to wrap and then *rollback* one or multiple transactions (ObjectContexts). This can optimize performance in requests that implement multi-database scenarios, e.g. it may only open the connection to a second database after the connection to the first database is opened and its commands are executed, hence reducing the uptime of the second connection and saving resources. In the end, the `TransactionScope.Complete` method has to be called in order to prevent a *rollback* of the whole `TransactionScope`.

Additionally, the ObjectContext class provides a transaction API for manually committing or making *rollback* to a database transaction similar to ADO.NET API which is further discussed in section 5.2. For details on ef transaction APIs see chapter 16.2 of [Ler09].

4.1.2 Optimistic Locking

Multi-user environments pose new challenges to EF. EF implements Optimistic Offline Lock by default. Again, ObjectContext is designed to exist in a single request and for one client only, also because it is not thread-safe. Hence additional measures have to be implemented to prevent such concurrency problems. The most used method is optimistic locking or concurrency and is supported in EF with the following strategies [Ler09]:

- By setting all the entity properties with attribute `concurrency mode` as `"fixed"` (configurable in Visual Studio designer), except the read-only properties such as object IDs. This assures that any SQL `update` command, calculated in the future with `SaveChanges`, will have all those properties filtered by their old values in the `WHERE` clause;

- Having an additional property or control field like `LastChanged` (`timestamp`) for each entity and then have it set to `"fixed"` in the `concurrency mode` as well. This requires additional code to set updates of this field whenever any entity field changes, which can be implemented by overriding object properties on a Virtual Proxy or using T4 templates to inject code at compile time.

4.1.3 Code Customization

Although frameworks such as EF tend to be generic for all businesses, often enterprise software requires specific code customization. That includes handling certain events, code refactoring or custom code generation.

In EF, because generated classes are set to `partial`, they can be extended with additional functionality in separate files, e.g. creating custom properties or overriding methods for each entity or the ObjectContext itself (which applies to all the attached entities) [Ler09].

Additionally, the generated code in EF declares `partial` methods such as `onContextCreated` for the ObjectContext, `onNameChanged` and `onNameChanging` for each property of each entity. These methods are used as in the Template Method pattern and can be implemented in the partial class [Ler09].

Another way to inject code in the DAL EF generated classes is to handle the following events: `ObjectContext.SavingChanges`, `EntityObject.PropertyChanging`, `EntityObject.PropertyChanged` and `RelatedEnd.AssociationChanged`. Injected code is then implemented in the partial classes in a new method that handles the above events. Note that whilst VB.NET allows event bindings to the methods, in `C#` additional code has to be managed in order to set up the event handlers. For instance, the `SavingChanges` event can be set up in the `onContextCreated` method [Ler09].

For some cases, however, the above solutions may not provide the expected flexibility. For that, there are two options: using T4 templates to change the generated classes before compile time or customize the EF code generation. Because T4 features and enhancements were not very appealing before EF 4.0, T4 templates are not discussed here. Customizing code generation can be implemented by using the `SampleEdmxCodegenerator` provided by EF team to generate the code-behind of EDMX in Visual Studio (see [Ler09] and [dpb08] for details), although this is an overly complex approach.

4.1.4 POCOs

In some cases like in N-Tier architectures, applications often require that the business entities are entirely decoupled from the persistence mechanisms such as the ObjectContext in EF.

Even though a client-server application may use WCF to create data contracts, it is important that, in the server, business logic and entities are as independent from the underlying frameworks with its evolving APIs as possible. That is achieved by using POCO entity classes, which if isolated in a portable library, can be imported by other tiers or different applications without additional framework libraries. That brings a better separation of tiers and a more abstract API.

In EF 1.0 there are the following options to achieve POCO entities:

- Create DTOs, requiring translation between EF entities to DTOs. It helps packaging entities and collections into a single result response to a client request. This prevents network overhead when the presentation tier runs on the client (e.g. Silverlight) and via web services (e.g. WCF) communicates with a server using data contracts. Note that DTOs will not use **Object Services** from EF. **Automapper** is a library that helps automating the translation process between DTOs and EF entities. An alternative for the DTOs to keep track of changes in the client is by implementing a local `EntityState` as presented in [Ler09]. For that, DTOs have to contain some behaviour and must be made isolated and portable so they can be used in both the client and the server tiers, thus hindering the isolation of tiers;

- Use **Persistence Ignorance (POCO) Adapter for Entity Framework V1** (see chapter 23.2 of [Ler09] for details). It generates POCOs via command line, provides little documentation and does not process schema updates. This adapter, generates POCOs as DTOs, manages translation between DTOs and EF entities, supports Unit of Work and Lazy Load features for the POCO entities;

- Create entity classes and have them inherit from `EntityObject`, be decorated with mapping attributes and import additional EF API. It can trigger some conflicts with designer generated classes. Also, this approach does not provide entirely persistence ignorant entities [Ler09];

- Implementing `IPOCO` interfaces which is less compromising than inheriting `EntityObject` but still requires mapping attributes and importing additional EF API. This approach consists of implementing certain interfaces so the change tracking and relationship management capabilities can be used. See chapter 19.2 of [Ler09] for details.

To use POCO entities in EF 4.0 the following options are available:

- Disable code generation by removing the `Custom Tool` in the Visual Studio EDMX properties. Then write the code for each entity as well as the object context for handling the Repositories.

- Use T4 templates for code generation. There are two T4 templates that come with EF 4.0. One for generating entities that inherit from `EntityObject` and another for self-tracking entities implementing a couple interfaces for keeping track of changes in the entity properties. However there is little persistence ignorance with these two templates. For that, a third template has to be installed: the **ADO.NET VB.NET (`C#`) POCO Entity Generator**. Withal, these POCO entities take advantage of Unit of Work and Lazy Load capabilities. At runtime, because of `virtual` properties, the EF factories (via Factory Method) **DynamicProxies** (Virtual Proxy) for encapsulating each POCO entity with its Unit of Work **object registration** and Lazy Load behaviour. Note that this behaviour only works for entities that are attached to the ObjectContext;

4.1.5 Testing EF

This section describes a practical analysis of the EF 1.0 beginning with a basic notion of configuration settings followed by an example of model generation from the database and then querying test, analysis and benchmark.

4.1.5.1 Configuration and model testing

Both SQL Server and MySQL were tested with EF 1.0. Comparatively, the setting up process is similar except MySQL requires the installation of an additional dotConnect driver.

The same database schema was replicated in both the RDBMSs. The generated EDMX files for the two databases contain minimal differences only on the first lines:

```
<!--MySQL-->
<Schema Namespace="cazModel.Store" Alias="Self"
Provider="MySql.Data.MySqlClient" ProviderManifestToken="5.1"
xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStore
SchemaGenerator" xmlns="http://schemas.microsoft.com/ado/2006/04/edm/ssdl">

<!--SQLServer-->
<Schema Namespace="cazModel.Store" Alias="Self"
Provider="System.Data.SqlClient" ProviderManifestToken="2008"
xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStore
```

```
SchemaGenerator" xmlns="http://schemas.microsoft.com/ado/2006/04/edm/ssdl">
```

There is no difference in the code-behind (VB.NET or `C#` file containing the ObjectContext and the entity classes mapped by the EDMX file definitions) that is generated by EF for both databases. Figure 4.1 presents a diagram with the components managed in the configuration of EF 1.0. One `AppConfig` file can contain various `connectionStrings` which can only link to one database provider and each EDMX file generates one ObjectContext with a set of entities.

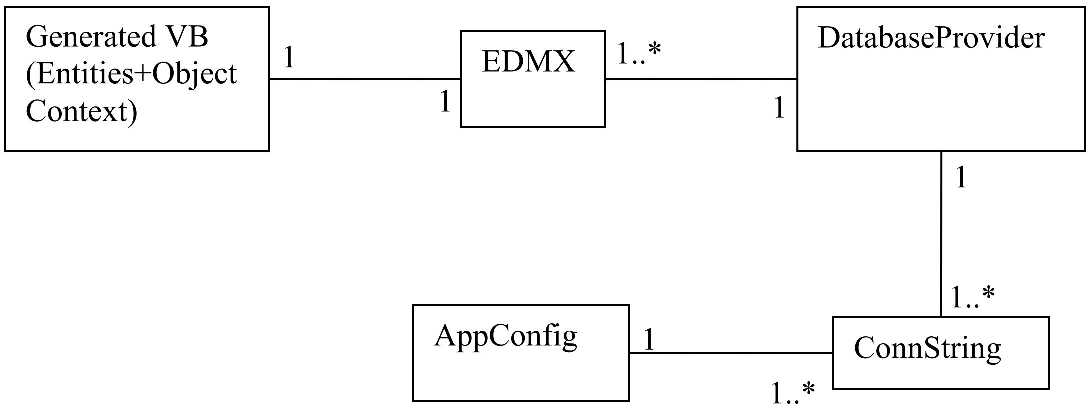


Figure 4.1: Diagram of the various components and files managed in the configuration of EF 1.0

After having an idea of how the EF is configured, a database schema was created in MySQL 5.1 using the **InnoDB** storage engine. Rather than **MyISAM**, **InnoDB** uses foreign key constraints which helps the EF generate one-to-many and many-to-many associations automatically from the database. In order to have a many-to-many association generated by EF, an intermediary table has to be created and contain only the foreign keys (combined as a compound primary key) for both the associated entities (see section 2.4.2 for details).

In EF 1.0, only database-first is supported, i.e. generation and update of the entity model from database schema [Ler09]. On the other hand, EF 4.0 supports model first features, which are not covered in here.

Figure 4.2 presents a test database schema (in MySQL 5.1 **InnoDB**) used to by EF to generate the entity data model in Figure 4.3. As expected, only three classes were generated and with right association mapping.

From that model and testing the Query Object implementation of EF, it was possible to find three different ways of fetching a `User`:

- Using Linq-to-Entities:

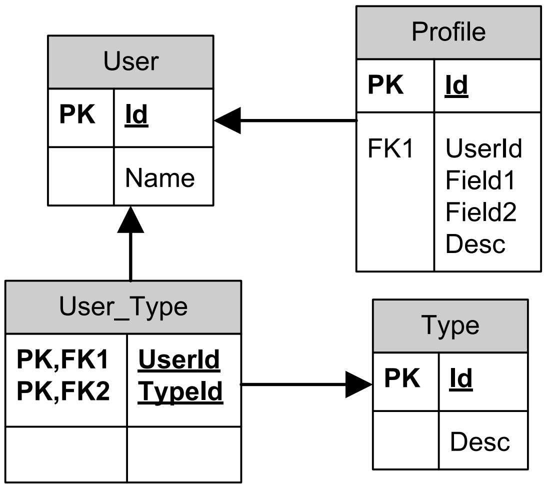


Figure 4.2: MySQL Test database for EF 1.0

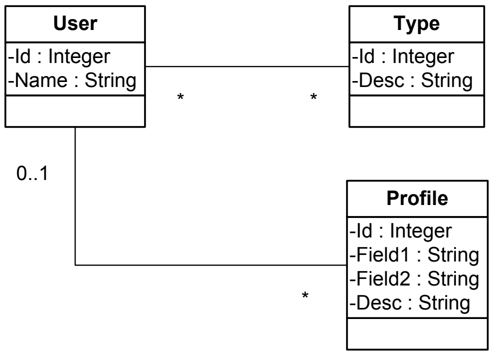


Figure 4.3: Entity Data Model generated in EF 1.0 from MySQL Test database schema

```
Dim u As User = cazEntitiesContext.User.Where(Function(c) c.Id = 1)
```

- Using EF API:

```
Dim u As user = cazEntitiesContext.GetObjectByKey(New
EntityKey("cazEntities.User", "Id", 1))
```

- Using EQL:

```
Dim u As User = cazEntitiesContext.CreateQuery(Of User)("SELECT VALUE
u FROM cazEntities.User AS u WHERE u.Id = 1").First()
```

Linq-to-Entities is the EF implementation of LINQ and the successor of Linq-to-SQL. It can be used as both QBL or QBA. EQL, is a close to SQL language, following a QBL approach. It does not use type checking which can cause trouble debugging, although is more versatile and flexible than Linq-to-Entities (QBA).

Saving objects in EF requires adding the new detached object to a Repository of that type (provided by ObjectContext), which attaches it to the Unit of Work. Updates do not need this. Another important test is to verify if updates involving associated objects work as expected. For that, EF was tested regarding cascading updates on bidirectional associations. For instance, when a new `Profile` is added to an existing `User`, EF implicitly adds the `Profile` reference to the `User` as well.

In EF 1.0, composite keys are identified automatically from the database which already generates the appropriate code for managing them. It is possible to define composite keys manually in the EDMX file.

In order to fetch an object using the composite key as a parameter, EQL or Linq-to-Entities queries treat the various identity fields as normal fields. EF API, on the other hand, requires that a `Dictionary(Of String, Object)` is defined for a building the composite key parameter to be used in `ObjectContext.GetObjectByKey` method.

4.1.5.2 Basic querying

From the designed model in p. 96, some basic query tests were implemented with the goal of analysing their performance. The is section confronts the results of EF API, Linq-to-Entities, EQL and SQL examples. The code of each example is in appendix B.1. For details on how queries are processed in EF see chapter 9.2 in [Ler09].

Each query example is presented with the VB.NET code, followed by the SQL code that EF generates. The SQL was retrieved via MySQL logging. All the examples were submitted to performance tests, running some or all sections of code 10^n times and according to Table 4.1. Note that the comments describing how many times a block of code is run comes under that same code with a different colour.

The performance tests displayed in Table 4.1 were conducted using EF 1.0 and MySQL 5.1 with logging enabled. Also the table `User` only contains one row so that the time variable is less subject to MySQL query planning and execution. The goal of this benchmark is to find the faster and most optimized EF querying mechanism and then compare it with the native SQL examples which access database directly through the database provider API.

It is visible in Table 4.1 that, Linq-to-Entities is slightly slower than EQL (example 1 in p. 203 and 3 in p. 204) and that by default, the EF API applies caching automatically. If the same EQL query is used several times it is convenient to have it created only once (example 2 in p. 203).

As for example 5 (in p. 204), it was verified that the time cost would be the same

Table 4.1: EF query examples benchmark

Examples	**SQL queries executed**	**10^3 operations in seconds (n=3)**	**10^4 operations in seconds (n=4)**	**10^5 operations in seconds (n=5)**
1. EQL 1	10^n	2.39063	19.98438	198.78125
2. EQL 2	10^n	1.15625	8.03125	77.75000
3. Linq-to-Entities 1	10^n	2.40625	19.29688	193.75000
4. EF API 1	1	0.48438	0.57813	1.25000
5. EF API 2	10^n	1.35938	9.32813	89.48438
6. Linq-to-Entities 2	10^n	2.54688	20.90625	204.84375
7. Native-SQL 1	10^n	0.90625	7.60938	73.81250
8. Native-SQL 2	10^n	0.45313	3.04688	29.48438
9. Native-SQL 3	10^n	0.46875	3.06250	28.59375

for reading the same row 10^n times (no caching) or only reading the one row 1 time and produce $10^n - 1$ more `selects` with no result.

Regarding native SQL, prepared statements do not cause time cost changes for this scenario. In fact, both the examples 8 and 9 (in p. 205 and p. 206), in Table 4.1, were tested by adding to the `WHERE` clause, about 300 new conditions (`1=1`) concatenated by `AND`, so that the SQL query held more than 2000 characters. The result was a variation of about 0.5 seconds more cost for the example 8 with 10^4 operations. From the example 7 (in p. 205) to 8, although, it is evident that optimizing the use of connections can greatly enhance performance.

In the end, it is possible to conclude that the EF API is the fastest loading mechanism, especially when providing caching. Although it is the least flexible, therefore not suitable for filtering or complex querying. Also, it is obvious that native SQL outperforms any of the EF examples.

4.1.5.3 Eager and deferred load

Because EF 1.0 does not support the implicit Lazy Load strategies, the querying strategies tested here include eager loading and deferred loading. This analysis uses the 5 examples in appendix B.2 which consist of loading a specific `User` and its corresponding `Profile` rows. The different examples use EQL, Linq-to-Entities or EF API.

Eager loading consists of having the association explicitly asked in the query itself, e.g. executing a query that loads a `User` and its `Profile` rows at once. Deferred loading is the same as the Explicit Initialize pattern, which is a primordial form of Lazy Load. It consists of loading the `User` object, leaving its collection of `Profiles` empty, and only when the business code is explicitly called to fill that association, are the `Profile` rows effectively fetched and the collection filled.

See the chapter 4.8 in [Ler09] for details on this matter. Note that EF 4.0 and NHibernate use implicit Lazy Load that, by creating a Virtual Proxy at runtime, do load the object properties as needed, while at the same time preserving some degree of persistence ignorance.

With eager loading (in p. 207), the EF produces a query that results in an outer join of all the fields of a `User` row with all its corresponding `Profile` rows. Analysing its generated SQL, the outermost select is only for sorting the tuples by the `User` primary key column (`Id`) and by the `Profile Id`. In this case, as there is only one `User Id`, it is irrelevant to sort by `User Id`. In fact it adds unnecessary query plan for sorting a relation. If the relation is small enough to fit in memory, either an index scan or a table scan with an in-memory sorting algorithm will perform well [UGMW01]. If the relation is too large, multi-pass algorithms like sort-merge join a better solution (see appendix A for details).

With the SQL produced by EF in eager loading (in p. 207), using a left outer join (rather than a left inner join) is unnecessary, because there is only one `User` row in the driving table. Additionally, it is not clear what importance have the two `CASE` clauses and the corresponding fields, although they seem irrelevant as well.

The actual MySQL 5.1 query plan is given by the following events: first it performs an operation for reading one row only (of table `User`); second it runs a full table scan fetching all the rows (1000) in `Profile` table that have the condition `Profile.UserId = User.Id` and while performing an in-memory sort algorithm (`filesort`). Depending on the clustering factor MySQL would either use full table scan or index search. For this case, in which all the 1000 rows in `Profile` are relevant to the result, having an index on `Profile.UserId` column is irrelevant, i.e. MySQL uses full table scan anyway.

On the other hand, the generated SQL with deferred loading (in p. 208) is much cleaner and simpler to understand.

Table 4.2 displays the results of a performance test running a query that loads a `User` with its 1000 `Profiles`.

Comparing the eager loading with the deferred loading examples, the latter performs slightly faster, which is likely due to the slow SQL query join with sorting and nested selects. Also note that `Person` is only fetched once in deferred loading.

All three eager loading queries (example 1, 2 and 3 in p. 207) have similar cost times. The only variable is the LINQ compilation time, which is not significant in here.

Regarding the deferred loading examples, using Linq-to-Entities and then the `Load` method (in p. 208), results on fetching the `User` once and then all its 1000 `Profiles`

Table 4.2: EF eager and deferred load query examples benchmark

Examples	**SQL executed queries**	**100 operations in seconds (n=2)**	**1000 operations in seconds (n=3)**
1. EQL 1	10^n	7.843750	74.015625
2. Linq-to-Entities 1	10^n	7.843750	73.890625
3. Linq-to-Entities 2	10^n	7.968750	75.156250
4. EF API with deferred load	$1 + 10^n$	6.984375	64.562500
5. Linq-to-Entities 1 with deferred load	$1 + 10^n$	7.359375	66.953125

when the `Load` method is called. Using EF API, generates the same SQL queries, but runs slightly faster because it avoids the LINQ compilation time.

In the end, even though eager loading is slower in the examples above, the overhead of communicating with the database throughout the network and the bottlenecks can be higher in real cases. That typically makes the deferred loading approach less efficient for more complex queries and multi-user scenarios.

4.1.6 Dynamism in EF

The most important requirement of Cachapuz's CazFramework is the ability to define dynamic lists at runtime based on SQL and on top of database schema changes at runtime, rather than relying on a code generation of new entities and rebuild of the application whenever changes are needed. Also, at times, it is important to be able to project some of the fields in an entity without the trouble of specifying a new entity, thus optimizing speed.

Dynamism can be implemented at various levels in EF 4.0. Loading dynamic objects from the database can be implemented at various levels with the following strategies:

- Projections with Linq-to-Entities, returning **anonymous types** (EF 1.0). See p. 103;
- Projections in EQL using **Named Type Constructors** (EF 1.0). See p. 104;
- Use the native provider database API and waive the EF features. EF 4.0 provides API (`ExecuteStoreQuery(Of T)`) to run SQL directly to the RDBMS and then automatically populating an entity or DTO;
- Create entities and graphs dynamically. This consists of defining entities and associations in runtime using reflection and a more dynamic EF API. It can use dynamic

structures like strings or `KeyValuePair` collections as input. See the example in chapter 17.8 of [Ler09]. Note that the same example is supported by an additional assembly that defines the POCO objects to be transformed into EF entities. To create actual object types in runtime, in .NET, requires code instrumentation.

Note that the examples bellow use a slightly different database and object schema than that of the other examples in 4.1.5.1. Also, here, the DBMS used is SQL Server 2008. The object `User` is mapped with table `Users`, containing more fields. Also the Repository of users in the ObjectContext is called `Users` rather than `User` of the other examples. Hence, these naming conventions are more readable than those in 4.1.5.1.

Projections with Linq-to-Entities

In the example below, the Linq-to-Entities query only fetches two fields of the object `User`. The LINQ `Select` command returns an `IQueryable(Of <anonymous type >)`. This is available since EF 1.

Code:

```
Dim query = ctx.Users.Select(Function (c) New With {.Name = c.Name,
                                                    .Email = c.Email})
Dim users = x.Take(10).ToList()
Dim name = users.First().Name
```

SQL generated:

```
SELECT
    [Limit1].[C1] AS [C1],
    [Limit1].[Name] AS [Name],
    [Limit1].[Email] AS [Email]
FROM (
    SELECT TOP (10)
        [Extent1].[Name] AS [Name],
        [Extent1].[Email] AS [Email],
        1 AS [C1]
    FROM [dbo].[User] AS [Extent1]
) AS [Limit1]
```

If it is previously implemented a DTO class, e.g. a `UserShort` containing its name and email only, that result can be *casted* to a `UserShort`. Otherwise it is still possible to manage this **anonymous type** like a DTO but without being able to serialize it.

The **anonymous type** is a compile trick, i.e. it is strongly typed and made available in compile time automatically, which alleviates the burden of writing each and every managed object. However, this does not operate at runtime and therefore it lacks some dynamism. Also, the Unit of Work is unavailable with **anonymous types**

More on this subject can be found in chapter 4.2 of [Ler09].

Projections with EQL

Since EF 1.0 it is possible to project fields using EQL as well. This is done using a **Named Type Constructor** as in the following example:

Code:

```
Dim query As IEnumerable(Of User) = ctx.CreateQuery(Of User)( _
"SELECT VALUE CazApplication.Entities.User (u.userID, null, u.Password, & _
    "u.Email, u.UserProfileID, u.IsAdmin, u.IsActive)" & _
"FROM Users as u")
Dim users = query.Take(100).ToList()
users.First().Name = "John"
ctx.SaveChanges()
```

SQL generated:

```
SELECT
    CAST(NULL AS int) AS [C1],
    [Extent1].[UserID] AS [UserID],
    CAST(NULL AS varchar(1)) AS [C2],
    [Extent1].[Password] AS [Password],
    [Extent1].[Email] AS [Email],
    [Extent1].[UserProfileID] AS [UserProfileID],
    [Extent1].[IsAdmin] AS [IsAdmin],
    [Extent1].[IsActive] AS [IsActive],
FROM [dbo].[User] AS [Extent1]
```

Thus, inside the EQL, the named type constructor must contain all the fields of the entity in the correct order. The not required fields must be specified with null. Here, the Unit of Work does not operate as with the **anonymous types**. Note that the constructor has to be written with its full name, including the *namespace* where the entity resides.

4.2 NHibernate

NHibernate is an ORM framework for .NET and a port from the well known Hibernate for Java, which has been widely used for a long time in enterprise software. One of the key points of this tool is its determined DDD orientation, supporting the management of a dominant Domain Model over a submissive database that can easily change its schema to match the Domain Model. The essential features of NHibernate are the following:

- Domain Model pattern;
- Interoperability among different database providers and SQL dialects;
- Query Object (HQL, Linq-to-NHibernate, **Criteria** API);
- Unit of Work (Session), Identity Map and optimistic locking (control fields);
- Code customization for audit logging, via IoC;
- Lazy Load via Virtual Proxies created at runtime;
- Persistence ignorance POCO entities;
- Reflective Metadata Mapping with generic Data Mappers;
- Composite keys;
- Unidirectional and bidirectional associations of one-to-one, one-to-many and many-to-many;
- Object-relational hierarchy patterns of table-per-class-family, table-per-class and table-per-concrete-class;
- Generation and management of updates to the database, i.e. NHibernate takes care of the database schema and the developer only has to worry about the object schema.

However, the lack of support in Visual Studio designer and GUI widgets makes NHibernate rather unappealing for .NET developers. Even though there are paid frameworks and Visual Studio add-ins to generate NHibernate Metadata Mapping definitions and POCO entities code from the database, by default it requires the developer to code them manually.

This section analyses the NHibernate (version 2.1 and 3) with some theory and practical tests as well. Most of the supporting documentation used here is of Hibernate and NHibernate 3.

4.2.1 Unit of Work

Similar to the EF ObjectContext, in NHibernate, the Session is the core and most important class. That is, it implements a full featured Unit of Work that handles transactions, caching (Identity Map), optimistic locking, Virtual Proxy domain objects with Lazy Load, Data Mapper and Repository via an external LINQ implementation (Linq-to-NHibernate) as well as other Query Object implementations (HQL and `Criteria` API).

The `SessionFactory` class manages the whole Domain Model schema, at runtime and generically, using reflection from the Metadata Mapping definitions. It is thread-safe, unlike Session class and as the name suggests, it creates Session instances.

When updating an object to the database, Session checks the *changeset* and only updates what changed. Note that within the same Session, if the transaction is not committed to database the changes made to the persistent objects are saved in the Unit of Work (Session) itself. All users that operate within the same Session will see these objects changed, even if not committed.

Moreover, NHibernate is able to manage POCO entities but at the same time traceable by a Session. For that, the instance of `ISession` (Session) is injected via constructor injection into the Virtual Proxies, generated at runtime, that inherit from the actual POCO domain entities. This provides flexibility on control over the life-cycle of every domain object that is attached to a Unit of Work.

Additionally, each of these runtime subclasses contain some flag to determine if changes ought to be propagated to the Unit of Work. Also any subclass must override all the setters from the corresponding POCO entity and implement the Unit of Work **object registration** strategy, which implies accessing the Session it is attached to (i.e. contains reference of `ISession`) and mark that object and property dirty. Note that to enable this kind of behaviour, all the properties of that POCO entity must be `overridable` or `virtual`.

Thus, whenever the Session loads an entity, it instantiates a Virtual Proxy subclass of that same entity and sets the flag to not mark the object dirty before its data is loaded from the database through its setters.

An important consideration is that, by default, NHibernate only updates the dirty objects but as a whole, i.e. the SQL updates contain all the columns in a table even if only one entity field has changed. However, it can be configured to generate dynamic updates in the Metadata Mapping file (e.g. `Person.hbm.xml`) via the class attribute `dynamic-update`. Hence, NHibernate will update only the fields that changed, and for that it uses dynamic proxies. Note that this strategy does not always enhance performance

[KBA+12].

For creating dynamic proxies, NHibernate uses an IoC container. It relies on .NET reflection and runtime class enhancement (via **Castle.DynamicProxy** library), also known as *bytecode* instrumentation [Sta11].

A Session is a non thread-safe lightweight object that provides a level of abstracting from the database transaction management. The simplest way of coding a transaction is opening a session, reading and updating objects and then flush it (using `session.Flush()`), so the changes are persisted to the database at once. Similar to EF, nonetheless, it is possible to acquire more control over a transaction by using `ITransaction` API including `Commit` and *Rollback* methods. For more information and code examples see [KBA+12].

Note that using multiple NHibernate transactions does not grant that in DBMS multiple transactions will be used. Also, there is no guarantee that some method in Session will always perform the SQL call to the database because NHibernate always tries to delay that call as long as possible. That is except for `ISession.Flush` and `ITransaction.Commit` methods which force database synchronization (Unit of Work commit).

Often the Session object instance is misinterpreted as being a data store because it behaves like an OODBMS, which can hinder performance if misused. One example of a bad practice is using *session-per-operation* anti-pattern (consists of opening a session for each query or update to the database) for it generates too many calls to the database and excessive overhead.

The most common and encouraged pattern in hibernate documentation ([KBA+12]) is the *session-per-request*, suitable for most applications. That leads to a one-to-one relationship between Session and database transaction.

Sometimes, however, a request may seem too brief when, for instance a user interacts with the system by sending multiple requests in a short time period. For that there are two options: *session-per-request-with-detached-objects* or *session-per-conversation* (reconnects and disconnects the session rather than closing it and without the need to reattach objects). See [KBA+12] for details.

Another common mistake is having a *session-per-application*, which is a shared Session instance working as an in-memory data store (Repository). This approach presents a number of challenges:

- In-memory changes over objects that are loaded from the Session will apply to all users of that application instance;
- Multi-threaded Session requires additional complex code for locking management;

- Session will have stale data if other applications update the same database often;

- Clear Session once in a while is required, including empty *changeset*, Identity Map and other in-memory caches so that it does not reach out of memory exceptions;

Having a shared Session is renouncing the already implemented RDBMSs ACID properties and therefore it is rarely required for N-Tier applications. Although, for single user applications with single user databases it may perform well, it still needs to clear the Session from time to time.

4.2.2 Optimistic Locking

NHibernate implements Optimistic Offline Lock by default. Optimistic locking is configurable using control fields in the Metadata Mapping files (e.g. `Person.hbm.xml`) with class and property attribute `optimistic-lock` ([Sta11] and [KBA+12]:

- In the class, `optimistic-lock` can be set to: `version` (checks a version column on SQL updates), `all` (all the fields are compared in the SQL update), `dirty` (only the dirty fields are compared in the SQL update) and `none`. With `optimistic-lock="version"` a new column (often a `Long` type) has to be created and mapped to a field, in the Metadata Mapping file via `version` or `timestamp` element. Both the elements allow the version column to be managed by the database rather than the NHibernate, which implies using the attribute `generated="always"`;

- In the property, `optimistic-lock` can be either true or false (default is true), which specifies if the class version should increment when that property changes. It only applies if the class itself is set to use optimistic lock with `version` or `timestamp`.

For pessimistic locking, NHibernate Session provides additional locking mechanisms for reads, writes, upgrades, etc.

4.2.3 Lazy Load

Lazy Load is one of the most discussed patterns of Hibernate and NHibernate for its high variety of implementations and complexity of configuration as well as the performance enhancements and problems it brings about. The Lazy Load strategies implemented in this framework match some similarities to the ones presented in 3.3.3. Hence, NHibernate defines the different fetching methods ([Sta11], [KBA+12]):

- Immediate fetching, i.e. when an entity is loaded, its one-to-one and one-to-many (collection) associations or class attributes are fetched immediately. For instance, if a `Person` has a one-to-many association with `Profile`, when a `Person` object is loaded, it fetches all its profiles as well. This option can be defined within the elements of the associations (in Metadata Mapping file) with `fetch="join"` (generates one SQL query with a join) or `fetch="select"` (default option that generates one SQL query for `Person` plus `N` SQL queries for the profiles unless the batch size is set to more than 1 for that collection). Also, note that collections are set to lazy by default, which require `lazy=false` definition to enable immediate fetching;

- Lazy collection fetching, i.e. a collection is only loaded from the database when the application invokes an operation upon that collection. It does not require dynamic proxies, rather the lazy collection type has its iterator implement some data loader. See 3.3.3 for details on how a `ProxyList` is implemented;

- Batch fetching, i.e. a collection is fetched by chunks of a fixed batch size while keeping track of a limit and an offset. For instance, if a collection of profiles is defined with a batch size of 10, as that collection is iterated, at most, it performs `N/10` queries with `N` being the length of the collection. In one-to-many associations, if the collection is intended to be fully iterated, it is more efficient to eagerly load it (with a `fetch="join"`) in the first place. Batch fetching is used together with lazy collection fetching;

- Proxy fetching, i.e. a single-valued association (one-to-one or many-to-one) is fetched only when any of the properties of the associated object are invoked, except its id. For instance, if a `Person` has a many-to-one association with `Profile` and it is set to use proxy fetching, when a `Person` is loaded, it will not fetch the associated `Profile` immediately. That association is instantiated with a Ghost object that only contains its id with all the other properties to null. The `Profile` will actually be loaded (entirely) only when its properties other than the `id` are called.

  Note that a many-to-one association is mapped to the database with a foreign key constraint in the `PersonTable` that points to the primary key of the `ProfileTable` and thus it is possible to instantiate the Profile object with the id before accessing the `ProfileTable`. A one-to-one association can be either mapped in the same way or having the same primary key in both tables.

  Proxy fetching is implemented in NHibernate as in EF through *subclassing* at runtime, which replaces `Profile` class with a subclass `ProfileProxy` that overrides its getter properties. For that, in the Metadata Mapping file, the `Person` many-to-one association must be defined with `lazy="proxy"` (only implemented since NHibernate 3 and new default) and, in the class, the getter property has to be set virtual so it can be `overridable`. For a better understanding see the following example of pseudo-code:

```
class Person
   virtual property Profile() : Profile
       returns _profile

'at runtime
class ProfileProxy inherits Profile
   override property Name() : String
      if _name is null
         _session.Load(Of Profile)(this) 'db call
      return _name
```

With the above example, when a `Person` is loaded, its `Profile` getter property returns a `ProfileProxy` (encapsulated with `Profile` type) which is a Ghost object containing only the `Profile` id from the foreign key in `PersonTable`. Casting problems arise when using inheritance with proxies (unless using table-per-class-family). More details can be found in section 3.3.3. Proxy fetching requires *bytecode* instrumentation;

- Lazy attribute fetching, i.e. a class attribute (single-valued other than collections and associations) is fetched when the instance variable is first accessed (implemented since NHibernate 3). This approach is similar to proxy fetching, except it is applied to the basic typed properties which can be set to `lazy="true"` in the Metadata Mapping. This requires *bytecode* instrumentation and is rarely an efficient optimization, also if entities contain too many fields, that is often a sign of poor Domain Model design. An alternative to avoid lazy attribute fetching is using queries with field projection;

- No-proxy, i.e. single-valued association (one-to-one or many-to-one) is fetched when the instance variable is first accessed. This does not use a proxy for the associated object, which avoids some casting problems explained in section 3.3.3 (implemented since NHibernate 3.2). For instance, if a Person has a many-to-one association with `Profile`, when the `Profile` getter property is accessed, it will instantiate the `Profile` object (POCO instead of a proxy) with all its properties eagerly fetched. Note that the `Person` object must have lazy initialization behaviour injected on its property getter `Profile` as the following example of pseudo-code demonstrates:

```
[commandchars=\\\{\}]
class Person
   virtual property Profile() : Profile
       return _profile

\verbHighlight{'at runtime:}
class PersonProxy inherits Person
   override property Profile() : Profile
      _profile =_session.GetNamedQuery("ProfileByPerson")
                        .SetParameter("id",this.id)
                        .FirstOrDefault() \verbHighlight{'db call}
      return _profile
```

This feature is enabled, in the Metadata Mapping file, setting the association element with `lazy="no-proxy"`. Requires *bytecode* instrumentation and is rarely needed.

By default, NHibernate uses lazy collections for one-to-many or many-to-many associations and proxy fetching for single-valued associations (one-to-one and many-to-one). Proxy fetching for single-valued is not implemented in NHibernate 2. Note that other than lazy collection fetching, lazy single-valued attributes always require proxies.

Proxied single-valued attributes always require some getter property be decorated via code injection with code to specifically load something. NHibernate accomplishes this with **Castle.DynamicProxy** library that generates dynamic proxies at runtime with *bytecode* instrumentation.

Thus by default, NHibernate 3 generates proxies for all persistent classes and uses them to enable lazy fetching of single-valued associations and attributes. The proxies are subclasses of the POCO entities and require that a default constructor and virtual properties are implemented in the POCO objects [KBA+12].

4.2.4 Code Customization: Audit Logging

Enterprise layered applications and frameworks often rely on the combination and integration of different services together. Customizing code generation, event handling or intercepting method calls are important features that allow the default behaviour of an application to be changed especially when the requirements are variable.

Typically, audit logging and accessibility or security control are some of those services, for they exist in most applications. Although more specific business components are also very common, e.g. automatic translation, message censorship or very specific GUI customizations for special customers.

This kind of services is commonly related to non-functional requirements, not coupled to the core business logic of the application. For that, they ought to be disabled without affecting the rest of the application.

Audit logging is one of those services, which consists of tracking data changes often keeping some log record containing the user responsible for the changes, the date and time of the event, the type of event and the data affected.

Although auditing can be done manually (continuous effort of coding tedious code and more code is susceptible to more bugs) or using database triggers (slow, can cause bottlenecks, compromises business logic to the database and may become slower as logic complexity escalates), NHibernate provides built-in mechanisms for intercepting persistence events, without affecting the core business logic of the application.

For that, NHibernate uses an implicit IoC container approach via `IInterceptor` implementation which intercepts some specific behaviour or persistence events that occur on certain entities.

The example in Figure 4.4 is based on the one in chapter 8.4 of the book [KBHK09]. That same example demonstrates the implementation of `IInterceptor` approach, which is built upon the following steps:

- Define information to be logged: user id, date and time, type of log, entity affected, etc;
- Create an `AuditRecord` entity, with all the fields to be logged and define it in the NHibernateMetadata Mapping as an immutable entity, which means once a record is inserted it cannot be updated;
- Have the audit-able entities implement an `IAuditable` interface;
- Implement the interface `IInterceptor` (all methods) or create an `AuditInterceptor` class that inherits from `EmptyInterceptor` and overrides only the methods intended to use. `EmptyInterceptor` is a class that implements all the methods in `IInterceptor` with no behaviour (empty).

There are some particular aspects of `NHibernate.IInterceptor`. Because it intercepts some methods called in a Session, it defines a `SetSession` method to configure a session to be intercepted by some interceptor class (implementing `IInterceptor`). The most relevant methods in `IInterceptor` are:

- `OnSave` - called when an object is set to be saved with SQL `INSERT` but not necessarily immediately;

- `OnFlushDirty` - called when an object is detected to be dirty during a flush;

- `OnPostFulsh` - called after the database commit.

Figure 4.4 defines a multi-layered approach for auditing a `Bank` web service. All the numbered classes from 1 to 5 correspond to the various layers of code (IoC container code not included) that are used within the server, when a `Bank` service is requested.

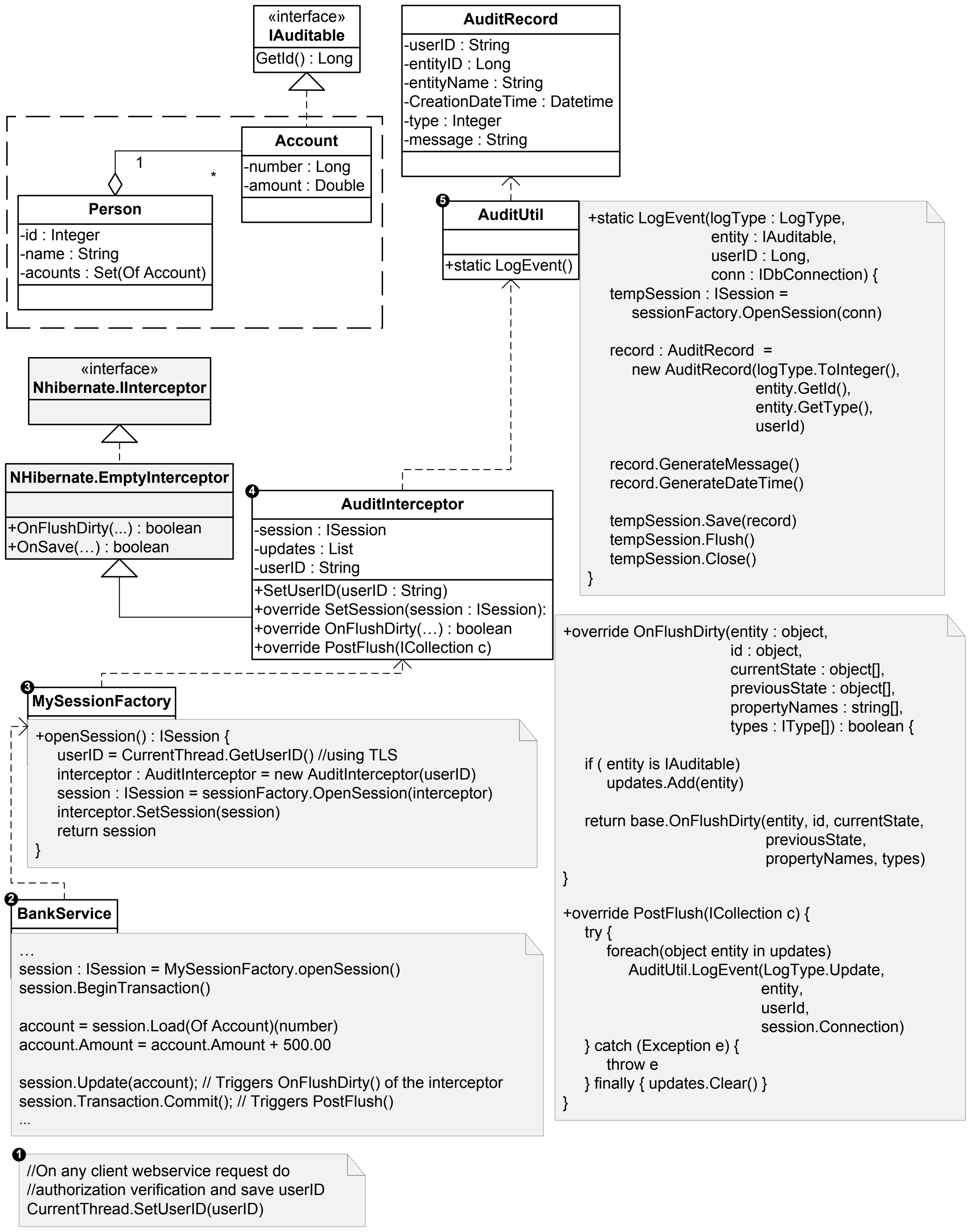
Note that `AuditInterceptor` contains an additional `userID` field for keeping track on who is responsible for committing the changes.

Another feature in Figure 4.4 is implicit authentication. .NET N-Tier applications often use WCF services that already support authentication features and allow interceptors for calls on service methods. Other options are using TLS to manage the current caller id (the id of the user that performed that service request) and avoid passing this id to the business logic on every method as a parameter. This requires, at the beginning of the service call, that the user id is fetched from a http session or cookie and then set the current thread to be aware of this user id.

Thus, the example (Figure 4.4) implements Decorator to decorate the `sessionFactory.openSession` method in order to factory a Session with both the `AuditInterceptor` configured and the id fetched from the current thread (Abstract Factory). It is not pointed in the example but `MySessionFactory` would replace `SessionFactory`, which then would require the former to implement a lot more methods.

Instead of using NHibernate to persist logs, libraries like **log4net** can be used. Although it is recommended to encapsulate the persistence mechanism as in N-Tier architectures so it becomes easier to change that same mechanism.

Other methods for implementing more specific IoC, would require the use of libraries such as **Castle Windsor** or **Spring.NET**.



```
+static LogEvent(logType : LogType,
                 entity : IAuditable,
                 userID : Long,
                 conn : IDbConnection) {
    tempSession : ISession =
        sessionFactory.OpenSession(conn)

    record : AuditRecord  =
        new AuditRecord(logType.ToInteger(),
                        entity.GetId(),
                        entity.GetType(),
                        userId)

    record.GenerateMessage()
    record.GenerateDateTime()

    tempSession.Save(record)
    tempSession.Flush()
    tempSession.Close()
}
```

```
+openSession() : ISession {
    userID = CurrentThread.GetUserID() //using TLS
    interceptor : AuditInterceptor = new AuditInterceptor(userID)
    session : ISession = sessionFactory.OpenSession(interceptor)
    interceptor.SetSession(session)
    return session
}
```

```
…
session : ISession = MySessionFactory.openSession()
session.BeginTransaction()

account = session.Load(Of Account)(number)
account.Amount = account.Amount + 500.00

session.Update(account); // Triggers OnFlushDirty() of the interceptor
session.Transaction.Commit(); // Triggers PostFlush()
...
```

```
+override OnFlushDirty(entity : object,
                       id : object,
                       currentState : object[],
                       previousState : object[],
                       propertyNames : string[],
                       types : IType[]) : boolean {

    if ( entity is IAuditable)
        updates.Add(entity)

    return base.OnFlushDirty(entity, id, currentState,
                             previousState,
                             propertyNames, types)
}

+override PostFlush(ICollection c) {
    try {
        foreach(object entity in updates)
            AuditUtil.LogEvent(LogType.Update,
                               entity,
                               userId,
                               session.Connection)
    } catch (Exception e) {
        throw e
    } finally { updates.Clear() }
}
```

```
//On any client webservice request do
//authorization verification and save userID
CurrentThread.SetUserID(userID)
```

Figure 4.4: Implementation of `AuditInterceptor` in NHibernate

4.2.5 Testing NHibernate

This section describes a practical analysis of the features in NHibernate 2.1. Unlike EF, NHibernate does not feature Visual Studio designer, GUI widgets or code generators. Coding the NHibernate Metadata Mapping definitions, the persistent objects and other configuration settings by hand can be a tedious job that any developer expects to avoid when using an ORM framework. For that reason, this NHibernate analysis used Sculpture 2.1 to help such process. Sculpture is a model driven code generation framework that integrates with other technologies in an N-Tier architecture. The data tier can either be configured to use EF or NHibernate ORM frameworks. Note that this analysis is not intended to test Sculpture but NHibernate.

This analysis begins with Sculpture generating a Domain Model (POCOs) and the NHibernate Metadata Mapping definitions from the database schema (in Figure 4.5) using MySQL 5.1. Figure 4.6 presents the Visual Studio designer (Sculpture feature) with the domain classes (POCOs) generated by Sculpture. In order to generate the class associations, the MySQL database schema had to be configured with **InnoDB**.

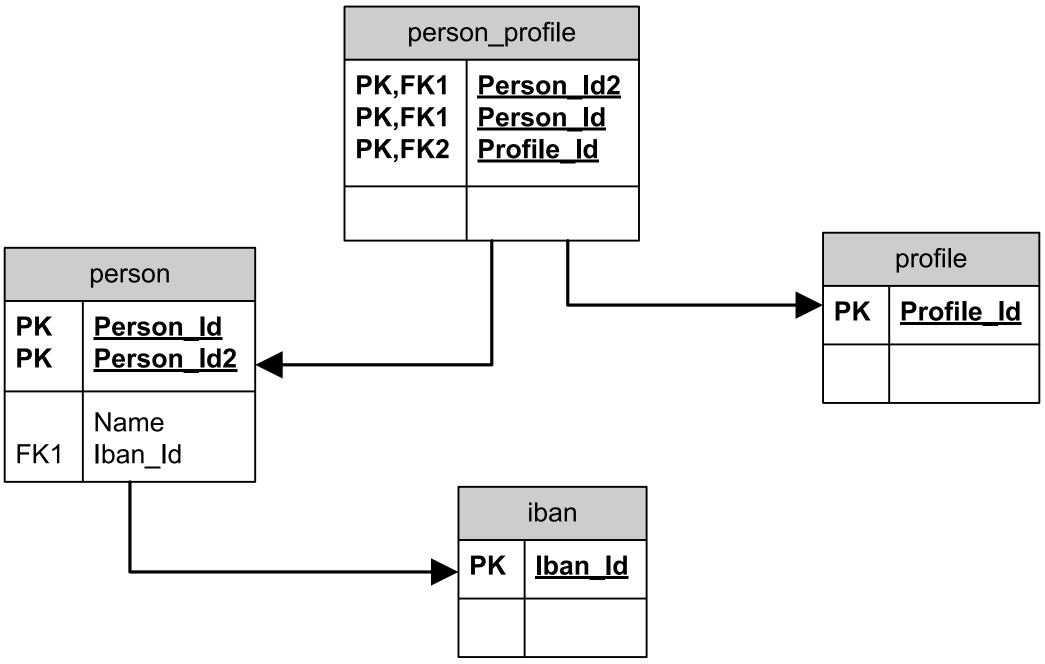


Figure 4.5: Testing NHibernate: MySQL database schema

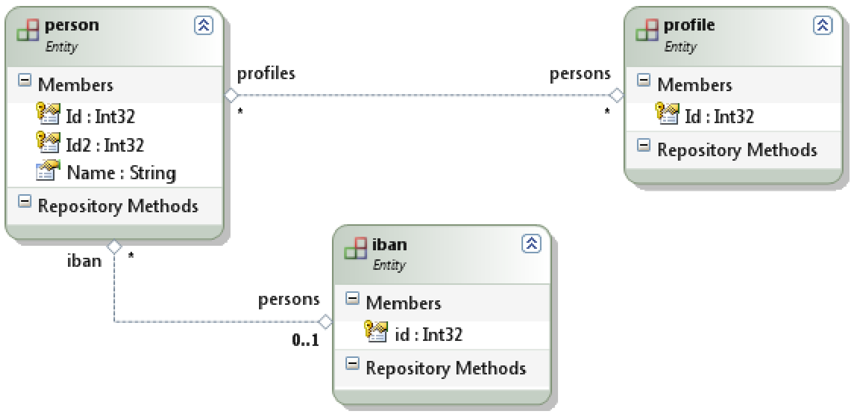


Figure 4.6: Testing NHibernate: generated object schema

Thus, Sculpture generates two Visual Studio projects:

- `DataAccess` project: contains the NHibernate configurations (Metadata Mapping, Session configuration helpers) and Repository definitions as demonstrated in Figure 4.7;

- `Entities` project: contains the generated entities. One example is demonstrated in Figure 4.8.

Note that Sculpture has `personRepositoryBase` and `personBase` as the classes with generated code and then it provides `personRepository` and `person` empty classes that inherit from the `Base` classes and can be manually coded with additional features.

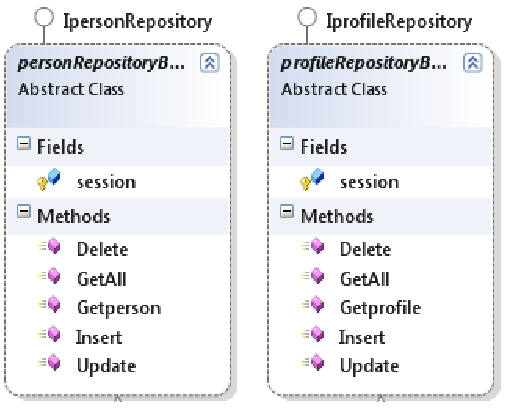


Figure 4.7: Testing NHibernate: Repository classes generated by Sculpture

Each generated Repository class (Figure 4.7) encapsulate a Session, which is a singleton Session variable instantiated once per application start-up in a helper. Here, Sculpture implements the session-per-application pattern which is not recommended because the Session object is not thread safe and is designed to be very shot-lived (see section 4.2.1). The other methods in the repository implement the basic CRUD NHibernate API.

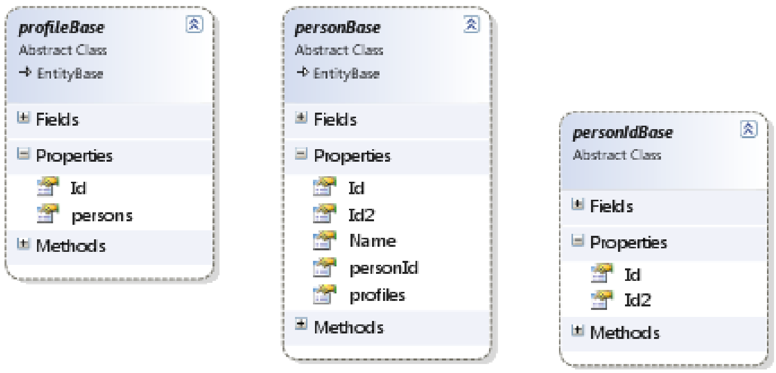


Figure 4.8: Testing NHibernate: POCO `Person` classes generated by Sculpture

In NHibernate, composite keys have to be mapped to a new type. Thus, Sculpture generated the `personId` class (see its superclass `personIdBase` in Figure 4.8) to embed both the ids corresponding to the composite key of the table. The Metadata Mapping definitions, specified in file `person.hbm.xml` for configuring the composite key as well as its many-to-many association with `profile`, are given by the following code:

```
<!--person.hbm.xml-->

<composite-id name="personId" class="personId">
   <key-property name="Id" column="Id" type="Int32"/>
   <key-property name="Id2" column="Id2" type="Int32"/>
</composite-id>

<set name="profiles" table="person_profile" lazy="true" cascade="save-update">
   <key>
      <column name="Person_Id"/>
      <column name="person_Id2"/>
   </key>
   <many-to-many class="profile">
      <column name="Profile_Id"/>
   </many-to-many>
</set>
```

Note that the `profile set` (many-to-many association) is set to use the lazy collections strategy and cascading save (insert new) or update. The same definitions of this association are set in `profile.hbm.xml` file which are similar to the above example but with inverted keys. With cascading, in this bidirectional association, if a new `profile` is added to the `profiles set` of a `person` and then only the `person` is saved, the changes will propagate through the association and call the SQL insert for the `profile` before inserting a new row in the Association Table Mapping `person_profile`.

The `MySql.Data.dll` and `NHibernate.ByteCode.Castle.dll` files have to be added to the `Build` folder in order to compile the Visual Studio projects generated by Sculpture. That is because both the Dynamically Linked Library (DLL) assemblies are loaded with reflection in runtime to provide database provider driver and *bytecode* instrumentation (proxies).

The following examples provide a test to the NHibernate API using the test database, domain model and generated code above.

4.2.5.1 Simple Load and Identity Map

Many-to-one load association on `person`

Code:

```
session.Load(Of person)(new personId(2,1))
```

SQL generated, using mapping settings on the many-to-one association (`iban`) of `person` with default settings: `fetch="join"` or `outer-join="true"`:

```
SELECT this_.Id as Id2_1_, this_.Id2 as Id2_2_1_,
   this_.Name as Name2_1_, this_.ibanId as ibanId2_1_,
   iban2_.id as id3_0_
FROM person this_
left outer join iban iban2_ on this_.ibanId=iban2_.id
WHERE this_.Id = 2 and this_.Id2 = 1 limit 2
```

SQL generated, using mapping settings on the many-to-one association (`iban`) of `person` with default settings: `fetch="select"` or `outer-join="false"`:

```
SELECT this_.Id as Id2_0_, this_.Id2 as Id2_2_0_,
this_.Name as Name2_0_, this_.ibanId as ibanId2_0_
FROM person this_
WHERE this_.Id = 2 and this_.Id2 = 1 limit 2

SELECT iban0_.id as id3_0_ FROM iban iban0_
WHERE iban0_.id=111
```

Many-to-one associations are eager by default and cannot be set lazy in NHibernate 2.1 (proxy fetching was implemented since NHibernate 3, see section 4.2.3). Here, the fetching strategy can either be set to `join` (default) or `select`. The former executes a single SQL select joining both tables (person and iban) and the latter executes 2 separate SQL select (one for each table).

In this case, because any loaded person only gets one iban, it is convenient to use the join fetch strategy because it produces less calls to the database. Note that due to NHibernate default lazy collections, profiles are not eagerly loaded.

Identity Map on profile

Code:

```
session.Load(Of profile)(1) 'loads from db
session.Load(Of profile)(2) 'loads from db
session.Load(Of profile)(1) 'loads from cache
session.Load(Of profile)(2) 'loads from cache
```

SQL generated:

```
SELECT profile0_.Id as Id0_0_
FROM profile profile0_ WHERE profile0_.Id=1

SELECT profile0_.Id as Id0_0_
FROM profile profile0_ WHERE profile0_.Id=2
```

This demonstrates the behaviour of NHibernate level 1 cache, also known as Identity Map. This map is stored in a Session which is intended to be short-lived or be flushed and cleaned regularly. Note that this behaviour is not expected when other queries like Linq-to-NHibernate or HQL are executed, although there are other manageable levels of caching, not covered here.

4.2.5.2 Linq-to-NHibernate join query examples

Because NHibernate 2.1 does not implement LINQ, a contribute library was built to provide Linq-to-NHibernate rather than using criteria API or session.QueryOver(Of T). For that it is required to import the NHibernate.Linq *namespace* and then use session.Linq(Of T)(). NHibernate 3 already provides a built-in LINQ provider by using session.Query(Of T)(). Note that the following examples use the NHibernate 2.1 LINQ provider.

Fetching a person filtered through its iban id

Code:

```
query = From per As person In session.Linq(Of person)
        Where per.iban.id = 333 Select per
person = query.First
```

SQL generated:

```
SELECT this_.Id as Id2_1_, this_.Id2 as Id2_2_1_,
   this_.Name as Name2_1_, this_.ibanId as ibanId2_1_,
   iban1_.id as id3_0_
FROM person this_
   left outer join iban iban1_ on this_.ibanId=iban1_.id
WHERE iban1_.id = 333 limit 1
```

Although in the relational paradigm, the inner join is more advertised, runs faster and is adequate to a number of use cases than the outer join, the fact is that in ORM frameworks, it makes more sense for simple data listing, to fetch all the persons, even those that do not contain an `iban`.

Fetching all profiles that have any persons associated

Code:

```
profiles = session.Linq(Of profile)
   .Where(Function(u) u.persons.Any).ToList()
'OR
query = From p In session.Linq(Of profile)
   Where p.persons.Any
profiles = query.ToList()
```

SQL generated:

```
SELECT this_.Id as Id0_0_ FROM profile this_
WHERE this_.Id in (
   SELECT this_0_.Id as y0_ FROM profile this_0_
   WHERE exists(
      select 1 from person_profile
      where this_0_.Id=Profile_Id))
```

Fetching all the profiles that contain any persons associated falls within the typical nested-loop join (see appendix section A.4.1 for details on nested-loop joins). Because the actual `person` data was not requested, NHibernate opted for a different solution than the inner join. Although it seams the outer most select is irrelevant for such query.

Fetching all persons that have a profile with id=1

Code:

```
query = From per In session.Linq(Of person),
   pro In p.profiles Where pro.Id = 1 Select per
persons = query.ToList()
```

SQL generated:

```
SELECT this_.Id AS Id3_2_, this_.Id2 AS Id2_3_2_,
   this_.Name AS Name3_2_, this_.ibanId AS ibanId3_2_,
   iban3_.id AS id2_0_, profiles4_.Person_Id AS Person2_4_,
   profiles4_.person_Id2 AS person3_4_,
   p2x1_.Id AS Profile1_4_, p2x1_.Id AS Id0_1_
FROM person this_
   LEFT OUTER JOIN iban iban3_ ON this_.ibanId=iban3_.id
   LEFT OUTER JOIN person_profile profiles4_ ON
      this_.Id=profiles4_.Person_Id AND
      this_.Id2=profiles4_.person_Id2
   LEFT OUTER JOIN profile p2x1_
      ON profiles4_.Profile_Id=p2x1_.Id
WHERE p2x1_.Id = 1
```

Despite the many left outer joins and the size difference between the Query Object and the SQL command, the above example is very simple and straightforward. All it does is perform an eager fetch for the persons while filtering.

4.2.5.3 Lazy Collections examples

Default one-to-many association navigation

Code:

```
profile = person.profiles(0)
```

SQL generated:

```
SELECT profiles0_.Person_Id as Person2_1_,
   profiles0_.person_Id2 as person3_1_,
   profiles0_.Profile_Id as Profile1_1_,
   profile1_.Id as Id0_0_
FROM person_profile profiles0_
   left outer join profile profile1_
      on profiles0_.Profile_Id=profile1_.Id
WHERE profiles0_.Person_Id=2 and profiles0_.person_Id2=1
```

Although collections are lazy by default, once even the first element is accessed, all the collection is loaded at once.

Default one-to-many association navigation (inverted)

Code:

```
count = profile.persons.Count
'OR
person = profile.persons.First
```

SQL generated:

```
SELECT persons0_.Profile_Id as Profile1_2_,
   persons0_.Person_Id as Person2_2_,
   persons0_.person_Id2 as person3_2_,
   person1_.Id as Id2_0_, person1_.Id2 as Id2_2_0_,
   person1_.Name as Name2_0_, person1_.ibanId as ibanId2_0_,
   iban2_.id as id3_1_
FROM person_profile persons0_
   left outer join person person1_
      on persons0_.Person_Id=person1_.Id and
         persons0_.person_Id2=person1_.Id2
   left outer join iban iban2_ on person1_.ibanId=iban2_.id
WHERE persons0_.Profile_Id=1
```

The above example presents the inverted association of the prior example and it generates a similar SQL statement.

One-to-many association with eager load

The following example demonstrates the behaviour of an eager collection. For that, in the `profile` mappings, the `persons` set (one-to-many association) is set to `lazy="false"`.

Code:

```
profiles = session.Linq(Of profile)().ToList()
```

SQL generated:

```
SELECT this_.Id as Id0_0_ FROM profile this

SELECT persons0_.Profile_Id as Profile1_2_,
   persons0_.Person_Id as Person2_2_,
    persons0_.person_Id2 as person3_2_,
    person1_.Id as Id3_0_, person1_.Id2 as Id2_3_0_,
    person1_.Name as Name3_0_, person1_.ibanId as ibanId3_0_,
    iban2_.id as id2_1_
FROM person_profile persons0_
    left outer join person person1_ on
        persons0_.Person_Id=person1_.Id and persons0_.person_Id2=person1_.Id2
    left outer join iban iban2_ on person1_.ibanId=iban2_.id
WHERE persons0_.Profile_Id=3

SELECT persons0_.Profile_Id (...) WHERE persons0_.Profile_Id=2

SELECT persons0_.Profile_Id (...) WHERE persons0_.Profile_Id=1
```

In the above example, first NHibernate fetches all the profiles and then, for each `profile` it loads the according persons eagerly. In here, NHibernate decided to perform N+1 different `selects` rather than running a single `select`. While the N+1 approach retrieves $Rows(profile)+Rows(profile)*Rows(person)$ rows, the single `select` approach only retrieves $Rows(profile) * Rows(person)$ rows. However, the latter is a cross join which returns a Cartesian product between the profiles and the persons. In turn, that returns a lot of repeated data which ends up being slower than the N+1 approach.

Note that eager load may incur in fetching the whole database at once into memory. The advantage of this is that all data can be passed to upper layers and become detached from the Session without facing lazy exceptions.

One-to-many association with lazy load (default)

The following example demonstrates the behaviour of a lazy collection. For that, in the `profile` mappings, the `persons` set (one-to-many association) is set to `lazy="true"`, which is the default.

Code:

```
profiles = session.Linq(Of profile)().ToList()
For Each pro In profiles
    pro.persons(0).Id
Next
```

This approach generates the same N+1 SQL `selects` as the last example (4.2.5.3), except each `select` is executed at different stages. The `ToList()` generates the first `select` which fetches all the profiles with no associations filled. Then, on each iteration of the `for` cycle, NHibernate executes one SQL for fetching the `persons` set of each `profile`

One-to-many association with lazy load and batch-size The following example adds to the above, the batch-size feature applied to the `persons` collection in the `profile` mappings with `batch-size="2"`. Note that the code is the same as the prior example and only the SQL generated is different.

SQL generated:

```
SELECT this_.Id as Id0_0_ FROM profile this

SELECT persons0_.Profile_Id as Profile1_2_,
   persons0_.Person_Id as Person2_2_,
   persons0_.person_Id2 as person3_2_,
   person1_.Id as Id3_0_, person1_.Id2 as Id2_3_0_,
   person1_.Name as Name3_0_, person1_.ibanId as ibanId3_0_,
   iban2_.id as id2_1_
FROM person_profile persons0_
   left outer join person person1_ on
      persons0_.Person_Id=person1_.Id and persons0_.person_Id2=person1_.Id2
   left outer join iban iban2_ on person1_.ibanId=iban2_.id
WHERE persons0_.Profile_Id in (1, 2)

SELECT persons0_.Profile_Id (...) WHERE persons0_.Profile_Id=3
```

In here, NHibernate fetches the lazy collection (one-to-many association) `persons` but in groups of 2 elements rather than one by one, during the cycle. Thus it performs the following number of `selects` (`div` returns the integer division, `mod` returns the remainder of integer division, N is the number of persons and B is the batch-size):

$$\begin{cases} div(N,B)+1, & \text{if } mod(N,B)=0 \\ div(N,B)+2, & \text{if } mod(N,B)>0 \end{cases}$$

For this example, NHibernate executes 3 `selects`.

Another way to enable this batch-size behaviour is to set the `person` class mapping to `batch-size="2"`. However, this approach will take effect on all the associations that involve a collection of persons.

Note that HQL and Linq-to-NHibernate always fetch a collection eagerly when the `ToList()`, or other method that calls `GetEnumeratior()`, is invoked on that `IQuery` or

`IQueryable` interface which is returned by `session.CreateQuery()` or extension methods `session.Linq(Of T)` and `session.Query(Of T)`.

Another important note about collections is that NHibernate does not only support the collection `set`. Using a `set`, on running `person.profiles.add(...)`, NHibernate fetches all the profiles from database before inserting the new one. That is because a `set` does not allow duplicated elements and that kind of validation is only possible if the whole collection is loaded to memory first. Other collection types such as `map` or `bag` do not require this kind of validation and thus are able to insert a new element to a lazy collection without entirely loading it.

4.2.5.4 Cascading Delete operations

Simple delete without cascading on delete

The following example can either use no cascade or `cascade="save-update"` in `persons` association of `profile` class mappings.

Code:

```
session.Delete(session.Load(Of profile)(3))
```

SQL generated:

```
SELECT this_.Id as Id0_0_ FROM profile this_
WHERE this_.Id = 3 limit 2

DELETE FROM person_profile WHERE Profile_Id = 3

DELETE FROM profile WHERE Id = 3
```

Sometimes it can be a good idea to wrap this whole operation in a single database transaction to prevent other users from updating that `profile` right before being deleted.

Simple delete with cascading on delete

The following example configures the `iban` class mappings to use a `cascade="all"` on the `persons set` (one-to-many association).

Code:

```
session.Delete(session.Load(Of iban)(333))
```

SQL generated:

```
SELECT this_.id as id3_0_ FROM iban this_
WHERE this_.id = 333 limit 2

SELECT persons0_.ibanId as ibanId1_, persons0_.Id as Id1_,
   persons0_.Id2 as Id2_1_, persons0_.Id as Id2_0_,
   persons0_.Id2 as Id2_2_0_, persons0_.Name as Name2_0_,
   persons0_.ibanId as ibanId2_0_
FROM person persons0_ WHERE persons0_.ibanId=333

DELETE FROM person_profile
WHERE Person_Id = 4 AND person_Id2 = 1

DELETE FROM person WHERE Id = 4 AND Id2 = 1

DELETE FROM iban WHERE id = 333
```

The second `select` fetches all the persons using that `iban` so that they can all be deleted with the `iban`. Because in the `person` association of profiles there is no cascade on delete, the profiles have been spared. This approach can be dangerous when the Domain Model is interlinked with many associations, for the whole database can be deleted at once.

4.2.5.5 HQL examples

The most flexible way of querying with NHibernate is using HQL which has a very similar syntax with the SQL except it queries the object model rather than the relational model. Its learning curve is slower than Linq-to-NHibernate but it provides a wider set of commands. HQL is compiled at runtime using ANTLR (the `Antlr3.Runtime.dll` has to be included in the `build` folder) unlike Linq-to-NHibernate which is built at compile time.

An HQL query is treated as a string so it does not provide strong type checking. The method `session.CreateQuery(HQLQueryString)` is used to return a `NHibernate.IQuery` that is only executed when a method like `IQuery.List()` is invoked.

HQL simple load (1)

Note that HQL queries are lazy by default, meaning that all the persons were fetched but with lazy one-to-many associations all according to the Metadata Mapping definitions.

Although it is possible to eagerly fetch the associations even though they are defined as lazy.

Code:

```
query = session.CreateQuery("from person as p")
ilist = query.List()
```

SQL generated:

```
select person0_.Id as Id3_, person0_.Id2 as Id2_3_,
   person0_.Name as Name3_, person0_.ibanId as ibanId3_
from person person0_

SELECT iban0_.id as id2_0_ FROM iban iban0_ WHERE iban0_.id=222

SELECT iban0_.id as id2_0_ FROM iban iban0_ WHERE iban0_.id=111

SELECT iban0_.id as id2_0_ FROM iban iban0_ WHERE iban0_.id=333
```

All the 4 SQL statements above were generated when executing the HQL query (running the `IQuery.List()` method). The query returns a `System.Collections.IList` filled with `person` objects.

HQL simple load (2)

Code:

```
ilist.Item(0).profiles(0)
```

SQL generated:

```
SELECT profiles0_.Person_Id as Person2_1_,
   profiles0_.person_Id2 as person3_1_,
   profiles0_.Profile_Id as Profile1_1_,
   profile1_.Id as Id0_0_
FROM person_profile profiles0_
   left outer join profile profile1_ on
      profiles0_.Profile_Id=profile1_.Id
WHERE profiles0_.Person_Id=1 and profiles0_.person_Id2=2
```

Only when that `person.profiles` association is iterated are the profiles actually loaded.

HQL join example (1)

The following example returns all the persons that have at least one profile.

Code:

```
session.CreateQuery("from person as p join p.profiles as prof")
       .List()
```

SQL generated:

```
select person0_.Id as Id3_0_, person0_.Id2 as Id2_3_0_,
   profile2_.Id as Id0_1_, person0_.Name as Name3_0_,
   person0_.ibanId as ibanId3_0_
   from person person0_
   inner join person_profile profiles1_ on
      person0_.Id=profiles1_.Person_Id and
      person0_.Id2=profiles1_.person_Id2
   inner join profile profile2_ on
      profiles1_.Profile_Id=profile2_.Id

SELECT iban0_.id as id2_0_ FROM iban iban0_ WHERE iban0_.id=222

SELECT iban0_.id as id2_0_ FROM iban iban0_ WHERE iban0_.id=111

SELECT iban0_.id as id2_0_ FROM iban iban0_ WHERE iban0_.id=333
```

Because it was not specified what to collect from that join (select or project) NHibernate returns an `IList` of pairs in the form of array objects with length 2 and containing both the `person` and the `profile` as a tuple.

Essentially, HQL supports `inner join` (`join`), `left outer join` and `right outer join`. Because in this example, there is no left or right join, the query only fetches the persons that contain at least one profile. Thus, NHibernate uses an SQL inner join. An HQL `left outer join`, on the other hand, would imply an SQL outer join to fetch all the persons even those with no profiles.

Note that an HQL join does not fill the lazy associations within an object unless the command `fetch` is used. For instance to set the query above to return persons and filling the lazy association profiles it would require the following HQL code:

```
from person as p join fetch p.profiles as prof
```

Note that the above example uses generates an SQL inner join. For an outer join it would have to be rewritten as the following:

```
from person as p left join fetch p.profiles as prof
```

HQL join example (2)

The following example demonstrates the behaviour of an inner join with field projection.

Code:

```
session.CreateQuery("select p.Name, prof.Id from person as p" & _
                     "join p.profiles as prof").List()
```

SQL generated:

```
select person0_.Name as col_0_0_, profile2_.Id as col_1_0_
from person person0_
   inner join person_profile profiles1_ on
      person0_.Id=profiles1_.Person_Id and
      person0_.Id2=profiles1_.person_Id2
   inner join profile profile2_ on
      profiles1_.Profile_Id=profile2_.Id
```

This query returns an `IList` of array objects with length 2 containing a tuple of `string` and `int`.

4.2.5.6 Dynamic LINQ

Dynamic LINQ is LINQ without strong type checking, i.e. using strings in the clauses `Where`, `Select`, `OrderBy`, etc. Thus, LINQ becomes *serializable* and writeable at runtime.

Dynamic LINQ is a sample code provided in Visual Studio 2008 in the folder:

```
...\ProgramFiles\Microsoft Visual Studio 9.0\Samples\1033\CSharpSamples
\LinqSamples\DynamicQuerys
```

. The `Dynamic.cs`, when imported in some project, it extends the `IQueryable` interface with extension methods providing new LINQ methods that take strings as arguments. Note that EF already provides **Dynamic LINQ** without requiring this.

Dynamic LINQ sample 1

Code:

```
session.Linq(Of person).Where("personId.Id = 1 &&" & _
   "personId.Id2=1").FirstOrDefault()").List()
```

SQL generated:

```
SELECT this_.Id as Id3_1_, this_.Id2 as Id2_3_1_,
   this_.Name as Name3_1_, this_.ibanId as ibanId3_1_,
   iban2_.id as id2_0_
   FROM person this_
      left outer join iban iban2_ on this_.ibanId=iban2_.id
   WHERE (this_.Id = 1 and this_.Id2 = 1) limit 1
```

Note that the composite key is accessed by the `personId` class created by Sculpture.

Dynamic LINQ sample 2

Code:

```
Dim ienum As IEnumerator = session.Linq<person>
   .Where("it.personId.Id = 1")
   .OrderBy("iban.Id desc")
   .Select("New (Name,iban.Id)")
   .GetEnumerator()
Dim array = GetArrayList(ienum)
```

SQL generated:

```
SELECT this_.Name as y0_, iban1_.id as y1_
FROM person this_
   left outer join iban iban1_ on this_.ibanId=iban1_.id
WHERE this_.Id = 1
ORDER BY iban1_.id desc
```

The above example demonstrates the projection features of **Dynamic LINQ**. `IEnumerator` is dynamic and through reflection it is able to convert the result into an `ArrayList` of `Dictionaries(propertyName,value)`.

Note that joins are not implemented in **Dynamic LINQ**.

4.2.5.7 Database Synchronization

NHibernate provides database schema updates from domain objects and Metadata Mappings in runtime. The following code uses reflection to load the NHibernate configurations (database provider, SQL dialect, `connectionString`, etc) and load the assembly containing the entity POCO classes with the Metadata Mapping files:

```
Configuration configuration = new Configuration();
configuration.Configure(Assembly.GetExecutingAssembly(),
   "SculptureSolution.DataAccess.GeneratedCode.hibernate.cfg.xml");
configuration.AddAssembly("SculptureSolution.Entities");
```

Update schema

Code:

```
var update = new NHibernate.Tool.hbm2ddl.SchemaUpdate(configuration);
update.Execute(true, true);
```

SQL generated:

```
create table iban (id INTEGER not null, primary key (id))
alter table person add column ibanId INTEGER
```

This example updates the database schema consulting the `information_schema` tables and columns of MySQL. If the `iban` table and the reference in person to `iban` was previous deleted in the database, this update will create the table `iban` and the column `ibanId` in person table.

Create and Drop schema

The following code demonstrates how to manage schema exports for Create and Drop operations with NHibernate API:

```
Dim export = New NHibernate.Tool.hbm2ddl.SchemaExport(configuration);

''only drops all tables
export.Execute(true, true, true); ''justDrop = true
''or
export.Drop(true, true);

''first drops and then creates
export.Execute(true, true, false); ''justDrop = false
''or
export.Create(true, true);
```

Note that the second option does not update, it drops and then creates the database schema. All registers are lost.

4.2.6 Dynamism in NHibernate

Dynamic LINQ (see 4.2.5.6 for examples) consists of querying objects in a criteria like way. It is simple, well structured and *serializable*, but fails for more complex queries especially because there is no support for joins. Joins could be implemented but this still would not make it a good solution for complex queries.

HQL (see 4.2.5.5 for examples) is ORM specific, which would tie the business logic to the ORM specific API. It is very similar to SQL and very feature-rich which is good for complex queries. HQL is portable or *serializable* as a single string object and has a similar syntax to LINQ but with other more database specific features within the object paradigm.

The disadvantages of HQL are especially the typical problems of using an ORM, like the dual-schema problem. That is, it needs the object schema updated whenever something changes in the database, otherwise it will not be able to perform the correct mappings.

If a new `connectionString` is dynamically added to the application, in order to run a remote query, it is impossible to set up the model so it can be immediately queried with HQL. That is unless the object schema (entity classes) is generated again and the `SessionFactory` is created with the updated schema definitions (Metadata Mapping). This can be done programmatically because the `SessionFactory` receives this schema in XML as a parameter during its creation. Thus, the same application can instantiate multiple `SessionFactory` instances with different Metadata Mapping definitions and loading new DLL assembles containing new or updated entities via reflection if needed.

However, NHibernate is designed to work well with DDD and not with legacy database centric systems.

4.3 Conclusions

This chapter analysed and described some relevant features of both ORM frameworks EF and NHibernate. Various ORM patterns (presented in chapter 3) were identified in these frameworks, such as Unit of Work, Lazy Load, Optimistic Offline Lock, Query Object, etc.

EF and NHibernate were analysed according to the requirements of Cachapuz (presented at the beginning of this chapter). Both ORM frameworks support database interoperability among various providers and RDBMSs and composite keys are managed in a similar way. Optimistic locking can be easily configured by both ORM frameworks by setting any field to a version (timestamp) column in the database or use the control fields defined by Cachapuz legacy database.

EF 1.0 manages non POCO entities implementing Unit of Work behaviour on setter properties, which is not a very portable solution. On the other hand, EF 4.0 already uses dynamic proxies for overriding the POCO entity setters like NHibernate. However, NHibernate does not enable proxies by default, which disables dynamic updates if not specified otherwise. This is due to the fact that proxies do not always bring performance enhancements and sometimes generate some issues. In here, it is clear that NHibernate attempts to keep a closer to PIM model with entirely serializable and portable POCO entities. On the other, EF 4.0 does not manage POCO entities by default.

Lazy Load is only supported by EF 4.0 and NHibernate, although the latter is more versatile. Both EF and NHibernate drive Partial-Object problems, especially on client-server applications.

Regarding the learning curve of both frameworks, the lack of Visual Studio support on NHibernate is determinant. Also, NHibernate adopts a configuration over convention philosophy with more extensive and complex API, e.g. `Session.Update` does not update the object until `Session.Flush` is called. EF only uses `SaveChanges` for committing all changes inflicted to the in-memory attached domain objects, which is simpler.

Code customization in EF 1.0 is easy to implement due to (defensive) extension points in the `ObjectContext` class (generated in Visual Studio). NHibernate does not provide code generation, it rather uses IoC interceptors for code injection, which is a cleaner approach but also more complicated.

Both EF and NHibernate implement LINQ and OQL with very similar query features. HQL is equivalent to EQL and both are versatile methods for writing simpler object queries that generate complex and bigger SQL queries. In fact they are very similar to SQL. Those not familiar with SQL can use the LINQ implementations for simpler querying and better abstraction. Linq-to-Entities with eager load (using implicit joins) apparently generates overly complicated SQL in some cases (see example in p. 207).

Additional performance tests will be conducted in chapter 6, although it is obvious that these frameworks add some overhead on applications.

Regarding the dynamism requested by Cachapuz, both ORM frameworks provide field projections and joins among non related objects, which returns dynamic structures similar to DataReaders and without any Unit of Work, caching or Lazy Load features.

However, these dynamic features by the two ORM solutions were not entirely satisfying to Cachapuz. Both EF and NHibernate take too much control of the database, especially because they do not solve the Dual-Schema problem. The dynamism required for adding new tables, domain objects and querying complex SQL is not fulfilled with these ORM frameworks. Even though these ORM frameworks provide some level of database-first code generation, legacy databases and bottom-up development is not a good practice with such tools for they tend to hinder substantially the flexibility of the object model.

Hence, the solution falls within ClassBuilder enhancements.

Chapter 5

Implementation of CazDataProvider

Cachapuz development environment is set around large legacy databases and CazFramework. Fundamentally, CazFramework is a very generic and dynamic framework for managing data lists (results of SQL queries). Almost all of its DAL classes are generated by ClassBuilder ORM tool, which is developed by Cachapuz as well.

The rise of new .NET ORM frameworks like EF, lead to the possibility of migrating Cachapuz DAL mechanism to one of those frameworks and thus discarding ClassBuilder. However, because the ORM frameworks analysed in chapter 4 did not meet the needs of Cachapuz, it was decided to keep using the same paradigm and technology with ClassBuilder.

Meanwhile, it became a usual requirement of Cachapuz customers to use CazFramework with different RDBMSs other than SQL Server (e.g. MySQL, SQLite). Thus, database interoperability became a priority feature to be implemented in ClassBuilder.

To meet this requirement, the CazFramework DAL needed to replace the specific data provider API (Object Linking and Embedding, Database (OLEDB)) with an abstract one. For that, the ClassBuilder DAL generated classes had to replace the OLEDB provider API dependency with an abstract provider API that encapsulates the different database providers and SQL dialects. For details on how ClassBuilder (before implementing CazDataProvider) works see 5.1.

Therefore, CazDataProvider, a custom data provider, was developed in .NET to meet the needs of Cachapuz and encapsulate a group of .NET data providers. Overall, this chapter describes the implementation of CazDataProvider. Its API was further adopted by ClassBuilder to replace the OLEDB API.

Apart from the research carried out in this dissertation until this chapter, the implementation of CazDataProvider demanded for some more analysis. First, this chapter presents an analysis on ClassBuilder and a research on ADO.NET data providers. The following sections present various possible architectures of CazDataProvider and its implementation.

5.1 Analysis of ClassBuilder

ClassBuilder is a light ORM tool built by Cachapuz to support a continuous development of DAL components working on top of legacy databases. Thus, ClassBuilder generates entities, Metadata Mapping definitions and implements a CRUD API all in VB.NET code, pluggable to the CazFramework DAL.

The key advantage of ClassBuilder over other ORMs is its simplicity for it only depends on one schema (the relational). Providing a Relational Domain Model (see section 5.1.1 for details) to the business logic, it manages one POCO entity class per table of the database. Also, because Cachapuz legacy databases do not use foreign key constraints, due to conventions of performance and other problems, associations and hierarchy can not be generated automatically by any tool. Therefore, in ClassBuilder these features can only be implemented manually.

Overall, ClassBuilder supports the following features:

- Relational Domain Model;
- Database-first development with one-way code generation;
- Persistence ignorance with POCO entities;
- Simple Unit of Work implementation with optimistic locking, transactional support via data provider API (`IDbTransaction`) and controlled updates (only dirty objects are updated to the database)
- Audit logging;
- Composite keys;
- CRUD API with SQL query filtering;
- SQL Server 2005 and 2008 support (data provider and SQL dialect).

Unlike other ORMs like NHibernate or EF, ClassBuilder does not implement a full featured Unit of Work, Identity Map, Lazy Load, Query Object or database interoperability.

An alternative to deal with a fixed relational schema (e.g. on legacy databases) could consist of using the Table Module pattern and the DataSet .NET API. Although, DataSet is particularly resource-consuming, it lacks customization features, flexibility and N-Tier (POCO) support.

The following subsections provide insight on what is the Relational Domain Model, how ClassBuilder implements Data Mapper, Unit of Work and audit logging.

5.1.1 Relational Domain Model

This section describes a domain logic pattern discovered in ClassBuilder to characterize the DAL strategy used in CazFramework. Also, it helps understand how ClassBuilder avoids certain ORM problems.

The Relational Domain Model is no more than a simple Domain Model(see section 3.1.3 for details) whose model does not take advantage of inheritance or associations. Therefore, it avoids certain ORM problems, meaning a significant reduction in complexity. Legacy systems of database centric applications and bottom-up development strategies find here a more attractive and simple solution than with Domain Model pattern.

Relational Domain Model is characterized by having a one-to-one mapping between classes and tables, similar to the Table Module but without using Record Set. Additionally, it defines a Data Mapper to enable this mapping and provide basic CRUD API.

A Relational Domain Model can be identified and designed with the following characteristics:

- Each entity is modelled as if it were a table row representation, which includes mapping all the columns, including the primary and foreign keys to object fields;
- Typically the entities work as DTOs;
- The domain objects remain isolated from each other without defining any associations or inheritance relationships;
- Data constraints, integrity and relationships between tables are managed manually in domain logic;
- Each domain object requires a Data Mapper that manages a close-to-SQL API;
- The domain objects and Data Mappers must be easily regenerated when the database schema changes.

Some advantages of Relational Domain Model are:

- Because domain objects are POCOs without behaviour, their use in presentation layer is less problematic;
- Data Mappers and domain objects can be generated easily form the database schema using custom tools like ClassBuilder;
- It is a good solution for legacy database centric systems;
- Changing from relational paradigm to object paradigm, abandon SQL, engaging into migration processes and learning of new frameworks is no longer necessary;
- As there is no navigable object graph, Lazy Load is not applicable and thus the avoidance of Partial-Object problems;
- The absence of a persistence context (Unit of Work) allows the domain objects to be easily *serializable* as DTOs.

However, this pattern also encounters some disadvantages such as the following:

- Modelling complex domain logic can become confusing and overly complex without object oriented features like associations and inheritance;
- The Data Mapper CRUD API may be too simplistic for complex querying. More data is likely to be fetched into memory than the actually needed, which breaks performance. A solution can be to define additional methods, in the Data Mapper, containing complex domain logic aware SQL queries.
- Due to the simplicity of Relational Domain Model, there may be no implementation of Unit of Work, Lazy Load, caching, abstraction to SQL, concurrency control, schema synchronizations and other features provided by ORMs like NHibernate or EF, which optimize performance substantially;
- Typically, there is no database interoperability as opposed to ORM frameworks;
- There is no persistence ignorance and therefore, the data integrity is responsibility of the domain logic developer.

Figure 5.1 confronts Relational Domain Model with other domain logic patterns from the section 3.1). This figure is equivalent to the one in the discussion of the section 3.1.4. Although Relational Domain Model requires some effort from the start, with a proper code generator like ClassBuilder, the development of new domain objects can be simple and fast as long as the domain logic complexity does not increase too much.

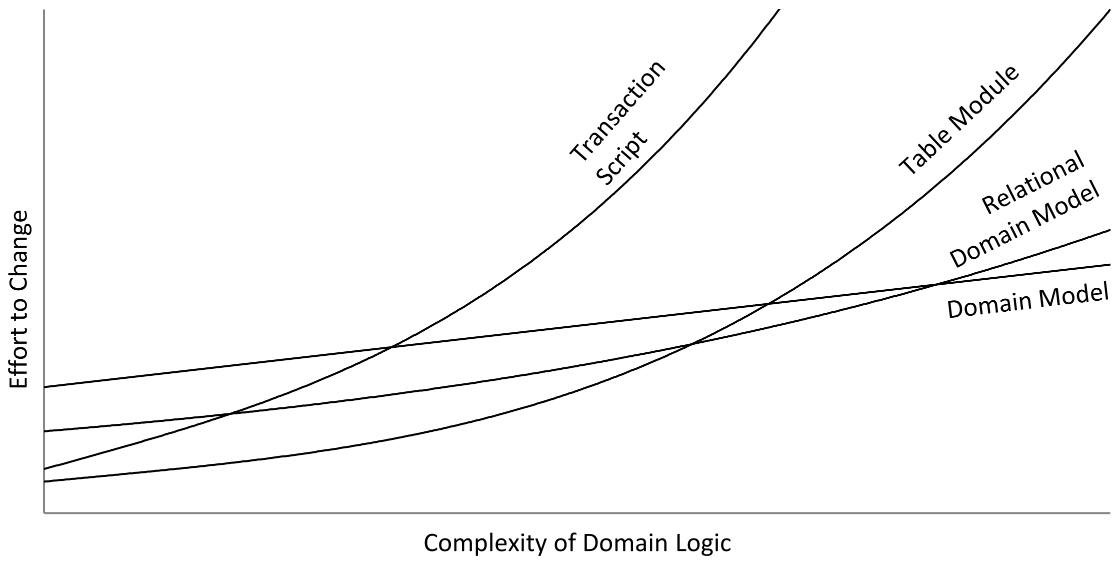


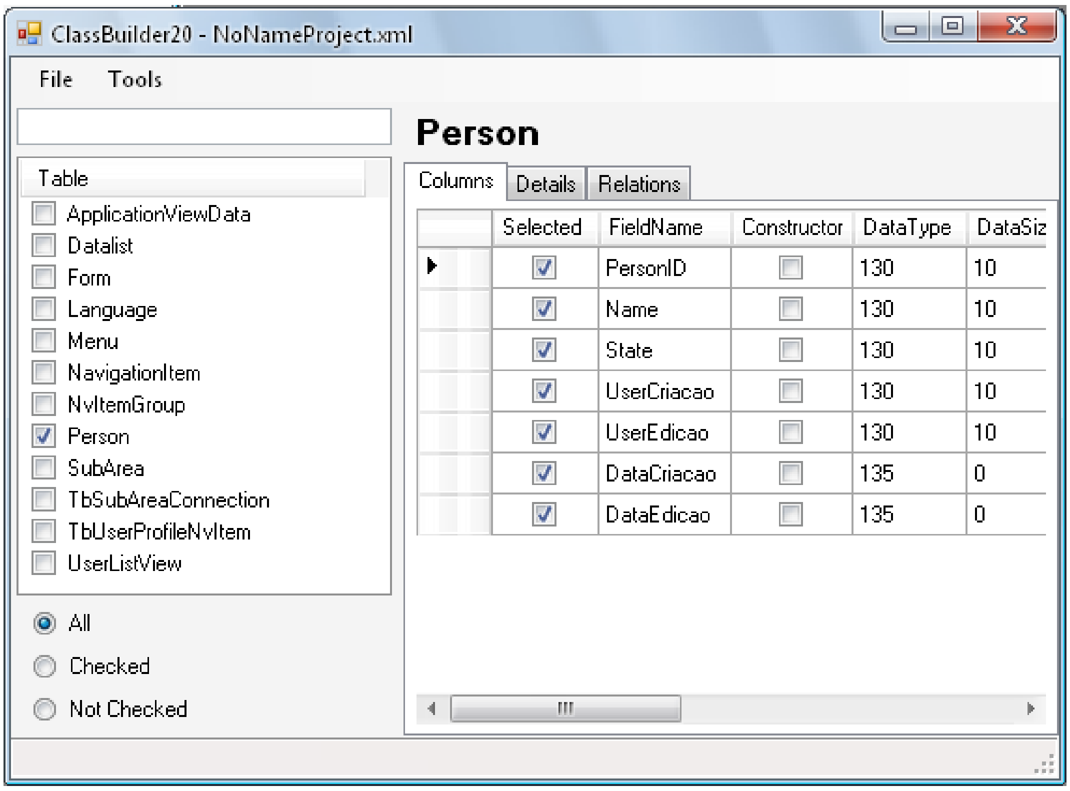
Figure 5.1: Sense of the effort changing rate as a domain logic complexity evolves in Relational Domain Model confronted with the other domain logic patterns (adapted from [Fow02])

5.1.2 Data Mapper

Essentially, ClassBuilder is a GUI windows application (developed in VB.NET) that connects to a database and then displays a set of tables in the schema to be selected for code generation (Figure 5.2). For instance, with a `Person` table selected, it generates the following classes (see Figure 5.3 for details):

- `Person`: POCO entity with one field per column and an additional flag to check if that entity was loaded from the database or is a new `Person`;
- `PersonCol`: a typed collection of `Person`. It inherits from `List(Of Person)` and implements additional methods to handle the entity id;
- `PersonConn`: object that encapsulates SQL queries, CRUD methods, database provider API with connection and transaction management. It implements a Data Mapper, in code Metadata Mapping definitions (which is slightly faster than using reflection), controlled updates (only dirty objects are submitted to database) optimistic locking (optional and implemented via control fields) and audit logging (optional);

The `Person` entity object (in Figure 5.2) can have its fields mapped to some or all of the columns in the according table as if it was a database view. In fact, it is convenient for tables that have a large number of columns (often occurs in legacy databases that



Person

Table
☐ ApplicationViewData
☐ Datalist
☐ Form
☐ Language
☐ Menu
☐ NavigationItem
☐ NvItemGroup
✓ Person
☐ SubArea
☐ TbSubAreaConnection
☐ TbUserProfileNvItem
☐ UserListView

[✓ All] Checked Not Checked

	Selected	FieldName	Constructor	DataType	DataSiz
▶	✓	PersonID	☐	130	10
	✓	Name	☐	130	10
	✓	State	☐	130	10
	✓	UserCriacao	☐	130	10
	✓	UserEdicao	☐	130	10
	✓	DataCriacao	☐	135	0
	✓	DataEdicao	☐	135	0

Figure 5.2: ClassBuilder screenshot before generating `Person` code

have incremental updates to its schema) to manage multiple views of that same table. ClassBuilder can generate different classes for the same table as long as they have different names, e.g `PersonFullView`, `PersonShortView`.

Figure 5.3 demonstrates that `PersonConn` isolates both the relational and the object schemas from each other (Data Mapper). Even though it maps two schemas together, the relational is still mandatory, however any changes to the database schema are not expected to compromise the previously generated code. That is possible by avoiding updates or deletes on the table structure (columns).

The `PersonConn` API (in Figure 5.3) implements basic CRUD with SQL filtering and Explicit Initialize methods to load a `Person` object without *big fields* or only with key fields. Also, it manages the OLEDB data provider API as an abstraction to more specific APIs.

5.1.3 Unit of Work and Optimistic Locking

Although ClassBuilder does not keep track of changes within the relational domain objects like many ORMs, it still provides controlled updates (only dirty objects are updated to the database) and an Optimistic Offline Lock mechanism to manage concurrency control.

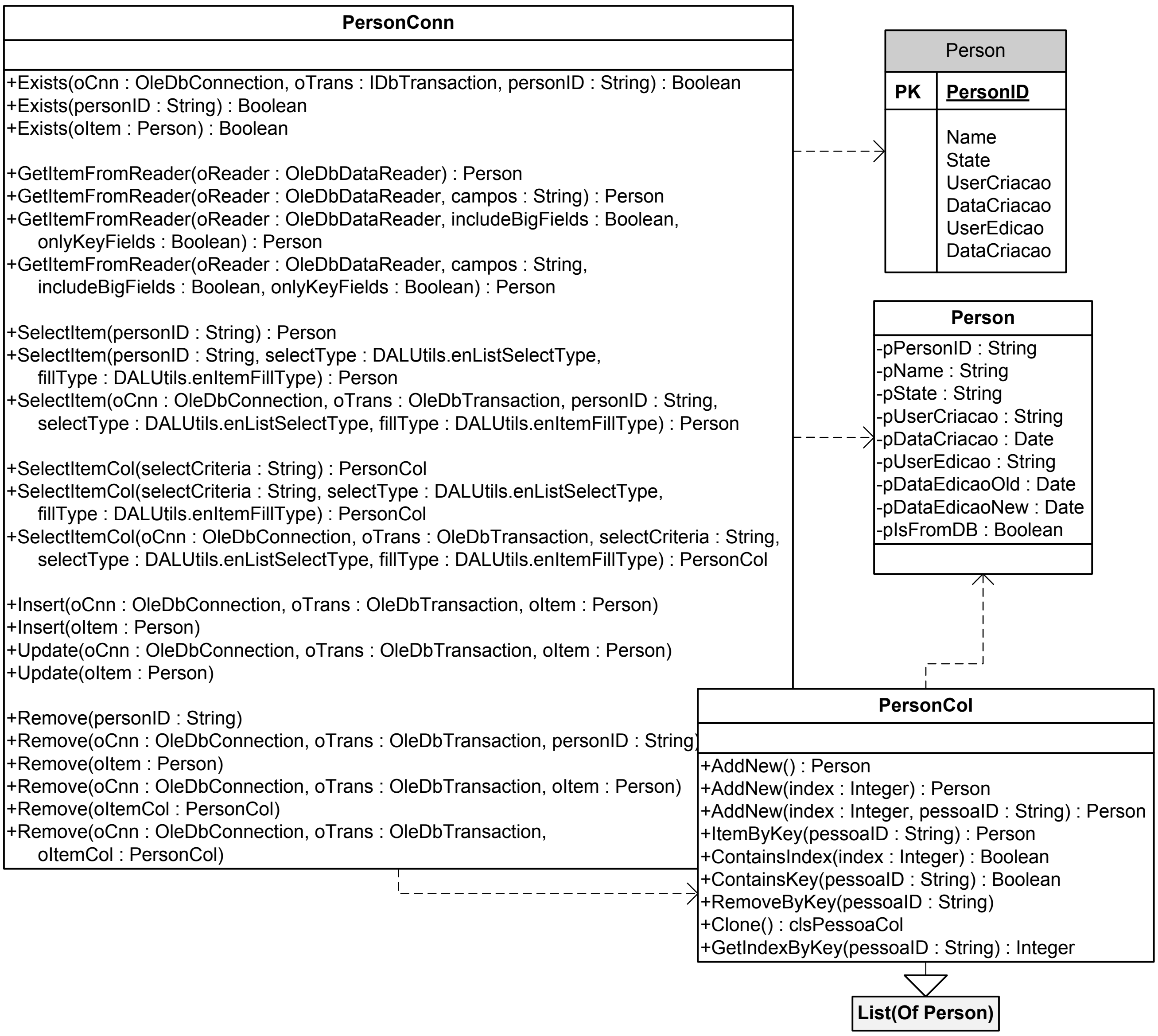
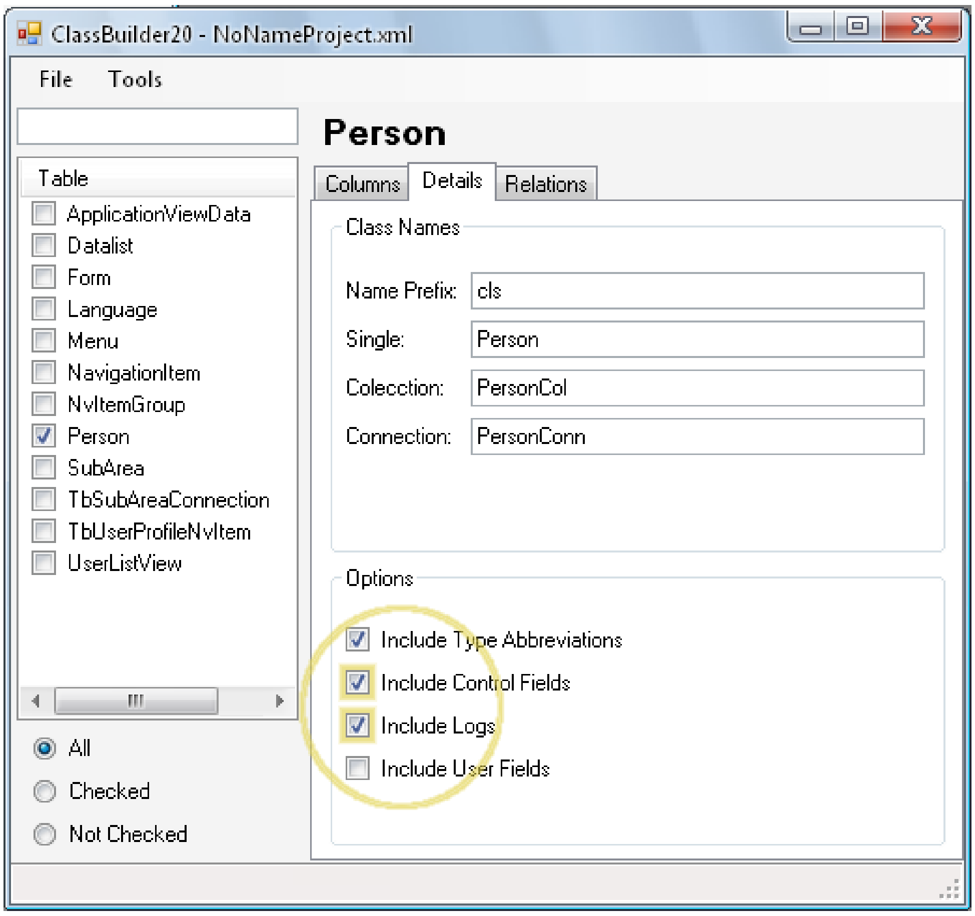

PersonConn
+Exists(oCnn : OleDbConnection, oTrans : IDbTransaction, personID : String) : Boolean
+Exists(personID : String) : Boolean
+Exists(oItem : Person) : Boolean
+GetItemFromReader(oReader : OleDbDataReader) : Person
+GetItemFromReader(oReader : OleDbDataReader, campos : String) : Person
+GetItemFromReader(oReader : OleDbDataReader, includeBigFields : Boolean, onlyKeyFields : Boolean) : Person
+GetItemFromReader(oReader : OleDbDataReader, campos : String, includeBigFields : Boolean, onlyKeyFields : Boolean) : Person
+SelectItem(personID : String) : Person
+SelectItem(personID : String, selectType : DALUtils.enListSelectType, fillType : DALUtils.enItemFillType) : Person
+SelectItem(oCnn : OleDbConnection, oTrans : OleDbTransaction, personID : String, selectType : DALUtils.enListSelectType, fillType : DALUtils.enItemFillType) : Person
+SelectItemCol(selectCriteria : String) : PersonCol
+SelectItemCol(selectCriteria : String, selectType : DALUtils.enListSelectType, fillType : DALUtils.enItemFillType) : PersonCol
+SelectItemCol(oCnn : OleDbConnection, oTrans : OleDbTransaction, selectCriteria : String, selectType : DALUtils.enListSelectType, fillType : DALUtils.enItemFillType) : PersonCol
+Insert(oCnn : OleDbConnection, oTrans : OleDbTransaction, oItem : Person)
+Insert(oItem : Person)
+Update(oCnn : OleDbConnection, oTrans : OleDbTransaction, oItem : Person)
+Update(oItem : Person)
+Remove(personID : String)
+Remove(oCnn : OleDbConnection, oTrans : OleDbTransaction, personID : String)
+Remove(oItem : Person)
+Remove(oCnn : OleDbConnection, oTrans : OleDbTransaction, oItem : Person)
+Remove(oItemCol : PersonCol)
+Remove(oCnn : OleDbConnection, oTrans : OleDbTransaction, oItemCol : PersonCol)
Person
PK
PersonID
Name
State
UserCriacao
DataCriacao
UserEdicao
DataCriacao
Person
-pPersonID : String
-pName : String
-pState : String
-pUserCriacao : String
-pDataCriacao : Date
-pUserEdicao : String
-pDataEdicaoOld : Date
-pDataEdicaoNew : Date
-pIsFromDB : Boolean
PersonCol
+AddNew() : Person
+AddNew(index : Integer) : Person
+AddNew(index : Integer, pessoaID : String) : Person
+ItemByKey(pessoaID : String) : Person
+ContainsIndex(index : Integer) : Boolean
+ContainsKey(pessoaID : String) : Boolean
+RemoveByKey(pessoaID : String)
+Clone() : clsPessoaCol
+GetIndexByKey(pessoaID : String) : Integer
List(Of Person)

Figure 5.3: Example of generated code with ClassBuilder

Optimistic locking is optional and can be enabled by activating the control fields before generating the code, as demonstrated in Figure 5.4. For that, the table must contain the following columns: `UserCriacao` (logged user responsible for inserting that entity into the database), `DataCriacao` (date and time of the creation), `UserEdicao` (logged user responsible for updating that entity into the database) and `DataEdicao` (date and time of the update).

Apart from the `DataEdicao` column, the others are only relevant for logging. As it is visible in Figure 5.3, all the columns are mapped to the according object fields except `DataEdicao` column which origins two fields: `DataEdicaoNew` and `DataEdicaoOld`.

The Unit of Work in ClassBuilder does not keep an Identity Map nor does register


ClassBuilder20 - NoNameProject.xml

File Tools

Person

Columns | Details | Relations

Table
- [ ] ApplicationViewData
- [ ] Datalist
- [ ] Form
- [ ] Language
- [ ] Menu
- [ ] NavigationItem
- [ ] NvItemGroup
- [✓] Person
- [ ] SubArea
- [ ] TbSubAreaConnection
- [ ] TbUserProfileNvItem
- [ ] UserListView

[✓ All] Checked Not Checked

Class Names

Name Prefix:	cls
Single:	Person
Colecction:	PersonCol
Connection:	PersonConn

Options
- [✓] Include Type Abbreviations
- [✓] Include Control Fields
- [✓] Include Logs
- [ ] Include User Fields

Figure 5.4: ClassBuilder screenshot for generating audit logging and control fields

the objects clean or dirty. When control fields are enabled, the generated entities cease to be persistence ignorant (POCO) for they contain behaviour on all the property setters except for `DataEdicaoNew` and `DataEdicaoOld`. This behaviour sets `DataEdicaoNew` to the current time whenever the property setter is invoked, similar to setting the object dirty. This is part of the implementation for both controlled updates and optimistic locking.

The controlled updates work because when updating an object, ClassBuilder checks if its `DataEdicaoNew` is different from `DataEdicaoOld`. If so it forwards it to the updating process. Otherwise it assumes the object has not been modified and therefore ignores the update.

Further on the updating behaviour, the SQL update statement is set to update all the object fields and the `Where` clause to check for the object key fields as well as if the `DataEdicaoOld` is still the same in the according database row. This last condition is what enables the optimistic locking with only one trip to the database. If the SQL update fails, it is likely due to some other user having updated that same object in the meantime and for that it must have changed its `DataEdicao` value. In that case, this function will throw an exception alerting the user that the register had been touched and the operation revoked.

Note that private method `PersonConn.GetItemFromReader` takes an `IDataReader` and translates it into a `Person` instance. During that method call, all the `Person` property setters are invoked which runs the behaviour that verifies if the object was changed (enables the Optimistic Offline Lock). For that it changes the `DataEdicaoNew` to the current time (`pdatDataEdicaoNew = System.DateTime.Now`) on every setter. However, all this behaviour is supposed to be deactivated at the time of instantiating the object in order to not mark it dirty before providing it to the user. Although this is solved at the end of the `PersonConn.GetItemFromReader` call, for it sets `DataEdicaoNew` back to the old (original) value. Ideally, only at the business logic level should the object keep track of changes within itself. This problem is discussed in detail in section 3.3.1.

Unlike what happens with updates (optimistic locking), deletes are not validated concurrently, i.e. if a user loads a `Person` and before updating it, some other user deletes it, the update will fail.

Transactional support is provided in ClassBuilder by allowing the user to access the data provider API and `IDbTransaction` objects directly. The CRUD methods of the Data Mapper (`PersonConn`) accept an optional `IDbTransaction` as argument, and if omitted, no database transaction is executed.

5.1.4 Audit Logging

Audit logging can be enabled by activating the option *Include Logs* in Figure 5.4. Thus, ClassBuilder creates private methods `InsereLog` (for inserting log) and `PreencheCamposRegistoLog` (to fill the fields of a log register) within the Data Mapper (`PersonConn`).

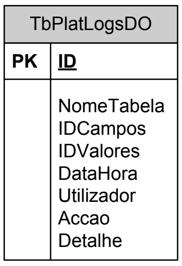


Figure 5.5: ClassBuilder database table for log keeping

Additionally, the database must contain the table `TbPlatLogsDO` as demonstrated in Figure 5.5. The fields of a `clsPlatLogDO` class (mapped to table `TbPlatLogsDO`) consist of:

- `IDCampos`: the field name, e.g. `"PersonID"`;
- `IDValores`: value of the id, e.g. `per.PersonID`;
- `DataHora`: current time;
- `Utilizador`: the user id;
- `Accao`: type of log `"Insert|Update|Delete"`;
- `Detalhe`: XML with serialized log object (`Log.Registo`) containing the object field names and values.

If logs are activated, every time an entity is updated, the Data Mapper loads that same entity again from the database before updating it. Then it inserts a log register from the information of both the old and the new (dirty) entity object and finally the new (dirty) entity is updated to the database. On insert or delete, the Data Mapper only inserts the log after inserting the entity to the database and that log record only contains information of one version of the entity object, new or old (instead of having both versions like in update).

`UserCriacao`, `UserEdicao`, `DataCriacao`, `DataEdicao` are used here as logging information. Another relevant logged information is the `userId`. This field is passed to the Data Mapper (`PersonConn`) when the Data Mapper is instantiated. Then, when data is changed by that user and methods like insert or update are invoked, the entity properties `UserCriacao` or `UserEdicao` are set to the current `userId`. Note that if TLS was used here to store the user id, it would be accessible implicitly on every layer. It would not have to be passed as an argument at the instantiation of the Data Mapper neither on a bunch of other business logic methods.

5.1.5 Conclusions

This section discusses the advantages and disadvantages between continuing to use the ClassBuilder in Cachapuz or migrate to another ORM framework like EF or NHibernate. Additionally, further ClassBuilder enhancements are addressed in here.

The advantages of keeping ClassBuilder are:

- The developer team is accustomed to ClassBuilder and the relational model;
- ClassBuilder works well with legacy databases with mandatory relational schemas;

- The time and effort spent to build ClassBuilder will not have been in vain if it is not discarded;
- ClassBuilder is customizable and enhanceable;

The disadvantages of continuing to use ClassBuilder are:

- ClassBuilder does not respond to the dynamic querying of CazFramework lists feature;
- ClassBuilder is dependent on the database provider. It uses OLEDB provider API for SQL Server. Even though it only implements basic SQL, if other databases are to be experimented but still with the according OLEDB driver, it can have trouble with dates and auto-increments;
- It does not support associations and therefore makes navigation much difficult;
- ClassBuilder is another tool to be supported by Cachapuz;
- ClassBuilder is not N-Tier friendly especially because POCO entities are not entirely persistence ignorant for they implement audit logging and optimistic locking behaviour;
- ClassBuilder does not provide caching, cascading.

ClassBuilder future possible enhancements:

- Implement one-to-one, one-to-many and many-to-many (uni or bidirectional) associations. This implies migrating to the object model, which involves the management of two more dissimilar schemas than before: the relational schema and the object schema rather that the relational oriented object schema (Relational Domain Model);
- As a consequence of having navigable associations, Lazy Load (see 3.3.3) or Explicit Initialize can be implemented as well. Explicit Initialize can be defined by a level or depth of the association navigation within the graph, e.g. if the function `PersonConn.Select(PersonID = 1, NavigationDepth = 1)` is invoked, the `Person` object is loaded and all its associations are fetched eagerly at the same time, but any of the associated objects will not load any of its associations. Another approach is to have the method specify the name of the association to be eagerly fetched. These fetching strategies can be defined in separate metadata files and may provide more than one fetching profile;

- Database interoperability: provide to ClassBuilder the ability to use both SQL Server and MySQL at the same time. This can be implemented using an Abstract Factory. The impediment here is that ClassBuilder uses the following OLEDB classes: `OleDbConnection`, `OleDbCommand`, `OleDbParameter` and `OleDbDataReader`. ClassBuilder would have to replace this API with a custom made data provider;

- Have a query API that executes SQL and returns a *queryable* interface (e.g. implementing LINQ) that encapsulates a dynamic structure result. This helps on database interoperability and enables features like filtering, ordering and pagination, which are useful when the tables are not mapped to the domain model and also fits well in other components that operate with LINQ interfaces, like GUI data grids.

- Implement cascading on update and on delete. This can either be part of the associations feature or needs to have a manual configuration for defining the object dependency graph, (note that foreign key constraints are not used in Cachapuz databases);

- Have a Unit of Work with Identity Map and keeping track of property changes within an object so the update would not send all the fields back to the database.

The most relevant feature to implement is database interoperability due to the requirement of many CazFramework customers. The fourth possible enhancement item comes as a consequence of this. Implementing associations and other object oriented features would, in time, enhance ClassBuilder to behave more like EF or NHibernate. This implies a change of paradigm and with it the rise of a Dual-Schema problem which does not work well with legacy databases and dynamic lists defined in runtime.

5.2 .NET Data Providers

This section is an intro on ADO.NET data providers and gives some insight on strategies to build an abstract ADO.NET data provider with database interoperability.

While most applications use RDBMSs like SQL Server, MySQL or Oracle, others may use simpler solutions such as SQLite, **Excel** or text files and even other sorts of storage systems like Not only SQL (NoSQL) or OODBMSs. However, each kind of data source has to implement a set of interfaces specified by a technology or environment (like ADO.NET) in order to provide data to that same environment. In other words, a data provider is a middle-ware that links a technology (programming language or environment) to a data source. Also, each data source can be supported by multiple data providers.

ADO.NET comes with .NET framework as a set of tools to access file or server based data sources. ADO.NET libraries appear in `System.Data` *namespace* which contains classes providing communication with the data source.

One of the new features of ADO.NET is the support of the DataSet API, a disconnected database-agnostic tabular data container. DataSet and DataTable classes were covered in 3.1.2.

The other part of ADO.NET API consists of connected classes, which require an open connection available in order to work. The only relevant ADO.NET connected classes to CazFramework and ClassBuilder are (chapter 1 of [Mal05]):

- `Connection`: it opens a connection with the data source. Here, ADO.NET may recycle physical connections to create new instances of the `Connection` class;
- `Transaction`: it runs multiple commands as an atomic operation, managing isolation levels in the relational database;
- `Command`: represents an executable SQL statement for queries, inserts, updates, schema manipulation or any other command supported by the RDBMS in use;
- `Parameter`: multiple parameters can be used in a single command. This is useful for SQL statements that take different arguments (usually specified with '?'), including stored procedures, named queries or SQL queries;
- `DataReader`: light read-only iterator which keeps a cursor to the *fetchable* data, result of the SQL statement in the database.

Unlike the other ADO.NET classes, these connected classes have to be implemented differently per RDBMS. That implementation ships inside a .NET data provider.

Figure 5.6 (based on chapter 2 of [MH03]) presents a relationship between a group of .NET data providers, the data sources and the ADO.NET classes and interfaces. Note that the *namespace* is only displayed on those providers that directly implement the ADO.NET interfaces. Most of the providers in the figure are further analysed in section 5.4 so they can be effectively supported by CazDataProvider.

Thus, SQL Server specific classes appear under `System.Data.SqlClient` *namespace* in the same way as OLEDB classes do in `System.Data.OleDb` and Oracle classes in `System.Data.OracleClient`. Although there are other naming conventions like in MySQL which has specific implementation classes in `MySql.Data.MySqlClient` *namespace*.

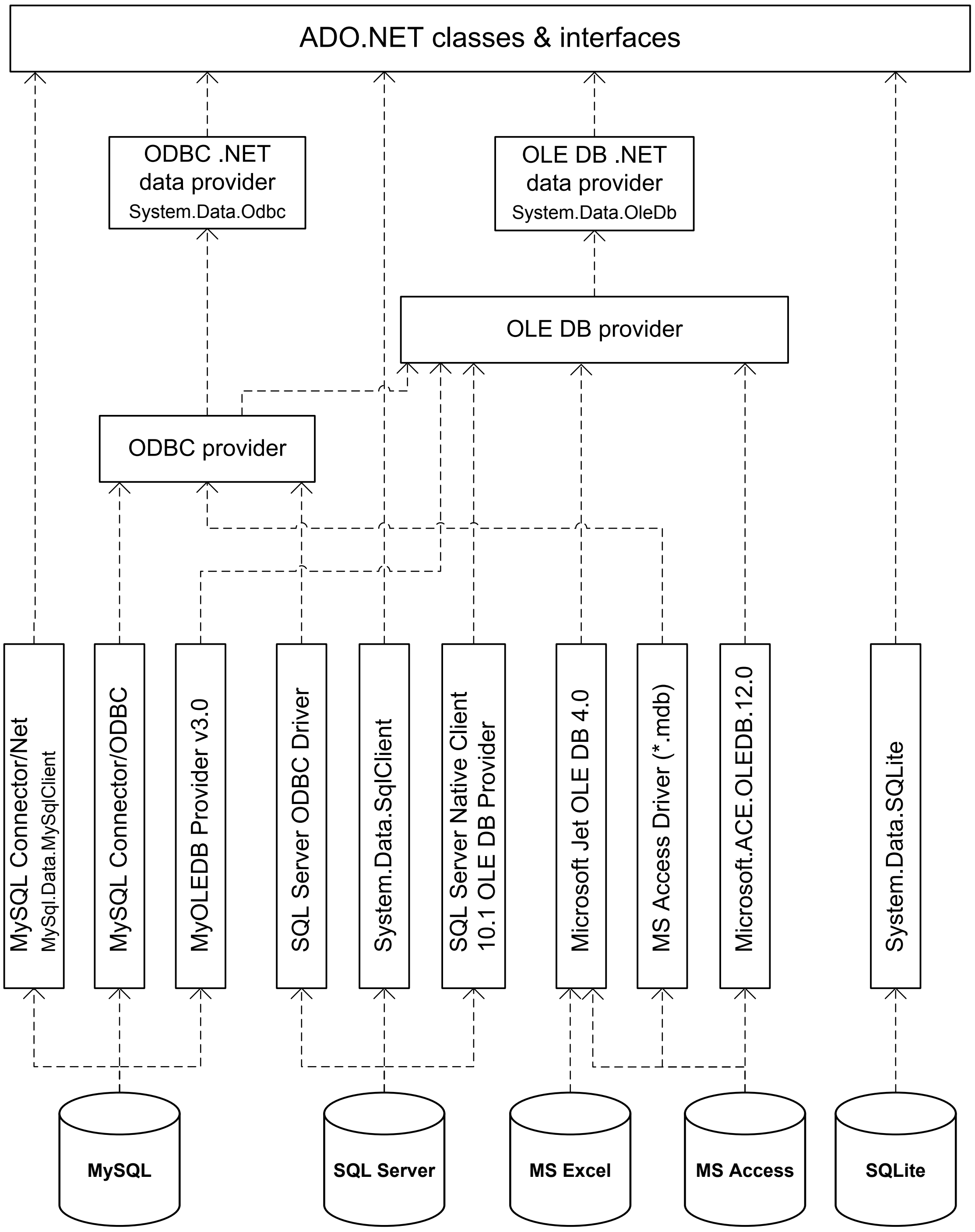


Figure 5.6: .NET data providers for a group of databases

Each .NET data provider contains implementations of the ADO.NET connected classes and interfaces. For example `SqlConnection`[1], `OleDbConnection`[2] and `MySqlConnection`[3] they all implement `IDbConnection`[4] interface.

Even though OLEDB .NET data provider is a generic provider, i.e. not database specific, and thus can be used for a wide variety of databases, it still requires a different OLEDB implementation for each database. In fact, OLEDB is just another abstraction layer similar to ADO.NET.

There are some advantages of using database specific providers like **SqlClient** or **MySql Connector/Net**[5] over generic data providers like OLEDB or Open Database Connectivity (ODBC). Specific providers are better equipped with database specific functionality and they provide better performance and support for specific data types. For example, **SqlClient** provider uses Tabular Data Stream (TDS) protocol to communicate directly with SQL Server, which delivers higher performance than using an OLEDB layer with **COM Interop** (.NET middleware to enable communication between Component Object Model (COM) objects and .NET objects) [MH03].

However, there are cases like CazFramework, that require database interoperability features. For that, in ADO.NET it is possible to have provider agnostic code with either of the following solutions:

- Use generic data providers like OLEDB or ODBC which implies sacrificing performance and specific database functionality. Additional problems may arise when using different SQL dialects, data types and OLEDB provider specific implementation idiosyncrasies (see the *Implementation* section in 5.4 for details);

- Develop a custom data provider can be a more flexible solution with improved performance and maintainability. This is possible by implementing the ADO.NET interfaces `IDbConnection`[4], `IDbTransaction`[6], `IDbCommand`[7], `IDbDataParameter`[8] and `IDataReader`[9] [MH03]. Although, database interoperability typically requires an abstraction layer to work on top of other data providers rather than have a custom data provider implemented from scratch. Later on this chapter, more insight is given on how to build such abstraction layer in order to support the core of CazDataProvider;

[1] http://msdn.microsoft.com/en-us/library/system.data.sqlclient.sqlconnection.aspx
[2] http://msdn.microsoft.com/en-us/library/system.data.oledb.oledbconnection.aspx
[3] http://dev.mysql.com/doc/refman/5.1/en/connector-net-ref-mysqlclient.html#connector-net-ref-mysqlclient-mysqlconnection
[4] http://msdn.microsoft.com/en-us/library/system.data.idbconnection.aspx
[5] http://dev.mysql.com/doc/refman/5.0/en/connector-net.html
[6] http://msdn.microsoft.com/en-us/library/system.data.idbtransaction.aspx
[7] http://msdn.microsoft.com/en-us/library/system.data.idbcommand.aspx
[8] http://msdn.microsoft.com/en-us/library/system.data.idbdataparameter.aspx
[9] http://msdn.microsoft.com/en-us/library/system.data.idatareader.aspx

- Implement a provider factory by taking advantage of the Strategy pattern already implemented in ADO.NET classes, e.g. the `MySqlConnection` and `SqlConnection` are both concrete strategies of `IDbConnection`). Then choose one of the following:

  - Create a static Factory Method to instantiate the appropriate connected objects (e.g. `MySqlConnection` or `SqlConnection`) from a given connection string or other domain logic parameter. For instance, the application may have a configurations database containing a variable set of database connection strings and then, the business logic decides which user is granted the access to which database by opening the according connection implicitly (Abstract Factory). To implicitly manage accessibility control the user validation can use TLS;
  - ADO.NET 2.0 already provides a factory class to achieve a similar goal but having the connection strings defined in the default application XML configurations file (e.g. `App.Config` or `Web.Config`) rather than in a database. This is possible by using the `System.Data.Common.DbProviderFactories` class with the static method `GetFactory(providerInvariantName : String) : DbProviderFactory`. The argument `providerInvariantName` is the name defined in the XML configurations file, for the element containing all the settings of a provider connection (assemblies, connection string, etc). That method returns a `DbProviderFactory` (abstract class) instance which encapsulates a provider specific factory, e.g. an `SqlClientFactory` or a `MySqlProviderFactory`. All the ADO.NET specific providers do implement `DbProviderFactory`. This class returns methods to get `Connection` and `Command` instances, similar to the `IDbConnection` API. More details on this approach can be found in chapter 1 of [Mal05].

  Although, a provider factory is simple to develop and requires very few code, it stumbles across the same problems of database interoperability as those of generic providers like OLEDB or ODBC;

- Another approach is using the ADO.NET *Provider Model* design pattern defined and implemented by Microsoft for ADO.NET 2.0 [How04]. Essentially, it implements Factory Method and Strategy patterns to enable the development of a DAL component pluggable to the application in the XML configurations file in the same way as the ADO.NET providers are typically configured in `App.Config`. Also, this DAL component encapsulates all the data access logic including multiple data providers, e.g. **MySql Connector/Net** or **SqlClient**. Primarily, the main data provider class shall inherit from `System.Configuration.Provider.ProviderBase` abstract class and then implement `Initialize` method from its superclass as well as a group of CRUD methods to expose to the client similar to a Table Data Gateway. Although, those methods shall manage generic database interoperability rather than implement any business logic, and for that reason they should provide similar API to the `IDbConnection` and `IDbCommand` interfaces. Note that each database will have a separate data provider class managing SQL dialects and ADO.NET data provider particularities. Additionally, it is also required to have a provider manager class with a static (thread-safe) collection of providers for the client data accesses. Although this is a more flexible approach than the others, it only defines a Strategy and Factory Method architecture of the solution. All the development effort is under the implementation code. Also, it is convenient to keep the code as platform independent as possible (PIM), and for that reason this approach may not be the finest for the requirements of CazFramework. More implementation details on .NET *Provider Model* can be found in the article *How to write a Provider Model* [Nay08].

5.3 Designing an Architecture

This section presents the designing process of CazDataProvider proposing 6 different solutions for the same problem and in the end the chosen one to be implemented.

From sections 5.1 and 5.2 it is set that ClassBuilder needs to generate DAL code that accesses ADO.NET data providers indirectly through CazDataProvider classes. So the classes to replace are:

- `OleDbConnection`;
- `OleDbCommand`;
- `OleDbParameter`;
- `OleDbDataReader`;
- `OleDbTransaction`.

Having analysed different data providers of different RDBMSs, in 5.2, it is safe to state that all the specific provider ADO.NET classes (e.g. `OleDbConnection`, `MySqlConnection`, etc.) implement the same interfaces (e.g. `IDbConnection`). Therefore, all the above OLEDB classes set to be replaced in ClassBuilder, do implement the following interfaces accordingly:

- `IDbConnection`;
- `IDbCommand`;
- `IDataParameter`;
- `IDataReader`;
- `IDbTransaction`.

It is also important to analyse how the above interfaces work and, for that, Figure 5.7 provides an overall view on the associations and *creational* dependencies (in the methods that return an instance of the other interface) they hold together.

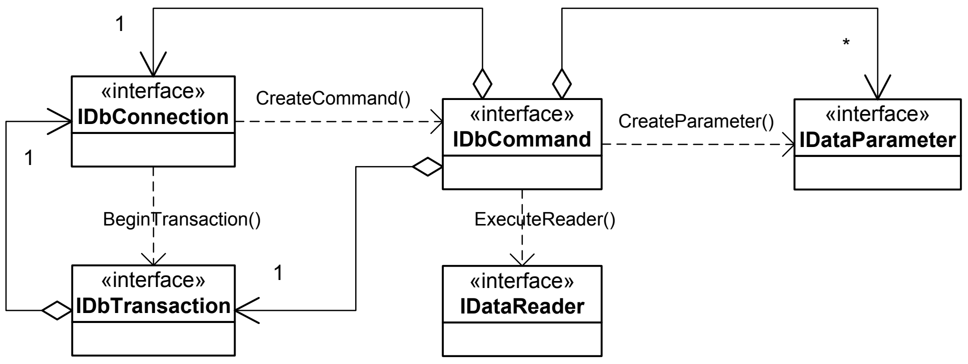


Figure 5.7: `IDb*` and `IData*` interfaces, the associations and *creational* dependencies they hold together

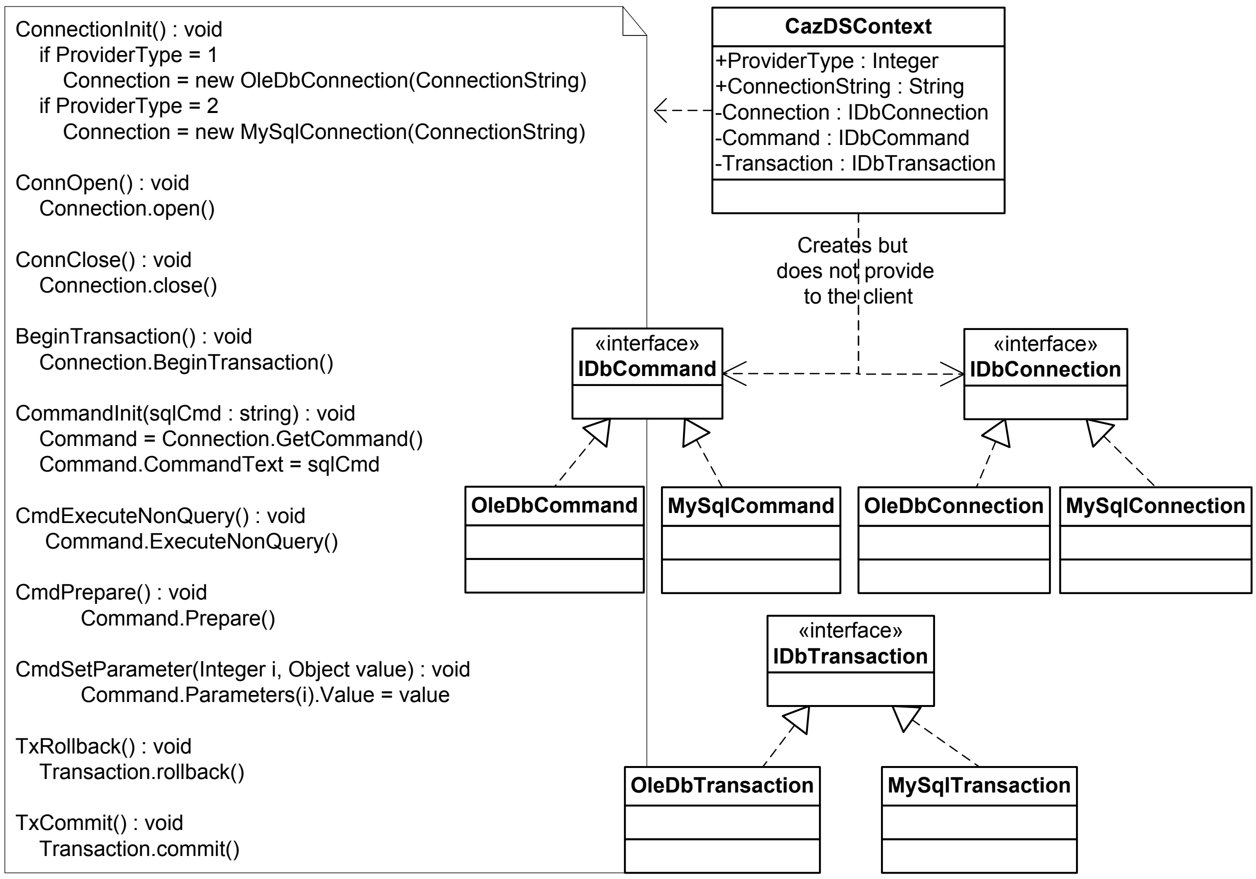
As it was already stated in section 5.2, these ADO.NET classes implement the Strategy pattern in which, for instance, the `IDbConnection` is a *strategy* and the `MySqlConnection` is the *concrete strategy*. This and other GoF patterns prove to be essential in all the 6 different solutions of this section.

5.3.1 Solution 1: Data Context Facade and Factory

This solution was designed to encapsulate the `IDb*` interfaces similarly to the approach defined by Joydip Kanjilal[1]. The diagram representation of this solution is given by Figure 5.8 which implements Facade, factory and delegation patterns.

One particular feature of this approach is that `CazDSContext` encapsulates all the methods of ADO.NET, which in turn, also enables it to encapsulate other APIs that have nothing to do with relational databases. For instance, if the customer wants the application to support plain text file persistence via file stream rather than ODBC API, it would

[1]`http://aspalliance.com/837_Implementing_a_Data_Access_Layer_in_C`



```
ConnectionInit() : void
    if ProviderType = 1
        Connection = new OleDbConnection(ConnectionString)
    if ProviderType = 2
        Connection = new MySqlConnection(ConnectionString)

ConnOpen() : void
    Connection.open()

ConnClose() : void
    Connection.close()

BeginTransaction() : void
    Connection.BeginTransaction()

CommandInit(sqlCmd : string) : void
    Command = Connection.GetCommand()
    Command.CommandText = sqlCmd

CmdExecuteNonQuery() : void
     Command.ExecuteNonQuery()

CmdPrepare() : void
        Command.Prepare()

CmdSetParameter(Integer i, Object value) : void
        Command.Parameters(i).Value = value

TxRollback() : void
    Transaction.rollback()

TxCommit() : void
    Transaction.commit()
```

CazDSContext
+ProviderType : Integer
+ConnectionString : String
-Connection : IDbConnection
-Command : IDbCommand
-Transaction : IDbTransaction
Creates but does not provide to the client
«interface» IDbCommand
«interface» IDbConnection
OleDbCommand
MySqlCommand
OleDbConnection
MySqlConnection
«interface» IDbTransaction
OleDbTransaction
MySqlTransaction

Figure 5.8: Solution 1: Data Context Facade and factory

require `CazDSContext.ConnOpen()` to instantiate a `StreamReader` and a `StreamWriter` (or an encapsulation of both with Lazy Initialization or Virtual Proxy). It is likely that a new `TextFileProvider` class would have to be implemented and would not support transaction management. Also, because this solution is relational-based, SQL queries could have to be parsed and translated in order to fetch tabular data on text files.

Note that in Figure 5.8, `CazDSContext` only implements a Factory Method for `IDbConnection`. The instantiation of `IDbCommand` and `IDbTransaction` is done via the `IDbConnection` calls (see more details in Figure 5.7). Thus, it is impossible to have one `IDbCommand` to work with different `IDbConnections` as it is impossible to have an `OleDbCommand` being executed through an `SqlConnection`.

In here, because `CazDSContext` encapsulates `IDb*` interfaces, the application layers above, are protected from changes that can occur on those interfaces. Nevertheless, excessive unreasonable use of encapsulation can bring a rather tedious code. In the context of Cachapuz, it is desired that CazFramework works with multiple RDBMSs and do not ponder on changing that technology any time soon, especially due to legacy databases.

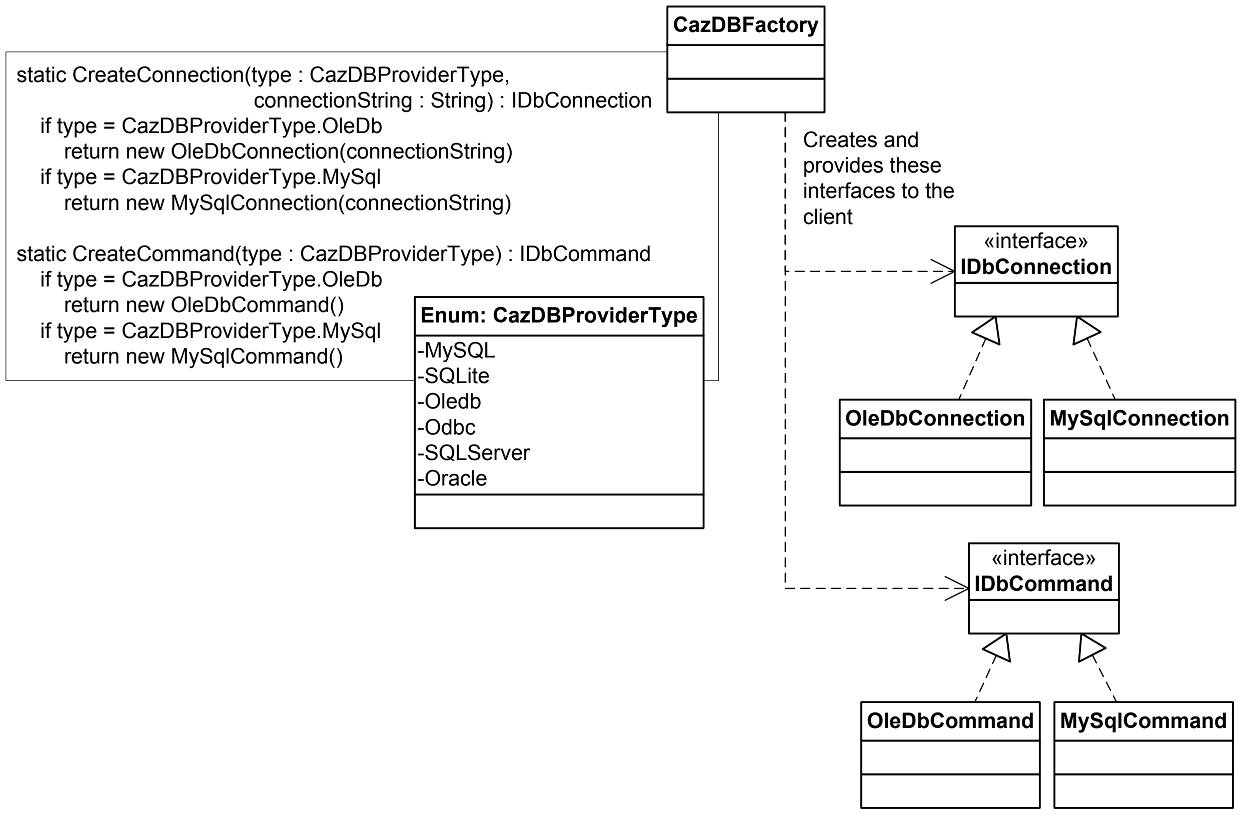
Also, it is not a burden to rely on a subset of ADO.NET interfaces.

One advantage of this approach is its density, for it concentrates all the data providers encapsulation and data access API in one single class. Any RDBMS SQL dialect or specific provider differences can all be managed within `CazDSContext`. However, as the number of supported providers increase, the code becomes harder to maintain.

5.3.2 Solution 2: Provider Factory

Like *Solution 1*, this solution was designed to encapsulate the `IDb*` interfaces as well. Although, in here there is a closer similarly to the provider factory approach supported by ADO.NET 2.0 and presented at the end of section 5.2.

This solution implements a Factory Method and enables ADO.NET provider encapsulation using a very simple architecture. Figure 5.9 presents a provider factory implementation.



```
static CreateConnection(type : CazDBProviderType,
                                  connectionString : String) : IDbConnection
  if type = CazDBProviderType.OleDb
    return new OleDbConnection(connectionString)
  if type = CazDBProviderType.MySql
    return new MySqlConnection(connectionString)

static CreateCommand(type : CazDBProviderType) : IDbCommand
  if type = CazDBProviderType.OleDb
    return new OleDbCommand()
  if type = CazDBProviderType.MySql
    return new MySqlCommand()
```

Figure 5.9: Solution 2: Provider Factory

Note that the class `CazDBFactory` provides static Factory Methods and therefore, the Factory Method `CazDBFactory.CreateConnection()` is accessible in any part of the application code with no need for argument propagation. Although, from Figure 5.7, the implementation of `CazDBFactory.CreateCommand()` may seem irrelevant, it provides an alternative for instantiating an `IDbCommand` before passing an `IDbConnection` to it. Note that both must encapsulate the same specific provider.

The key to this provider factory is its simplicity of implementation for it only requires the `OleDb*` classes to be *refactored* into `IDb*` and `IData*` interfaces in the ClassBuilder generated code.

Even though very simple SQL queries may be transversal to different RDBMSs, two different ADO.NET data providers may not even work in the same way with the same RDBMS. For example, whereas the OLEDB providers use '?' parameters in the SQL statements, the **SqlClient** provider only supports named parameters such as '@name'. The current **MySql Connector/Net** only supports named parameters (specified as '?name'), although the new version 6.6.3 is planned to feature unnamed parameters as well. Prior MySQL versions used the '@name' form, however that changed due to conflicts with user variables.

Therefore, the provider factory in this solution, is unable to provide database interoperability.

5.3.3 Solution 3: Abstract Provider Factory

This solution is similar to the *Solution 2* (in 5.3.2) except here it implements an Abstract Factory with TLS to make the current user and its connection settings accessible anywhere and whenever a new connection has to be opened. See Figure 5.10.

That is due to the fact that there are requirements such as those of CazFramework, in which each user might have to manage a different database, depending on its role or relying on the location and business of that application solution.

Therefore, whilst the *Solution 2* had the `connectionString` propagated as arguments in the method calls, in here, the Abstract Factory avoids that a user specific `connection-String` is passed from the business logic calls to the DAL and CazDataProvider. This brings a more implicit connection management.

However, such solution breaks the N-Tier model as the data tier becomes dependent on very specific logic within the business tier. That same logic is responsible for defining how the users are associated with the connection settings and how they access the databases.

Another problem with this strategy is the impossibility to provide and extend database interoperability, like the *Solution 2*.

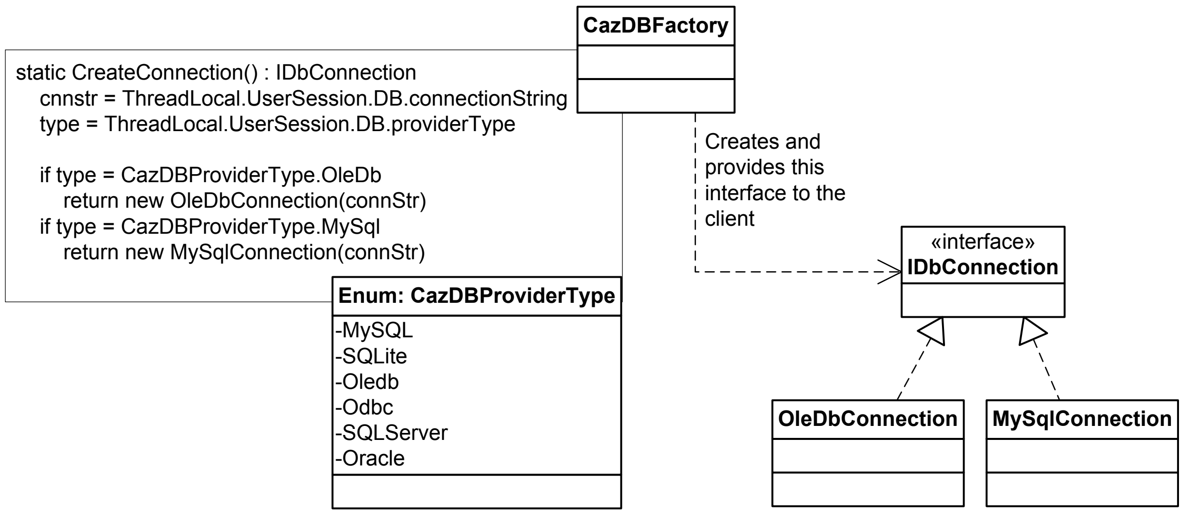

CazDBFactory
static CreateConnection() : IDbConnection
cnnstr = ThreadLocal.UserSession.DB.connectionString
type = ThreadLocal.UserSession.DB.providerType
if type = CazDBProviderType.OleDb
return new OleDbConnection(connStr)
if type = CazDBProviderType.MySql
return new MySqlConnection(connStr)
Enum: CazDBProviderType
-MySQL
-SQLite
-Oledb
-Odbc
-SQLServer
-Oracle
Creates and provides this interface to the client
«interface» IDbConnection
OleDbConnection
MySqlConnection

Figure 5.10: Solution 3: Abstract Provider Factory

5.3.4 Solution 4: Provider Factory with Subclassing

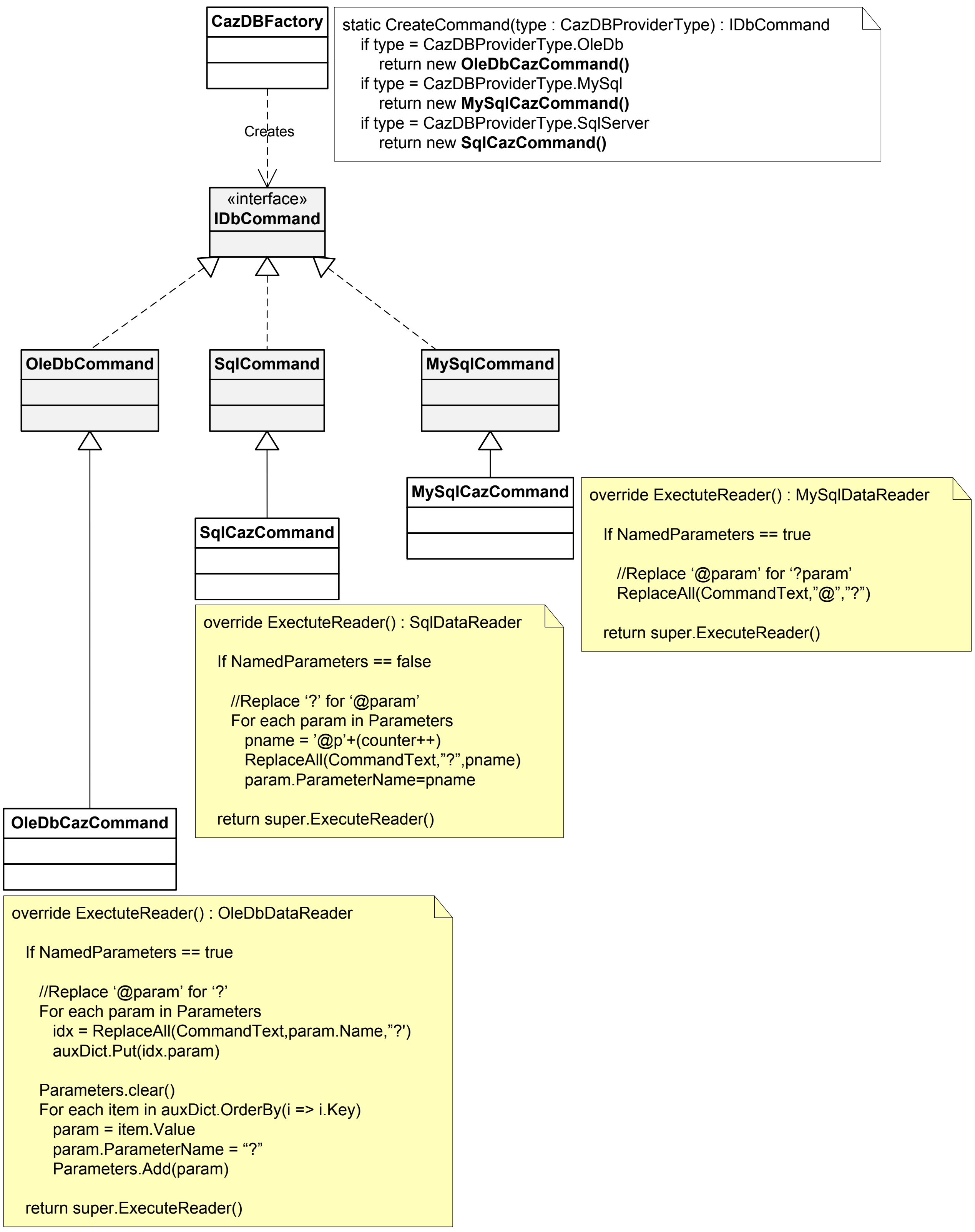
From the last approaches, only the *Solution 1* is able to deal with SQL dialects and provider differences and even so it does not support a very extensible model for database interoperability.

Therefore, this solution presents a code refactoring approach, taking advantage of the ADO.NET Strategy organization (already discussed at the beginning of this section) and implementing a Factory Method to instantiate subclasses of the specific providers.

Thus, subclassing is needed to reimplement API, or simply making some conversions (types and parameters) and translations (SQL). The problem of parameters has been discussed in 5.3.2 to involve at least the OLEDB, **MySql Connector/Net** and **SqlClient** data providers.

It has been a common practice, in CazFramework, to use unnamed parameters in legacy code especially that which is generated by ClassBuilder, due to the use of OLEDB providers. Although named parameters may provide better readability of code. Therefore, CazDataProvider may support: unnamed parameters, named parameters or both.

Figure 5.11 presents the classes involved in the implementation of a provider factory. The Factory Method is similar to the *Solution 2*. All the specific provider `Commands` that need refactoring, are extended. For instance, because `SqlCommand` only supports named parameters, its parameters and the SQL statement must replace the symbol '?' with something like '@name', before executing a query and after having defined all the parameters. A good place to inject that code is overriding `ExecuteReader` and in the end, call the superclass `ExecuteReader`. Note that `ExecuteNonQuery` would need this behaviour as well.



```
static CreateCommand(type : CazDBProviderType) : IDbCommand
    if type = CazDBProviderType.OleDb
        return new OleDbCazCommand()
    if type = CazDBProviderType.MySql
        return new MySqlCazCommand()
    if type = CazDBProviderType.SqlServer
        return new SqlCazCommand()
```

```
override ExectuteReader() : MySqlDataReader

    If NamedParameters == true

        //Replace '@param' for '?param'
        ReplaceAll(CommandText,"@","?")

    return super.ExecuteReader()
```

```
override ExectuteReader() : SqlDataReader

    If NamedParameters == false

        //Replace '?' for '@param'
        For each param in Parameters
            pname = '@p'+(counter++)
            ReplaceAll(CommandText,"?",pname)
            param.ParameterName=pname

    return super.ExecuteReader()
```

```
override ExectuteReader() : OleDbDataReader

    If NamedParameters == true

        //Replace '@param' for '?'
        For each param in Parameters
            idx = ReplaceAll(CommandText,param.Name,"?')
            auxDict.Put(idx.param)

        Parameters.clear()
        For each item in auxDict.OrderBy(i => i.Key)
            param = item.Value
            param.ParameterName = "?"
            Parameters.Add(param)

    return super.ExecuteReader()
```

Figure 5.11: Solution 4: Provider Factory with Subclassing

In order to solve the parameters issue, refactoring `SqlCommand` (via subclassing) would be enough for CazDataProvider along with the establishment of unnamed parameter policy (using '?'). However, the model in Figure 5.11 goes further. It defines a flag `NamedParameters` to manage the support of named parameters. This flag can either be set manually in the factory instantiation or controlled automatically by parsing the SQL statement looking for the appropriate symbols.

Thus, only `OleDbCommand` has to implement named parameters from scratch (see `OleD-bCazCommand` in 5.11). Also, considering the default policy of named parameters is set to use '@name' (`SqlCommand` form) rather than '?name' (`MySqlCommand` form), `MySqlCaz-Command` has to make a character translation, but at the same time prevent conflicts with MySQL user variables.

One important fact omitted by Figure 5.11 is the absence of a Factory Method for instantiating an `IDbConnection`. If `CazDBFactory` were to factory an `IDbConnection`, it would provide the untouched specific provider API implementation entirely, which includes the method `IDbConnection.GetCommand`. Because `GetCommand` does not factory `*CazCommands`, this would leave a trap for other developers to fall upon.

Therefore, the following suitable solutions can be:

- Using the delegation of *Solution 1* for encapsulating `IDbConnection` and possibly `IDbTransaction` as well. The API is more controllable here;
- Adopting subclassing for `IDbConnections` in the same way as with `IDbCommands` and then reimplement `GetConnection` to factory `*CazCommands`. Here, subclassing would imply an increasing number of useless classes with very little implementation.

Although using subclassing seems a good solution at first, it brings some disadvantages as well. Code refactoring is not easy to implement and although sometimes it makes sense to use generalization for inheriting functionality like that of `SqlCommand`, this typically falls within the *Call Super* or *BaseBean* anti-patterns, hence a sign of bad application design.

BaseBean defines a solution in which some class is extended purely for its functionality and as a consequence of that, *Call Super* occurs when the subclass overrides some superclass method and must mandatorily call the overridden method inside the new implementation.

Because `SqlCommand` was not designed to be extended, the responsibility falls entirely onto CazDataProvider to keep its refactored `*CazCommands` working without conflicts. Also, because `SqlCommand` is out of control, it may become sealed or change its internal API without warning, although it is likely to keep stable for the next couple of years.

Conceptually, `SqlCazCommand` is not really part of an ADO.NET data provider but rather of an abstraction for unified relational database access. For that, it ought to avoid inheriting from any `IDbCommand` implementation, including `SqlCommand`.

Delegation, on the other hand, provides a better control over the `SqlCommand`. It allows disabling some unnecessary or unmanaged API and provides a more flexible model, allowing the `*CazCommand` classes to share some implementation, for instance through Template Method. In other words, supposing that the model in Figure 5.11, was to add support for Oracle ADO.NET data provider and because like `SqlCommand`, `OracleCommand` does not implement unnamed parameters as well, both `SqlCazCommand` and `OracleCazCommand` would be able to share the same implementation.

One advantage of this solution, though, is its simplicity and small amount of code required in the short-term. *Solution 5* (in 5.3.5) describes an alternative to this inheritance tight coupling problem with the Template Method suggested by Fowler.

5.3.5 Solution 5: Provider Factory with Template Method

The need to improve *Solution 4*, due to its design problems, led to an analysis on both the Decorator and the Template Method design patterns

Ideally, *Solution 4* could work by simply injecting some extension points in `SqlCommand`. For that, `SqlCommand` would have to provide extension points as demonstrated in Figure 5.12 which is an implementation of Template Method pattern.

Note that with Template Method, the inheritance relationship inverts its responsibilities, i.e. it is the superclass that is dependent on the subclass through the Template Methods.

In Figure 5.12, `SqlCommand` could be abstract so the client would be forced to work with `SqlCazCommand`. Although this way, the client can either instantiate an `SqlCommand` or a `SqlCazCommand` from the according Factory Method.

There are a few advantages in this model towards the *Call Super*. For instance, if the `SqlCommand` writer wants to add new behaviour to the `ExecuteReader` function (or even change its name), he is free to do so, without influencing the `SqlCazCommand` as long as the extension points are not erased.

However, whoever wrote `SqlCommand` did not want to take responsibility for any deriving classes, which makes this Template Method approach hypothetical.

Therefore a new approach had to be thought of and so the Decorator pattern was modelled according to Figure 5.13. Here, the *Component* interface is `IDbCommand`, the *ConcreteComponent* is `SqlCommand`, the *Decorator* is `CazCommand` and the *ConcreteDec-*

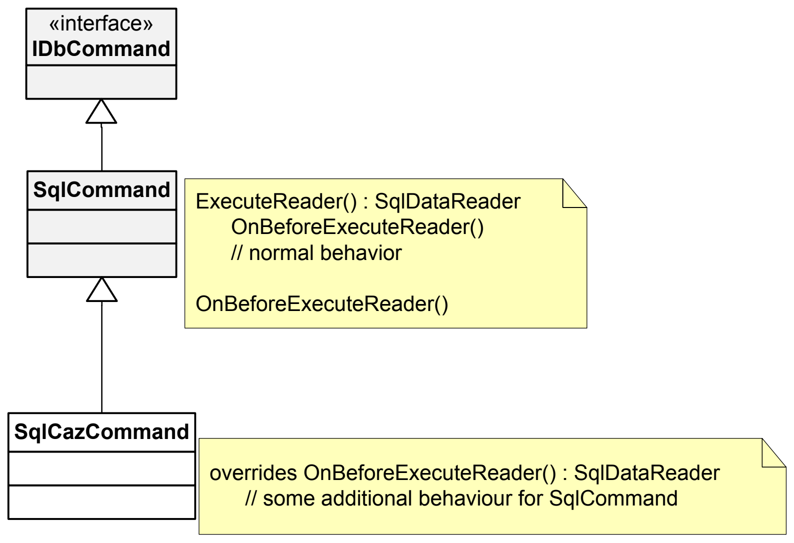

«interface»
IDbCommand
SqlCommand
ExecuteReader() : SqlDataReader
OnBeforeExecuteReader()
// normal behavior
OnBeforeExecuteReader()
SqlCazCommand
overrides OnBeforeExecuteReader() : SqlDataReader
// some additional behaviour for SqlCommand

Figure 5.12: Ideal `SqlCommand` refactoring with Template Method

orator is `SqlCazCommand`.

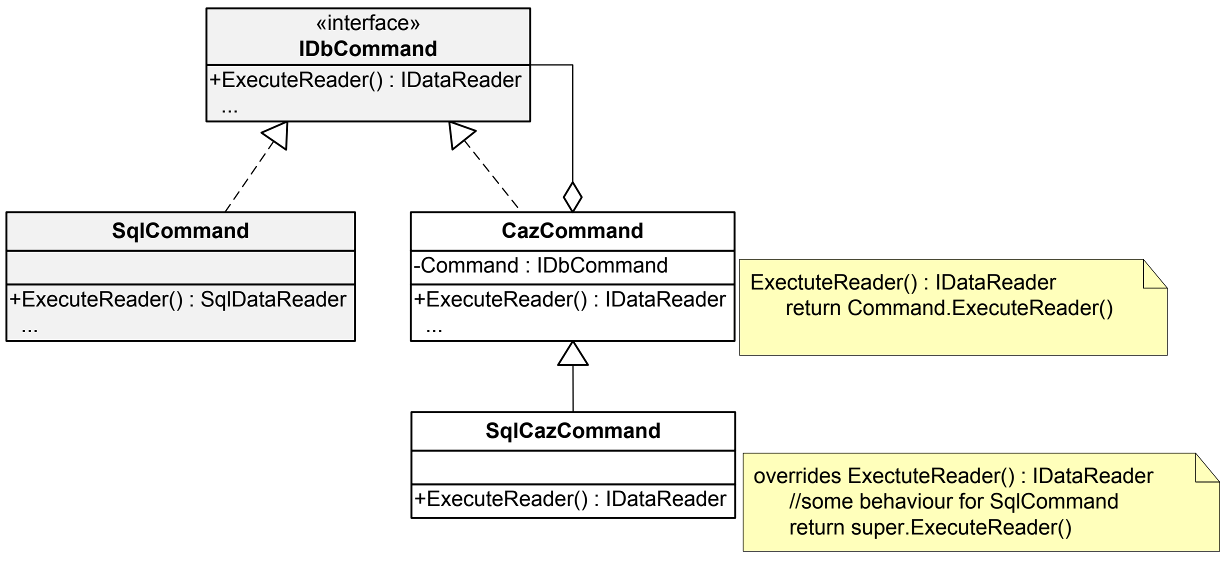

«interface»
IDbCommand
+ExecuteReader() : IDataReader
...
SqlCommand
+ExecuteReader() : SqlDataReader
...
CazCommand
-Command : IDbCommand
+ExecuteReader() : IDataReader
...
ExecuteReader() : IDataReader
return Command.ExecuteReader()
SqlCazCommand
+ExecuteReader() : IDataReader
overrides ExecuteReader() : IDataReader
//some behaviour for SqlCommand
return super.ExecuteReader()

Figure 5.13: `SqlCommand` refactoring with Decorator

Note that once again, the `SqlCazCommand` (*ConcreteDecorator*) implements *Call Super*. Meanwhile this time , the superclass is more controllable, plus it may be further refactored to contain common behaviour among the different *ConcreteDecorators*.

If, however, this Decorator implementation uses Template Method rather than *Call Super* (see Figure 5.14), this becomes cleaner and better modelled. That is, the `Caz-Command` becomes abstract, which is safer for preventing unexpected instantiation, and the `SqlCazCommand` avoids tight coupling to its superclass API. Also, there is slightly less code and responsibility but less control as well for the *ConcreteDecorator* to manage.

Another aspect of Decorator (of Figure 5.14) that can be improved for the case study of CazDataProvider is prevent the `CazCommand` of implementing all the `IDbCommand` methods, including unnecessary and unmanaged behaviour. Also, with this, a `CazCommand` will be able to strictly encapsulate `IDbCommands` such as `SqlCommand` or `MySqlCommand` rather than freely encapsulate `CazCommands` and any of its subclasses as well.

The above aspects lead to the actual *Solution 5* demonstrated in Figure 5.14. It provides a defensive model expecting more data provider interoperability problems and therefore is set for refactoring in long-term. These problems are likely to occur especially because the `IDb*` specific classes are encapsulated and managed according to the black box principle[1] and often with a closed implementation.

Hence, *Solution 5* borrows some Decorator aspects, uses Template Method and requires Factory Methods as well.

Figure 5.14 provides two additional aspects. First, the abstract `CazCommand` implements not only some of the `IDbCommand` API and extension points (Template Methods) but also contains additional methods. For instance, `AddParameter` simplifies the call and also may standardize the way of doing it, e.g. set the parameter name to '?' if unnamed parameters is the policy to be used;

The other aspect is part of the problem on how to manage SQL dialect differences. In Figure 5.14, the considered default policy for writing queries is SQL Server dialect. The tested example consists on how to manage the `Top N` SQL Server specific SQL command when using other RDBMSs and data providers. Regarding this problem, the three `*CazCommands` use the following implementation:

- `SqlCazCommand` - because SQL Server uses the default syntax `Top N` and that policy is not changeable, there is no implementation in here;
- `MySqlCazCommand` - MySQL uses `Limit N` syntax and therefore must replace all occurrences of `Top N` in the `CommandText` with the former;
- `OracleCazCommand` - the equivalent to `Top N` in oracle is `rownum < N` and for that it must replace the latter in the `CommandText`.

The solution for this SQL dialect problem is once again to have an extension point in the abstract class so each subclass can inject different conversions right before the SQL query is executed.

[1] http://it.toolbox.com/blogs/enterprise-solutions/design-principles-black-boxes-16028

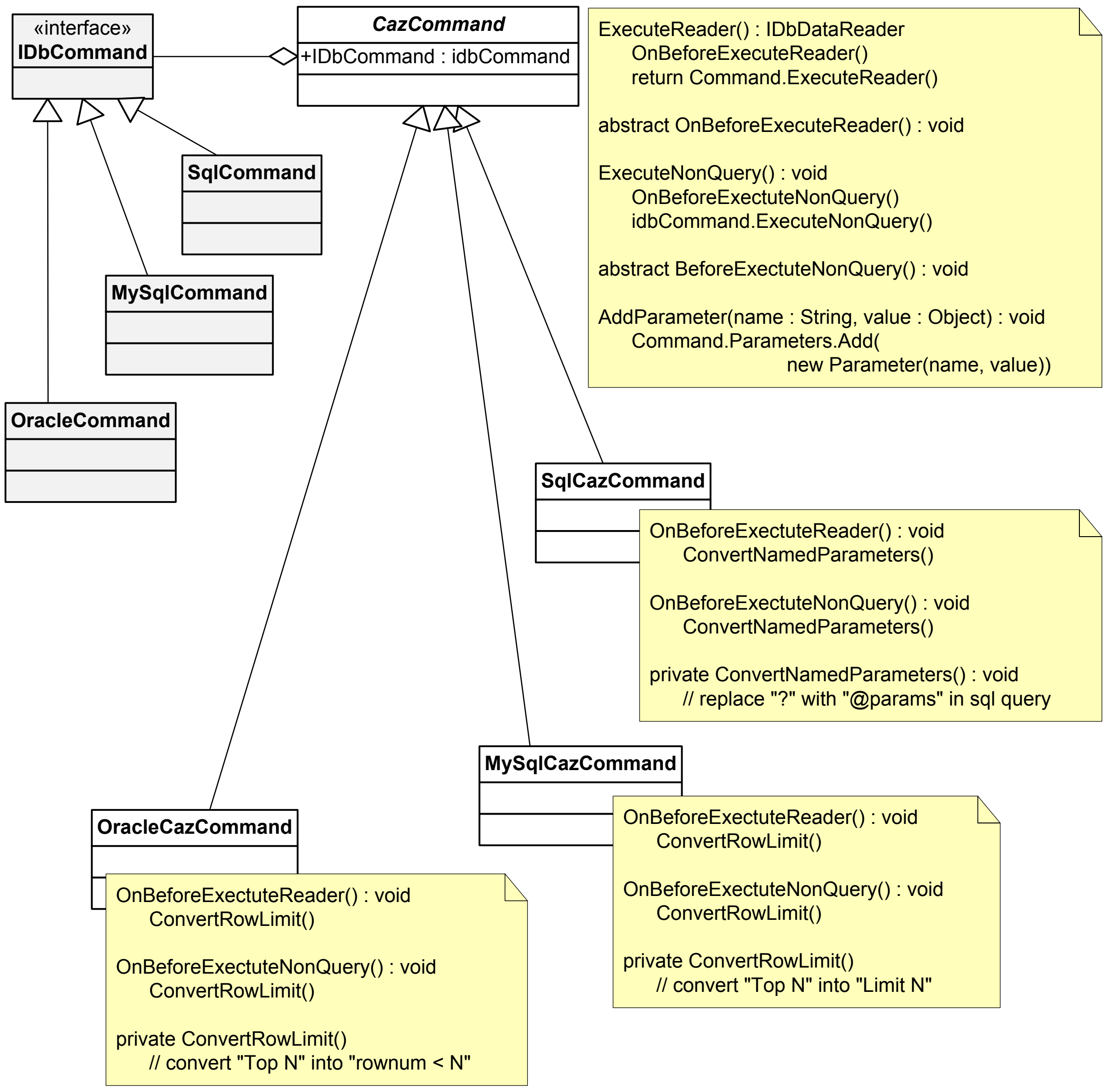
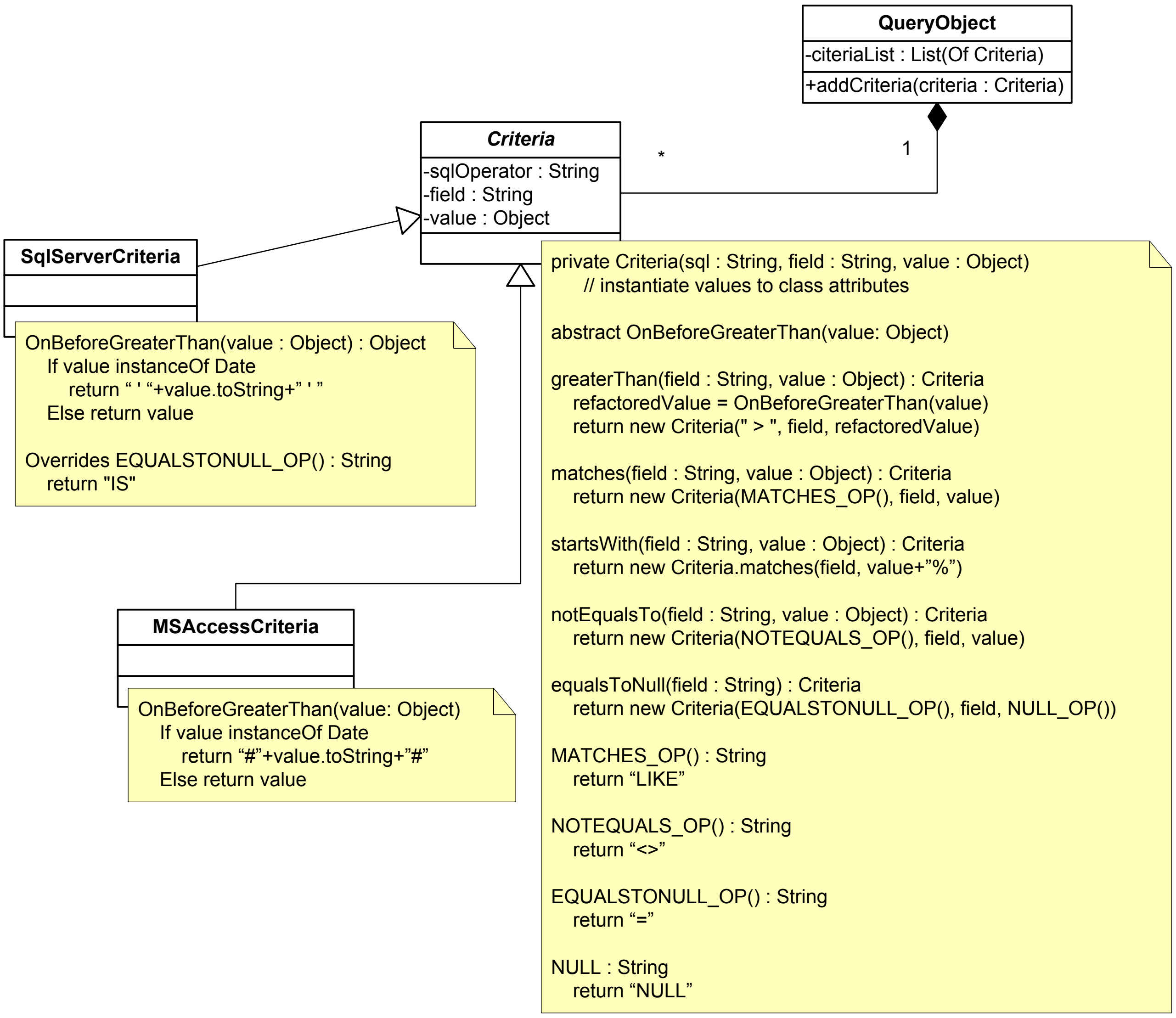

«interface»
IDbCommand
CazCommand
+IDbCommand : idbCommand
SqlCommand
MySqlCommand
OracleCommand
SqlCazCommand
MySqlCazCommand
OracleCazCommand
ExecuteReader() : IDbDataReader
OnBeforeExecuteReader()
return Command.ExecuteReader()
abstract OnBeforeExecuteReader() : void
ExecuteNonQuery() : void
OnBeforeExectuteNonQuery()
idbCommand.ExecuteNonQuery()
abstract BeforeExectuteNonQuery() : void
AddParameter(name : String, value : Object) : void
Command.Parameters.Add(
new Parameter(name, value))
OnBeforeExectuteReader() : void
ConvertNamedParameters()
OnBeforeExectuteNonQuery() : void
ConvertNamedParameters()
private ConvertNamedParameters() : void
// replace "?" with "@params" in sql query
OnBeforeExectuteReader() : void
ConvertRowLimit()
OnBeforeExectuteNonQuery() : void
ConvertRowLimit()
private ConvertRowLimit()
// convert "Top N" into "Limit N"
OnBeforeExectuteReader() : void
ConvertRowLimit()
OnBeforeExectuteNonQuery() : void
ConvertRowLimit()
private ConvertRowLimit()
// convert "Top N" into "rownum < N"

Figure 5.14: Solution 5: Provider Factory with Template Method

However, this approach adds to the responsibility of abstracting the data provider, the responsibility of managing abstract unified SQL in the `*CazCommand` which again breaks the single responsibility principle. Also, for instance the MySQL OLEDB provider will require the same implementation methods of `MySqlCazCommand` that manage the SQL dialect conversions.

A good approach to solve query parsing and conversion is using Query Object pattern. This is debated in the in *Solution 6* (5.3.6).



```
private Criteria(sql : String, field : String, value : Object)
    // instantiate values to class attributes

abstract OnBeforeGreaterThan(value: Object)

greaterThan(field : String, value : Object) : Criteria
    refactoredValue = OnBeforeGreaterThan(value)
    return new Criteria(" > ", field, refactoredValue)

matches(field : String, value : Object) : Criteria
    return new Criteria(MATCHES_OP(), field, value)

startsWith(field : String, value : Object) : Criteria
    return new Criteria.matches(field, value+"%")

notEqualsTo(field : String, value : Object) : Criteria
    return new Criteria(NOTEQUALS_OP(), field, value)

equalsToNull(field : String) : Criteria
    return new Criteria(EQUALSTONULL_OP(), field, NULL_OP())

MATCHES_OP() : String
    return "LIKE"

NOTEQUALS_OP() : String
    return "<>"

EQUALSTONULL_OP() : String
    return "="

NULL : String
    return "NULL"
```

```
OnBeforeGreaterThan(value : Object) : Object
    If value instanceOf Date
        return " ' "+value.toString+" ' "
    Else return value

Overrides EQUALSTONULL_OP() : String
    return "IS"
```

```
OnBeforeGreaterThan(value: Object)
    If value instanceOf Date
        return "#"+value.toString+"#"
    Else return value
```

Figure 5.15: Solution 6: Example of a Query Object implementation

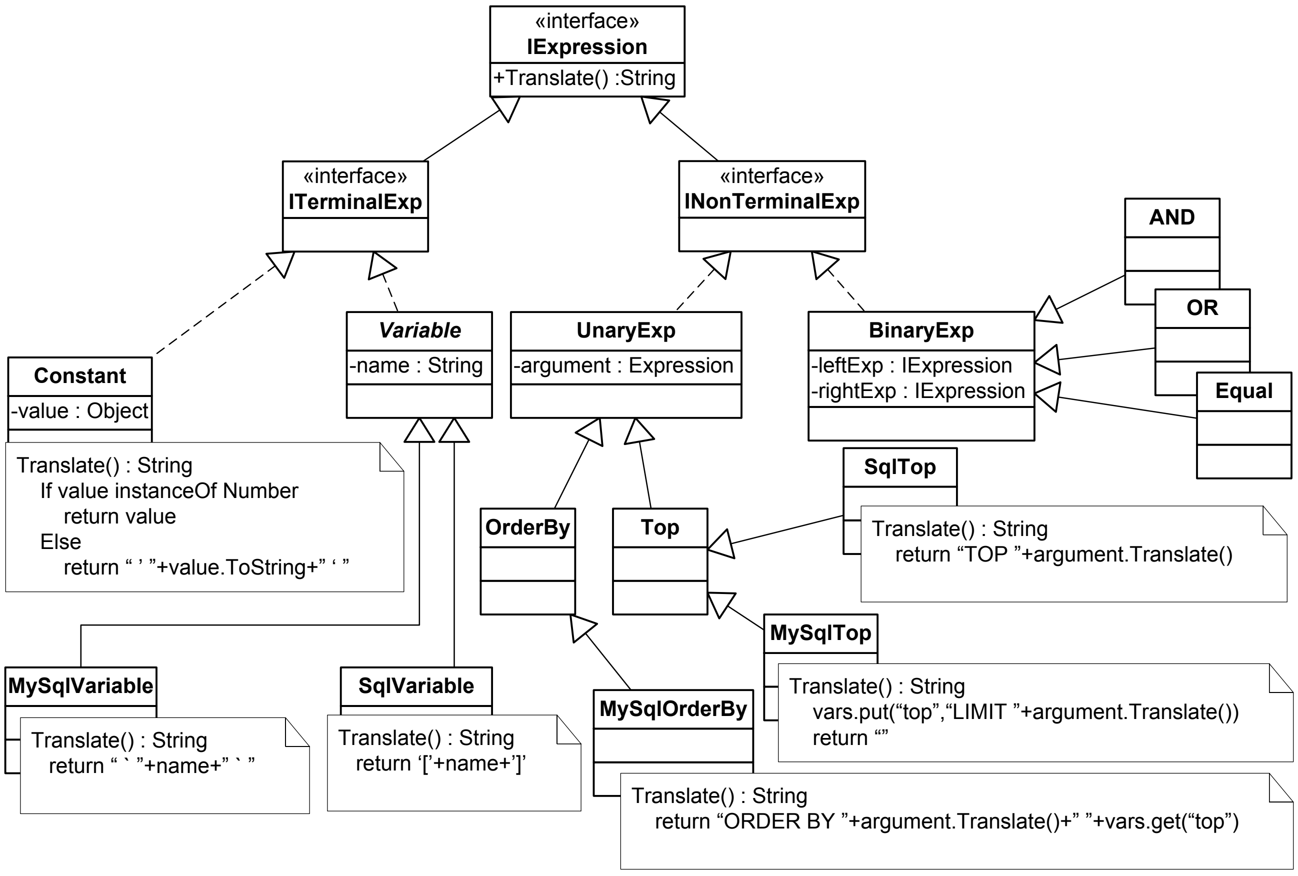
5.3.6 Solution 6: Query Object

There are some strategies often used for interpreting and parsing, supporting from simple expressions up to complex languages. Such strategies include, for instance, parser generator tools, implementation of Interpreter or Query Object patterns. The purpose of having an SQL Interpreter in CazDataProvider is to achieve a unified way of writing queries that are abstracted from the RDBMSs SQL dialects.

Because Cachapuz and CazFramework has always worked with no database other than SQL Server, it is agreed that the SQL Server dialect were to be the standard for writing SQL queries in CazDataProvider.

As already explained in section 3.4.2, full featured ORM frameworks like NHibernate implement this SQL dialect abstraction with Query-By-Example (QBE), QBA and QBL. Typically a complex QBL like HQL requires parsing tools and libraries such as ANTLR. Expression-based queries like NHibernate Criteria API or LINQ, can be implemented with the Query Object or Interpreter pattern.

A simple Query Object criteria API implementation is given by the model in Figure 5.15. With this approach, for instance, the client would create the criteria `Criteria.greaterThan("quantity",10)` and then add it to `criteriaList` in the `QueryObject` within the current `CazCommand`.



```
Translate() : String
    If value instanceOf Number
        return value
    Else
        return “ ’ ”+value.ToString+” ‘ ”
```

```
Translate() : String
  return “TOP ”+argument.Translate()
```

```
Translate() : String
   vars.put(“top”,“LIMIT ”+argument.Translate())
   return “”
```

```
Translate() : String
  return “ ` ”+name+” ` ”
```

```
Translate() : String
  return ‘[’+name+’]’
```

```
Translate() : String
  return “ORDER BY ”+argument.Translate()+” ”+vars.get(“top”)
```

Figure 5.16: Solution 6: Example of an SQL-based Interpreter implementation

Note that Figure 5.15 resolves problems with date types and other operators via the Template Method, but keeps a default behaviour in the abstract `Criteria` class.

An Interpreter implementation similar to LINQ, can be demonstrated by Figure 5.16. Expression or criteria interfaces are provided to the client like in the Query Object example. Note that in SQL Server, the command `Top` comes right after `SELECT` and in MySQL it comes right after `ORDER BY` clause. This required an additional variable to delay the translation of `Top` into `LIMIT` for MySQL.

However, because CazFramework works on top of relational data through often complex and dynamic SQL queries, a criteria API would not be a good solution. Also, QBA and LINQ implementations are not suitable for complex querying. Parsing the whole SQL statement would require a parser and a unnecessary performance loss. Thus, a good solution can consist of allowing the traditional SQL statement to be combined with simple criteria API (similar to Figure 5.15) for filtering, ordering, limiting and offsetting (useful for data pagination), essentially because most GUI grids provide interfaces for these operations. Although it is common that by default, .NET GUI data grids use LINQ interfaces and for that, implementing a LINQ for CazDataProvider can be an good solution as well.

5.4 Implementation

This section presents the details of CazDataProvider implementation. From the solutions proposed in section 5.3, *Solution 5: Provider Factory with Template Method* (5.3.5) was selected for building the core classes of CazDataProvider and *Solution 6: Query Object* (5.3.6) for achieving SQL dialect interoperability features.

At this phase, the main requirements for CazDataProvider were to provide relational database and data provider interoperability, i.e. it was expected to support the following ADO.NET providers of Figure 5.6 (in section 5.2):

- **MySql Connector/NET 6.2.3** (`MySql.Data.MySqlClient`);
- **MyOLEDB Provider v3.0** (enabled by `MySqlProv.3.0` in OLEDB connection string);
- **SqlClient** for both for .NET 3.5 and 4.0 (uses `System.Data.SqlClient` *namespace*);
- **SQL Server Native Client 10.1 OLE DB Provider** (enabled by `SQLNCLI10.1` in OLEDB connection string);
- **MSAccess OLEDB Provider** (enabled by `Microsoft.ACE.OLEDB.12.0` in OLEDB connection string);
- **System.Data.SQLite 1.0.65.0** (uses the same *namespace* as its name).

All 6 providers work on top of SQL Server 2008, MySQL 5.1, **MSAccess (JET Engine 4.0)** and SQLite 3.6.23 RDBMSs. Also the Integrated Development Environment (IDE) used for developing CazDataProvider was Visual Studio 2008.

The constraints were that SQL queries should be written in SQL Server dialect and with OLEDB provider behaviour, as it always has been part of CazFramework and ClassBuilder development environment.

From the architecture of the *Solution 5* (in 5.3.5), it is possible to define a simple test case. Thus, the code in Figure 5.17 is expected to run the same way using different providers.

```
//SQLServer OLEDB Provider
Dim conn As CazConnection
conn = CazFactory.CreateCazConnection(ProviderType.SqlServer,
   "Data Source=MIGUEL-LAPTOP;Initial Catalog=TST;Integrated Security=True")

//SQLServer Native Provider
conn = CazFactory.CreateCazConnection(ProviderType.OleDb,
   "Provider=SQLNCLI10.1;Data Source=Miguel-Laptop;Integrated Security=SSPI;
   Initial Catalog=TST")

//TEST CASE
conn.Open()

cmd = conn.CreateCommand()
cmd.CommandText = "Select TOP 2 * from People where name = ? or name = ?"
cmd.AddParameterWithValues("?", "jack")
cmd.AddParameterWithValues("?", "joe")

idr = cmd.ExecuteReader()
While (idr.Read())
   Console.WriteLine(idr.Item(1).ToString())
End While

conn.Close()
```

Figure 5.17: A priori test case for CazDataProvider

At the end of this phase, the model of Figure 5.18 was implemented containing the main classes of CazDataProvider. Here, the essential applied patterns were Template Method, Factory Method and Query Object.

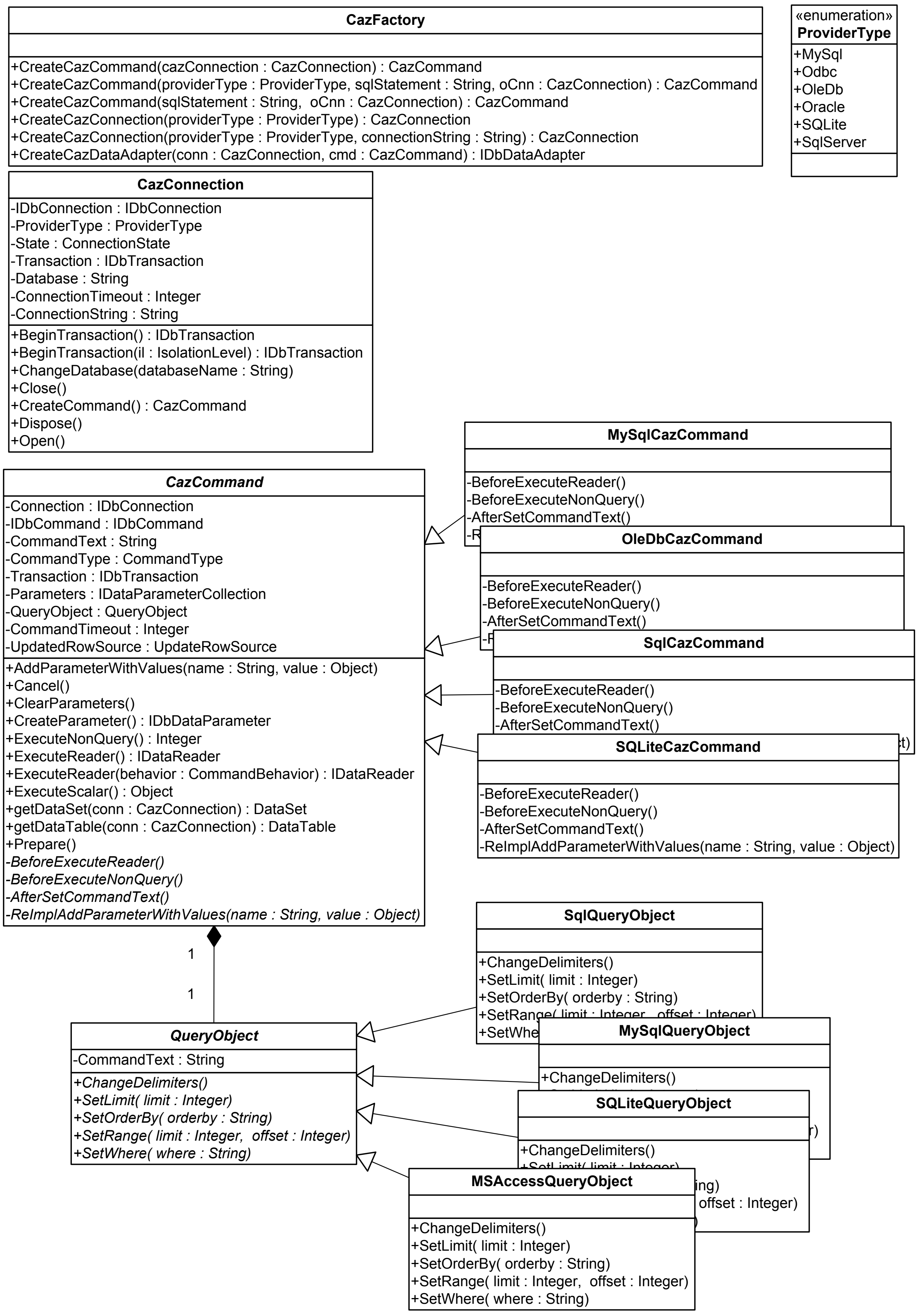

CazFactory
+CreateCazCommand(cazConnection : CazConnection) : CazCommand
+CreateCazCommand(providerType : ProviderType, sqlStatement : String, oCnn : CazConnection) : CazCommand
+CreateCazCommand(sqlStatement : String, oCnn : CazConnection) : CazCommand
+CreateCazConnection(providerType : ProviderType) : CazConnection
+CreateCazConnection(providerType : ProviderType, connectionString : String) : CazConnection
+CreateCazDataAdapter(conn : CazConnection, cmd : CazCommand) : IDbDataAdapter
«enumeration» ProviderType
+MySql
+Odbc
+OleDb
+Oracle
+SQLite
+SqlServer
CazConnection
-IDbConnection : IDbConnection
-ProviderType : ProviderType
-State : ConnectionState
-Transaction : IDbTransaction
-Database : String
-ConnectionTimeout : Integer
-ConnectionString : String
+BeginTransaction() : IDbTransaction
+BeginTransaction(il : IsolationLevel) : IDbTransaction
+ChangeDatabase(databaseName : String)
+Close()
+CreateCommand() : CazCommand
+Dispose()
+Open()
MySqlCazCommand
-BeforeExecuteReader()
-BeforeExecuteNonQuery()
-AfterSetCommandText()
OleDbCazCommand
-BeforeExecuteReader()
-BeforeExecuteNonQuery()
-AfterSetCommandText()
SqlCazCommand
-BeforeExecuteReader()
-BeforeExecuteNonQuery()
-AfterSetCommandText()
SQLiteCazCommand
-BeforeExecuteReader()
-BeforeExecuteNonQuery()
-AfterSetCommandText()
-ReImplAddParameterWithValues(name : String, value : Object)
CazCommand
-Connection : IDbConnection
-IDbCommand : IDbCommand
-CommandText : String
-CommandType : CommandType
-Transaction : IDbTransaction
-Parameters : IDataParameterCollection
-QueryObject : QueryObject
-CommandTimeout : Integer
-UpdatedRowSource : UpdateRowSource
+AddParameterWithValues(name : String, value : Object)
+Cancel()
+ClearParameters()
+CreateParameter() : IDbDataParameter
+ExecuteNonQuery() : Integer
+ExecuteReader() : IDataReader
+ExecuteReader(behavior : CommandBehavior) : IDataReader
+ExecuteScalar() : Object
+getDataSet(conn : CazConnection) : DataSet
+getDataTable(conn : CazConnection) : DataTable
+Prepare()
-BeforeExecuteReader()
-BeforeExecuteNonQuery()
-AfterSetCommandText()
-ReImplAddParameterWithValues(name : String, value : Object)
1
1
QueryObject
-CommandText : String
+ChangeDelimiters()
+SetLimit(limit : Integer)
+SetOrderBy(orderby : String)
+SetRange(limit : Integer, offset : Integer)
+SetWhere(where : String)
SqlQueryObject
+ChangeDelimiters()
+SetLimit(limit : Integer)
+SetOrderBy(orderby : String)
MySqlQueryObject
+ChangeDelimiters()
SQLiteQueryObject
+ChangeDelimiters()
MSAccessQueryObject
+ChangeDelimiters()
+SetLimit(limit : Integer)
+SetOrderBy(orderby : String)
+SetRange(limit : Integer, offset : Integer)
+SetWhere(where : String)

Figure 5.18: CazDataProvider: Implemented core classes

From the IDb* and IData* interfaces, initially marked for refactoring, only IDb-Connection and IDbCommand had to be changed into CazConnection and CazCommand accordingly. The interfaces IDbTransaction, IDataParameter, IDataParameterCollection and IDataReader are still managed by CazDataProvider with provider specific implementations. This is due to the fact that they are not expected to raise significant interoperability problems.

From the model in Figure 5.18, the client API consists of CazConnection, CazFactory (including the ProviderType enumerator), CazCommand and QueryObject. The Template Method enables additional provider specific CazCommands (e.g. MySqlCazCommand) and SQL dialect implementations of QueryObject (e.g. MySqlQueryObject)

CazConnection

Note that in order to make Template Method work, the superclass (CazCommand) has to be set abstract to prevent wrong instantiation. Also, the refactored methods intended to be overridden by the subclasses, must be set protected (not supposed to be seen by the client) and virtual or overriddable in the superclass.

An early version of CazDataProvider had an abstract CazConnection extended by provider specific Connections with Template Methods. This proved to be unnecessary because all the IDbConnection implementations share the same behaviour. However, the method CazConnection.GetCommand needs to create a CazCommand of the same ProviderType (of the CazConnection), with the appropriate QueryObject and provider specific IDbCommand.

The methods of CazConnection are similar to those of IDbConnection. Thus, with inheritance by delegation, all CazConnection methods are implemented simply by calling the appropriate method of IDbConnection, except for GetCommand which calls a Factory Method in CazFactory to instantiate the right CazCommand.

The following example demonstrates how to instantiate a CazConnection and create a CazCommand:

```
//MySql OLEDB Provider
Dim conn As CazConnection
conn = CazFactory.CreateCazConnection(ProviderType.MySql,
   "Provider=MySqlProv.3.0;Data Source=GA_CAZWEB;User ID=root;
   Location=localhost")
cmd = conn.CreateCommand()
```

CazFactory

CazFactory provides static Factory Methods to create a CazConnection given a Provider-

Type (one of the 4 supported: MySql, OleDb, SqlServer and SQLite). Additionally, it creates CazCommands given a CazConnection (the same method called by CazConnection.GetCommand). It is here that the external provider libs are used and loaded by reflection to avoid further compilation issues. An example of using CazFactory to instantiate a CazCommand is given by the following code:

```
cmd = CazFactory.CreateCazCommand("Select id,nome from Tabela where
    '1' = ? or '2' = ?", conn)
```

CazCommand

CazCommand is the most important class in the CazDataProvider model (see Figure 5.18). It is an abstract class and inherits most IDbCommand methods via delegation. The encapsulated IDbCommand inside can be any provider specific ADO.NET Command (e.g. OleDbCommand, MySqlCommand, SQLiteCommand or SqlConnection). The more CazCommand subclasses, more providers supported.

Unlike SqlParameterCollection or OleDbParameterCollection, the IDataParameterCollection API does not provide a user friendly Add method accepting the arguments of parameter and value. Thus, Template Method pattern was used to manage multiple implementations of methods like CazCommand.AddParameterWithValues. Also, before sending the SQL query to the database, there is an extension point that triggers specific provider adjustments.

The following example presents the use of unnamed parameters applied to a Query Object criteria, also set to take a range of rows from the result before executing the CazCommand:

```
cmd.AddParameterWithValues("?", "1")
cmd.QueryObject.SetRange(3, 1)
cmd.ExecuteReader()
```

Before running ExecuteReader, both MySqlCommand and SqlCommand replace the unnamed parameters with named parameters. For that, in these same specific Commands, the method AddParameterWithValues replaces the '?' symbol with a numbered name to complement the named parameters conversion

Parameters problems

While examining all the 6 data providers, interoperability parameter problems were found on IDbCommand implementations and they were further resolved in CazCommand along with its subclasses. The three encountered problems are:

- Some providers accept only named or unnamed parameters. `SQLiteCommand`, `MySqlCommand` and `SqlCommand` only accept named parameters, e.g. '@name'. MySQL and SQL Server `OleDbCommands` (**MySqlProv.3.0** and **SQLNCLI10.1**) only accept unnamed parameters with symbol '?'. **MSAccess** `OleDbCommand` (**Microsoft .ACE.OLEDB.12.0**) accepts both the approaches.

- Therefore, it is assumed that the client of `CazCommand` will only use unnamed parameters (with '?') in the SQL statement and not use named parameters. Also, the parameters must be added respecting the order in the SQL statement and using `AddParameterWithValues`, which ignores the parameter name.

- OLEDB providers only support string format date parameters (`yyyy-MM-dd HH:mm:ss (no GMT)`). `AddParameterWithValues` is set to still accept `System.Date` values and then, in the subclasses of `CazCommand`, it is converted into a string. Additionally, for the MySQL non OLEDB provider, `MySqlCommand` needs the `DbType` of the parameter to be specified as a `Datetime` in order to work;

- OLEDB **MySqlProvider.3.0** provider only supports string value parameters. When adding a new parameter with integer or boolean values, it crashes. This is managed in `OleDbCazCommand` with if conditions.

QueryObject

When a `CazCommand` is created, a `QueryObject` is instantiated inside it by the `CazFactory`. This `QueryObject` determines which SQL dialect is configured for that `CazCommand`. Typically, there are as many `QueryObjects` as RDBMSs supported.

This `QueryObject` was required because, for instance both **MySql Connector/NET 6.2.3** and **MyOLEDB Provider v3.0** or **SqlClient** and **SQL Server Native Client 10.1 OLE DB Provider** share the same abstraction to SQL dialect.

The CazDataProvider model (in Figure 5.18) contains `MySqlQueryObject`, `SqlQueryObject`, `SQLiteQueryObject` and `MSAccessQueryObject` all inheriting from `QueryObject` and implementing Template Method.

`QueryObject` is an abstract class responsible for adding criteria to the SQL statement in a provider and SQL dialect independent way. Its name is after the Query Object pattern and therefore follows a similar strategy from the *Solution 6* (5.3.6). Such criteria supports the following operations:

- `SetLimit` - similar to the `TOP N` in SQL Server, it fetches the N first rows from the result;

- SetRange - fetches the rows comprehended by the specified range in the full result;
- SetWhere - filters the result given a condition, i.e. it adds an SQL Where clause to the SQL statement;
- SetOrderBy - sorts the result in the database by adding an Order By clause at the end of the SQL statement.

Note that whenever any QueryObject criteria is invoked, it changes the relative CazCommand.CommandText (SQL statement).

QueryObject contains a copy of CommandText from CazCommand. This makes the ObjectQuery criteria call for query transformation to not immediately reflect on the CazCommand.CommandText. This problem occurs due to CommandText being a string and thus it does not pass as a reference to other classes or methods. The solution consisted on adding an update or synchronization of CommandTexts before executing the SQL command and also run the update on the getter property of CazCommand.CommandText. The setter property is responsible for changing its value and the value in ObjectQuery.CommandText immediately. A more simple solution would be for QueryObject to manage a reference to the same IDbCommand.

QueryObject problems

QueryObject criteria API manages the dissimilarities of SQL syntax over 4 different relational databases (MySQL, SQL Server, SQLite and **MSAccess**). One problem is the variety of delimiters used for column, table and variable names. By default, it is assumed that delimiters for column and table names come as [tablename] and [columname] in SQL Server. It is not expected that types and other SQL commands come specified by those same delimiters. For instance MySQL uses ‘xxx‘ and does not accept [xxx] whilst with SQL Server is the exact opposite. The use of delimiters is important because reserved words can lead to unexpected problems. See the following example to understand the differences between MySQL and SQL Server delimiters in queries:

```
// SQL Server example:
Select [u].[name] From [dbo].[Users] As [u]

// MySql example:
Select ‘u‘.‘name‘ From ‘dbo‘.‘Users‘ As ‘u‘
```

Thus, by the above example, MySqlQueryObject has to convert from SQL Server to MySQL delimiters if MySQL database is being used (SQL Server dialect is the default in CazDataProvider).

Another problem occurs with the SQL command for limiting the number of results. This is implemented in QueryObject with SetLimit criteria. MySQL and SQLite use

`Limit N` at the end of SQL query, after `Order By` clause. On the other hand, SQL Server and **MSAccess** use `Top N` and right after the `Select`.

However, `SetLimit` was relatively simple to implement when compared to `SetRange` criteria. Whilst MySQL and SQLite use simple `Limit N` and `Offset M` notation, SQL Server and **MSAccess** do not provide equivalent implementation to `Offset M`.

The further examples demonstrate the equivalent SQL (in SQL Server and **MSAccess**) for the following `SetRange` in MySQL:

```
Select ‘u‘.* From ‘dbo‘.‘Users‘ As ‘u‘ Order By ‘u‘.‘id‘ Limit 10 Offset 100
```

Note that the above example takes only from user row 101 to 110. The equivalent in SQL Server 2008 is:

```
SELECT *
FROM
(
   SELECT
       row_number()
   OVER
      (ORDER BY [INNERTABLE].id) AS [ROWNUM], [INNERTABLE].*
   FROM
     (Select [u].* From [dbo].[Users] As [u]) [INNERTABLE]
) AS [OUTERTABLE]
WHERE
 [OUTERTABLE].[ROWNUM] BETWEEN 101 AND 110
```

This approach contains 3 nested `Selects` (a table scan and a sort) while without offset feature it contains only 1 `Select`. Also, it needs to know the column name to use the `Order By` clause. Thus, if the `SetRange` criteria is used, `SqlCazCommand.CommandText` must define the projected column names in the outermost `Select` instead of using the symbol ‘*’. The implementation in `SqlQueryObject` uses the first column name to set the `Order By`. Note that `row_number()` was added in SQL Server 2005.

The equivalent in **MSAccess** is:

```
Select Top 10 * From [dbo].[Users]
WHERE [id] NOT IN
   (SELECT TOP 100 [id] FROM [dbo].[Users] ORDER BY [id])
ORDER BY [id];
```

MSAccess only supports some features of SQL Server, it does not support `row_number()`. Therefore, the above approach was the encountered solution. This defines 2 nested `Selects` (the original `Select` is defined twice), although it is slower than the SQL Server `row_num` algorithm because it scans the 100 first rows and then scans the other 10. Also a column name has to be specified in the outermost `Select` the same way as with SQL Server algorithm due to the `Order By`.

In `OleDbCommand` for **MSAccess**, if the number of (unnamed) parameters in the SQL statement and in the collection of parameters does not match, it is assumed that was because of the query transformation of `SetRange` criteria. Hence, `OleDbCommand` will double its parameters by replication.

5.5 LinqToCaz

LinqToCaz is a custom implementation of .NET LINQ interfaces. Essentially it provides LINQ query features to CazDataProvider, similar to the `QueryObject` criteria API defined in the last section (5.4).

LinqToCaz implements Query Object (see section 3.4.2), Interpreter (see *Solution 6* 5.3.6) and Repository (see the example of a *queryable* implementation in section 3.4.3)

The implemented LINQ interfaces were `Take`, `Skip`, `Where` and `OrderBy`, although the code is extensible for adding other LINQ operators.

The following example demonstrates how these features are used with encapsulated CazDataProvider:

```
Dim provider As New LinqToCaz.DbQueryProvider(ProviderType.OleDb, _
   "Provider=SQLOLEDB;Data Source=.;Initial Catalog=Northwind;" _
   & "Integrated Security=SSPI")

provider.Open()
Dim sqlQuery = provider.CreateQuery( _
   "Select t.TerritoryID [TID], t.* from Territories t")

Dim queryA = sqlQuery.Where(Function(c) (c.Item("RegionID") = 1 _
   Or Not c.Item("RegionID") = 2) And c.Item("RegionID") = 3) _
   .OrderByDescending(Function(c) c.Item("TerritoryDescription")) _
   .OrderBy(Function(c) c.Item("RegionID")) _
   .Take(10).Skip(5)

Dim queryB = From c In sqlQuery _
   Where (c.Item("RegionID") = 1 Or Not c.Item("RegionID") = 2) _
   And c.Item("RegionID") = 3 _
   Order By c.Item("TerritoryDescription") Descending Order By _
   c.Item("RegionID") _
   Take 10 Skip 5
```

```
Dim resultA = queryA.ToList()
Dim resultB = queryB.ToList()
```

The above example contains 2 different ways of writing LINQ queries in .NET and only fetches data from database when the `query.ToList()` is invoked. LinqToCaz allows the SQL query to be defined before applying LINQ criteria on the `IQueryable` object returned by `CreateQuery` method. The result is only computed in the end and returning a dynamic data structure as `IEnumerable(Of Dictionary(Of String, Object))`, representing a row set.

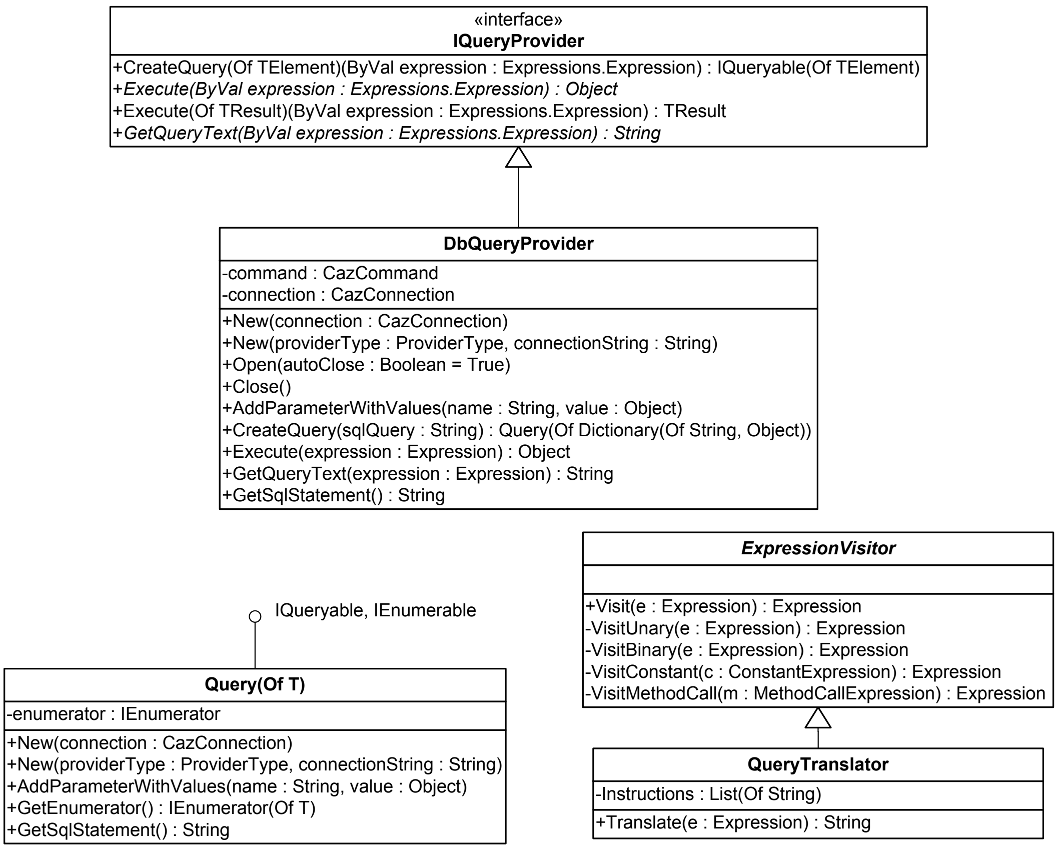

«interface»
IQueryProvider
+CreateQuery(Of TElement)(ByVal expression : Expressions.Expression) : IQueryable(Of TElement)
+Execute(ByVal expression : Expressions.Expression) : Object
+Execute(Of TResult)(ByVal expression : Expressions.Expression) : TResult
+GetQueryText(ByVal expression : Expressions.Expression) : String
DbQueryProvider
-command : CazCommand
-connection : CazConnection
+New(connection : CazConnection)
+New(providerType : ProviderType, connectionString : String)
+Open(autoClose : Boolean = True)
+Close()
+AddParameterWithValues(name : String, value : Object)
+CreateQuery(sqlQuery : String) : Query(Of Dictionary(Of String, Object))
+Execute(expression : Expression) : Object
+GetQueryText(expression : Expression) : String
+GetSqlStatement() : String
ExpressionVisitor
+Visit(e : Expression) : Expression
-VisitUnary(e : Expression) : Expression
-VisitBinary(e : Expression) : Expression
-VisitConstant(c : ConstantExpression) : Expression
-VisitMethodCall(m : MethodCallExpression) : Expression
IQueryable, IEnumerable
Query(Of T)
-enumerator : IEnumerator
+New(connection : CazConnection)
+New(providerType : ProviderType, connectionString : String)
+AddParameterWithValues(name : String, value : Object)
+GetEnumerator() : IEnumerator(Of T)
+GetSqlStatement() : String
QueryTranslator
-Instructions : List(Of String)
+Translate(e : Expression) : String

Figure 5.19: LinqToCaz: Class diagram

Figures 5.19 and 5.20 provide some insight on the LinqToCaz implementation details. Note that the LINQ expressions can be defined by a similar approach to the model in Figure 5.16 (Interpreter pattern).

`DbQueryProvider` is the class that creates the `IQueryable` `Query` and executes it. For that it has to implement `System.Linq.IQueryProvider` interface with `CreateQuery` and `Execute` methods.

When the `Query` is iterated, the `Query.GetEnumerator` is called which forwards to the `DbQueryProvider.Execute` method. For executing the SQL query, `DbQueryProvider`

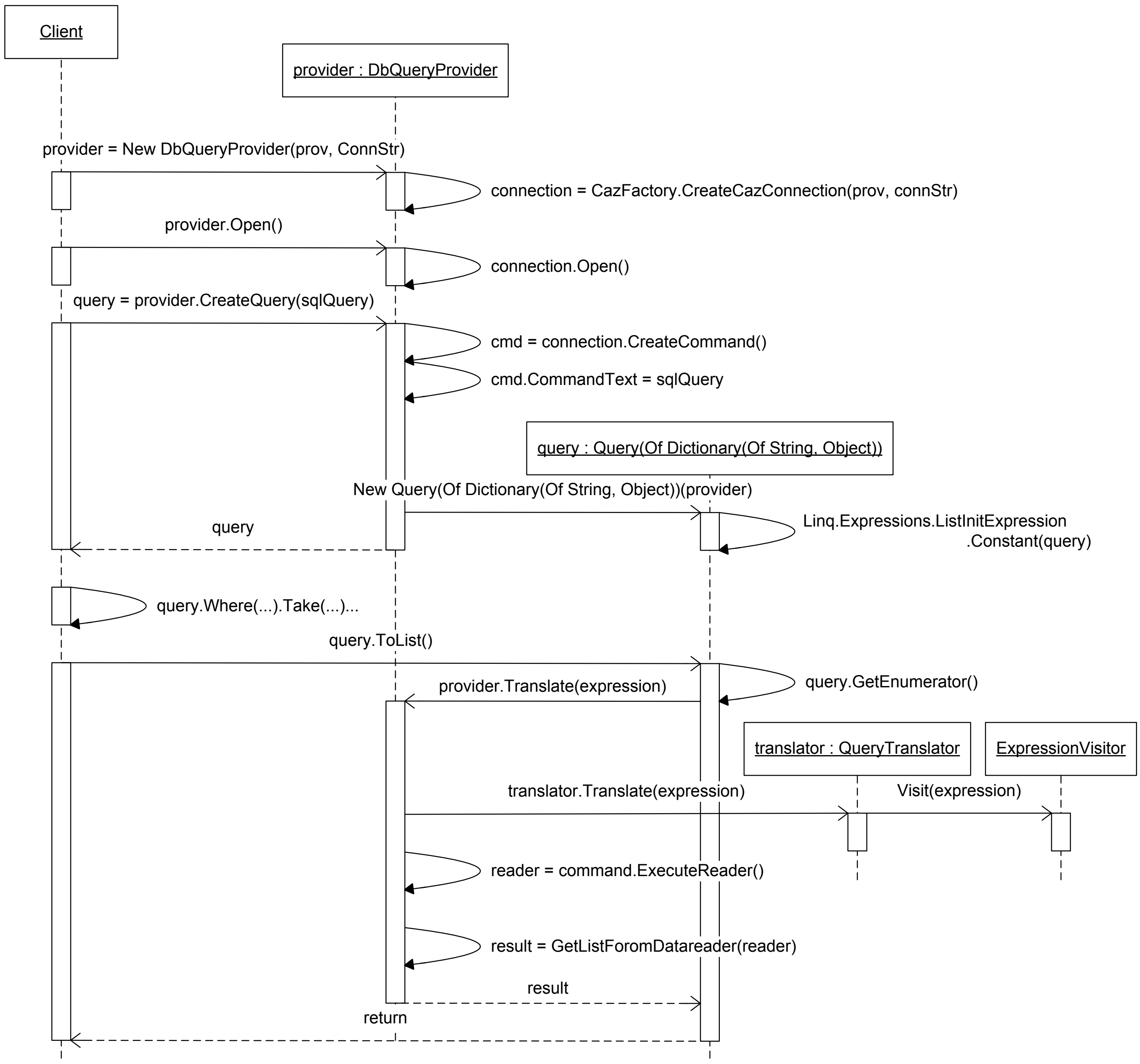

Client
provider : DbQueryProvider
provider = New DbQueryProvider(prov, ConnStr)
connection = CazFactory.CreateCazConnection(prov, connStr)
provider.Open()
connection.Open()
query = provider.CreateQuery(sqlQuery)
cmd = connection.CreateCommand()
cmd.CommandText = sqlQuery
query : Query(Of Dictionary(Of String, Object))
New Query(Of Dictionary(Of String, Object))(provider)
query
Linq.Expressions.ListInitExpression .Constant(query)
query.Where(...).Take(...)...
query.ToList()
query.GetEnumerator()
provider.Translate(expression)
translator : QueryTranslator
ExpressionVisitor
translator.Translate(expression)
Visit(expression)
reader = command.ExecuteReader()
result = GetListForomDatareader(reader)
result
return

Figure 5.20: LinqToCaz: Sequence diagram

uses CazDataProvider core API. Thus it needs to manage a `CazCommand` and a `CazConnection` instances. First it calls method `Translate` which takes a LINQ expression and calls `QueryTranslator` to convert this expression into a list of `Instructions`. These instructions are then iterated and if any of the commands `Where`, `OrderBy`, `Take` or `Skip` is found, it calls the corresponding `ObjectQuery` API criteria defined in CazDataProvider core `api`. Then, `CazCommand` executes the SQL command and the `IDataReader` is converted into a dynamic structure `List(Of Dictionary(Of String, Object))`.

`ExpressionVisitor` implements the Interpreter pattern as it defines a recursive LINQ expression visiting protocol. `QueryTranslator` overrides some methods to implement the behaviour that translates the `Where`, `OrderBy`, `Take` and `Skip` LINQ expressions into instructions defined as strings.

This LinqToCaz was developed using the code and guidance of Matt Warren with his article *LINQ: Building an `IQueryable` Provider*[1].

5.6 Conclusions

This chapter discussed the various steps of research, design and implementation of CazDataProvider and presents an alternative to the typical ORM frameworks like NHibernate or EF with the avoidance of most object-relational mismatch problems.

In here, ClassBuilder, a Cachapuz custom ORM tool was analysed with the identification of some common ORM design patterns. ClassBuilder was found to use a Relational Domain Model pattern which is the reason why it avoids most ORM problems like inheritance and association mapping, Partial-Object and Dual-Schema. Some features analysed in ClassBuilder include the Data Mapper API, audit logging, Unit of Work and optimistic locking.

Cachapuz demanded for a database-agnostic ClassBuilder and for that, ADO.NET providers and provider factories were analysed in order to find a solution for the architecture of CazDataProvider. Various architectural approaches were suggested, in section 5.3, surrounding the implementation of some design patterns like Strategy, Abstract Factory, Factory Method, *Call Super* or *BaseBean*, Template Method, Decorator, Query Object and Interpreter. In the end, *Solution 5: Provider Factory with Template Method* was chosen to be implemented in CazDataProvider and complemented with *Solution 6: Query Object*.

Section 5.4 discussed the implemented classes as well as the data provider and SQL dialect problems or barriers to the database interoperability realization. Additionally, LinqToCaz was implemented in order to provide LINQ query features to CazDataProvider.

CazDataProvider was successfully tested with the `mysql`, SQL Server, SQLite and **MSAccess** and thus, it currently supports the following ADO.NET data providers:

- **MySql Connector/NET 6.2.3**;
- **MyOLEDB Provider v3.0**;
- **SqlClient**;

[1]`http://blogs.msdn.com/b/mattwar/archive/2007/07/30/linq-building-an-iqueryable-provider-part-i.aspx`

- **SQL Server Native Client 10.1 OLE DB Provider**;
- **Microsoft.ACE.OLEDB.12.0**;
- **System.Data.SQLite 1.0.65.0**.

All in all, the key aspect of CazDataProvider is that it allows database agnostic SQL statements to return a dynamic *queryable* object structure. This feature enables a more dynamic approach than using ClassBuilder for it entirely avoids the object schema and with it, the recompilation of POCO entities when the relational schema changes). That is due to CazDataProvider converting a generic relational query into a relational database specific query (generic SQL to specific SQL) rather than using an object-relational translation approach (OQL to SQL) like typical ORM implementations.

In the end, ClassBuilder was modified to generate DAL code supported by CazDataProvider. This DAL code was further used in CazFramework. Hereafter, CazDataProvider will be enhanced to support more ADO.NET providers and RDBMSs like Oracle or **DB2**. LinqToCaz is set to be improved as well, in order to support more LINQ operators and work with GUI data grid features implicitly.

Chapter 6

Benchmark Tests: Comparing CazDAL, ORM Frameworks and OOSBMSs

This chapter presents a benchmark analysis of 5 different technologies relevant for this dissertation:

- **NHibernate 3.1.0**: Analysed ORM framework in section 4.2 and tested here using .NET 3.5 and SQL Server 2005;
- **EF 4.0**: Analysed ORM framework in section 4.1 with SQL Server 2005;
- **CazDAL**: ClassBuilder generated code using CazDataProvider on top of SQL Server 2005;
- **DB4O 8.0**: it is a proprietary OODBMS currently supporting Java and .NET platforms. It can run embedded or in server-mode and provides dynamic schema evolution at runtime, query criteria and LINQ implementation, ACID transactions and replication. Its lack of text-based indexing, support for many concurrent users (it is single-threaded) and other optimization features are still important disadvantages towards RDBMSs. For more details on OODBMSs see 2.5 and [VZ10];
- **Eloquera 3.0**: proprietary OODBMS similar to DB4O but for .NET only, multi-threaded (support for more concurrent users) and supporting OQL serializable queries.

All tests were developed in Visual Studio 2010. The test cases manage the object `Person` (see Figure 6.1) and in most cases a table `Person` as well. Note that a `Person` can have relations with other `Person` objects. Also, the `Person` id is set to identity (auto-increment) on SQL Server 2005.

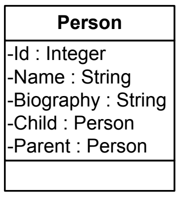


Figure 6.1: Benchmark Tests: Managed `Person` class

Some scenarios may use slightly different algorithms depending on the technology used. It is a priority to measure, at first, each technology's default behaviour, i.e. in the algorithms it is given stronger emphasis to the simpler API that requires a lesser understanding of that tool or framework. In the end, speed performance can be confronted with learning curve.

6.1 Creating

The algorithm behind this test consists of storing N different `Person` instances related to each other in groups of 3, i.e. there are $N/3$ families and each has a grandfather, a father and a son. Also, each `Person` is about 2KB of size due to the large `Bibliography` attribute. Note that the routine saves 3 `Persons` per iteration (e.g. 10,000 iterations persist 30,000 `Person` objects).

Figure 6.2 presents a speed performance test among all 5 technologies.

EF uses the `ObjectContext.SaveChanges` to commit the transaction twice on each iteration and therefore does not overload memory, unlike the NHibernate algorithm. Note that EF, in here, uses dynamic updates (default).

On each iteration **CazDAL** inserts 3 `Persons`, reads all 3 back from the database, sets the id references among all 3 objects and then updates them. Note that there is no support for dynamic updates in **CazDAL**. Therefore it becomes slower than most technologies tested here.

In this example, there is less overhead when using an OODBMS such as **Eloquera** or DB4O. The real `Person` id is persisted from its in-memory pointer, thus the attribute `id` is ignored. Therefore, only one call is invoked per object to the Object Database (ODB) API.

NHibernate uses `Session.Save` to persist each `Person`, fills the associations and then updates each `Person` object. The reason why NHibernate curve grows higher in Figure 6.2

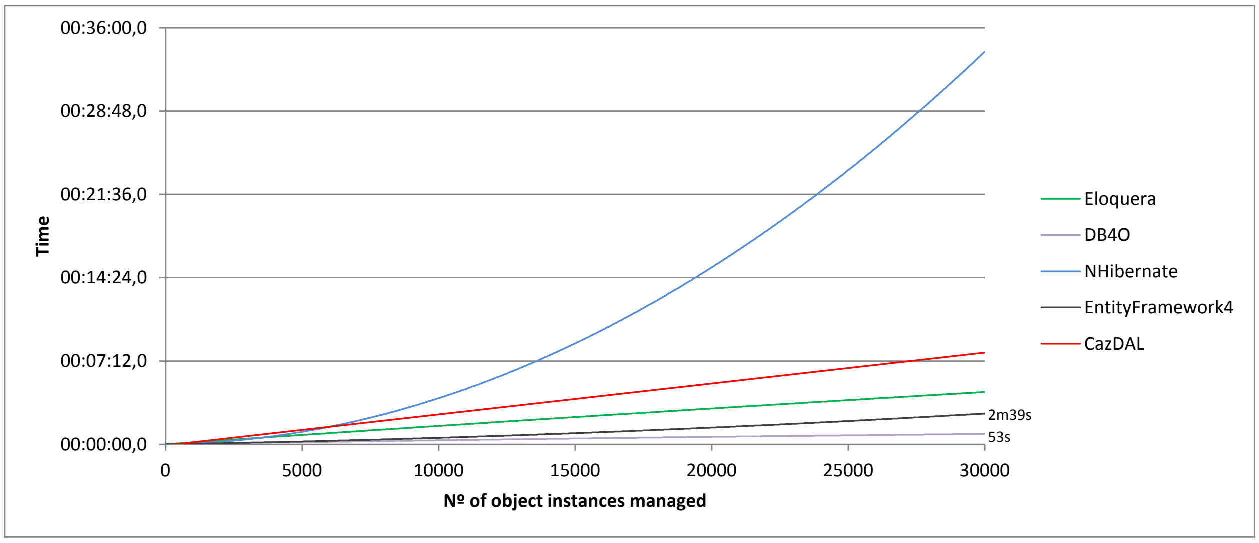


Figure 6.2: Benchmark Tests: Creating objects

is because the Unit of Work keeps track of every object in memory, i.e. the Session becomes larger and larger whilst more objects are persisted. This scenario can be avoided using transaction API with multiple commits or calling `Session.Flush` and `Session.Clear` once in a while, otherwise the Session will eventually run out of memory. Also, note that in here, NHibernate is set to not use dynamic updates (`dynamic-update="false"` is the default).

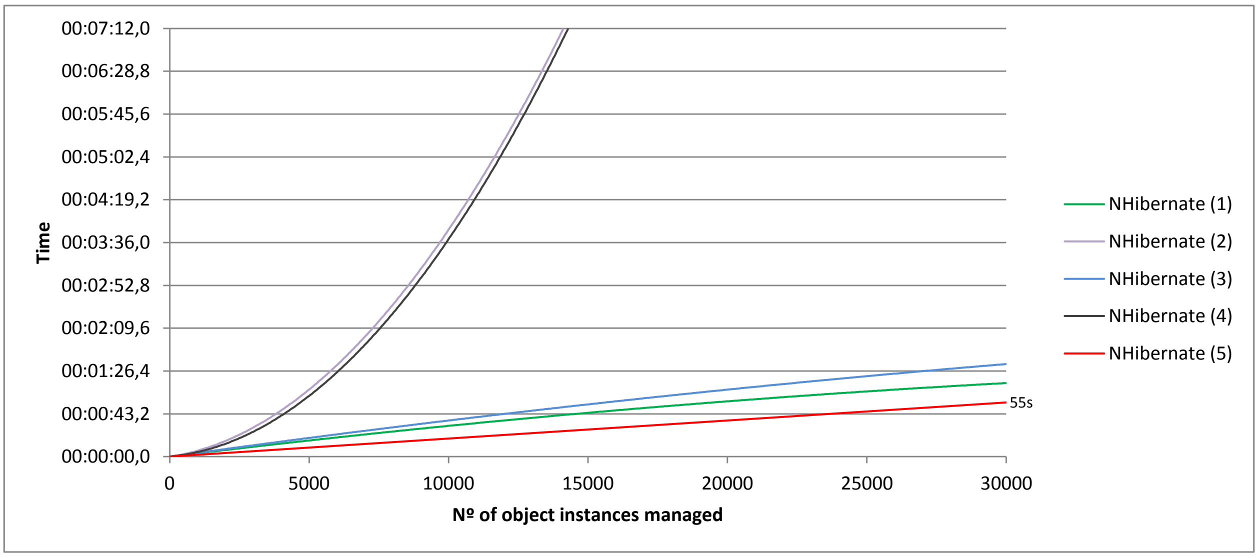


Figure 6.3: Benchmark Tests: Creating objects with NHibernate optimized algorithms

Figure 6.3 provides more tests on 5 different NHibernate algorithms:

- *NHibernate (1)*: Calls `Session.Flush` once at the end of the cycle;

- *NHibernate (2)*: Invokes `Session.Flush` per iteration;
- *NHibernate (3)*: Invokes `Session.Flush` and `Session.Clear` per iteration;
- *NHibernate (4)*: Wraps each iteration in a single transaction with $N/3$ commits;
- *NHibernate (5)*: Wraps all the iterations in a single transaction and only commits once at the end.

6.2 Loading All

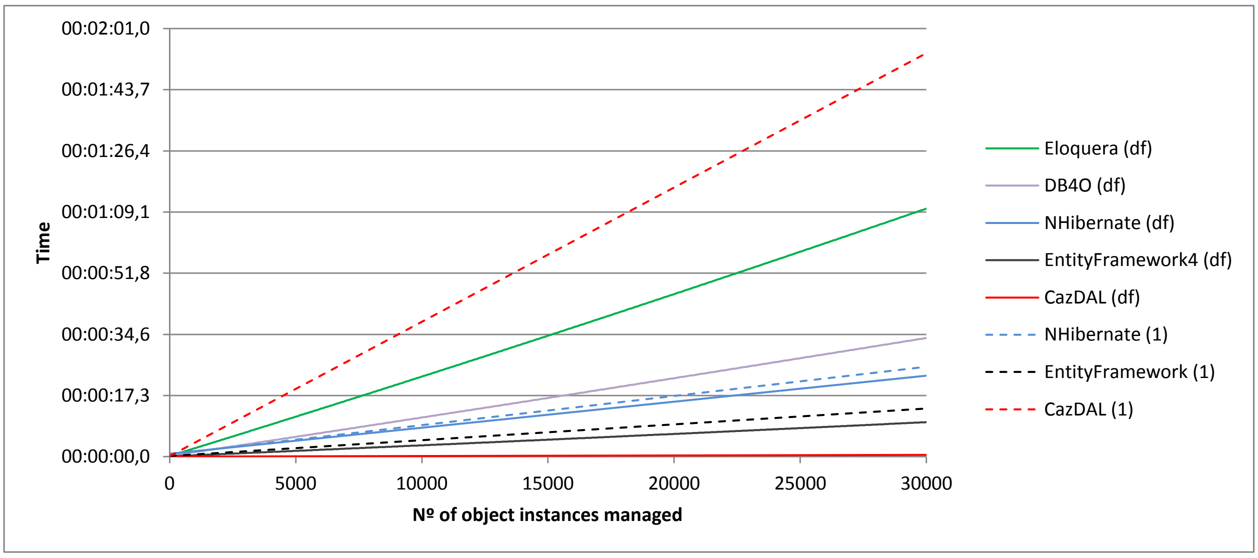


Figure 6.4: Benchmark Tests: Loading objects

In this test, the following 8 algorithms load N objects (all the persisted objects in section 6.1 are loaded) and then iterate over all of them while calling `Person.Name` (see Figure 6.4):

- *Eloquera (df)*: By default, **Eloquera** implicitly loads all the associations within each object;
- *DB4O (df)*: By default, DB4O implicitly loads all the associations within each object at a depth level of 5;
- *NHibernate (df)*: NHibernate is set to lazy associations and the large field `Biography` is lazy as well;
- *EntityFramework4 (df)*: The same as NHibernate;

- *CazDAL (df)*: Does not load associations neither large fields (`Biography`);
- *NHibernate (1)*: NHibernate is set to load eager associations with `join` fetching strategy;
- *EntityFramework4 (1)*: Uses explicit eager loading similar to NHibernate;
- *CazDAL (1)*: Explicitly loads each association (no caching) and the large field `Biography`.

Note that *CazDAL (1)* is slower because it does not cache early loaded objects and runs a single SQL `Select` for each loaded object.

6.3 Fetching One

In this test, all the algorithms fetch one `Person` object given the criteria `Name="Jack"`. In here the variable is the database size on the number of `Person` objects/rows. The default algorithms in Figure 6.5 follow the same configurations as in the above section 6.2. *Eloquera (df)*, *DB4O (df)*, *NHibernate (df)* and *EntityFramework4 (df)* use their own LINQ implementation. **CazDAL** is slightly faster because it uses SQL criteria, thus avoiding LINQ expression compile time.

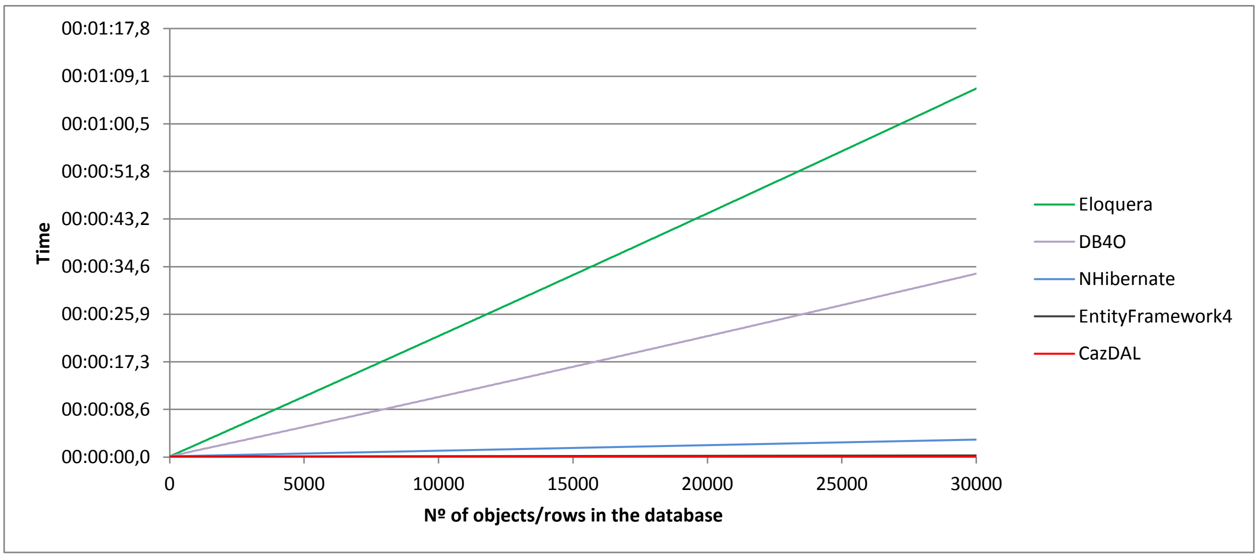


Figure 6.5: Benchmark Tests: Fetching one object with a variable database size

Because in Figure 6.5, the results for both OODBMSs diverge form the others, this test was performed again but with some optimizations.

The results of the second test are displayed in Figure 6.6. Here, both algorithms *NHibernate (df)* and *EntityFramework4 (df)* use the same default configurations of Figure 6.5 except the former runs HQL and the latter EQL rather than LINQ. In this new test, both *Eloquera (df)* and *Eloquera (idx)* algorithms run the **Eloquera** object SQL language (OQL). Also, *Eloquera (idx)* has a pre-created index on the `Person` field `Name`. Both *DB4O (df)* and *DB4O (idx)* run criteria API rather than LINQ. Additionally, *DB4O (idx)* has an index on the `Person` field `Name`.

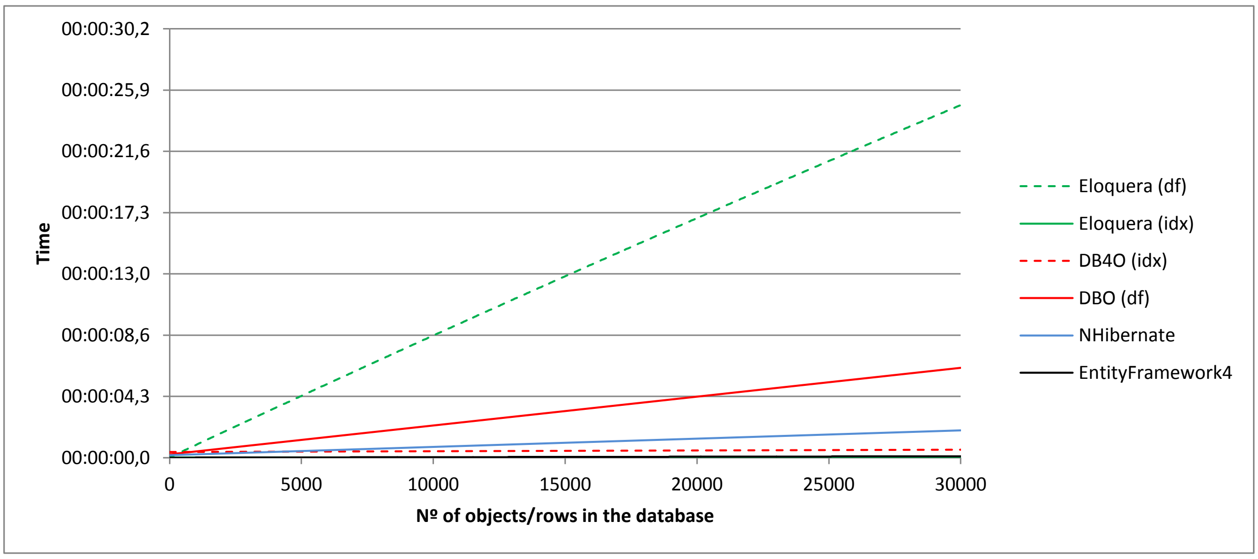


Figure 6.6: Benchmark Tests: Optimized fetching of one object with a variable database size

Figure 6.6 expresses a slight performance enhancement by using a different query API on both OODBMSs. This enhancement is greater when using indexes, which makes DB4O perform near to the ORM frameworks.

More tests were conducted with criteria `Name LIKE "J%"` for fetching more than one `Person`. Although the results did not vary significantly from the ones in Figure 6.5.

6.4 Updates and Deletes

Updates and deletes provide slightly different results than reads. Both tests in here, were performed by fetching a group of `Persons` with a criteria `Person.Child.Name LIKE "J%"` and then iterate over the results (23% of all the objects/rows in the database are returned by the query). Note that the variable here is still the database size.

The updates modify the `Person` field `Name` on each iteration. Figure 6.7 presents the following algorithms:

- *Eloquera (df)*: LINQ query and object saved on each iteration;
- *DB4O (df)*: Same as *Eloquera (df)*;
- *NHibernate (df)*: LINQ query (lazy associations and `Biography` field) and object updated with `Session.Update` and `Session.Flush` on each iteration (without dynamic updates);
- *EntityFramework4 (df)*: LINQ query (eager load of associations) and object saved with `ObjectContext.SaveChanges` on each iteration (with dynamic updates);
- *CazDAL (df)*: SQL query with a nested `Select` (load of all fields but without associations) and update of all `Person` fields on each iteration.

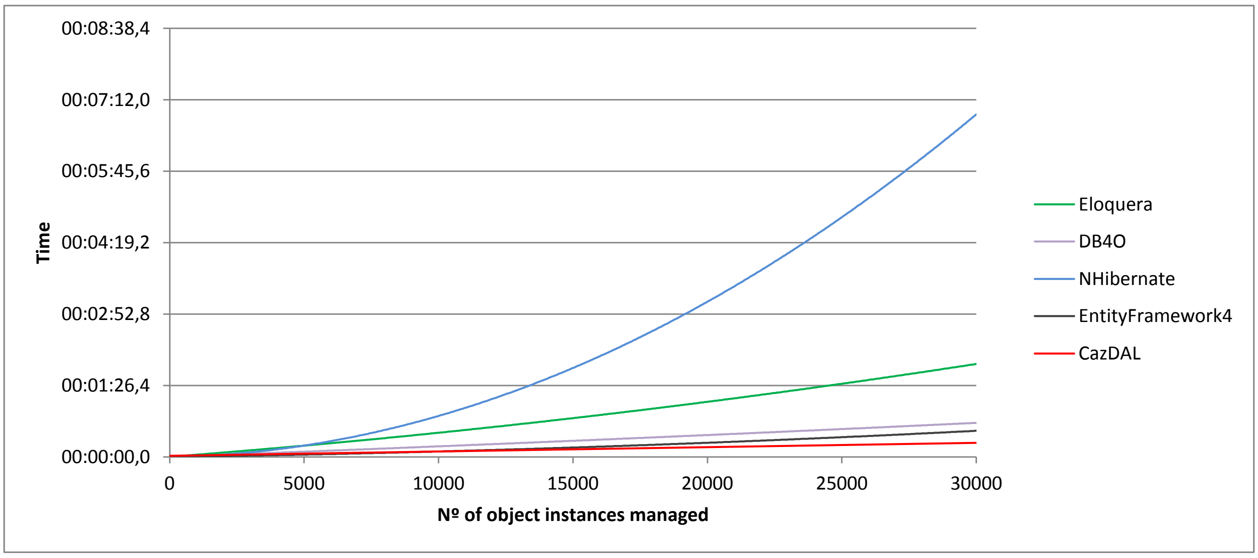


Figure 6.7: Benchmark Tests: Fetching and updating multiple objects with a variable database size

Note that *NHibernate (df)* is slow due to the same problem discussed in section 6.1. *CazDAL (df)* is fast because it does not load associations and manages SQL rather than LINQ. DB4O is also fast because it only updates the modified field.

With deletes, all 5 algorithms execute the same LINQ query as in the updates. The results are provided in Figure 6.8. Here, both *NHibernate (df)* and *EntityFramework4 (df)* have to load the associations `Child` and `Parent`, clear their references to the object about to be deleted and only then delete the object. Because *Eloquera (df)* and *DB4O (df)* manage object pointers rather than primary and foreign keys, each iteration only needs to delete the object. **CazDAL** only deletes the fetched rows without loading and updating (referencing) the associated rows. Other delete tests on **CazDAL** were conducted with eager associations and updating the associated rows and the result was only 17% slower.

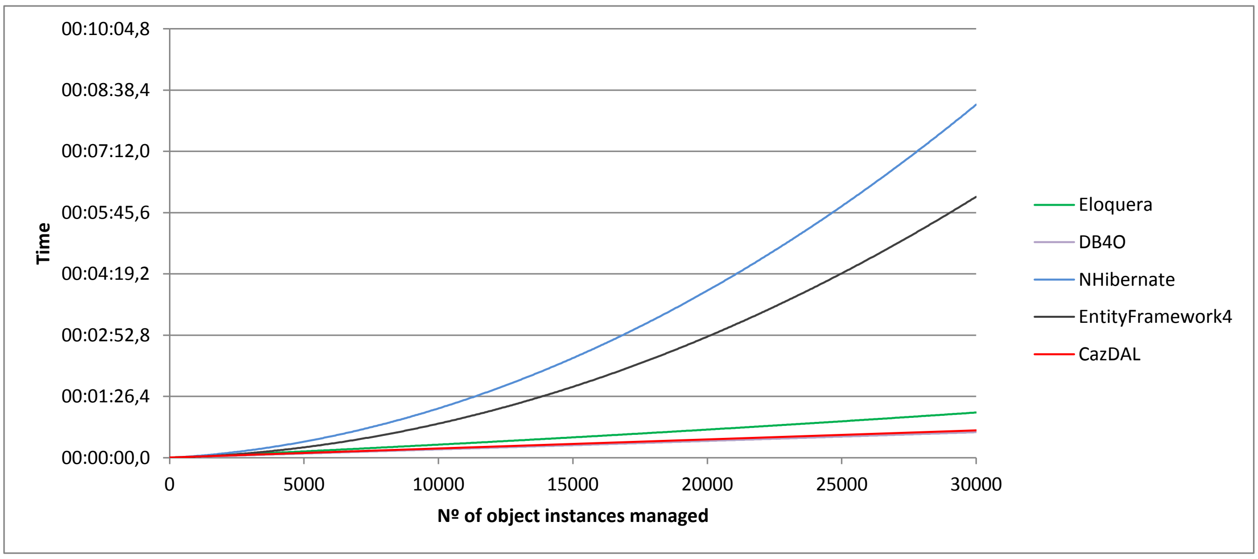


Figure 6.8: Benchmark Tests: Fetching and deleting multiple objects with a variable database size

Thus, according to these tests, it is apparent that DB4O and **Eloquera** run updates and deletes faster than ORM frameworks.

6.5 Joins

Joins were tested with NHibernate and DB4O only because, here, the intention was to compare ORM frameworks with OODBMSs. Also, because DB4O has similar results to **Eloquera** and NHibernate with EF. In here, the loaded object graph was tested with a variable depth up to 4000 associated objects similar to a linked list. Figure 6.9 demonstrates the performance of the following algorithms:

- *DB4O (1)*: Explicit load with graph depth size N;
- *DB4O (df)*: Default implicit load with graph depth size 5;
- *NHibernate (1)*: Eager load with N SQL `Selects`;
- *NHibernate (2)*: Eager load with join fetch running $N/2$ SQL `Selects`;
- *NHibernate (df)*: Lazy Load.

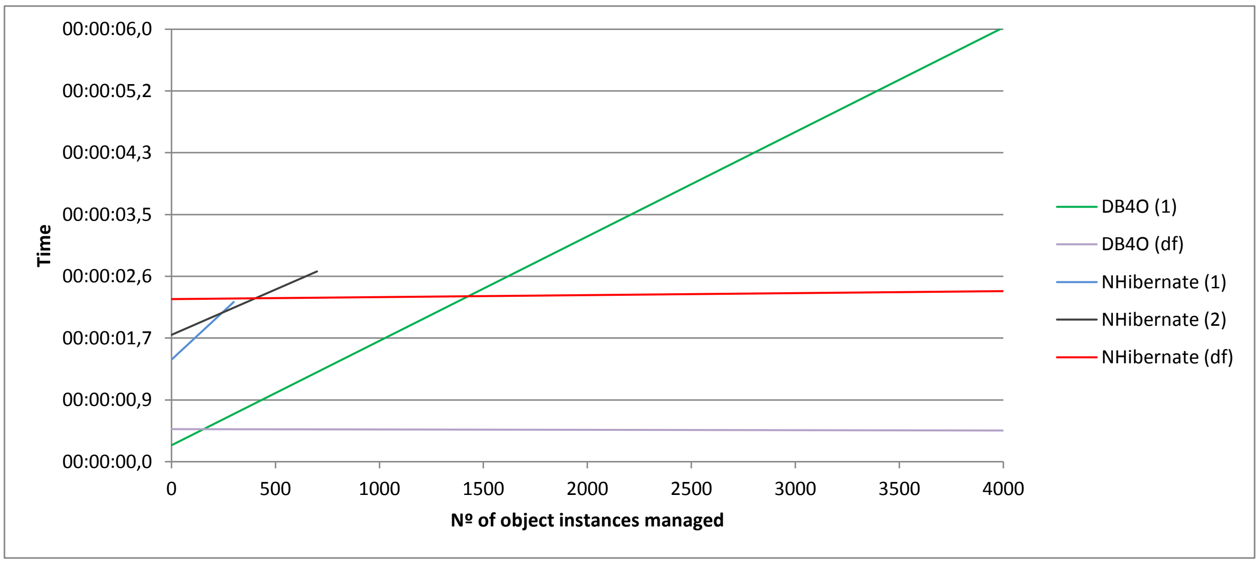


Figure 6.9: Benchmark Tests: Fetching an object graph with variable join operations

As expected, in DB4O the lazy strategies are constant and the eager loading is faster, supporting more objects than any of the eager load strategy of NHibernate which do not load more than 1000 objects. Fast joins is a typical characteristic of OODBMSs as opposed to ORM frameworks.

6.6 Conclusions

The various tests in here demonstrate that there is no best solution for all cases. NHibernate is a full featured ORM and for that it has the most difficult learning curve of all 5 technologies. It can provide good results if it is well optimized, despite the overly complex multi-layered architecture (ORM and relational database). EF performs similarly to NHibernate, although less feature rich and easier to learn. **CazDAL** is fast overall due to its close-to-relational API, although it does not provide much useful features of the ORM frameworks or OODBMSs. Even though ODBs still support a very small niche market, DB4O proves it can overcome RDBMSs in certain aspects like its simplicity and performance on join operations.

Chapter 7

Conclusion

The aim of this dissertation has been to discuss the persistence problem on object oriented applications and find the best solutions. The main focus lies on the Object Relational Mapping (ORM) limitations, patterns, technologies and alternatives.

7.1 Developed work

This dissertation was developed in Cachapuz under the project Global Weighting Solutions (GWS). Essentially, the objectives of GWS were focused on finding the optimal persistence layer for CazFramework, mostly providing database interoperability with close-to-Structured Query Language (SQL) querying. Recall the CazFramework is a generic solution working on top of legacy databases and its Data Access Layer (DAL) code had previously been generated with ClassBuilder, a custom ORM tool developed by Cachapuz. Overall, this dissertation described various approaches to achieving GWS' goals, from enhancing ClassBuilder to adopting a different ORM tool.

Chapter 1 provided an overview of CazFramework, introduced to the ORM problem and delineated the objectives of this work. In chapter 2 both the object and the relational paradigms were explained before identifying the object-relational mismatch. Here, some alternatives such as Object-Oriented Database Management Systems (OODBMSs) were presented, as well. Additionally, chapter 2 suggested that ORM can be addressed as a paradigm (based on Kuhn's book [Kuh96]). Chapter 3 defined the most relevant patterns and practices that drive ORM implementations. In chapter 4, two popular .NET ORM frameworks (NHibernate and Entity Framework (EF)) were analysed against requirements specified by Cachapuz and, with the help of ORM design patterns from the previous chapter.

Because the analysed ORM frameworks had not provided the expected functionality, the enhancement of ClassBuilder was the following step. That led to the development of CazDataProvider, the software artefact of this dissertation, which is discussed in chapter 5. Before explaining its architecture and implementation, an analysis on ClassBuilder and ADO.NET providers proved essential, in chapter 5, to fulfil the objectives. ClassBuilder was identified as a good alternative to some ORM problems like Dual-Schema and Partial-Object mostly due to its Relational Domain Model approach. The other sections provided insight on how CazDataProvider had enabled database interoperability features with dynamic querying relying only on the relational schema. Some issues arose, mostly due to ADO.NET data provider and SQL dialect idiosyncrasies. Also, CazDataProvider was complemented with LinqToCaz, a Language Integrated Query (LINQ) implementation for query criteria features.

Chapter 6 presented a set of performance tests run on ORM frameworks (NHibernate and EF) , OODBMSs (DB4O and **Eloquera**) and **CazDAL** (ClassBuilder with CazDataProvider).

7.2 Discussion

Even though ORM technology has been advancing fast in the last few years, the ORM paradigm might or might not be a good solution depending on a number of factors like the know-how of the developers and the domain logic complexity. In the case of Cachapuz, ORM was used but only up to a certain extent like that of ClassBuilder's Relational Domain Model.

CazDataProvider is a good solution for CazFramework to provide database interoperability and dynamic query features. Nevertheless, it still needs to support more Relational Database Management Systems (RDBMSs) and data providers, better LINQ support and continuous maintenance. Another disadvantage is the effort to produce something that already exists, i.e. database-agnostic features are already implemented and supported by other ORM frameworks like NHibernate or EF, even though not in the same way.

From the tested technologies in this dissertation, OODBMSs are a good alternative to ORM for its simplicity if performance is not an issue. ORM frameworks like NHibernate tend to be overly complicated towards others like EF, although in some cases it might compensate the extra effort to learn them. Also, typical ORM frameworks respond well to complex domain logic if they are allowed to take control over databases. Otherwise, ClassBuilder and CazDataProvider is a good solution for it produces lesser overhead and adapts better to legacy databases.

All in all, the content of this dissertation delivers a guide for developers who are on the verge of implementing or adopting an ORM tool and hope to find answers for their doubts or problems before taking unnecessary risks. Perhaps they can find in ClassBuilder and CazDataProvider the answers they have been looking for.

Appendix

Appendix A

Relational Database Mechanics

One essential step to better comprehend the Object Relational Mapping (ORM) patterns, is to acknowledge how well a Relational Database Management System (RDBMS) performs when submitted to a various range of operations. Also, because an ORM mediates the process of data retrieval for the application via Structured Query Language (SQL) statements, it is important to know how to write them in the most optimal way possible.

Throughout the last decades, querying data became more and more complex greatly based on the growth of data schemas. The continuous evolution of SQL is proof of the effort to alleviate this problem. However, the whole work of data retrieval in a RDBMS goes far beyond SQL. The ability of fulfilling such operations in the most fast and consistent way is often definite to choose one database system over another.

Once the SQL query is compiled, the database optimizer delineates an appropriate execution plan. During the compilation and optimization, the plan assembles a set of physical operators in the best possible order so that the result can be delivered efficiently. Often the optimizer has to make algorithmic decisions such as whether to enforce a hash-join followed by a sort or a sort-merge join.

Disk access is the primary concern for optimizing database applications. Therefore, the dominant time cost for the physical operators is measured in disk input/outputs (I/Os). Other important variables include CPU cost and main memory size available. The smallest unit of physical or cache storage is a block or page. The database organizes table rows in blocks, usually 2KB-16KB of size. Generally each block contains from less than one hundred to a few hundred rows [Tow03].

Databases also rely on caching to avoid unnecessary physical I/Os. Thus the block buffer cache is a shared memory segment containing recently used blocks, accessible through logical I/O. That means any user can take advantage of cached blocks placed there by any other user. This cache organizes the most recently used blocks at the head of the list and the least recently used at the tail. The most recently used blocks are the fastest, allowing its access to be 30 to 200 times faster than physical I/O [Tow03].

Physical table growth and ageing is one more concern of databases. Whether to have a continuous growth or purge (delete) rows in a table can greatly impact caching and physical access.

The primary physical operators of the execution plan only perform access to the tuples of a relation which are: table-scan and index-scan. A table-scan inspects the tuples of a relation by single or multi block reads. An index-scan uses an index structure to find tuples through leaf blocks on B-tree indexes or hash buckets on hash-based indexes. These concepts are described in detail, below.

A.1 Full table-scans

The full table-scan consists of reading a whole table without an index. Thus, it is common for most databases to request physical I/Os that read multiple blocks each at a time, rather than a single block. However, file systems work with much larger block sizes than physical blocks mostly due to fragmentation issues, which makes database multi-block I/O not so critical. Also, because a multi-block read is an atomic operation, even if all the blocks are cached except one, the database still requests physical I/O for all the blocks [Tow03].

Large full table-scans generally put its rows in the tail of cache buffer because they are not likely to be needed often.

The I/O cost of a full table-scan is linear and proportional to the number of blocks in use to make the relation persistent.

A.2 Indexes

On the other hand there is Indexed table access. To match exact values or a range of values requested by a certain query criteria, the database starts at the root block of the according index tree with the exact value or the value that delineates the beginning of the range. From there it descends deeper into the tree through branch blocks and leaf blocks as it converges to the desired value. Once the leaf block that best meets the criteria is found, the database follows each `rowid` to the respective table block.

The most common type of index is the B-tree index which is a balanced tree structure in which a node can have two or more children. It has a natural sort order and is effective when query criteria contains equality or range conditions. In these conditions, function-based indexes are not supported, i.e. the index is often disabled if functions are applied to the indexed column rather than the constant. For instance, the condition `UPPER(Last_Name) LIKE 'SMITH%'` will not apply the index. Indexes use `rowids` as pointers to the actual rows in the table. A `rowid` is composed of a block address and a row number [Tow03].

A B-tree index structure has one single root block, followed by branch blocks and leaf blocks at the bottom. A single leaf block has up to 300 index values. Each index value is composed by both the value of indexed column and the pointer or `rowid`. If a B-tree index is two-deep, it points at 300 to 90,000 rows of a table and its root block points at up to 300 leaf blocks, without using branch blocks. If the table has less than 300 rows, then the root block is a leaf block itself. If the table has more than 90,000 rows, the index B-tree is three-deep or more [Tow03].

Every leaf block has a pointer to the next leaf block sorted by the indexed values. From here, index range scans read as many as the number of leaf blocks that meet the query range condition. It is often enough for medium-sized range scans to only touch one single leaf block given that each holds 300 values.

The problem of inefficiency and ambiguity left by null values greatly affects index performance. This problem is handled differently among the database vendors. For instance, Oracle indexes do not contain entries that point at rows having null values for their indexed columns. So an index on a mostly null column can be very small, even on a very large table.

The I/O cost of a single lookup in an index B-tree has logarithmic complexity. The higher the factor of child nodes per node the lesser disk I/O reads performed. CPU cost will, however increase. For example, a lookup in a binary tree that has 2 sub-ranges on every branch, costs $\log_2 90,000 \sim 16$ jumps or I/O reads (plus one I/O read for the actual row) but only performs one condition check per jump. In a B-tree with 300 sub-ranges on every branch, it will only make $log_{300} 90,000 = 2$ jumps but a linear cost of up to 299 condition checks per jump. Branch conditions can be optimized to have a logarithmic cost, though. In the end, both the algorithms have similar CPU cost, but the B-tree is better prepared to reduce disk I/O when the structure does not fit entirely in memory. A range scan takes as much cost as a single lookup to get to the starting row. From there it requires one I/O read per table row access.

Inserting a large number of new rows can be painful, for each insert forces the index to rebalance itself. The delete operations provoke empty blocks which are less efficient to cache. An update is usually the same as an insert of new value and delete of old value. This is less problematic for quiet tables or when the index is on the primary key column, which implies the values do not change. When there are many pending changes (inserts, updates or deletes) that can be run at once, it is advised to drop the index, make the changes and then rebuild it.

Clustered indexes are the same as regular indexes except they physically arrange the table rows, sorted by the key value and sequentially on disk. Therefore a table can only have one clustered index. At the bottom of the tree the leaf blocks point at the physical rows. For that reason, there is no need for `rowids` which allows reading more rows per block. Also, the disk overhead spent in range scans to retrieve each physical row, is avoided in clustered indexes, through optimal caching.

Multi-table clusters can also improve the performance of joins between tables.

A.3 Indexed access Vs Table-scan

Overall it is imperative to draw a comparison of performance on both indexed access and table-scan, which is provided by the following example. Considering a 40-block table of typically 3,200 rows at which it is executed a query criteria that defines a value range only met by 5 rows. If the database performs an index range scan, as it is a small table, the index B-tree is only two-deep. It is likely that all 5 `rowids` will be found on the same leaf block. Although, if the starting value of the range is at the end of one leaf block, an extra hop will be required to the next leaf block for completing the range. From this point, the database accesses the table blocks of all 5 `rowids` found. Once again it is likely that all 5 rows are close together, and thus a single logical I/O read to a table block is enough to cache all 5 rows at once. However that depends on the clustering factor which determines how well are the rows ordered in conformity with the index [Tow03].

In the same example, a full table-scan performs 5 multi-block I/O reads of 64KB each across all the 3,200 rows, assuming each block has 8KB and about 10 rows. At the same time, the CPU is responsible for checking and discarding all 3,200 rows, throughout 40 blocks, but the 5 rows that match the query criteria.

For the indexed access, at the worst clustering factor, caching is useless. Thus it requires 7 physical I/O reads, 2 for finding the leaf block in the B-tree and 5 for accessing individual table blocks. For small tables and small indexes such as in this case, caching is likely to be effective. Also, with the full table-scan, 5 logical I/O reads are alone more expensive than the 7 logical I/O reads of the index scan [Tow03]. Apart from that, there is a CPU load in the full table-scan that covers all 3200 rows and is avoided with the index plan. In a real test this example would be fast enough to let the difference between both plans pass unnoticed. Larger tables would, however, be more expensive in full table-scans due to its linear complexity as opposed to logarithmic from index tree access.

It is not always obvious whether indexed access performs better than full table-scan. Moreover, the database optimizer does not always make the right decision. Thus it is estimated that if the result hands more than 20% of the rows, the full table-scan outperforms the indexed access and if less than 0.5% otherwise. If between 0.5% and 20% rows are returned it will depend on the clustering factor and how well the cache performs [Tow03]. This characteristic of a single condition that identifies the percentage of filtered rows from the total, is called selectivity.

When the query criteria holds two conditions it is possible to achieve the same result with different costs through different strategies such as one multicolumn index, two independent indexes or one single index. However a more complex solution is often not worth the effort.

A.4 Joins

The most interesting problems of optimization occur on multi-table queries. The high variety of SQL join operations and algorithms confirms that. Typically a join takes two input relations and outputs a single relation as a result of the association of the other two. Among the least valuable is the Cartesian join despite it being often misused through deceiving SQL syntax.

The most used are the inner joins and the outer joins. The generic algorithm for these two joins consist of primarily scanning all the rows of the driving table. Also a search is performed during or after the previous step, in order to find all the rows in the other table that match the join condition with the rows in the driving table. Finally it arranges a result with the joined rows. The difference between an inner join and an outer join is that the latter returns all the rows in the driving table, even those that do not meet the join condition with any row in the other table.

Though there are some variations among the database vendors regarding join implementations, the execution plan generally always ends up choosing one of the three join algorithms: nested-loop join, hash-join or sort-merge join. The book "Database Systems The Complete Book" ([UGMW01]) was essential to fulfil the research on these three algorithms.

A.4.1 Nested-loop join

The earliest join algorithm remaining from the primordials of relational databases is called nested-loop join. Its basic implementation defines an outer loop that iterates over every tuple in the driving table. Within this outer loop, at least one nested loop iterates through all the tuples of the other table while verifying the join condition and filtering the resulting rows. This is called tuple-based nested-loop join and it covers both the relations tuple by tuple, careless of block buffering [UGMW01]. Consider the following example focusing on the cost of performing a join operation between the relations `R` (driving table) and `S` (other table). `R` has 3,000 rows throughout 10 blocks and `S` has 6,000 rows throughout 20 blocks which makes a rate of 300 rows per block.

Without caching, the tuple-based algorithm makes $Rows(R) \times Rows(S) = 3,000 \times 6,000 = 18,000,000$ I/Os. With block buffering, however, it makes $Blocks(R)+Rows(R)\times Blocks(S) = 10 + 3,000 \times 20 = 60,010$ logical I/Os, which would be more acceptable. In rare perfect caching scenarios it would only make $Blocks(R) + Blocks(S) = 30$ physical I/Os.

The tuple-based algorithm is not used in RDBMS for it has been fairly optimized and thus giving place to the block-based nested-loop join. The block-based algorithm reads each relation block by block while using a memory buffer. This buffer has the size of `M` blocks and only stores tuples from smallest relation in the outer loop. Thus, the outer loop iterates on the relation `R`, $\frac{Blocks(R)}{M}$ times while performing the same amount of physical I/Os. The inner loop iterates on every tuple of the relation `S` reading block by block. Within this inner loop the memory buffer, containing `M` blocks of relation `R`, is iterated to finally match the join condition with current tuple of `S`. In some cases (e.g. hash join), the buffer is organized into a searching structure such as a hash table, where the buffer iteration is avoided [UGMW01].

Hence, considering the memory buffer stores up to 5 blocks, the block-based nested-loop join performs $Blocks(R) + \frac{Blocks(R)}{M} \times Blocks(S) = 10 + \frac{10}{5} \times 20 = 50$ logical I/Os. If the blocks of `R` could fit entirely in memory buffer ($M = Blocks(R)$), this algorithm would perform 30 logical I/Os.

Withal, a nested-loop join is a simple and robust algorithm for it can deliver huge result sets even for huge arguments without ever running out of memory. When the main memory is not enough to fit the join arguments or input relations, two-pass algorithms are generally more efficient than nested-loop joins. They are either sort-based, hash-based or index-based and require disk as an intermediary memory [UGMW01].

A.4.2 Hash join

The one-pass hash join operates in similar steps with block-based nested-loop of memory buffer the size of the smallest relation and using a searching structure to avoid iterating on the buffer. For that matter, hash-based algorithms use a hash function to spread the argument tuples into single buckets in a hash table structure.

In this algorithm, the optimizer begins by comparing both relations and estimating which is expected to return less relevant tuples to the joined `rowset`. Then it takes this smaller `rowset` to calculate a hash table based on its join key. This means running over every single row, perform the hash function for each and insert it into the correspondent hash bucket.

The ideal hash algorithm prevents collisions, allocating each row to a different bucket so that the lookup cost is constant. For that matter the optimizer calculates the best hash function algorithm possible to perform the operation. The simplest hash join expects the smallest relation to engender a hash table that can fit entirely into memory. Although, if otherwise, a two-pass algorithm is the resorting solution to write that structure into disk and then retrieve it by chunks.

Once the hash table is built, the second relation is retrieved from disk by blocks. Then, for each row on this second relation, a hash lookup is performed on its join key. The hash code leads to a hash bucket in the hash table, containing the row of the first relation with same join key. Although in case of collisions the process is not that simple. In the end the joined rows are retrieved and the rows with no match on both relations are discarded in an inner join scenario.

If the hash structure fits in memory the I/O cost of this algorithm is $Blocks(R) + Blocks(S)$. Otherwise, there is an additional cost, of reading all the blocks of `R` and persist the same blocks within a hash structure, before performing the actual join. Thus the two-pass algorithm takes $3 \times Blocks(R) + Blocks(S)$ physical I/Os . The performance of this algorithm also relies on the hash lookup cost and concerns the I/O operations spent on unnecessary rows [UGMW01].

In the end, the variables that affect the use and efficiency of hash-joins are:

- Randomness. Since the returning rows of a hash join are expected to be in random order, this algorithm should be avoided if sorting is needed.
- Selectivity. When point queries are more relevant than range queries, the hash-based algorithms are preferred over index-based B-tree algorithms. In such cases, joins are favoured for they can create many point queries.
- Size of smallest relation. Hash-based algorithms such as hash-joins, require the smallest argument to be small so that they can avoid the additional disk I/Os of the two-pass. In these cases hash-joins are strictly more efficient than nested-loop joins.

A.4.3 Sort-merge join

The sort-merge join merges two previously sorted input relations and outputs the tuples that match the join condition.

A sort-based algorithm can be as simple as an in-memory sort, although most cases require a two-pass. A two-pass sort consists of slicing the argument into memory-sized sub-lists, i.e. it reads `M` blocks of a relation into the available buffers on every iteration and until all blocks are read. Each time, the sub-list inside the buffer, is sorted in-memory with an efficient one-pass algorithm. An in-memory sort can have slightly more than a linear cost, though it is not expected to exceed the I/O block reads. After the in-memory sort, the sub-list is written into `M` blocks of the disk. This step is necessary because relations are often too large to fit entirely in memory. Once every sorted sub-list is persisted, they are all brought up together to as many buffers, except only loading one block for each sub-list. Thus, the merging process can be done into a single sorted list, in a single step.

The final step of the two-pass sort-merge join occurs the same ways as the two-pass sort-based algorithm, except it has two already sorted inputs, thus enabling reading more blocks of both into memory.

The algorithm of the merge step for a sort-merge join between relations `R` and `S` is given by a loop which ends when the end of relation `S` is reached. Additionally the buffers have to be previously set to iterate over the relation `R` and `S`. Inside the loop, if the current iterated tuples of `R` and `S` do match the join condition, they will be sent to the output. If the `R` tuple join key is smaller or equal to the `S` tuple join key, then the R iterator advances to the next tuple. Otherwise, the S iterator advances. Note that in case of multiple `S` tuple matches with current `R` tuple, they all must reside in-memory so that they can be checked with other `R` tuples.

Generally, the sort-merge join performs efficiently because the second input or relation `S` has a join key which is a primary key or at least unique. However, at the extreme there is only one join key value throughout the operation and every tuple of relation `R` joins with every tuple of `S`. Since every tuple of `S` has the same key, not only sorting becomes useless but also all the `S` tuples must be kept in-memory. This is normally not possible because memory buffers do not often cover medium sized tables, which leaves the nested-loop join approach to be the obvious choice for the optimizer. Note that this rare scenario enforces the so-called product or Cartesian join as it outputs $Rows(R) \times Rows(S)$ rows.

Furthermore, this sorting and merging algorithm is most used in duplicate elimination or grouping. For duplicate elimination, the merge takes two sorted sub-lists generally into two buffers holding the current block of each. The following step is to identify all the tuples on both relations to have the same sort key, discarding all others.

The cost of a sort-merge join is of $3 \times (Blocks(R) + Blocks(S))$ physical I/O operations in which the first $2 \times (Blocks(R) + Blocks(S))$ is for sorting only [UGMW01].

In the end, sorted-based algorithms should be favourable when sorting or duplicate elimination is required. Otherwise, for instance, a sort-merge join is dependent on both sizes of the arguments to fit in main memory in order to avoid two-pass . The hash-based algorithms only rely on the size of the smallest relation, thus outperforming the sort-based when sorting is not required [UGMW01]. Regarding two-pass algorithms, data variability can greatly effect the performance of a sorting algorithm.

A.4.4 Index-based join

Index-based algorithms are very efficient for conditions of equality and range when there is low selectivity. Index-based joins are similar to hash joins except they already have a pre created structure on disk, plus the advantage of sorting by the indexed columns. This join is efficient when one argument is very small and the other has an index with high variability of values.

If both the arguments have a sorted index on the join key, they can be served as inputs to a sort-merge-join. Towards relieving the sorting cost, with B-tree indexes for the join key on both relations, it is possible to perform a sort-merge join that avoids the first steps of sorting each relation separately. Additionally, only the index structure of the driving table or relation `R` is fully iterated in memory, while the tuples of `S` that find no match on the join key are never loaded.

Eventually, the index-based join performs less iterations than nested-loop or hash join and as a consequence it avoids reading unnecessary rows. While the other two algorithms perform $Rows(R) \times Rows(S)$ iterations for join condition checks, the index-based join only iterates $Rows(R)$ times despite each iteration costing more.

Appendix B

Testing EF querying: Examples of code and SQL generated

This appendix provides VB.NET code for using Entity Framework (EF) querying API and the SQL generated by such code. The many examples in here are discussed in section 4.1.5.

B.1 Basic querying

This section presents 9 EF querying examples discussed in 4.1.5.2 regarding their efficiency and quality of SQL generated.

1. EQL 1 example

Code:

```
cazEntitiesContext.CreateQuery(Of User)("SELECT VALUE u
FROM cazEntities.User AS u WHERE u.Id = 1").First
```

SQL generated:

```
SELECT ‘Extent1‘.‘Id‘, ‘Extent1‘.‘Name‘
FROM ‘User‘ AS ‘Extent1‘
WHERE ‘Extent1‘.‘Id‘ = 1 LIMIT 1
```

2. EQL 2 example

This example pre-creates the query before executing it `n` times.

Code:

```
Dim q As System.Data.Objects.ObjectQuery(Of user) =
cazEntitiesContext.CreateQuery(Of user)("SELECT VALUE u FROM
cazEntities.user AS u WHERE u.Id = 1") 'exectuted 1 time

q.Execute(Objects.MergeOption.NoTracking).First() 'exectuted 10^n times
```

SQL generated:

```
SELECT `Extent1`.`Id`, `Extent1`.`Name`
FROM `User` AS `Extent1`
WHERE `Extent1`.`Id` = 1
```

3. Linq-to-Entities 1 example

This example uses dynamic Linq-to-Entities.

Code:

```
cazEntitiesContext.user.Where("it.Id = 1").First()
```

SQL generated:

```
SELECT `Extent1`.`Id`, `Extent1`.`Name`
FROM `User` AS `Extent1`
WHERE `Extent1`.`Id` = 1 LIMIT 1
```

4. EF API 1 example

Code:

```
cazEntitiesContext.GetObjectByKey(New EntityKey(
"cazEntities.User", "Id", 1))
```

SQL generated:

```
SELECT `Extent1`.`Id`, `Extent1`.`Name`
FROM `User` AS `Extent1`
WHERE `Extent1`.`Id` = 1
```

5. EF API 2 example

Code:

```
cazEntitiesContext.GetObjectByKey(New EntityKey(
"cazEntities.User", "Id", i)) 'so that i goes from 1 to 10^n
```

SQL generated:

```
SELECT ‘Extent1‘.‘Id‘, ‘Extent1‘.‘Name‘
FROM ‘User‘ AS ‘Extent1‘
WHERE ‘Extent1‘.‘Id‘ = i
```

6. Linq-to-Entities 2 example

This examples takes advantage of lambda expressions.

Code:

```
cazEntitiesContext.User.Where(Function(c) c.Id = 1).First()
```

SQL generated:

```
SELECT ‘Extent1‘.‘Id‘, ‘Extent1‘.‘Name‘
FROM ‘User‘ AS ‘Extent1‘
WHERE ‘Extent1‘.‘Id‘ = 1 LIMIT 1
```

7. Native-SQL 1 example

Code:

```
Dim conn As New MySqlConnection
conn.ConnectionString =
    "server=localhost;User Id=root;database=caz"
'exectuted 1 time

conn.Open()
Dim sql = "SELECT ‘Extent1‘.‘Id‘, ‘Extent1‘.‘Name‘ " & _
    "FROM ‘User‘ AS ‘Extent1‘ WHERE ‘Extent1‘.‘Id‘ = 1;"
Dim cmd As MySqlCommand = New MySqlCommand(sql, conn)
conn.Close()
'exectuted 10^n times
```

SQL generated:

```
SELECT ‘Extent1‘.‘Id‘, ‘Extent1‘.‘Name‘
FROM ‘User‘ AS ‘Extent1‘
WHERE ‘Extent1‘.‘Id‘ = 1
```

8. Native-SQL 2 example

Code:

```
Dim conn As New MySqlConnection
conn.ConnectionString =
    "server=localhost;User Id=root;database=caz"
conn.Open()
'exectuted 1 time

Dim sql = "SELECT ‘Extent1‘.‘Id‘, ‘Extent1‘.‘Name‘ " & _
    "FROM ‘User‘ AS ‘Extent1‘ WHERE ‘Extent1‘.‘Id‘ = 1;"
Dim cmd As MySqlCommand = New MySqlCommand(sql, conn)
'exectuted 10^n times

conn.Close()
'exectuted 1 time
```

SQL generated:

```
SELECT ‘Extent1‘.‘Id‘, ‘Extent1‘.‘Name‘
FROM ‘User‘ AS ‘Extent1‘
WHERE ‘Extent1‘.‘Id‘ = 1
```

9. Native-SQL 3 example

Code:

```
Dim conn As New MySqlConnection
conn.ConnectionString =
    "server=localhost;User Id=root;database=caz"
conn.Open()
Dim sql = "SELECT ‘Extent1‘.‘Id‘, ‘Extent1‘.‘Name‘ " & _
    "FROM ‘User‘ AS ‘Extent1‘ WHERE ‘Extent1‘.‘Id‘ = 1;"
Dim cmd As MySqlCommand = New MySqlCommand(sql, conn)
cmd.Prepare()
'exectuted 1 time

cmd.Parameters.AddWithValue("id", i)
Dim m = cmd.ExecuteReader()
m.Close()
'exectuted 10^n times

conn.Close()
'exectuted 1 time
```

SQL generated:

```
SELECT ‘Extent1‘.‘Id‘, ‘Extent1‘.‘Name‘
FROM ‘User‘ AS ‘Extent1‘
WHERE ‘Extent1‘.‘Id‘ = i
```

B.2 Eager and Deferred Loading

This section presents 3 querying examples of eager loading and 2 of deferred loading strategies in EF. These examples are discussed in 4.1.5.3 regarding its performance and SQL generated code.

Eager loading

1. **EQL 1** Code:

```
cazEntitiesContext.CreateQuery(Of User)("SELECT VALUE u " & _
"FROM cazEntities.User AS u WHERE u.Id = " & 1).Include("Profile").First
```

2. **Linq-to-Entities 1** Code:

```
cazEntitiesContext.User.Where("it.Id = " & 1) _
                       .Include("Profile") _
                       .First()
```

3. **Linq-to-Entities 2** Code:

```
cazEntitiesContext.User.Include("Profile") _
                       .Where(Function(c) c.Id = 1) _
                       .First()
```

SQL generated:

```
SELECT ‘Project2‘.‘Id‘,‘Project2‘.‘Name‘, ‘Project2‘.‘C1‘,
‘Project2‘.‘C3‘ AS ‘C2‘, ‘Project2‘.‘C2‘ AS ‘C3‘,
‘Project2‘.‘Field1‘, ‘Project2‘.‘Field2‘, ‘Project2‘.‘Desc‘,
‘Project2‘.‘Id1‘, ‘Project2‘.‘UserId‘
FROM (
    SELECT ‘Limit1‘.‘Id‘, ‘Limit1‘.‘Name‘, ‘Limit1‘.‘C1‘,
    ‘Extent2‘.‘Field1‘, ‘Extent2‘.‘Field2‘, ‘Extent2‘.‘Desc‘,
    ‘Extent2‘.‘Id‘ AS ‘Id1‘, ‘Extent2‘.‘UserId‘,
    CASE WHEN (‘Extent2‘.‘Id‘ IS NULL) THEN (NULL)
```

```
                                       ELSE (1) END AS ‘C2‘,
    CASE WHEN (‘Extent2‘.‘Id‘ IS NULL) THEN (NULL)
                                       ELSE (1) END AS ‘C3‘
    FROM (
        SELECT ‘Extent1‘.‘Id‘, ‘Extent1‘.‘Name‘, 1 AS ‘C1‘
        FROM ‘User‘ AS ‘Extent1‘
        WHERE ‘Extent1‘.‘Id‘ = 1 LIMIT 1) AS ‘Limit1‘
    LEFT OUTER JOIN ‘Profile‘ AS ‘Extent2‘
    ON ‘Limit1‘.‘Id‘ = ‘Extent2‘.‘UserId‘) AS ‘Project2‘
ORDER BY ‘Id‘ ASC, ‘C3‘ ASC
```

Deferred loading

4. EF API with deferred load Code:

```
Dim u As User
u = cazEntitiesContext.GetObjectByKey( _
    New EntityKey("cazEntities.User", "Id", 1))
u.Profile.Load()
```

5. Linq-to-Entities 1 with deferred load Code:

```
Dim u As User
u = cazEntitiesContext.User.Where("it.Id = " & 1).First()
u.Profile.Load()
```

SQL generated:

```
SELECT ‘Extent1‘.‘Id‘, ‘Extent1‘.‘Name‘
FROM ‘User‘ AS ‘Extent1‘
WHERE ‘Extent1‘.‘Id‘ = 1 LIMIT 1
--exectuted 1 time
--’LIMIT 1’ only executed in Linq-to-Entities 1 example

SELECT 1 AS ‘C1‘, ‘Extent1‘.‘Field1‘, ‘Extent1‘.‘Field2‘,
‘Extent1‘.‘Desc‘, ‘Extent1‘.‘Id‘, ‘Extent1‘.‘UserId‘
FROM ‘Profile‘ AS ‘Extent1‘
WHERE (‘Extent1‘.‘UserId‘ IS NOT NULL) AND (‘Extent1‘.‘UserId‘ = 1)
--exectuted 10^n times
```

Bibliography

[AMC01] D. Alur, D. Malks, and J. Crupi. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[Bec97] K Beck. *Smalltalk: best practice patterns*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[BK06] C. Bauer and G. King. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA, 2006.

[BW99] P. Bishop and N. Warren. Java in practice: Design styles and idioms for effective java, 1999.

[Cat11] R.G.G. Cattell. Scalable sql and nosql data stores. *SIGMOD Rec.*, 39(4):12–27, May 2011.

[CB00] R.G.G. Cattell and D.K. Barry. *The object data standard: ODMG 3.0*. Morgan Kaufmann Pub, 2000.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13:377–387, June 1970.

[Dat04] C. J. Date. *An Introduction to Database Systems*. Pearson Addison-Wesley, Boston, MA, 8. edition, 2004.

[Dat07] C.J. Date. *Logic and Databases: The Roots of Relational Theory*. Trafford on Demand Pub, 2007.

[dpb08] dpblogs. Sampleedmxcodegenerator sources @ONLINE. `http://blogs.msdn.com/b/adonet/archive/2008/01/24/sampleedmxcodegenerator-sources.aspx`, January 2008.

[Fow02] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[Fus97] M.L. Fussell. Foundations of object relational mapping. *White Paper*, 1997.

[GHJV95] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[GL02] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[How04] R. Howard. Provider model design pattern and specification, part 1 @ONLINE. `http://msdn.microsoft.com/en-us/library/ms972319.aspx`, March 2004.

[KBA$^+$12] G. King, C. Bauer, M.R. Andersen, E. Bernard, S. Ebersole, and H. Ferentschik. Hibernate reference documentation v3.6.10. 2012.

[KBHK09] P.H. Kuaté, C. Bauer, T. Harris, and G. King. *NHibernate in action*. Manning Pubs Co Series. Manning, 2009.

[Kim90] W. Kim. *Introduction to object-oriented databases*. 1990.

[Kuh96] T.S. Kuhn. *The structure of scientific revolutions*. University of Chicago press, 1996.

[Ler09] J. Lerman. *Programming Entity Framework*. O'Reilly Media, Inc., 1st edition, 2009.

[Mal05] S. Malik. *Pro ADO.NET 2.0*. Apresspod Series. Apress, 2005.

[Mar03] R.C. Martin. *Agile software development: principles, patterns, and practices*. Alan Apt series. Prentice Hall, 2003.

[MH03] M. MacDonald and B. Hamilton. *ADO.NET in a Nutshell*. O'Reilly Media, Inc., 2003.

[Nay08] K. Nayyeri. How to write a provider model @ONLINE. `http://dotnetslackers.com/articles/designpatterns/HowToWriteAProviderModel.aspx`, January 2008.

[New06] T. Neward. The vietnam of computer science. *Blog post, The Blog Ride, Ted Neward's Technical Blog (June 2006)*, 2006.

[Nur05] P. Nurani. Object relational mapping. 2005 Georgia Oracle Users Conference, 2005.

[SSRB00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[Sta11] NHibernate Staff. Nhibernate reference documentation. In *http://nhforge.org/doc/nh/en/index.html*. nforge, 2011.

[Sto05] M. Stonebraker. One size fits all: An idea whose time has come and gone. In *Proceedings of the International Conference on Data Engineering (ICDE*, pages 2–11, 2005.

[Tow03] D. Tow. *SQL tuning - generating optimal execution plans: covers Oracle, DB2 and SQL server*. O'Reilly, 2003.

[UGMW01] J. D. Ullman, H. Garcia-Molina, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

[VZ10] P. Van Zyl. Performance investigation into selected object persistence stores, 2010.

Glossary

.NET The Microsoft platform .NET. i, iii, xii, 2, 5, 7, 29, 41, 46, 49, 60, 62, 72, 74, 84, 85, 89, 90, 103–105, 107, 113, 135, 137, 147–149, 151, 165, 173, 174, 179, 189

Global Weighting Solutions Global Weighting Solutions is a project aiming to constitute an unified software development approach for all the enterprises in Europe that belong to Bilanciai Group. 220

`C++` `C++` programming language is an enhancement of C language. 10

`C#` `C#` object-oriented programming language for .NET. 8, 10, 28, 38, 85, 94, 96, 97

3-Tier Three Tier Architecture (Presentation, Business and Data). 56

Abstract Factory GoF creational pattern. 113, 146, 150, 155, 176

Active Record PoEAA data source architectural pattern. xi, 43, 54, 62–65, 78, 81

Adapter GoF structural pattern. 44, 92

ASP.NET Microsoft's .NET Web application framework. 50

Association Table Mapping PoEAA object-relational structural mapping pattern. 27, 65, 118

C C programming language. 10, 11

Cachapuz Cachapuz Weighting Solutions (based in Braga, Portugal). i, iii, v, 2, 3, 7, 89–91, 102, 132, 133, 135, 136, 144–146, 153, 163, 176, 189, 190

CazDataProvider DAL abstraction layer proposed in this dissertation, to be used in CazFramework. i, iii, xiii, 5, 6, 88, 135, 136, 147, 149, 151, 155, 156, 158, 161, 163, 165–171, 173, 175–177, 179, 190, 191

CazFramework The solutions framework built and maintained in Cachapuz and inserted in Project GWS. i, iii, 2, 3, 5, 8, 102, 135–137, 145–147, 149, 151, 153, 155, 156, 163, 165, 166, 177, 189, 190

ClassBuilder It is an ORM tool maintained by Cachapuz. xii, 3, 5–8, 88–90, 134–147, 151, 155, 156, 166, 176, 177, 179, 189–191

Data Mapper PoEAA data source architectural pattern. xi, 54, 55, 57, 62, 64–68, 70, 71, 73–76, 78, 80, 82, 85, 86, 88, 91, 92, 105, 106, 137–140, 143, 144, 176

DataAdapter Class from ADO.NET API. 50

DataReader Class from ADO.NET API. 46, 52, 61, 133, 147

DataRow Class from ADO.NET API. 50, 52

DataSet Class from ADO.NET API. xi, 47–53, 56, 60–62, 74, 86, 88, 90, 137, 147

DataTable Class from ADO.NET API. 48–50, 52, 147

Decorator GoF structural pattern. xii, 79, 113, 159–161, 176

Dependent Mapping PoEAA object-relational structural mapping pattern. 27, 65, 75

Domain Model PoEAA domain logic pattern. i, iii, xi, 29, 44, 47, 54–62, 64–69, 74, 76, 81, 82, 84, 86, 88, 90, 105, 106, 110, 115, 126, 137

dotConnect Database connectivity solution in ADO.NET supporting integration with some relevant databases. 91, 96

Dual-Schema ORM problem described by Neward. 27, 41, 58, 64, 65, 90, 133, 146, 176, 190

Entity Bean J2EE server side component EJB which represents a persistent data object. 56, 57

EntityManager JPA class that implements the Entity Manager pattern. 29

Explicit Initialize OO pattern [Bec97] implementing an API, explicitly called by the client, to fill the properties or associations of an object after it has been first created. 77, 100, 140, 145

Facade GoF structural pattern. xii, 44, 55, 61, 73, 152, 153

Factory Method GoF creational pattern. 44, 47, 67, 71, 72, 78, 80, 96, 150, 151, 153, 154, 156, 158, 159, 161, 166, 168, 176

Foreign Key Mapping PoEAA object-relational structural mapping pattern. 27, 63, 65

Gateway PoEAA base pattern. 44, 46, 61

Ghost An object state used in Lazy Load (PoEAA). 32, 77, 80, 81, 109, 110

Hibernate ORM framework for Java. 1, 4, 16, 17, 28, 30, 31, 55, 57, 73, 75, 84–86, 104, 105, 108

Identity Field PoEAA object-relational structural mapping pattern. 29, 63

Identity Map PoEAA object-relational behavioural pattern. 33, 55, 65, 67, 69–71, 73, 75, 76, 81, 88, 90–92, 105, 106, 108, 119, 136, 141, 146

Interpreter GoF behavioural pattern. xiii, 85, 163, 164, 173–176

Java The Java software platform from Sun Microsystems. 8–10, 28, 29, 35, 38, 41, 46, 52, 61, 62, 72, 84, 85, 104, 179

Layer Supertype PoEAA base pattern. 67, 70, 92

Lazy Initialization OO pattern [Bec97] for delaying the creation of an object instance to the time it is actually accessed. xii, 32, 51, 52, 77, 78, 81, 86, 87, 153

Lazy Load PoEAA object-relational behavioural pattern. xii, 27, 32, 33, 43, 55–57, 59, 66, 75–82, 88, 91, 92, 95, 96, 100, 101, 105, 106, 108, 132, 133, 136, 138, 145, 186

Linq-to-Entities LINQ implementation of EF querying. 91, 92, 97–103, 133, 204

Linq-to-NHibernate LINQ implementation of NHibernate querying. 105, 106, 119, 124, 126

Linq-to-SQL A .NET ORM framework, now obsolete. 85, 90, 98

LinqToCaz LINQ implementation for CazDataProvider. xiii, 173–177, 190

Mapper PoEAA base pattern. 64

Metadata Mapping PoEAA object-relational metadata mapping pattern. xii, 59, 66, 70, 75, 82–87, 90, 91, 105, 106, 108–112, 115–117, 126, 130, 132, 136, 139

Microsoft Microsoft Corporation. 90, 150

Money PoEAA base pattern. 44, 46

MySQL Oracle's open source relational database system. xii, 34, 35, 37, 96–99, 101, 115, 131, 135, 146, 147, 155, 158, 161, 162, 164, 165, 170–172

N-Tier Multi Tier Architecture. 32, 36, 40, 43, 54, 64, 90–92, 94, 108, 113, 115, 137, 145, 155

NHibernate The .NET Hibernate port. i, iii, xii, xiii, 1, 5, 6, 31, 60, 73, 74, 90, 101, 104–120, 123–126, 128, 130–133, 136, 138, 144, 146, 163, 164, 176, 179–183, 186, 187, 189, 190

ObjectContext Entity Framework class that implements Unit of Work pattern. 74, 91–94, 96, 97, 99, 103, 106

Observer GoF behavioural pattern. 73

Optimistic Offline Lock PoEAA offline concurrency pattern. 73, 93, 108, 132, 140, 143

Oracle Oracle Database Server. 7, 9, 11, 15, 35, 146, 147, 159, 177, 195

Partial-Object ORM problem described by Neward. 32, 41, 54, 67, 76, 81, 133, 138, 176, 190

Perl Perl programming language. 11

Pessimistic Offline Lock PoEAA offline concurrency pattern. 73

Plugin PoEAA base pattern. 44

Proxy GoF structural pattern. 78, 79, 92

Query Object PoEAA object-relational metadata mapping pattern. xii, 43, 55, 59, 75, 84–86, 91, 92, 97, 105, 106, 121, 132, 136, 162–164, 166, 169, 170, 173, 176

Record Set PoEAA base pattern. 44, 47, 48, 50, 62, 137

Relational Domain Model Domain logic pattern discovered in ClassBuilder. xii, 136–139, 145, 176, 190

Repository PoEAA object-relational metadata mapping pattern. xii, 75, 85–87, 91, 92, 96, 99, 103, 106, 107, 116, 173

ResultSet Class from Java API. 46, 52, 61–63, 65

Ripple Loading Lazy Load problem described in Core J2EE patterns. 81

Sculpture .NET open source Model-Driven Development code generation framework. xii, 115–118, 130

Session Hibernate and NHibernate Session class that implements Unit of Work pattern. 29, 55, 56, 74, 105–108, 112, 113, 116, 117, 119, 123, 181

Session Bean J2EE server side component EJB suitable for implementing tasks and keeping conversations with clients. 56

Silverlight Microsoft Silverlight application framework (similar to WPF) for running Rich Internet Applications essentially on the browser (like Adobe Flash). 95

SQL Server Microsoft SQL Server is a relational database system. 2, 9, 11, 96, 103, 135, 136, 145–147, 149, 161, 163–166, 170–173, 176, 179, 180

SQLite Open source embedded relational database system. 135, 146, 165, 171, 172, 176

SQLJ Embedded SQL in Java. 41

Strategy GoF behavioural pattern. 80, 150–152, 156, 176

Table Data Gateway PoEAA data source architectural pattern. xi, 44–50, 52, 61, 62, 151

Table Module PoEAA domain logic pattern. xi, 44, 47–52, 54, 60–62, 87, 137

Table-per-class Object-relational structural mapping pattern introduced by Neward [New06]. xi, 18, 19, 21–23, 65, 75, 91, 105

Table-per-class-family Object-relational structural mapping pattern introduced by Neward [New06]. xi, 18, 22, 23, 29, 65, 75, 78, 91, 105, 110

Table-per-concrete-class Object-relational structural mapping pattern introduced by Neward [New06]. xi, 18, 21–23, 65, 75, 91, 105

Template Method GoF behavioural pattern. xii, 94, 159–162, 164, 166, 168–170, 176

TopLink ORM framework for Java. 73

Transaction Script PoEAA domain logic pattern. xi, 44–46, 60–62, 87

Unit of Work PoEAA object-relational behavioural pattern. xi, xii, 40, 43, 47, 55, 57, 59, 66, 67, 69–74, 76, 78, 82, 88, 90–93, 95, 96, 99, 104–107, 132, 133, 136–138, 141, 146, 176, 181

Value Holder A strategy for implementing Lazy Load (PoEAA). 77, 79, 80

Value Object PoEAA base pattern. 44

Virtual Proxy A Proxy GoF structural pattern for loading expensive objects on-demand. xii, 32, 52, 77–81, 92, 94, 96, 101, 105, 106, 153

Visual Studio Integrated Development Environment(IDE) for Microsoft .NET platform. 3, 60, 89–91, 93, 94, 96, 105, 115, 116, 118, 129, 133, 165, 179

Windows Forms Microsoft's graphical API. 50

Acronyms

ACID Atomicity, Consistency, Isolation and Durability. 12, 13, 29, 35, 37, 39, 46, 108, 179

ADO.NET ActiveX Data Object for .NET. i, iii, xi, 1, 6, 41, 45, 47, 52, 53, 61, 91, 93, 135, 146–152, 154–156, 159, 165, 169, 176, 177, 190

ANTLR ANother Tool for Language Recognition. 85, 126, 164

AOP Aspect Oriented Programming. 73, 92

API Application Programming Interface. 3, 7, 29, 30, 34, 35, 37, 39, 44, 46, 47, 50, 52, 55, 60–64, 67, 69, 70, 75, 85, 86, 88, 91–93, 95, 98–100, 102, 105–107, 117–119, 131–133, 135–140, 143, 145–147, 150–152, 154, 156, 158–161, 164, 165, 168, 169, 171, 173, 175, 176, 180, 181, 184, 187, 203

CMP Container Managed Persistence. 56, 57

COM Component Object Model. 149

CPU Central Processing Unit. 34, 193, 195, 196

CRUD Create, Read, Update, Delete. 3, 44, 47, 50, 52, 62, 63, 65, 74, 117, 136–140, 143, 151

DAL Data Access Layer. 3, 5, 7, 8, 44, 52, 62, 90, 94, 135–137, 150, 151, 155, 177, 189

DAO Data Access Object. 44, 45

DB4O Database for Objects. 36, 179, 180, 182, 184–187, 190

DBA Database Administrator. 13, 19, 23, 27, 28, 41, 58, 61

DBMS Database Management System. 41, 69, 103, 107

DDD Domain Driven Design. 58, 60, 84, 104, 132

DDL Data Definition Language. 39, 84

DLL Dynamically Linked Library. 3, 118, 132

DTO Data Transfer Object. 32, 44, 50, 52, 54, 57, 61, 64, 81, 95, 102, 103, 137, 138

EDMX Entity Data Model XML. 84, 94, 96, 97, 99

EF Entity Framework. i, iii, xii, xv, 1, 5, 6, 17, 30, 31, 55, 60, 74, 84, 85, 90–104, 106, 107, 109, 115, 129, 132, 133, 135, 136, 138, 144, 146, 176, 179, 180, 186, 187, 189, 190, 203, 207

EJB Enterprise JavaBeans. 1, 56, 57

EQL Entity Query Language. 31, 91, 98–100, 102, 104, 133, 184

GC Garbage Collector. 81

GoF Gang of Four. 4, 6, 15, 32, 43, 44, 54, 59, 152

GUI Graphical User Interface. 2, 3, 32, 41, 50, 91, 105, 112, 115, 139, 146, 165, 177

GUID Globally Unique Identifier. 29

GWS Global Weighting Solutions. i, iii, 2, 3, 5, 7, 189, 213

HBM Hibernate Mapping. 84

HQL Hibernate Query Language. 31, 85, 105, 106, 119, 124, 126–129, 132, 133, 164, 184

I/O input/output. 12, 20, 193–196, 198–201

IDE Integrated Development Environment. 30, 31, 165

IoC Inversion of Control. 74, 82, 105, 107, 112, 113, 133

J2EE Java Platform, Enterprise Edition. 7, 56

JDBC Java Database Connectivity. 1, 37, 41

JDO Java Data Objects. 38

JPA Java Persistence API. 38, 56

JVM Java Virtual Machine. 78

LINQ Language Integrated Query. 30, 41, 85, 98, 101–103, 106, 119, 129, 130, 132, 133, 146, 164, 165, 173–177, 179, 183–185, 190

MDA Model-Driven Architectire. 58

NoSQL Not only SQL. 5, 34–37, 146

ODB Object Database. 180, 187

ODBC Open Database Connectivity. 1, 149, 150, 152

ODL Object Definition Language. 39, 84

ODMG Object Data Management Group. 38, 39, 85

OLAP Online Analytical Processing. 31

OLEDB Object Linking and Embedding, Database. 3, 135, 140, 145–147, 149–151, 155, 156, 162, 165, 166, 170

OLTP Online Transaction Processing. 34

OML Object Manipulation Language. 39

OODBMS Object-Oriented Database Management System. i, iii, 1, 5, 6, 12, 15, 33, 36–41, 55, 73, 76, 85, 86, 107, 146, 179, 180, 183, 184, 186, 187, 189, 190

OQL Object Query Language. 38, 39, 55, 85, 86, 133, 177, 179, 184

ORM Object Relational Mapping. i, iii, 1–9, 13, 15–17, 27, 28, 30–38, 40, 41, 43, 44, 51, 52, 54–58, 60, 64, 67, 68, 73–76, 81, 82, 84–90, 104, 115, 120, 132–138, 140, 144, 163, 176, 177, 179, 184, 186, 187, 189–191, 193

PIM Platform Independent Model. 33, 57, 73, 82, 133, 151

POCO Plain Old CLR Object. xii, 90, 92, 95, 96, 103, 105, 106, 110, 111, 115, 117, 130, 133, 136–139, 142, 145, 177

PoEAA Patterns of Enterprise Application Architecture. 15, 43, 44, 46–48, 55, 87

POJO Plain Old Java Object. 35, 57

PSM Platform Specific Model. 57

QBA Query-By-API. 30, 85, 86, 98, 164, 165

QBE Query-By-Example. 30, 164

QBL Query-By-Language. 30, 31, 85, 86, 98, 164

RDB Relational Database. 18

RDBMS Relational Database Management System. i, iii, 1, 3–5, 7–9, 11, 12, 22, 26, 34, 35, 37, 38, 40, 41, 84, 96, 102, 108, 132, 135, 146, 147, 151, 153–155, 161, 163, 165, 170, 177, 179, 187, 190, 193, 198

RMI Remote Method Invocation. 57

SAP System, Applications and Products in Data Processing. 7

SQL Structured Query Language. i, iii, xiii, 3–5, 8, 9, 12, 15, 20, 22, 24, 30, 31, 33–35, 38, 39, 44, 45, 47, 50–52, 60–63, 66, 67, 69, 70, 84–86, 90, 92, 93, 98–109, 113, 118–140, 142, 145–147, 149, 151, 153–156, 158, 161–166, 168–177, 183–186, 189, 190, 193, 197, 203–208

T4 Text Template Transformation Toolkit. 91, 94, 96

TDS Tabular Data Stream. 149

TLS Thread Local Storage (see specification in [SSRB00]). 72, 113, 144, 150, 155

UML Unified Modeling Language. 23, 27

VB.NET Visual Basic .NET. 3, 94, 97, 99, 136, 139, 203

WCF Windows Communication Foundation. 51, 91, 95, 113

XML eXtensible Markup Language. 35, 57, 66, 82, 84, 132, 144, 150