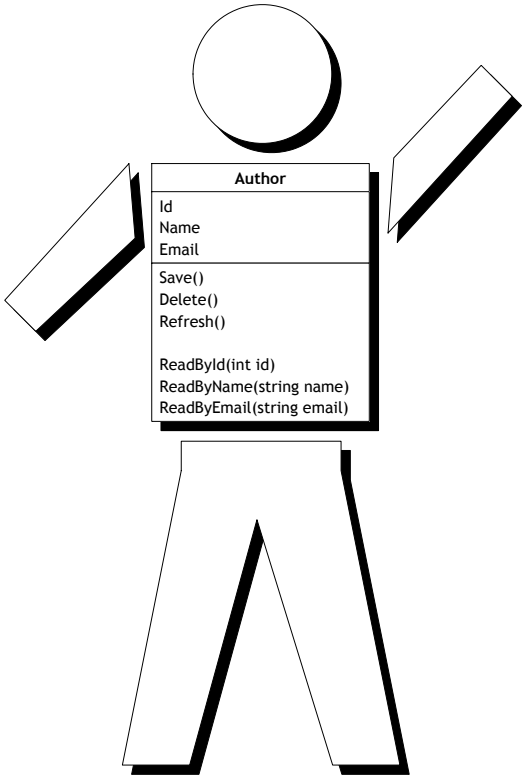

Vivid Data Objects

AALBORG UNIVERSITET

Department of Computer Science
Fredrik Bajers Vej 7, building E
DK-9220 Aalborg
Telephone: (45) 9635 8080
Telefax: (45) 9815 9889
http://www.cs.aau.dk

Title:
Vivid Data Objects

Theme:
Programming Technology

Project period:
DAT6/CIS4, spring 2007

Project group:
d632a

Group members:

Alex Henning Johannesen

Jacob Volstrup

Supervisor:

Kurt Nørmark

Number printed: 6

Number of pages: 159

Submitted: June 6, 2007

Synopsis:

This thesis documents the process of developing a framework for communicating with an RDBMS for applications that have a limited need for database query capabilities are characterized of by being "forms-over-data".
The motivation for making this framework is rooted in a subset of the impedance mismatch problem together with issues that are associated with using data from an RDBMS in an object-oriented environment. This led us to automatic code generation of a specialized API that implements four different design patterns together with a synchronization mechanism that enables a lightweight concurrency model in keeping local data consistent with changes in the RDBMS.
The VDO framework is documented in detail and an evaluation of the it is given, where we state our achievements together with the limitations it has.
The thesis concludes that the designed framework assists an application developer by reducing unnecessary reproduction of effort, when developing applications that need to safely interact with an RDBMS together with other clients.

Preface

Before commencing with this master thesis we would like to draw the reader's attention to the fact that the thesis is a result of a DAT6/CIS4-project at the Department of Computer Science, Aalborg University. In this context we would like to point out that we composed a report in the autumn of 2006, from which Section 2.1 on page 7 employs some of the findings from the research of that report.

Prerequisites

The thesis caters for readers with interest in software development in general and especially issues that are impediment for developers, who deal with software applications that employ a relational database as a storage mechanism for accessing and manipulating information.

With this in mind, we presuppose that the reader is conversant with object-oriented programming and has a basic understanding of relational databases. Furthermore, as the thesis deals with concepts from code generation it is considered to be advantageous for the reader if he has a basic knowledge of this as well.

Reading Notes

References in the thesis are represented in the form: [27, p. 16], where *p* is an abridgment for page, *sl.* for slide, and *ch.* for chapter. A list of references can be found on page 155.

The thesis is structured into five main parts. The first part in the thesis deals with the problems that we wish to provide a solution to, whereas the next part follows up with background material that applies to the design of this solution. The third part is about aspects that relate to the design and realization of the solution. The fourth part comprises an evaluation of the implemented solution, and finally the fifth part is the closing of the thesis.

Typography

During the course of this thesis an extensive number of abbreviations are used. When we introduce a new concept for which an abbreviation is applicable, the full name is presented, followed up with the abbreviation of that concept in parenthesis, e.g. Vivid Data Objects (VDO).

Moreover, when we refer to code in the text, then a special font is employed, e.g. `IEnumerable`. Code examples are given in listings that have the following structure:

```
1 // Comments go here..
2 Console.WriteLine("Welcome to Vivid Data Objects!");
```

Listing 1: Caption for the code example.

In Listing 1 there are line numbers on the left side. These numbers are used to refer to specific code fragments.

Source Code & Tools

In the course of this project, we have made use of several tools. VDO is implemented in C# 2.0 [2] and runs on the .NET 2.0 platform [8]. The integrated development environment (IDE) used for the implementation is Microsoft Visual Studio 2005 [6]. Furthermore, relational database management systems that we currently provide support for in VDO is Microsoft SQL Server 2005 [5].

Additionally, it is recommended that the reader has access to these tools if he wishes to try the implemented solution. From June 14th source code for the VDO framework is available from our VDO web site:

- http://vdo.avanceret.dk

Contents

List of Figures

Listings

I

Problem

1 Introduction

Since the early days of developing software for computers there has been a huge focus on operating with data, which ended up with the first database management system (DBMS) in the mid 1970s with a corresponding language for data manipulation [44, p. 21]. From the first DBMS up until now there has been great developments in how these systems work and how they interact with programming languages. Applications that use databases are an essential part of many enterprises today since these constitute much of the information infrastructure. These systems generally use programming languages for general-purpose computation and databases for storage and retrieval, which for instance involves control of concurrent data access, searching for data, and updating data securely and reliably.

These systems have, as any other group of applications, gone through an evolution as programming languages have evolved. At present time most of such systems are developed by using a combination of object-oriented programming languages and relational database management systems (RDBMS). This approach is popular due to several reasons. Object-oriented programming languages are widely employed on the grounds that they, among other things, promote greater flexibility and maintainability in programming and offer methods for encapsulation and information hiding. RDBMS are by far most widely used, which also is reflected by the fact that most discussed database solutions in the commercial world are relational. Given this popularity, our interest is to pursue the integration of these two areas. This means that if we at a later point in the text refer to a database, then it is a relational database unless otherwise stated.

Object-oriented programming languages and database query languages are based on different semantic foundations as their origins are in very different worlds. Some of these differences are imperative programs compared with declarative queries, algorithms and data structures versus relations and indexes, null pointers versus null for missing data, and different approaches used for modularity and encapsulation. Moreover, in concert with these differences there exist points of distinction with regards to handling concurrency and referential integrity

in these two paradigms.

Given the aforementioned it is evident that a great challenge in programming languages is to reduce the complexity of accessing and integrating information that is not natively defined using object-oriented technology.

The general problem of integrating databases and programming languages is often referred to as the term *impedance mismatch* [30]. In the context of computer science, this mismatch manifests itself in the inherent disconnect between databases and programming languages. The term *impedance mismatch* is not well-defined, but principally it encompasses the fact that the boundary between (object-oriented) programming languages and relational databases constitutes a range of issues that hinders a unified integration. In order to remove this barrier it would be necessary to either use a programming language, which supports the data model known from standard DBMS (like Prolog [46, 47]) or to employ a DBMS that supports the data model known from standard object-oriented programming languages (like Java or C#). More information about issues that constitute the impedance mismatch can be found at [34, 51].

During the development of applications that take advantage of an underlying RDBMS the developer often has to switch his mind from the object-oriented paradigm for the application host language and the declarative paradigm for manipulating the data storage in the RDBMS. As the two paradigms are so different in their nature the task of switching can be exhausting and distracting, which in effect often results in errors in the developed application, thus giving a need for extra debugging.

One of the most common mechanism employed by applications to interact with a RDBMS is a call level interface (CLI) such as ODBC [44, ch. 4] or a derivation of it like JDBC [11] or ADO.NET [1]. While there are many advantages in using a CLI, for instance performance and expressiveness, there are also disadvantages such as possibilities for bad query syntax, misspelled column and table names, data type mismatches and security lapses.

When the CLI approach is employed it frequently entails that the developer has to maintain database connection and query code in separate places within the application code. The database specific code can with advantage be put into special classes in order to ensure that hard-coded query code is not scattered out in too many places such that most of the application is decoupled from it. However, in this setting that developer often has to perform the tedious task of coding and maintaining several classes with few differences with regards to database specific code - this problem also manifests itself in the case with basic CRUD (create, read, update and delete) applications.

The aforementioned disadvantages associated with using CLI has prompted the software industry to provide better solutions for interacting with RDBMS seen from an object-oriented language perspective. These solutions take different approaches in dealing with the impedance mismatch and differ in level of sophistication and complexity. This diversity can be classified into categories, which include

language extensions [20, 26, 25], object/relational mapping [40, 33, 28, 32, 19] and persistent object systems [12, 40, 4]. Although many of the cited solutions attempt to overcome issues that constitute the impedance mismatch, some of them address a broader range of issues than others. The various solutions also differ in how they impact the way the developer has to implement an application, for instance Safe Query Objects [19] (SQO) provides an API that lets a developer to define database queries in fairly object-oriented terms, whereas DLINQ [20] brings about a query syntax that is very similar to SQL by extending C# with new keywords, thus forcing the developer to work in different paradigms.

Generally, we maintain that extending a general-purpose programming language with domain specific language elements is not always the most expedient approach. On the basis of this point of view we select to concentrate on the application programming interface (API) approach in this thesis, which we term *Vivid Data Objects* (VDO). Further argumentation for taking this approach is given in Section 2.2 on page 34.

During our preliminary research for this thesis [27] we examined a range of different solutions that try to overcome some of the inherent problems associated with the impedance mismatch. In order to manifest the framework for the following work in this thesis we sum up the most apparent and relevant findings we identified from the examined API-based solutions in Section 2.1 on page 7.

2 Motivation

In this chapter we examine those factors which motivate us in commencing with the realization of an API solution that employs design patterns and deals with a particular subset of the impedance mismatch.

Section 2.1 is concerned with our previous work and those results we came upon during this research. The reader should be especially aware of the impedance mismatch issues that have our attention, which are outlined in this section for further reference. Our previous work is a foundation for our further interests and work, and can be thought of as a motivational catalyst for what we are doing in this project.

In Section 2.2 on page 34 we use our previous work in concert with new found interests to state our fields of interest for this project and how these things relate to each other.

2.1 Previous Work Revisited

In this section we revisit selected parts of our previous research [27] that we find appreciable and relevant in the context of this thesis. It is not our intention to bring about an in-depth induction to our previous findings, but rather to establish a sufficient frame of reference for the reader that is directly applicable to our further work. Some of our previous work that is presented in this section has been slightly modified and edited in order to comply with the overall end of this thesis.

Last semester we performed a detailed evaluation of a series of existing solutions, which provide an interaction mechanism to a RDBMS from an object-oriented language. The evaluation was carried out on the basis of a perspective that comprised a subset of the impedance mismatch. An elaboration of this subset is presented in Section 2.1.1 on the next page. The elaboration only comprises and applies to those issues that we concentrate on in this thesis.

In order to frame related work we present reviews that elaborate on substantial

merits and demerits of the API-based solutions that were examined in [27]. This is done in Section 2.1.3 on page 13.

During the evaluation of the examined solutions we identified a series of characteristics in order to obtain a standard of reference, which could be used as support in future work that took our concerns with regards to the impedance mismatch into consideration. We end this section by listing the most usable and relevant characteristics, which were identified in the previous work in Section 2.1.4 on page 32. These characteristics are intended to serve as a foundation in the design of our solution.

2.1.1 Impedance Mismatch Subset

Given the scope and complexity of the impedance mismatch, the focus in our previous work was directed to a subset of the impedance mismatch. This subset provides the basis for our current work, however in continuation of this our focus is further confined to a smaller subset.

As stated in Section 2.1 on the preceding page our elaboration of the impedance mismatch subset only comprises and applies to those issues that we currently concentrate on. The issues we focus on in this work can principally be described under the category *safe queries*.

During the preliminary work of this thesis our comprehension of the category safe queries has changed since our previous work. We realized that the type checking issues that we dealt with fit under safe queries. Thus we expand safe queries to include type checking issues as well. In the following we put forward a definition of the issues that we see fit under the category safe queries. In Section 2.1.2 on the next page we provide matching examples that exemplify the respective problems of each issue within safe queries.

With the aforementioned in view, the issues that constitute safe queries are defined as follows:

- **Query Vulnerability**: This issue comprises SQL injection attacks, which is a subset of an unverified/unsanitized user input vulnerability that can yield security violations such as unauthorized access to a database.
- **Query Correctness**: This encompasses correctness checks of generated SQL queries, which consists of validating that representations of columns and tables that are employed in the programming languages correspond to the database schema, as well as ensuring a sensible usage of SQL queries.
- **Type Checking**: Here we concentrate on checking the validity and the correspondence between data types used in the database schema and the data types used in the programming language. Data type correspondence means that the types used in both ends should be compatible. An issue that is relevant to probe into is determining whether data type correspondences

are *size compatible*, for instance checking whether a certain type of a column in the database has the sufficient size to accommodate the corresponding type in the programming language or vice versa.

The reason that we focus on these particular issues is that we maintain that the reviewed solutions are not sufficient, when the complexity of a problem constitutes all the issues within this category combined.

2.1.2 Identifying the Problems

In following we list those problems, which we find to be in our greatest interest and inspired us to start our research. We are aware that more problems exist inside the problem domain, but to retain our focus we have narrowed the problems down to five particular ones that can be associated with the category *safe queries*. Together with each problem we list one or more possible problem scenarios such the issues are exemplified to the reader.

The problems we list are a mixture of problems we have experienced by ourself and problems we have learned about through other solutions, as well as heard about from other developers, thus these problems are of great significance in the everyday programming life when using a database as a back-end for storing data.

2.1.2.1 String Types

When working with strings in modern object-oriented programming languages like Java or C# there is no explicit limit to their lengths unlike their common predecessor C - actually there is no actual string type in C so instead the developer has to use an array of characters, thus having a fixed maximum length of the string. In the database world the most used string types are `char` and `varchar` as they have fixed length in the files, where the table records are saved [44, ch. 11]. These two types have a maximum of how many characters they can hold, which is specified in the database schema and is similar to the size of character arrays known from C. Both `char` and `varchar` have an upper limit to the size depending on the DBMS used, but offer a string alternative without any size limit - the `text` type.

It should be obvious to realize that there is a problem with the string types if some of the size limited types in the DBMS are used together with the regular string types in the object-oriented programming language being employed. As many database schema designers prefer to use either `char` or `varchar` instead of `text` this is a problem that needs attention. One could argue that this problem is nonexistent as the DBMS is expected to report an error whenever the application is trying to update the database with unfit variables for the records, but we still find it to be important not to create SQL query strings which do not comply with the database schema as we would like to move as much of the fault detection as possible away from the DBMS and into the application.

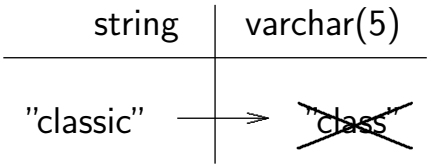

Figure 2.1: String longer than `varchar`.

As just lined up and as shown in Figure 2.1 there can be a possible problem when updating the database if the affected column does not allow a string of the given length. However this is not guaranteed to be a problem if the given string is of same or lower length than allowed, as shown in Figure 2.2.

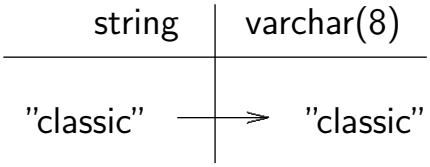

Figure 2.2: String shorter than `varchar`.

Although some DBMS like PostgreSQL claim that there is no actual performance difference between the different string types many database designers still prefer to use either `char` or `varchar` [10], though `text` would be the type to fit best to the string types from object-oriented programming languages as they can hold a string of any size. One of many reason for this decision among the database designers could be the fact that when using `char` or `varchar` the size used for each record will be fixed, but with `text` there is no limit to each record, thus being able to take up more space than the database designer would like.

When using these non-fitting string types together it is necessary to perform some check of the string variables to ensure that the SQL queries sent to the DBMS are not violating the database schema. Though many of these checks can be performed at compile-time many string variables are generated at run-time mostly as result from user input. Thus it is necessary to check some, if not all, of the string during run-time to ensure the validity of the SQL queries.

2.1.2.2 Numeric Types

When using primitive types like the numeric types there is an upper and a lower limit to the size of the number at hand, and this relies in both the object-oriented and the relational world [7, 10, 3]. This upper and lower limit is a direct result of how much storage is being used for the variables. From our experience, in databases with many records it seems to be an ongoing tendency to save as much disk space as possible with each column, thus selecting the numeric type that

takes up as little space as possible while still being able to hold the needed values for the variables.

For many of the numeric types used in databases there is an equivalent type in object-oriented programming languages, like `smallint` in both PostgreSQL MySQL and `short` in Java [7, 10, 3]. Even though this numeric type has an equivalent in the two worlds unfortunately they are not always used together, thus sometimes a `smallint` is used in the DBMS together with a `int` in the programming language. This does not cause a problem when reading a value from the database as the `smallint` has a more narrow range (from $-32,768$ to $32,768$) than the `int` (with range from -2^{31} to 2^{31}) . If trying to save an integer from the programming language to the database a problem will occur if the integer has a value above $32,768$ or below $-32,768$ as these values do not fit into the `smallint` column in the database.

If there were only used equivalent types in the DBMS and the application no problem would occur leading us to the conclusion that it is best to disallow the developer from using incompatible types. Even though this would solve some of the problems it is not possible in all situations as some numeric types from the DBMS have a different range than any of those from the application language - like `tinyint` in MySQL which has no equivalent in the Java programming language [7, 3].

Whenever some non-equivalent numeric types are used when some equivalent types are available it would be preferable to disallow the compilation by giving the developer errors about these incompatible types. Whenever some non-equivalent numeric types are used and there are no compatible types the compiler should give the developer a warning at compile-time and further check the variables during run-time to prevent SQL queries with values outside the allowed range for a given column.

2.1.2.3 Null Values

In object-oriented programming languages like Java any uninitialized variable for a primitive type is assigned to some standard value, e.g. 0 for most numeric types [3]. In contrast to this the object types are not initialized with a standard value as they often are complex, but instead they are initialized as a pointer to `null`. In DBMS it is possible for columns to allow its value not to be set, thus being a `null` value.

When translating values from the DBMS to the running application it is possible to retrieve a null value. Although this is not a problem when retrieving a value that is to be stored directly as an object in the application, like a string. However, it is mostly values for primitive types that are retrieved from the DBMS. As primitive types in e.g. Java are not able to hold the `null` value it is necessary to either use the standard value for the type in the application language, let the developer choose some standard value, or completely forbid using columns in

which `null` values can occur.

2.1.2.4 SQL Injection Attacks

Most applications accept user input and some of this can be harmful to the DBMS serving the SQL queries generated from the application - these queries are called SQL injection attacks as they exploit vulnerabilities in the application to inject erroneous code into the SQL query through the variables from the user used directly in the SQL query. The most focus on SQL injection attacks is in regard to web applications for the internet [24, 17], but any public available application with input from the user (e.g. at train stations, supermarkets, etc.) is of interest when preventing SQL injection attacks as they are all vulnerable to the inputs. Hence, it is of general interest to include methods for preventing SQL injection attacks in our solution.

If a person is interested in discovering vulnerabilities in a running application it is easy to find if no counter-measures have been taken by the application developer to prevent these attacks [22]. Basically the problem arises when there is no verification of the variables given from the user to the program. If these variables are used directly in the SQL query strings it is possible to modify the query and execute the queries in ways which is not preferable. In Listing 2.1 we show a method that could exist in an application with user verification - the method is invoked with two parameters for username and password, it passes these directly to the SQL query string. In Listing 2.2 we see the SQL query string produced by calling `getUser` with `uname` as the username and `pass` as the password. In Listing 2.3 on the facing page we attempt to perform a SQL injection attack by setting the username parameter to `admin' --`, thus using the fact that anything after `--` is commented out by the DBMS in the query string. With this injection there is no check against the password, which leaves the application vulnerable.

```
1  public ResultSet getUser(String uname, String pass) {
2    String query = "SELECT id FROM users "
3                   + " WHERE username='" + uname + "'"
4                   + " AND password='" + pass + "'"
5    Statement stmt;
6    ResultSet rs;
7    stmt = connection.createStatement();
8    rs = stmt.executeQuery(query);
9
10   return rs;
11 }
```

Listing 2.1: Unsafe usage of input variables.

```
1  SELECT id FROM users WHERE username='uname' AND password='pass'
```

Listing 2.2: The query produced in 2.1.

```
1 SELECT id FROM users WHERE username='admin' --' AND password='
    pass'
```

Listing 2.3: The query produced in 2.1 on the facing page with injection attack.

Although the vulnerability shown in Listing 2.3 can be considered harmful to the running application it is possible to do severe damage to the DBMS by adding more queries or by executing commands. Each query could be decoupled by the SQL query separator the semicolon. By exploiting this vulnerability it is possible for the attacker to create, alter, and drop tables which can have severe consequences - an example of this is shown in Listing 2.4 where the username variable has been set to `'; DROP TABLE users --`. As the goal for this injection attack is to do harm to the database the username variable in the SQL query string has been set to be the empty string - after execution the `users` table has been deleted, thus it is no longer possible to use the application.

```
1 SELECT id FROM users WHERE username=''; DROP TABLE users --' AND
    password='pass'
```

Listing 2.4: The query produced in 2.1 on the facing page with a critical injection attack.

It should be obvious that it is important to prevent any kind of injection attack to the SQL query string during run-time as any attack that is a result of malicious user input is not detectable during compile-time. Even though it is not possible to do the checks during compile-time it is possible to point out those occurrences in the source that are vulnerable due to user input.

2.1.2.5 Developer Mistyping

When creating application source code there is always the risk that the developer mistypes or misspells parts of the program, which entails a risk of mistyping in the parts that are responsible for communication with the DBMS.

Most compilers tell the developer about these mistypings during compile-time, but due to the fact that SQL queries are present in the application source code as strings they are not checked to see whether they comply with the database schema, thus giving errors at run-time.

The mistyping done by the developer may be names for tables (e.g. `user` instead of `users`, which can be hard to localize as an error when looking through the application source code) or columns where some compile-time errors would help the developer narrow down his search to locate errors.

2.1.3 Evaluated Solutions

In [27] we evaluated eight existing solutions that had different origins, some came from academia and others were open source initiatives or commercial solutions.

Although we examined eight solutions, we only choose to include three of those in the following. This choice is based on the fact that these are most closely related to what we wish to deal with.

The reviews are centered around describing the structure and the most distinguishing features of the respective solutions in concert with whether the solutions address the concerns of the impedance mismatch subset that were outlined in Sections 2.1.1 on page 8.

2.1.3.1 Hibernate

In short, Hibernate can be described as a object/relational persistence and query service for Java and .NET. As we find Hibernate to be a rather complex project, we only look at those things that we find interesting and relevant.

Hibernate works as a layer that takes care of the mapping from Java classes to database tables and from Java data types to SQL data types. Hibernate requires no interfaces of base classes for persistent classes and makes it possible to make any class or data structure persistent. Furthermore, it employs run-time reflection to gain information about the persistence of objects. Class instances can be transient or persistent, where the former state is the default for all classes.

The mapping done in Hibernate is by way of XML documents, which define the ORM and generate table and constraint creation scripts. It supports a variety of inheritance mapping approaches and all entity association mapping styles, which include one-to-many, many-to-one, one-to-one and many-to-many. However, the XML mapping files are written by hand. This leaves room for errors such as misspellings of column names, which can be tedious to find. On the other hand, once the mapping files are defined Hibernate offers a range of tools for assistance in the development process. These include tools for generating Java classes (or database tables) from existing mapping files, and also for generating mapping files from existing database tables.

An illustration that depicts the overall architecture of Hibernate is given in Figure 2.3 on the facing page.

As illustrated in Figure 2.3 on the next page, all communication is being done via instances of `Session`, which are single-threaded, short-lived objects representing the communication between the application and the data store. A `Session` holds a mandatory cache of persistent objects, which are used for navigating the object graph or looking up objects by identifier. A `SessionFactory` is an immutable cache that represents all compiled mappings for a single database and configurations of Hibernate. A `Session` works as a factory for instances of `Transaction` that also are single-threaded, short-lived objects, which make it possible to perform commits or rollbacks as known from databases. More elaborate details about the architecture can be found in [39, ch. 2]. An example that shows how the various parts in the architecture interact is given in Listing 2.5 on the facing page.

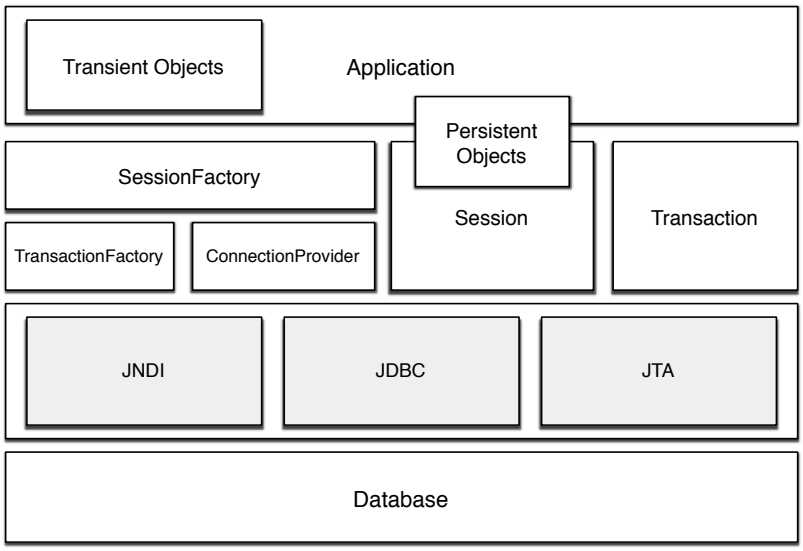

Figure 2.3: The architecture of Hibernate.

In the example we create an `Event` object and hand it over to Hibernate, which takes care of the SQL and executes `INSERT` on the database. In order to shield our code from the actual underlying transaction system we use the `Transaction` API.

```
1  ...
2  public class EventManager {
3    public static void main(String[] args) {
4
5      EventManager mgr = new EventManager();
6      mgr.createAndStoreEvent("New Event", new Date());
7      HibernateUtil.getSessionFactory().close();
8    }
9
10   private void createAndStoreEvent(String title,
11                                    Date date) {
12
13     Session session =
14     HibernateUtil.getSessionFactory().getCurrentSession();
15
16     session.beginTransaction();
17
18     Event event = new Event();
19     event.setTitle(title);
20     event.setDate(date);
21
22     session.save(event);
23     session.getTransaction().commit();
24   }
```

```
25 }
```

Listing 2.5: Storing an object.

In order to express queries, Hibernate gives you three options [39, ch. 10]. The first is to use native SQL, which deviates a bit depending on the database. Native SQL is useful if you want to utilize database specific features, but it has the same pitfalls associated with CLI. An example of a SQL query is given in Listing 2.6. Native SQL has optional support from Hibernate for result set conversion into objects.

```
1 List cats = session.createSQLQuery(
2     "SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM < 10",
3     "cat",
4     Cat.class
5 ).list();
```

Listing 2.6: SQL query.

The next option is Hibernate's own query language, Hibernate Query Language (HQL). HQL is object-oriented and understands notions like inheritance, polymorphism and association. HQL employs a syntax that is similar to SQL as it is designed to resemble it. It uses many of the same keywords known from SQL and queries may consist of clauses, subqueries, expressions and aggregate functions. Expressions allowed in the `WHERE` clause include most of the kind of things you could write in SQL, the same is true for the use of aggregate functions [39, ch. 14]. Clauses like `ORDER BY` and `GROUP BY` are also supported by HQL and work similar as in SQL. In addition, it is also possible to perform inner, left outer, right outer and full joins in HQL. Even though HQL queries support object-orientation they still suffer from being strings like native SQL. An example of HQL queries is given in Listing 2.6.

```
1  ...
2  List kittens = session.createQuery(
3      "from Cat as cat where cat.mother = ?")
4      .setEntity(0, pk)
5      .list();
6
7  Cat mother = (Cat) session.createQuery(
8      "select cat.mother from Cat as cat where cat = ?")
9      .setEntity(0, izi)
10     .uniqueResult();
```

Listing 2.7: HQL queries.

The last option is criteria queries, which distinguish themselves from the other two options. They make it possible to build queries dynamically, using an object-oriented API. Hibernate provides a criteria query API, `Criteria`. An example of criteria query is shown in Listing 2.8 on the next page.

```
1 List cats = sess.createCriteria(Cat.class)
2     .add( Restrictions.like("name", "F%")
3     .addOrder( Order.asc("name") )
4     .addOrder( Order.desc("age") )
5     .setMaxResults(50)
6     .list();
```

Listing 2.8: Criteria query.

As can be seen in Listing 2.8 it is possible to add orderings and restrictions using methods. Hibernate does the translation of the criteria instance into a query.

An interesting thing that we would like to point out is how data types are handled in the mapping done by Hibernate. In some cases, a mapping element may lack a type attribute, see line 6 in Listing 2.9.

```
1 <hibernate-mapping>
2     <class name="events.Event" table="EVENTS">
3         <id name="id" column="EVENT_ID">
4             <generator class="native"/>
5         </id>
6         <property name="title"/>
7     </class>
8 </hibernate-mapping>
```

Listing 2.9: Hibernate mapping file.

The types that are declared and employed in the mapping files are neither Java nor SQL data types. These types are so called Hibernate mapping types, converters which can translate from Java to SQL data types and the other way around. Hibernate tries to determine the correct conversion and mapping type itself if the type attribute is not present in the mapping file. This leaves room for errors, because the correspondence between the types may be entered in the XML document by hand. Moreover, in some cases this automatic detection done by Hibernate might not have the default you would expect or need. An example is the date property `java.util.Date`. Here Hibernate can not determine if the property should map to a SQL `date`, `timestamp` or `time` column.

In order to understand the terminology in examples that are to come, we first introduce the concept of *entity* and *value* types that are used in the context of Hibernate. Formally an entity is any class whose instances have their own persistence identity. A value type is a class that does not define some kind of persistent identity. Basically this means that entity types are classes with identity properties, and value type classes depend on an entity [39, ch. 5].

The `save` and `delete` methods of `Session` apply to instances of entity classes, never to value type instances. The persistence lifecycle of a value type instance is entirely depended on the lifecycle of the owning entity instance. Furthermore, in Hibernate a value type may define some associations, where it is possible to

navigate from a value type instance to some other entity. However, it is never possible to navigate from the other entity back to the value type instance. Associations always point to entities, which means that value type instances are owned by exactly one entity when they are retrieved from the database. Moreover, at the level of the database, any table is considered to be an entity.

In the following we look at how types are handled with regards to queries. HQL and native SQL queries are represented with an instance of `org.hibernate.Query`, which offers methods for parameter binding, result set handling, and for the execution of the actual query. A query is usually executed by invoking the `list` method, the result of the query is loaded into a collection in memory. In addition, entity instances retrieved by a query are in persistent state.

The most basic SQL query is to get a list of scalars. The query in Listing 2.10 returns a `List` of `Object[]` with scalar values for each column in the `CATS` table.

```
1 session.createQuery("SELECT * FROM CATS").list();
```

Listing 2.10: Scalar result from a simple SQL query.

Hibernate employs result set metadata to deduce the actual order and types of the returned scalar values. However, even though the types may be determined there is still a disconnect between specifying the query and the usage of the result, e.g. if the query string is altered, no warning is given to the developer at compile-time if he tries to access a scalar value that is not in `Object[]`.

The example in Listing 2.10 was about returning scalar values or the "raw" values from the result set. The code in Listing 2.11 exemplifies how to get entity objects from a native SQL query by way of using the method `addEntity`.

```
1 session.createSQLQuery("SELECT * FROM CATS")
2 .addEntity(Cat.class);
```

Listing 2.11: Entity result from a simple SQL query.

Assuming that `Cat` is mapped as a class with the columns `ID`, `NAME` and `BIRTHDATE` the query in Listing 2.11 returns a `List` where each element is a `Cat` entity, but still we do not get static type checking.

When using HQL you are restricted to the same pitfalls as with native SQL with regards to static type checking. However, you have some more convenient ways of specifying what kind of result you expect from the query. This information is specified in the query string. An example of this is given in Listing 2.12, which contains queries that return multiple objects as an array of type `Object[]`, as a `List` or as an type-safe Java object, respectively.

```
1 select mother, offspr, mate.name
2 from DomesticCat as mother
3     inner join mother.mate as mate
4     left outer join mother.kittens as offspr
```

```
5
6  select new list(mother, offspr, mate.name)
7  from DomesticCat as mother
8      inner join mother.mate as mate
9      left outer join mother.kittens as offspr
10
11 select new Family(mother, mate, offspr)
12 from DomesticCat as mother
13     join mother.mate as mate
14     left join mother.kittens as offspr
```

Listing 2.12: Different types of results using HQL queries.

We were interested in how Hibernate deals with size compatibility, so we compiled a small application that tries to insert a too large Java string into a column of type `varchar(5)`. As expected we only got a run-time error from the underlying system. The error was an exception, `com.mysql.jdbc.MysqlDataTruncation`, which informed us that the input data was too long for the column in question. We maintain that such potential errors as this one should be informed to the developer as a warning.

Another experiment we performed was with regards to injection attacks. Hibernate provides several ways of communicating with the database and we did not try all of them. When trying to execute the code in Listing 2.13, we did not manage to drop the table `Events`. We suspect the reason that the command `drop table` is not performed is because of the used database MySQL 5.0, which does not allow more than one query in one input. We did not examine if this is the case with other databases.

```
1 String ok = "\'; DROP TABLE events; --";
2 session.createSQLQuery(
3 "SELECT * FROM Events WHERE title = '" + ok +"'");
```

Listing 2.13: SQL injection attack.

However, we managed to perform another type of attack successfully, which is given in Listing 2.14. It returned all rows that were in the table `Events`.

```
1 String ok = "' or title is not null or title='";
2 session.createSQLQuery(
3 SELECT * FROM Events WHERE title = '" + ok +"'");
```

Listing 2.14: SQL injection attack.

The concatenation used in Listing 2.13 and 2.14 is notoriously unsafe and the most common vector in injection attacks. There is a safer way by using parameters in Hibernate. Listing 2.15 on the following page shows code that does the same, but is safe because it uses the method `setString`, which accepts positional parameters. Internally Hibernate uses prepared statements, which guard

against SQL injection. In a prepared statement, arguments are bound to the statement rather than having a plain text SQL statement being issued. The code in Listing 2.15 is in fact the recommended approach.

```
1 String ok = "' OR title IS NOT NULL OR title='";
2 session.createSQLQuery(
3 "SELECT * FROM Events WHERE title = ?").setString(0, ok)
```

Listing 2.15: SQL injection attack.

However, an inconsiderate developer may well use the approach with string concatenation and therefore we must conclude that Hibernate does not completely protect you from injection attacks.

As for debugging HQL or SQL queries Hibernate does not assist you during compile-time or when you write the query code. However, modern development environments like Eclipse can assist a developer with auto-completion, but misspellings of column names or bad syntax in query strings are not detected before run-time.

2.1.3.2 Safe Query Objects

Safe Query Objects (SQO) is an integrated approach to the impedance mismatch that allows query behavior to be defined using statically typed objects and methods, which supports query shipping[1] by automatically generating code to execute queries remotely in a relational database [19].

Safe queries are Java objects that adhere to a specific programming pattern to define the behavioral components of a query. The fact that safe queries are Java objects implies that they are semantically integrated and type-checked statically.

The prototype of SQO employs the OpenJava [9] macro system for metaprogramming at compile-time to generate code to invoke the Java Data Objects (JDO) persistence library [4] - in this context that is translating from safe query classes to database access classes.

In short, macros in OpenJava are compile-time transformations on class definitions and expressions that reference them, and are run after initial type analysis, but before complete type checking has been performed. An OpenJava compile-time transformation is specified by a *metaclass*. OpenJava extends the Java programming language in a way that it is possible for a class definition to specify a metaclass using the `instantiates` keyword.

JDO is used as a foundation for safe queries. It is a standard for interfacing Java with persistent data in relational and non-relational data stores, and supports both object-relational mapping and a call level interface. JDO supports a subset of relational query behavior that includes dynamic filters and filters that involve joins between multiple tables, sorting, parameterization (like user input), and existential quantification. The examined prototype inherits the capabilities and limitations of JDO 1.0 [18], which means that it does not support multi-table query results or general aggregation.

JDO provides access to persistent objects via `PersistentManager`, which is an interface to making an object persistent, as well as to retrieving persisted objects, and removing them from persistent storage. The method `newQuery` of `PersistentManager` is used to create a JDO `Query` object, which is a call level interface. This interface is decomposed into methods for ordering, filtering and declarations of query parameters, imports and variables.

SQO makes extensive use of the concept of filters, since the authors maintain that defining a query as a filter method is natural in the sense that it is a simple and effective Java programming pattern: it modularizes query behavior into methods that have the right form to be translated for remote execution in a relational database. Filter conditions in SQO are translated into the JDO Query Language (JDOQL), which allows automatic conversations between types, overloads < and > for boxed values including dates and strings.

[1] Query shipping refers to the practice of transferring high-level operations, like filtering and joining, to the database.

In JDO every query has a *candidate type* that defines the class of objects that are returned by the query. This type is specified in the call to the method `newQuery`. An example of a JDO query is depicted in Listing 2.16, where all employees whose salary is greater than their manager's salary is selected.

```
1 import javax.jdo.*;
2 Collection executePayCheck(PersistenceManager pm) {
3     Query payCheck = pm.newQuery(Employee.class);
4     payCheck.setFilter("salary > manager.salary");
5     Object result = payCheck.execute();
6     return (Collection) result;
7 }
```

Listing 2.16: Filtering in JDO.

The filter specification in Listing 2.16 is evaluated for each instance of the candidate type, `Employee`, which means that references to `manager` and `salary` access the members of the instance. In order to navigate through object-valued fields, such as `manager.salary`, JDO employs joins. The result of this query is the subset of all candidate instances in the database for which the filter evaluates to true. JDO handles the translation of database rows to Java objects. The static return type of the method `execute` in Listing 2.16 is `Object`, but at run-time the return value is a read-only `Collection` that contains instances of the candidate type.

The simplest form of a safe query is an object that contains a boolean method that can be employed to filter a collection of candidate objects. The example in Listing 2.16 is shown as a safe query implementation in Listing 2.17.

```
1 class PayCheckQuery extends SafeQuery<Employee> {
2     boolean filter(Employee emp) {
3         return emp.salary > emp.manager.salary;
4     }
5 }
```

Listing 2.17: Filtering using a safe query object.

Given that the filter in Listing 2.17 is plain Java code implies that syntax and types are checked during compile-time.

The behavior of a query object is characterized by the base class `SafeQuery`, given in Listing 2.18, where the generic type `T` defines the candidate type of the query. Moreover, safe query objects are instances of *safe query classes*, which all must extend `SafeQuery<T>`. Furtermore, safe query classes must adhere to some behavioral restrictions, which for instance includes that the filter method has to be side-effect free - the full list of these restrictions is given in [19].

```
1 class SafeQuery<T> {
2     boolean filter(T item) { return true; }
```

```
3 }
```

Listing 2.18: The `SafeQuery` base class.

Given the fact that safe queries are Java classes makes it possible to execute them locally to filter any collection of objects. However, since our focus is on interfacing with a database we only concentrate on remote execution.

In order to implement remote execution, SQO employes OpenJava to generate necessary methods and attributes that enable a query to be shipped to and executed on a database. The behavior for compile-time metaprogramming is encapsulated in the metaclass `RemoteQueryJDO`, which is applied to the safe query using the `instantiates` keyword. An example of this is given in Listing 2.19.

```
1 class PayCheck instantiates RemoteQueryJDO
2                 extends SafeQuery<Employee> {
3
4     boolean filter(Employee emp) {
5         return emp.salary > emp.manager.salary;
6     }
7 }
```

Listing 2.19: Remote execution: safe query class that invokes a metaclass.

During compile-time, OpenJava runs the metaclass and supplies the definition of `PayCheck` as input. At this point, the metaclass can examine the partially compiled definition of the class and modify or extend the class. In the case with Listing 2.19, `RemoteQueryJDO` gets `PayCheck` as input and generates the method `execute`, which implements a remote version of the `PayCheck` filter method by passing appropriate strings to the JDO interface. The automatically generated `execute` method for `PayCheck` is given in Listing 2.20. Note that the generated method is the same as the potentially unsafe code in Listing 2.16 on the facing page, but the crucial difference is that the code in Listing 2.20 is automatically generated from a type-checked Java method, thus the safe query version is type-safe.

```
1 import javax.jdo.*;
2 ...
3 Collection<Employee> execute(PersistenceManager pm) {
4     Query q = pm.newQuery(Employee.class);
5     q.setQuery("salary > manager.salary");
6
7     return (Collection<Employee>) q.execute();
8 }
9 ...
```

Listing 2.20: Remote execution: automatically generated method for `PayCheck`.

In addition to filters, SQO also provides means for sorting. Relational query languages define a sort order by deriving a list of sortable values from each element of the candidate type. This list of values can be marked to indicate whether the

sort order should be ascending or descending. In SQO sorting is specified by associating a list of sortable values with each object in the result set. In the same manner as with the `filter` method, safe queries can also specify an `order` method that takes a candidate class and returns a list of sortable objects. This list is represented as a linked list of `Sort` objects, where each contains a comparable value, a flag indicating the sort order, and an optional secondary `Sort` value. An example of a safe query that specifies two sort orders is listed in Listing 2.21.

```
1  class SortQuery instantiates RemoteQueryJDO
2                  extends SafeQuery<Employee> {
3
4    Sort order(Employee emp) {
5
6      return new Sort(emp.department.name,
7                   Sort.Direction.ASCENDING,
8                   new Sort(emp.salary,
9                   Sort.Direction.DESCENDING));
10     }
11 }
```

Listing 2.21: Sorting: safe query with an `order` method.

The code specified in the `order` method in Listing 2.21 is translated into code that has a proper syntax, which is accepted by JDO, depicted in Listing 2.22, after being processed by OpenJava.

```
1  import javax.jdo.*;
2  ...
3  Collection<Employee> execute(PersistenceManager pm) {
4      Query q = pm.newQuery(Employee.class);
5      q.setOrdering("department.name ascending, "
6                   + "salary descending");
7
8      return (Collection<Employee>) q.execute();
9  }
10 ...
```

Listing 2.22: Sorting: Automatically generated remote execution.

In addition to filtering and sorting, we are going to look at two other significant query features of SQO, *parameterized queries* and *dynamic queries*, that we find imperative with regards to our objectives.

In SQO parameterized queries are used when their behavior depends upon input values. Query parameters are values that can have an effect on the filtering and ordering of query results. The way that SQO makes them accessible from the `filter` and `order` methods is to encapsulate them as instance variables in the query object. Formal parameters on the class constructor are being used to initialize these variables. An example of a parameterized safe query class is given

in Listing 2.23. The code is used to find employees with a salary greater than a limit.

```
1  class SalaryLimit instantiates RemoteQueryJDO
2                    extends SafeQuery<Employee>
3  {
4      double limit;   /* parameter */
5      SalaryLimit(double limit) {
6          this.limit = limit;
7      }
8
9      boolean filter(Employee emp) {
10         return emp.salary > this.limit;
11     }
12 }
```

Listing 2.23: Parameterized safe query object.

The parameter in Listing 2.23 is a normal Java variable, which entails that its declaration, usage, and binding are all checked for consistency during compilation. This is in contrast to a hand-coded parameterized JDO query, where the essential connection between declaration, usage, and binding is broken up in disjoint API calls, which leaves room for errors that only are discovered at run-time. The result after translating the code in Listing 2.23 is given in Listing 2.24. The `RemoteQueryJDO` metaclass translates the query parameter into corresponding JDO calls. Given the fact that the translation is automated entails that there are no syntax or type errors in the strings passed to the CLI. More information about the translation mechanism involved can be found in [19, p. 9].

```
1  import javax.jdo.*;
2  ...
3  Collection<Employee> execute(PersistenceManager pm)
4  {
5      Query q = pm.newQuery(Employee.class);
6      q.setFilter("salary > limit");
7      q.declareParameters("double limit");
8      Map paramMap = new HashMap();
9      paramMap.put("limit", limit);  // boxed
10
11     Object result = q.executeWithMap(paramMap);
12     return (Collection<Employee>) result;
13 }
14 ...
```

Listing 2.24: Automatically generated code.

The last query feature of SQO we discuss in depth is dynamic queries. This kind of query constitutes filters, parameters, or orderings that are built during run-time. Dynamic queries are employed when different filter criteria have to be combined to construct a complete filter. For instance, one can imagine a set

of optional search criteria that has to be specified in an application. In this scenario the filters that result from different combinations of criteria will differ, and since the difference between filters is structural, parameterized queries are not sufficient.

In SQO dynamic filters are implemented as a special case of filters and can be expressed as a `filter` method by way of using conditionals or short-circuit evaluation. The idea here is that parts of the overall filter are only evaluated if certain conditions are met. An example of a dynamic filter is given in Listing 2.25.

```
1  class DynQuery instantiates RemoteQueryJDO
2                 extends SafeQuery<Employee>
3  {
4      private String namePrefix; // may be null
5      private Double minSalary;  // may be null
6
7      DynQuery (String namePrefix, Double minSalary){
8         this.namePrefix = namePrefix;
9         this.minSalary  = minSalary;
10     }
11     boolean filter(Employee item){
12        return (namePrefix == null
13                      item.name.startsWith(namePrefix))
14                     && (minSalary == null
15                      item.salary >= minSalary);
16     }
17 }
```

Listing 2.25: Safe query using dynamic filter.

The short-circuit evaluation of `||` in Listing 2.25 is necessary in order to avert a `null`-pointer exception. SQO employs *abstract partial evaluation* to find the cases where a `null` argument triggers short-circuit evaluation. The approach for interpretation of `null` values is to determine whether an expression being evaluated can be simplified given the subset of parameters that are `null`. If this is possible then the original expression is replaced by its simplest form. An example is the Java expression `(emp == null) || (emp.salary > 100)`. This expression simplifies to `true` when `emp` is `null` and `emp.salary > 100` when `emp` is not `null`. In the examined prototype of SQO this analysis only supports dynamic queries based on tests for `null` arguments. More elaborate details about the evaluation approach are in [19, p. 11].

Lastly, we would like to point out that we have not found any documentation that specifies how SQO deals with mapping from a nullable SQL type to a corresponding primitive type in Java.

2.1.3.3 SQL DOM

A solution that has some points of resemblance with SQO is SQL DOM: Compile Time Checking of Dynamic SQL Statements (SQL DOM) [32]. This solution can in its simplest form be described as a set of classes that are strongly-typed to a database schema. The primary goal of SQL DOM is to provide a way to have the full expressive power of dynamic SQL statements without inherent problems such as bad syntax, misspelled columns and table names, data type mismatches and SQL injection attacks.

The foundation of SQL DOM can be decomposed into two main parts. The first part is an *abstract object model* and the second part is an executable, sqldomgen, that is used for execution against a database schema to generate a concrete instantiation of the abstract object model.

The development of sqldomgen is done by using C# and the .NET framework. Its task consists of three main steps. First step is to obtain the schema from the database. The next step is to iterate through the tables and columns in the schema and output a number of files that contain a strongly-typed instance of the abstract object model. The last step is the actual output from sqldomgen, which is a dynamic link library that contains the classes that are strongly-typed to a database schema. These classes are what constitute the DOM in SQL DOM, the SQL Domain Object Model, and which also are employed to construct dynamic SQL statements without manipulating strings.

The approach that SQL DOM takes is to have a compiler to eliminate the possibility of problems with SQL syntax, developer misspellings and data type inconsistency. This elimination is feasible because names of tables and columns are embedded into the SQL DOM via class names or enumeration members, where data types of columns correspond with types of constructors and method parameters.

The object model constitutes three main types of classes, which are SQL statements, columns and `WHERE` conditions. For each of the four types of SQL statements, `SELECT`, `INSERT`, `UPDATE` and `DELETE`, a class is created for each table in the database schema. It is these classes that are used to create SQL statements. For instance, if you were about to create a `SELECT` statement for an employee table, you would use an instance of the `EmployeeTblSelectSQLStmt` class. An interesting property to note is that `SELECT` SQL statements are equipped with a `JoinTo<tablename>` method for each table with which they have a foreign key relationship. This alleviates the developer from remembering the names of foreign and primary key columns.

Each class is associated with a single table, and the constructors of each class are typed to take only parameters representing columns of the table with which each respective class is associated.

The column classes are used as parameters to the constructers and methods of the SQL statement class mentioned above. They specify which columns are to

be selected, updated or inserted. Furthermore, it is the column classes that hold data in instance variables of the same type as the columns in table with which they are associated. As it is possible that tables can have columns of same name, namespaces are used to avert name collisions.

An attractive property of SQL DOM that we find relevant to point out is security with regards to injection attacks. The interesting part here is that SQL DOM represents a single point of defense as all SQL statements are constructed by the SQL DOM. All database user input passes via constructors of classes in the SQL DOM and necessary precautions such as escaping and data type validation is handled within the realm of these constructors.

Actually it is within the column classes that SQL injection attacks are handled. Column classes that have a data type of `string` parse and possibly modify values that are given to them. An example is a single quote in a string, which would be escaped to two single quotes in order to avert an injection attack. Moreover, other data types do not need escaping because of the strongly-typed nature of the classes. For instance, a class that represents a column with the data type `int` only accepts values that are valid `int`'s. However, we have not come across any information about how SQL DOM handles nullable SQL types that have a corresponding value type in C#.

The last main type of class in the object model are the `WHERE` condition classes. These are used to specify conditions in `WHERE` conditions of `SELECT`, `UPDATE` and `DELETE` SQL statements. Instances of `WHERE` condition classes are added to SQL statements via the `AddWhereCondition` method, which is overloaded according to the columns in the database schema. Furthermore, it is possible to group together and randomly nest `WHERE` condition classes in order to construct complex conditions.

In the following we show some code examples that exemplify the benefits that are gained by using the features of SQL DOM, mentioned above.

Problems associated with SQL strings that we are interested in are syntax errors and misspellings. An example of this is depicted in Listing 2.26, which is a function, `GetCustomers`, that employs string concatenation to dynamically filter the result of a `SELECT` statement.

```
1  public string GetCustomers( string companyName,
2                  string contactName,
3                  string city,
4                  string region,
5                  string country )
6  {
7    bool firstCondition = true;
8    StringBuilder sql =
9    new StringBuilder("SELECT * FROM Customers ");
10
11   if((companyName != null) && (companyName.Length > 0)) {
```

```
12
13      if(firstCondition) {
14
15        firstCondition = false;
16        sql.Append(" WHERE ");
17
18      } else {
19        sql.Append(" AND");
20      }
21      sql.Append("CompnyName = '");
22      sql.Append(companyName);
23      sql.Append("'");
24    }
25
26    // similar code would be placed here for each of the
27    // other possible conditions.
28
29  return sql.ToString();
30  }
```

Listing 2.26: SQL statement with syntax error and misspelling.

Two errors that are easily made are illustrated on line 19 and line 21. The first one, line 19, is that there will not be any space between the `AND` and following column name, which results in a SQL statement that is not valid. The second error, line 21, is a misspelled column name that also will result in a run-time error. An improved version of `GetCustomers` that uses SQL DOM can be seen in Listing 2.27.

```
1   public string GetCustomers( string companyName,
2                   string contactName,
3                   string city,
4                   string region,
5                   string country )
6   {
7     CustomersTblSelectSQLStmt sql =
8       new CustomersTblSelectSQLStmt();
9
10    if((companyName != null) && (companyName.Length > 0)) {
11
12      sql.AddWhereCondition(
13        new CompanyNameWhereCond(companyName));
14    }
15
16    // similar code would be placed here for each of the
17    // other possible conditions.
18
19  return sql.GetSQL();
20  }
```

Listing 2.27: Improved `GetCustomers` using SQL DOM.

The improved version of `GetCustomers` in Listing 2.27 on the preceding page has several advantages over the old version. One of these is that the SQL syntax is automatically generated by the SQL DOM and therefore mistakes like misspellings or SQL syntax errors are eliminated.

Another example that we find relevant to point out is how SQL DOM assists in data type correctness. Listing 2.28 contains a function, `SetUnitsInStock`, which is used to update the `UnitsInStock` column of the `Products` table. The SQL data type of the `UnitsInStock` column is a `smallint`, which corresponds to `short` in C#.

```
1  public string SetUnitsInStock( int productID,
2                     int unitsInStock )
3  {
4    ProductsTblUpdateSQLStmt sql =
5      new ProductsTblUpdateSQLStmt();
6
7    sql.UnitsInStock = unitsInStock;
8
9    sql.AddWhereCondition(
10     new ProductIDWhereCond(productID));
11
12 return sql.GetSQL();
13 }
```

Listing 2.28: Data type mismatch bug.

As can been seen on line 7, the data type of the `unitsInStock` is `int`. This scenario generates a compiler error message that the compiler cannot implicitly convert type `int` to `short`. This is possible because of sqldomgen created a `UnitsInStock` property with a data type of `short`. Internally, the `UnitsInStock` property instantiates the `UnitsInStockUpdateColumn` class to handle the necessary work. This is interesting to us, namely because we are concerned about size compatibility, see Section 2.1.1 on page 8.

The last example we look at is about injection attacks. Listing 2.29 shows a function, `UpdateCustomer`, that is used for updating customer information.

```
1  public string UpdateCustomer( int customerID,
2                    string companyName)
3  {
4    CustomersTblUpdateSQLStmt sql =
5    new CustomersTblUpdateSQLStmt();
6
7    sql.CompanyName = companyName;
8
9  return sql.GetSQL();
10 }
```

Listing 2.29: Prevention of an injection attack.

A malicious user could attempt to perform an injection attack by submitting a specially constructed string like "`Kevin Mitnick'; drop table Customers --`" for the `companyName` parameter. However, this would not work because the attack would be prevented by the escaping done by the `CompanyNameUpdateColumn` class, which is created internally by the `CompanyName` property, line 7.

2.1.4 Identified Characteristics

During our previous research in [27] we focused on how other projects deal with the problems in the impedance mismatch subset described in 2.1.1 on page 8. Inspired by the classical way of developing languages and language extensions [43, p. 8] we made a list of characteristics from these projects. In order to support us in finding suitable characteristics we used a series of criteria [43, ch. 3] that reflected our interest and focus in regard to the impedance mismatch, thus being able to highlight which criteria each characteristic supports.

For each of the examined projects we thoroughly compared features with our criteria and whenever we identified something useful it was added to the list of characteristics. After examining the projects we ended up with 11 different characteristics.

Although we maintain that all the characteristics are of great importance when developing any kind of language, language extension or something similar, we have come to realize that not all of these characteristics can be unified, thus it is natural to select those characteristics which conform mostly to a given target. Hence, in the following we give an account of the characteristics *class generation*, *type mapping* and *run-time monitoring*, which we see fit within the overall objective in our current work.

2.1.4.1 Class generation

The reason for generating classes automatically should be obvious as it reduces the possibility of errors in the application as the classes are generated from the database schemas, e.g. from a document with the schema in an usable syntax. With these classes it is possible for a standard compiler to check for type mismatches without introducing any new methodologies.

The database classes are generated with the knowledge of the used database and tables, thus they are generated with the right types as parameters. The application developer is ensured that if the application is compiling at least the used types are correct.

Through warnings and errors the application developer gets during compilation if unsupported types are used, forces him to either use the correct type or make explicit type casting when this is not possible, thus the application developer is aware that problems can arise due to the type casting (e.g. different value ranges with numeric types).

Lastly some security vulnerabilities can be handled by the fact that it is possible to create precautionary measures against SQL injection attacks in the automatically generated classes. This is for instance used in SQL DOM as a strategy for taking measures against injection attacks.

2.1.4.2 Type mapping

As covered in Section 2.1.2 on page 9 there can be problems between the used types in the object-oriented language and the DBMS in use. By making a description of how to bridge between the types from the two very different worlds it is possible for the compiler to make valid checks of whether the application source code conforms with the database schema. Moreover, it is possible to map non-primitive types from the object-oriented world which can be saved as a primitive in the database, e.g. date types.

The characteristic *type mapping* is supporting safe queries by defining, which types from the different worlds are compatible. With this mapping it is possible for either the compiler or the run-time environment to make a qualified analysis of the communication with the DBMS and thereby preventing queries with incompatible types. Furthermore, the compiler or the run-time environment is able to give the application developer suitable errors and warnings based on the type mapping.

2.1.4.3 Run-time monitoring

By analyzing the SQL queries during run-time, prior to execution by the DBMS, it is possible to ensure that user input does not invalidate the queries compared to the behavior intended by the application developer.

The *run-time monitoring* characteristic supports safe queries by checking queries before they are sent to the DBMS for execution. By checking the query prior to execution it is possible to ensure that the query is consistent with the database schema with regards to the used types and the possibility of mismatch between the used types is taken care of as covered in Section 2.1.2 on page 9, but also that the query is correct in sense of the SQL query language. Moreover, injection attacks concerns are handled by checking that the user input does not invalidate the query.

2.1.5 Concluding Remarks

In this section we have gone through some subjects that influence and are relevant to the work that is ahead of us. We introduced the impedance mismatch subset that has our attention in our own solution, followed by a series of exemplifying problems that put the issues within the subset into perspective.

Three of the most relevant solutions from last years work were reviewed, in which we put forward the most prominent features along with some merits and demerits. The findings from the reviews in 2.1.3 on page 13 is something that is going to influence the approach and design of our own API. However, we wish to construct a solution that addresses the issues in the impedance mismatch subset that the other solutions do not accommodate in concert with that it should be

characterized by our own ideas and address some additional problems. This is what the next section is about, where we take a more overall angle on the matters that "stamp" the more structural and design related areas of our solution.

2.2 Fields of Interest

In this project we address problems that are associated with using data from a RDBMS in an object-oriented environment when developing applications.

Whenever using data from a database the developer has to decide how to represent the data together with how to manipulate the data, save the data and communicate with the database. If changes are made to the database schema, then the developer has to edit and examine classes within the application in order to ensure that no errors occur as a result of the changes. Moreover, if the communication is by way of regular SQL strings and a popular CLI such as JDBC, ODBC or ADO.NET, then this opens up for the possibility of typing errors as no correctness check is done on SQL strings by these CLIs.

The task of creating classes for using database tables may be a repetitive and time-consuming task for the developer. This is due to this process is often done by creating several classes or methods that are similar in code, but still deviate in code fragments that are adapted to the used database tables and the communication with the database.

Further, from our experience it is a common omission to make wrong mappings between the data types in the database and the host language, as well as forgetting to take measures against security breaches like injection attacks.

As stated in Section 2.1.4 on page 32, we found the three characteristics *class generation*, *run-time monitoring* and *type mapping* during our previous work. These are employed in our further work to provide a foundation for the design of our solution, and at the same time work as a catalyst for new and supporting ideas.

In order to supply *run-time monitoring* for a running program we maintain that an application programming interface (API) is an expedient approach. With an API it is possible to wrap any kind of communication with the database in use. This entails that we are able to ensure safe queries by preventing the developer of being able to explicitly operate with SQL strings and instead provide the developer with a restricted and strongly typed interface for communicating with the database. This effectively means that we want to eliminate errors that are connected with variable usage and perform an implicit *type mapping* from data types used in the database to corresponding data types used in the object-oriented environment. Both correct mapping of data types and prevention of injection attacks can be taken care of by wrapping the communication.

While a *run-time environment* can prevent faulty SQL queries from being ex-

ecuted it cannot reduce the workload of the developer in relation to representing and manipulating the data from the database. To further minimize the workload of the application developer we want to automatically generate database access classes. The generated classes should reduce unnecessary reproduction of effort with regards to retrieving, representing and manipulating data in the database. Secondly, the generated classes should take advantage of a *run-time environment*, as we can place general functionality for communicating with a database and manipulating data in the *run-time environment*, and the functionality for communicating with specific tables in the database in the generated classes.

It is important to realize that we are trying to satisfy both the relational and the object-oriented world, thus giving some restrictions. As foundation for the automatic *class generation* we use the relational world, thus mapping the tables from the database to classes without features that are only available in the object-oriented world like inheritance and polymorphism as there is no pendant in the relational world.

As the generated classes are intended to aid the application developer they should consist of basic methods for manipulating data in the class instances, and also in the underlying database. As the data manipulation language (DML) part of SQL is very exhaustive we are only engaged with a small subset of the operations in order to accommodate the other concerns we present in this section. Additionally, it is not our objective to provide an API that has the same degree of expressiveness as SQL, as the operations we are interested in are basic *create*, *read*, *update* and *delete* (CRUD) operations that work with a database environment that has multiple clients. These are the most common operations in the applications we are addressing with our project, cf. Section 2.3 on page 37.

When designing the classes that are to be generated some kind of guiding principle is very useful in order to keep them from becoming too detailed and complicated. As methods for some problems mature into best practices these are the obvious choice for us to employ as a guiding principle when designing our solution. In the world of programming these best practices are mentioned as design patterns [23], and in the field of database connectivity there are several different patterns for communicating with a database [21].

In the context of design patterns we wish to investigate how feasible it is to automatically integrate different patterns in the generated classes while still being "faithful" to these design patterns. We also wish to explore the field of using a design pattern, which is not normally used together with databases in order to add functionality that may lessen the programming burden of the application developer in some scenarios.

Database specific design patterns often have a large architectural impact on an API. This is particularly true for the *Active Record* pattern that we employ,

as it influences the structure of the generated classes. We find *Active Record* to be the pattern that is most conformable with our interests in using the database schema as foundation for our API, as we wish to map each table to a class, thus having an object in the running application for each corresponding row in the tables.

During run-time it is not unusual to retrieve the same row several times from a table, which leaves room for errors as the developer may alter two different objects that correspond to the same row. To keep objects in the running program from being duplicated each time a specific data row is being used we further wish to integrate the *Identity Map* pattern into our generated classes. This pattern ensures that each data row is represented by at most one *data object*, thus an identity map is needed to keep track of the loaded data objects.

As we wish to map the relational model onto the object-oriented model entails that relations between tables are kept intact even when the data is represented in objects. As a result of the fact that the tables often are related in a very extensive manner we wish to reduce the communication load when reading an object from the database by not reading the related objects until they are needed. This way of only loading information when needed is the *Lazy Load* design pattern.

In a setting where multiple clients are using the same database it can be a very complex operation to ensure even a lightweight version of concurrency such that a user is always working with data that is up to date or just being able to inform a user if the data has changed in the database.

Applications that are running simultaneously on several clients that share the same database often have problems in keeping their state consistent with regards to the state of data in the database. As a result of using the *Identity Map* we have a list of the loaded data objects, which can be used to poll the database for changes at a pre-defined interval. Instead of letting the developer performing this polling we wish to integrate an automatic synchronization mechanism into each generated class that keeps data objects in a state that is in accordance with their corresponding rows. This implies that if a row is either deleted or updated then this is reflected in the corresponding data object subsequent to each synchronization. Moreover, as we wish to keep relations between tables intact further entails that the synchronization mechanism should take into account when a new row is added to a table. This requires that data objects that represent rows, which share a relationship with a row that is added, updated or deleted, are automatically updated in order to reflect this.

As mentioned in a previous paragraph we wish to explore the possibility of using a regular design pattern together with the database to add functionality that may be usable in some scenarios. Being able to keep data objects in an up to date state gives rise to take this one step further, which is to provide a means to automatically inform other parts of an application about changes in data objects. This gives the application developer the possibility to take actions

as a result of changes caused on the data objects. We have found the *Observer* pattern to be a sound candidate for this particular problem as objects that are added as observers to a data object are informed about changes in the data object, thus giving the application developer the freedom to use this functionality when a simulated concurrency model is needed.

For the most of the included patterns the generated classes should be implemented such that the application developer can use them without any extra programming effort. However, the *Observer* pattern entails that the application developer has to implement "half" of the code, as this is very specific in regards to how the changed data objects should be handled.

The patterns we have chosen are examined in greater detail in Section 5 on page 55 together with a description of how we are planning on utilizing them.

Before commencing with the realization of VDO we knew that we would need to decide on a programming language. The languages we shared an interest in were C# and Java. Both authors have previously been exposed to both languages, but in connection with the work in [27] it came to our attention that C# 2.0 has some language features that we surmised could be conducive to reaching our goals. The choice of C# was also a consequence of the fact that we were curious about experimenting with its features. In conjunction with this choice we selected to concentrate on providing support for MS SQL Server 2005, though in a loosely coupled way such that we in the future are able to support other RDBMS.

2.3 Target Audience

The target audience of VDO comprises two parts. The first part is the category of applications that we target with our API and the second part consists of the type of users VDO is directed towards.

The type of applications we target are characterized of being "forms-over-data" applications, which we classify as simple CRUD applications that operate on tables that are normalized into either Boyce-Codd or third normal form [44, ch. 7]. Furthermore the applications that we aim for only require simple queries to the database. This implies that most industrial applications are not on our radar as they often require advanced query functionality that is available through SQL, sometimes referred to as Business Intelligence (BI) [50].

The target user of VDO is an application developer that is characterized by having a minimal need of performing advanced queries on data in the database in the applications he is developing. Conversely, he is characterized by developing applications that continuously monitor data in a database, i.e. have a need to be updated with data contained in tables according to actions caused of other clients using the same database as external storage, and require basic editing facilities

on that data. Furthermore, the targeted user does not have much knowledge of the impedance mismatch issues that we deal with or how to solve these, thus the requirement for solving these problems in VDO. In addition, the target user is not required to have much experience with the data access layer, but yet has a need to access data in tables in a manner that does not incur a steep learning curve, thus the requirement for a user-friendly way of setting up the access to a database by way of automatic generation of database access classes.

As we recognize and can relate to some of those needs we consider ourselves as a constituting part of the target audience. This entails a justification for the requirements that we later put forward with regards to the design of VDO as the motivational factor for developing VDO is rooted in our own needs, because we have in our experience encountered situations where this type of API is needed.

2.4 Summary

In this chapter we have outlined the motivational factors that impel us commencing with the realization of an API solution that employs design patterns and deals with a particular subset of the impedance mismatch. In the next section we state the overall objectives we wish to accomplish.

3 Objectives

The objectives for this thesis are based on what motivates us, which we outlined in the previous chapter. In the following we further concretize the overall goals that are pursued throughout this thesis.

Our principal objective is to serve an application developer by automating the task of both communicating with and representing data from an RDBMS in an object-oriented host language by providing a user-friendly solution that enhances productivity by reducing unnecessary reproduction of effort and solves the problems associated with safe queries as described in Section 2.1.1 on page 8.

In order to implement this solution we construct a code generator that is used for generating a specialized API that is based on the previously mentioned design patterns together with a synchronization mechanism. The specialized API should correspond to a provided relational database schema and its interdependent relations. Our goal for the code generation process is that it should require minimal effort and interaction from the user.

In the context of the aforementioned objectives it is in our interest to examine the following aspects:

- To which degree *Active Record* facilitates the implementation of safe queries.
- How far it is possible to integrate the observer pattern together with a synchronization mechanism in the specialized API such that it simulates a lightweight concurrency model that reduces the developer's workload of keeping local application data consistent with corresponding data, which is employed by multiple clients in a RDBMS.
- How language features of C# influence the design of the API.

II

Background

4 API Design

This chapter serves as purpose to identify characteristics of good API design and to list general best practices in API design that we follow in the design phase of VDO. The reason for saying practices is that API design is more a craft than a science.

The motivation for including subjects about API design has two primary aspects to it, one is to show the basis that provides the background for the design of VDO, the second aspect is that we wish to make the API as usable and intuitive as possible such that the user-experience is positive, thus the inclusion of characteristics of good API design to benchmark against.

Before commencing on presenting the subjects in this chapter we would like to emphasize that much of the structure and content of it is influenced by the work of *Joshua Bloch*, which gave a presentation on API design at *Google Tech Talk* [15]. Accommodating material to his work [14, 13] also serves as basis for the subject matter of this chapter, this material is the source from which we draw upon example code. Moreover, when other sources are used then these are denoted in the text.

It is important to realize that we are selective in choosing heuristics in relation to API design, thus the following subjects are by no means exhaustive, rather they are intended to support us throughout the design phase of VDO.

There are some general circumstances that have entailed our attention towards an incentive to comply with best practices in API design. In general, APIs can be among the greatest assets and liabilities for an enterprise , which stems from the fact that a good API is something that users may invest heavily in. This investment may be done in obvious ways and less obvious ways. Obvious ways are that they develop application that are build around the API, which means that they write to the API. The less obvious way is that they learn it, they may invest much time in learning it and once they have done that, it is reasonable to conclude that they may be reluctant to learn a new one if they have to replace their existing knowledge with something else that brings about new things to be

learned. In fact, we know from our own experience with different parts of the Java and the .NET platform that learning an API requires both time and effort, especially in situations where we have moved from one platform to the other and discovered that things do not always behave as expected.

Moreover, we need to bear in mind that it is important to realize that a public API is "forever" so to speak, because once it has a user-base you are not at liberty to change it at will, because people rely on it. Thus, you get one chance to get it right. In essence, this means that APIs are centered around the concept of contracts and interfaces. They can be thought of as means for developers to communicate intent to other developers.

Given these general reservations, we put forward some of the most apparent and important areas that we maintain are imperative to bear in mind during the design of VDO in the following sections.

As a final remark before getting into the deeper aspects of API design, we would like to bring to notice that the sign ⋄ is used as an indication for the presence of our own view or perspective in itemized lists. The ⋄ sign is used to indicate a comment to the text associated with a previous • or − sign in a list, which correspond to practices taken from [14, 13].

4.1 Characteristics of User-Friendly API Design

Before we set out on designing VDO it is appropriate to take some measures with regards to user-friendliness in order to ensure that VDO is designed with usability and positive user experience in mind. Therefore we have chosen to follow some general guiding principles that call for user-friendliness.

- It should be easy to learn and to use, preferably with minimal reliance on documentation.

⋄ We understand the essence of this as being that VDO should be intuitive and communicate its utility through structure and naming. Another way of saying this is that we should design VDO in a way that allows the user to know how to use it, that is giving as many clues as possible.

- It should be easy to memorize.

⋄ When we think about tasks that are easy to remember, we often realize that these are intuitive in nature. In the case of VDO we maintain that its structure and naming should communicate its behavior. It boils down to that we should design VDO such that the user should recognize rather than recall, which is what we understand about "easy to memorize".

- It should be hard to misuse. This in effect is the flip side of the two subjects above, because they imply that it should be hard to misuse. Effectively, this means that the API should "force" you to make the right thing.
- It should be easy to read and maintain code written to the API.
  - ◇ Although we have no control over code written to VDO, the message that we draw from this is if VDO is complicated to use, then code written against it is bound to reflect this.
- It should be sufficiently powerful to do what it has to do, which means that it should satisfy its requirements. Note, this does not mean that it should be powerful, because it is not necessarily the case that the more powerful it is the better it is. It should basically be powerful enough to do what is required of it and nothing more.
- Easy to evolve. The fact that the API should be sufficiently powerful implies that it should be easy to evolve, because there is a great chance that there are going to be new requirements at a later point.
- It should comply with the target audience, which means that it should provide utility and functionality that are highly valued by the target audience [42].
  - ◇ This implies that what is a good API for a business analyst is presumably not a good API for a mathematician, because they think in different problem domains and use different terminologies. Therefore the design of VDO should be aimed at its audience. For instance, as we do not require that the user of VDO has in-depth knowledge of databases and operations that can be performed on them it is imperative that we do not introduce terminologies that only are understood by users that program in database query languages.

The above characteristics are what we maintain is needed to adhere to in order to design VDO usable. In the remainder of this chapter we outline what is needed in order to achieve these characteristics.

4.2 The Process of API Design

There are design concepts and principles from software engineering that apply when designing an API [31] [37]. However in order to keep matters focused, we only concentrate on best practices given by [14] that tend to lead to good API design.

- Specify the requirements for the API.

- Employ use cases. The requirements should take form of use cases. This means the problems that VDO should be able to solve. These are important, because they provide the benchmark against which we can measure any proposed solution.
- Start out with a short specification, because in the early stage of the design process agility trumps completeness. As you gain better confidence that you are on the right track, then you refine it and make it more complete. This necessarily involves coding to the API that you are defining, however it does not mean coding to the implementation of the API.
- Code to the API definition early and often. This means that you pretend that it has been implemented.
  - Start coding before you have implemented it. It saves you from developing an implementation that you may throw away.
  - Start coding before you have specified it properly. This saves you from writing a specification you may throw away.
  - Continue writing to the API as you specify it more completely. It gives you better ability to prevent unpleasant surprises before you publish the API.
- Code lives on as examples. This forms one of the most important code that you write in regard to the API, because examples of API use tend to become emulated by users – if you get the examples right, then you have seeded the market with good uses of your API. Conversely, if you get them wrong then that may be emulated by the user as well. In essence, example programs should be exemplary.
- Maintain realistic expectations throughout the design process. Many API designs are over-constrained, because people want them to do more than they possibly can do. You have to make compromises as you cannot please everyone, if you do then you most likely end up with something big that is complicated and bloated. Therefore we should aim to displease everyone equally.

4.3 General Principles

In the following we outline some general principles that have an impact on good API design, because we are interested in stating the principles that influence our choices during the design process.

- An API should do one thing and do it well.

  - ◇ Our opinion is that parts of the API should do one thing and do it well, but this is dependent on how you define when something becomes an API. We consider the whole of VDO to constitute an API, whereas *Joshua Bloch* for instance refers to a single class as an API.

- Functionality should be easy to explain. This means that if it is not easy to explain then it is probably not doing one thing and doing it well. Moreover, if it is hard to name, then that is generally a bad sign. Good names drive good designs, because the names are the API talking back to you, so we should "listen" to it. Examples of this are:

  - **Good:**

    ```
    Font, Set, Vector, PrivateKey, Lock, TimeUnit, Future,
    ThreadFactory.
    ```

  - **Bad:**

    ```
    DynAnyFactoryOperations, _BindingIteratorImplBase,
    ENCODING_CDR_ENCAPS, OMGVMCID.
    ```

  The examples that show good names instantly communicate what they are, which is not the case in the examples of bad naming.

- The API should be as small as possible, but no smaller. This involves:

  - When in doubt, leave it out. This applies to every aspect of the API, that is functionality, classes, methods, parameters and so on.
  - ◇ The key thing that we have learned from this point is that we can always add functionality to VDO, but it is hard to remove functionality from it without imposing problems on the user. One could argue that there are ways around this issue, for instance by using the keyword `deprecated` to indicate that a method or a class is obsolete and developers are discouraged from using it. However, we still maintain that even though there exist opportunities for marking parts of VDO obsolete at a later point, it does not invalidate the rationale behind *when in doubt, leave it out.*
  - Conceptual weight is more important than bulk, e.g. number of classes, methods and so on. What is important is the number of concepts. When the user is learning an API it is imperative for him to know how many different things he has to learn about. There are number of ways to decrease the conceptual weight of the API. The most important one is reusing interfaces. An example of this is the collection framework in Java, in this framework there are many different implementations

of the `Set` interface. This way you do a lot without requiring the user to learn a lot of new concepts.

- ◇ We understand this as we should strive for a low conceptual weight in VDO. We maintain that conceptual weight is closely related to the user-friendliness and usability of VDO, namely because too high conceptual weight entails that the user may be required to be introduced for too many concepts, which in effect has an negative impact on the user-experience and learning curve. A way of decreasing the conceptual weight of VDO is to adhere to common practices in C#.

- Implementation should not impact the API. Implementation details are unnecessary and confuse users. Furthermore, they inhibit our freedom to change the implementation.

  - Be aware of what is an implementation detail. This encompasses that we have to be careful to not over-specify the behavior of methods. The specification of a method should not involve something that is an implementation detail and we would like to change it at a later point.

    An example of this is with regards to specification of hash methods. One may be tempted to think that the exact value that is returned by a hash code method is a proper part of a specification. In general this is incorrect, it is an implementation detail. The specification of the method should simply state that it returns an integer and with a high probability this integer will differ for two different objects. You should have the freedom to change the implementation as you learn about flaws in your hash method and new approaches in hashing.

  - Do not let implementation details "leak" into the API. One example is if you have an API that is about phone numbers that throws SQL exceptions. Imagine then if you want to re-implement it on top on some proprietary data store rather than a SQL data store, then your clients are already trying to catch SQL related exceptions. One solution to this problem is that you can emulate those SQL exceptions, but that is undesirable. This boils down to that we should make sure that the exceptions that we throw are on the same layer of abstraction as the rest of the API.

- Minimize the accessibility of everything. This comprises that we make members of classes as private as possible. Conversely, public classes should have no public fields. This maximizes information hiding [35, 36] as well as minimizes coupling, which has some attractive advantages attached to it. It allows for modules to be understood, used, built, debugged and tested independently.

- Names matter. An API can be thought of as a little language and users are going to have to learn that language and speak in it, which entails the following:
  - Names should be largely self-explanatory and cryptic abbreviations should be avoided.
  - Be Consistent. The same word means the same thing throughout the API (ideally across APIs on the platform). If two things mean the same thing, call them the same thing, if they mean different things, then call them something that differentiates them enough to tell them apart.
  - Strive for symmetry. For instance, if you have two verbs `Add` and `Remove`, and two nouns `Key` and `Entry` then you should have `AddEntry` and `RemoveEntry`.
  - If you get it right, then code reads like prose. An example is:

```
if(car.speed() > 2 * SPEED_LIMIT) {
  speaker.generateAlert("Watch out for cops!");
}
```

- Documentation is important. Document every class, interface, method, constructor, parameter and exception. The user should not be put in a situation, where he has to guess the intent of the code or to have to read the source code (if available), which entails that the implementation becomes the documentation.

◇ Especially apparent to us is to follow the following practices with regards to documentation:

  - **Class:** Always specify what an instance of a class represents.
  - **Method:** Specify the contract between the method and its client, which involves specifying pre-conditions, post-conditions and possible side-effects.
  - **Parameters:** Indicate units. An example of indicating units is the size of a block, in this case it is important to specify whether the size is in, say, bytes or megabytes, not just stating that it is the size of the block. Moreover, aspects like explaining the form, for instance if it is a string then one should explain the form it is in, e.g. XML or something else.

- Document state space for an object very carefully (if it is mutable). The user has to know when it is legal to call which methods and what happens after the call is made.

- The API must coexist peacefully with the platform. This involves that you do what is customary in the platform.

  ◇ In our case this is C# and .NET. More specifically it entails that we:

  - Obey standard naming conventions.
  - Avoid obsolete parameter and return types that are wrongful in the platform.
  - Mimic patterns in core APIs and language. This is important because people are likely to already know the core API.
  - Take advantage of API-friendly features, for instance generics, variable arguments, enumerations, default arguments and so on.
  - Know and avoid API traps and pitfalls.
  - Never transliterate the API, which means that we should not use conventions from a foreign language that do not fit or make sense in C#.

The aforementioned practices are general principles that apply to API design on different levels. In the next two sections we put forward more specific practices that apply to the design of classes, exceptions and methods, respectively.

4.3.1 Class & Exception Considerations

On a more specific level, there are some sound practices in class and exception design together with issues that a designer should consider and take into account, when creating an API. The most apparent practices we have an interest in complying with are given in the following.

- Minimize mutability. Generally, one should strive for minimizing mutability unless there is a good reason to do otherwise. This has advantages such as your classes get simple, thread-safe and instances of them are reusable, because you never have to generate a new one. Among the disadvantages, one is that you have a separate object for each value.

  If you have to make your class mutable, then keep the state space small and well-defined, which encompasses that you make clear when it is legal to call which method.

- Subclass only where it makes sense. Subclassing implies substitutability, which involves that subclassing is done when an *is-a* relationship exists. Otherwise use composition. Public classes should not subclass other public classes for ease of implementation. Examples of when it makes sense and when it does not to subclass are:

– **Bad:**

```
Stack extends Vector
```

When you ask yourself if every stack is a vector, then you come to the conclusion that it is not the case. You push and pop on a stack, which essentially is all what you do with a stack. You might also have a `peak` method and a `size` method, but a vector allows random access by index, which a stack does not.

– **Good:**

```
Set extends Collection
```

In this example the inheritance makes sense, because when you ask if every set is a collection then you come to the conclusion that it is. A set is just a special kind of a collection, a collection which does not allow for duplicate elements.

- Design and document for inheritance, otherwise prohibit it. The reason for this is because inheritance violates encapsulation [29]. This stems from the fact that a subclass may be sensitive to implementation details of the superclass. Thus, if we use subclassing, then we should document how methods use one another.

  A conservative and safe policy is to keep all concrete classes final.

As for exception design there are some things one should have in mind. Some of these things are influenced by how the language you use deals with exceptions. For instance, in Java you have checked exceptions, but in C# you do not have them. As we use C# we only look at some general practices you should adhere to, when using unchecked exceptions.

- Throw exceptions to indicate exceptional conditions. This implies that you should not force the client to use exceptions for control flow. On the other hand side, you should not fail silently.
- Unchecked exceptions should indicate a programming error.
- Always include failure-capture information in the exception, which allows for diagnosis and repair. In unchecked exceptions a message is sufficient.

4.3.2 Method Considerations

Like there are best practices with regards to class design there also are some practices that you should follow when designing methods of classes. Those that we maintain are important to adhere to are listed in the following.

- Do not make the user do anything that the library could do. This one of the fundamental rule of API design. It is not recommended to have the user doing something that the library can do, because it causes "boilerplate" code, which often is characterized by cut-and-paste code and being error-prone.

- Do not violate the principle of least astonishment. The user of the API should not be surprised by its behavior. Because if you surprise them then you may find yourself in the situation, where the users of your API think that they are doing something correct, but in reality the application is doing something else. An example from the `Thread` API in Java, where the the principle of least astonishment is violated:

```
public class Thread implements Runnable {
  // Tests whether current thread has been interrupted
  // Clears the interrupted status of current thread
  public static boolean interrupted();
}
```

  In the example there is a method called `interrupted`, which violates the principle of least astonishment. If you have a thread and want to check if it is interrupted then you call `Thread.interrupted()`, but what is wrong here is that the method *also* clears the interrupted status of the current thread; it has a *side effect*. Looking at the name, `Thread.interrupted()` does not communicate that it does this. The primary thing that this call does is to clear the interrupted status, thus it should be named `clearInterruptStatus` and documented that it returns the old interrupted status. The method is named based upon the second most important thing it does. Thus, every method should do the least surprising thing it could, given its name, and leave out unexpected behavior.

- Fail fast. The sooner you report a bug, the less damage it will do. Compile-time reporting gives users of the API the chance to find errors before their products are in the field. If you report errors at run-time, then report them at the first bad method invocation, for instance if the user passes something wrong into a method, then the user should be made aware of this as soon as possible.

- Provide programmatic access to data, which is available in string form. In essence, this means that whenever you have a method that returns something as a string, then always provide a corresponding method that returns the same stuff in programmatic form. Otherwise the users will parse strings, which is tedious. In fact, omitting this can cause the string format to be turned into a de facto part of API, because it prohibits you to add something to a string, because there may be user code that is parsing the string. A bad

example of this is with regards to a stack trace of an exception in the initial versions of Java. Before version 1.4 of Java the only way to get information about a stack trace was through a method called `printStackTrace`.

- Overload with care. Method overloading can be a good thing, but it tends to be overused. However, always avoid ambiguous overloading, which is multiple overloading that can do different things when passed the same values. A bad example of this is `TreeSet` in the Java collection framework. The reason for this is exemplified in the following example:

```
// Ignores order
public TreeSet(Collection c);
// Respects order
public TreeSet(SortedSet s);
```

`TreeSet` has two constructors, one that takes a collection and one that takes a sorted set. The first of these ignores the ordering, whereas the second one respects it. The problem is that if you have a sorted set that is casted to a collection then you get one result, and if it is not casted then you get another result.

The point here is that just because you can does not mean that you should. Often it is better to use a different name rather than overloading, but if you must provide overloading, then always ensure the same behavior for the same arguments.

- Use the right data type for the job. This implies that you use appropriate parameters and return types. There are a couple of different important things to remember:

  - Do not use a string if a better type exists, strings are cumbersome, error-prone and slow.
  - Use consistent parameter ordering across methods. Otherwise, the user may get it backwards. A bad example of not being consistent is given in the following:

```
char *strcpy (char *dest, char *src, size_t n);
void  bcopy  (void *src, void *dest, size_t n);
```

    - ◇ Here we have two input/output operations. The first function, `strcpy`, takes a destination, a source and a size. Conversely, the second function, `bcopy`, takes a source, a destination and a size. This is unfavorable and inconsistent ordering of `*src` and `*dest`, because it is possible that the user may assume a ordering where `*src` is a first parameter, which can result in that he ends up with filling source data with whatever is in the destination array.

- Avoid long parameter lists, especially those with multiple consecutive parameters of the same type. Long lists of identical typed parameters are harmful, because the user may transpose parameters by mistake. This does not hinder the application from compiling, but makes it misbehave at run-time.
- Avoid return values that demand exceptional processing. The user may forget to write the special-case code, leading to bugs. For example, return zero-length arrays or collections rather than nulls.

4.4 Concluding Remarks

In this chapter we have outlined some important characteristics of good API design together with some best practices that are required in order to achieve those characteristics. More specifically, we set out by listing general practices that apply to API on all levels. This was followed by some more specific sound practices that apply to class and method design together with issues that we should consider and take into account when designing VDO.

5 Applied Design Patterns

In this chapter we describe the most apparent and relevant patterns that are applied in the design of VDO, these are those patterns that constitute and characterize our solution.

The background material is aimed at our solution, therefore we chose to concentrate on matters that encompass relevant subjects that in some way have an impact on our solution or provide a better understanding on what constitutes and characterizes the design choices of VDO. The major content and structure of this chapter is based on [21].

Design patterns have been around for a long time, therefore it is not in our interest to give an in-depth elaboration of their history or applicability. Still, we find it to be noteworthy how the perception of design patterns is evolved from describing very complex operations [23] to a title for some best practice refinements over years.

Though the early design patterns described best practices they were still more general in their usage and not as specific as those presented by Fowler in [21]. The patterns explained by *Gang of Four* in [23] are by many seen as the *original* patterns, but also they only specifically describe limited usage due to their generality. As we explained in Section 2.2 on page 34 one of our main goals is to implement some best practices, this is our view and approach to design patterns, though not exclude the classical design patterns, just broaden the perspective.

> " *A key thing about patterns is that you can never just apply the solution blindly, which is why patterns tools have been such miserable failures. A phrase I like to use is that patterns are "half baked" - meaning that you always have to finish them off in the oven of your own project.* ***(Martin Fowler)*** *[21, p. 10]* "

With that in mind, the following three sections are about database specific

patterns. The first, *Active Record*, is an architectural pattern that has influenced and inspired some parts of the architecture of VDO. The two subsequent sections are about behavioral patterns, *Identity Map* and *Lazy Load*, respectively. These two patterns are used in VDO in concert with *Active Record*.

Lastly, in section we describe a "regular" behavioral pattern, *Observer*, which is used in the design and architecture of VDO.

5.1 Active Record

The *Active Record* pattern is a data source architectural pattern [21, ch. 12] that is used to read and manipulate data in a database. It is characterized by the fact that an object carries data and behavior, and much of this data is persistent and needs to be stored in a database. *Active Record* uses the most straightforward approach by encapsulating the database access logic in the domain object. In this setting we have a database table that is wrapped into a class, this consequently means that an instance of the class is associated to a single row in the table. After you have created an object, a new row is added to the table upon save. Each object that is loaded from the database gets its information from there, which means whenever an object is updated, the corresponding row in the table is updated as well.

The essence of an *Active Record* is a domain model in which the classes match very closely the table structure of an underlying database. Our approach to *Active Record* is characterized by having instance properties to represent a single record, that is one field in the class for each column in the table. Furthermore, instance methods are used to act on specific records, whereas class methods are used to act on several rows from the table.

The *Active Record* class is typically characterized by having methods that do the following [21]:

- Construct an instance of the *Active Record* from a SQL result set row.
- Construct a new instance for later insertion into the table. This can also be a constructor.
- Static lookup methods to wrap commonly used SQL queries and to return *Active Record* objects.
- Update the database and insert the data in the *Active Record* instance into it.
- Get and set methods for fields. E.g. properties in C#.

Furthermore, we envisage that properties can do intelligent things such as maintaining a correspondence between an SQL data type and a host language

data type. In addition, a property could represent a related table, and if this table is requested, the property can return the appropriate *Active Record*.

In this pattern the classes are convenient, but they do not hide the fact that a relational database is present as a data store. This is not crucial in our case, because it is not our objective to hide the database. The consequence is that you usually see fewer of the other object-relational mapping patterns present when *Active Record* is being employed [21, ch. 12].

Active Record has the primary advantage of simplicity as it is easy to understand, which is something that is imperative in the context of VDO. Generally, its primary problem is that it works well only if the *Active Record* classes correspond directly to the database tables. This is not an issue in our case, because we generate classes from the database schema.

Finally, *Active Record* is a good choice for applications that are not too complex, such as creates, reads, updates, and deletes [21, ch. 12].

5.2 Identity Map

An identity map can be employed when you wish to ensure that each object is only loaded once. This is done by keeping each loaded object in a map, whenever you want to fetch an object, you check the identity map first to see if it is already there [21, ch. 11]. The illustration in Figure 5.2 shows how a setup of an identity map can look like.

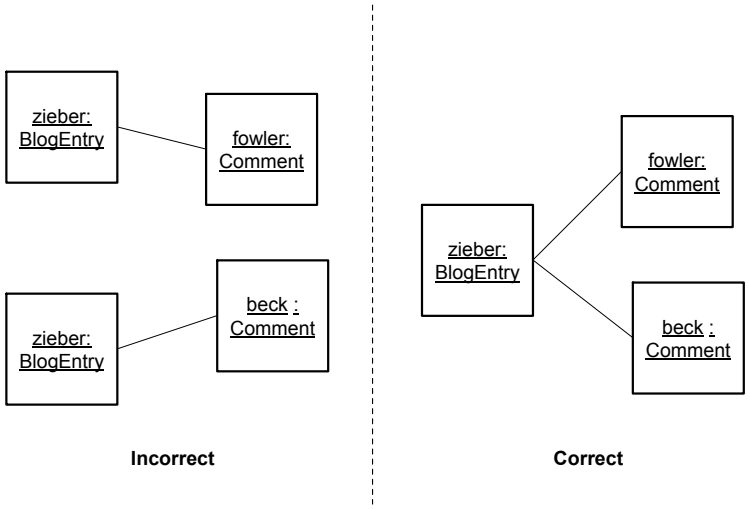

Figure 5.1: The problem of uniqueness.

Our primary incentive to use an identity map is to ensure *uniqueness* [48], which implies that we are always certain that we have not loaded data from the

same database record into two different objects. In general we consider uniqueness to be a good thing, but we have to bear in mind that future requests for old, stale data from an identity map in a given context may not be desirable. However, the situation that we wish to avoid is illustrated in Figure 5.1. On the left side we have two instances of the same `BlogEntry` fetched from the database, this is because no uniqueness is ensured and therefore we have two instances in memory.

The rationale behind our usage of the identity map is to have a series of maps that contain objects that are already fetched from the database. In our case we always have an isomorphic schema, which means that we can have one map per database table.

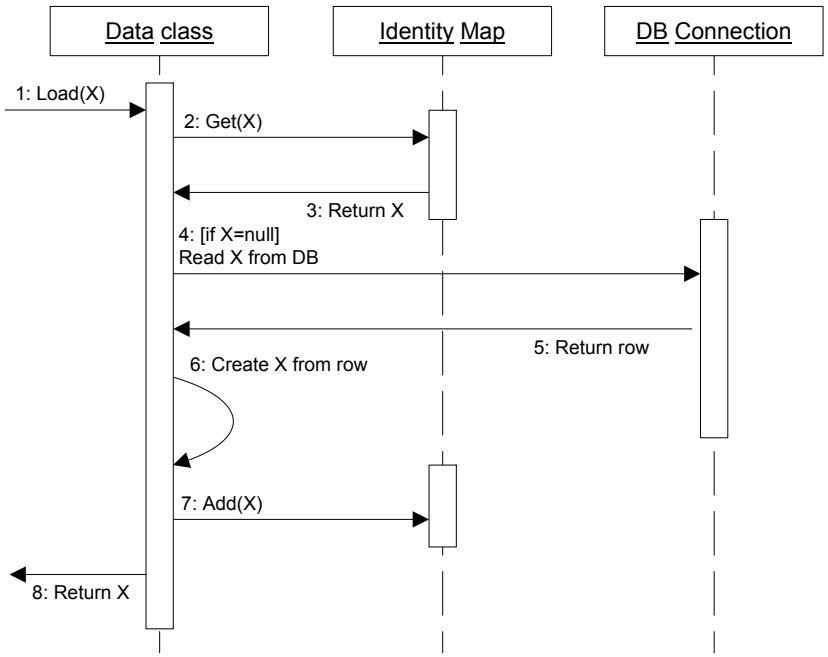

Figure 5.2: Sequence diagram for identity map.

A positive benefit we get by using an identity map is that we may improve the speed of lookups by using it, because if you load the same data more than once then you are incurring an extra cost in remote calls. However, we believe that the identity map is about ensuring uniqueness, not performance. This view is agreed by [21, ch. 3]. The reason that an identity map is more about uniqueness is because it can have an negative impact on performance as well. An example is that if you fetch a set of records, using a given search criteria, then you cannot know if all the entities in the identity map are the ones that are going to match

the query when you run it against the database. Discussions about this topic can be found at [45, 16].

When implementing an identity map there are a number of things to consider, one of these is the key for the map. An obvious choice is the primary key of the corresponding database table, which works well if the key is a single column and is immutable. More considerations about the identity map are in the context of matters like inheritance, transactional cache and immutable objects. As these considerations are not relevant to VDO we refer the reader to [21, ch. 11].

5.3 Lazy Load

Lazy load is applied where data is only obtained when it is needed, more specifically it is used with objects that do not contain all of their data you may need, but they know how to get the data if needed.

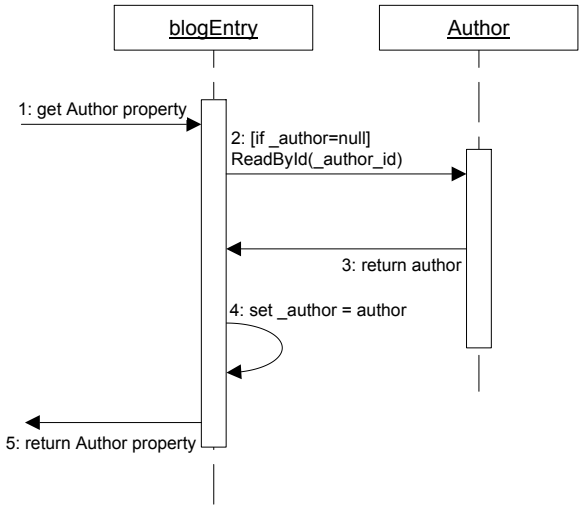

Figure 5.3: Example of sequence diagram for lazy load.

In regard to loading data from a database into memory it is expedient to design your API such that when you load an object of interest you also load the objects that are related to it. This makes the loading easier on the user that is using the object, who otherwise has to load all the objects he has interest in explicitly. However, there is a downside taking this approach, one object can have

the effect of causing a large and complex object graph by loading a huge number of related objects in chains, which is something that impairs performance when only a few of the objects are of actual interest. A lazy load approach halts this loading process for the moment, leaving a "marker" in the object such that if the data is needed it can be loaded only when it is used. An example of lazy loading is depicted in Figure 5.3.

There are four main varieties of lazy load [21, ch. 11]: *lazy initialization*, *virtual proxy*, *value holder* and *ghost*. These approaches are somewhat similar, but vary subtly and have various trade-offs.

However, the approach that is of relevance with regards to VDO is lazy initialization, which is the simplest of the four approaches. The basic idea is to have a special marker value, e.g. null, to indicate that a field is not loaded. Using this approach entails that every access to the field checks if the marker value is null, if it is then the value of the field is calculated before returning the field. In order to ensure this works, we have to keep the field self-encapsulated, which means that all access to the field is done through a getting method or property.

Using null as a signal mechanism to state that a field has not been loaded works well, unless null is a legal field value. In this scenario we need to use something else to signal that the field has not been loaded.

The positive aspect with lazy initialization is that it is simple, however it tends to force a dependency between the object and the database [21, ch. 11]. For that reason it works best in concert with architectural patterns like *Active Record* and *Table Row Gateway*.

According to [21] deciding when to use a lazy load pattern is all about deciding how much you want to fetch from the database as you load an object, and how many database calls that requires.

5.4 Observer

The observer pattern's intent is defined by [23, p. 273] as a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. The motivation for using the observer pattern stems from a common side-effect of partitioning an application into a series of cooperating classes, which is the need to maintain consistency between related objects without making the classes tightly coupled.

There exist many variations of the observer pattern, but the key objects in it are *subject* and *observer*. A subject may have any number of dependent observers, e.g. objects responsible for displaying data to a user. All observers are notified whenever a change occurs in the subject's state. In response, each observer queries the subject to synchronize its own state with the subject's new state. Figure 5.4 illustrates the logical relationship that exists between a subject and an observer.

The interaction that happens between a subject and an observer is also known

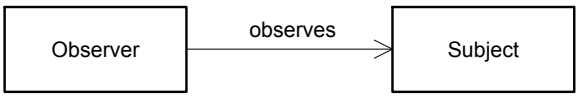

Figure 5.4: Observer and subject relationship.

as *publish-subscribe*. The subject is the publisher of notifications, without having to know who its observers are. Moreover, any number of observers can subscribe to receive notifications.

The observer pattern is especially applicable when the following conditions exist in your application [23, p. 274]:

- An abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects makes it possible to vary and reuse them independently.
- A change to one object requires changing others objects, and you do not know how many of them need to be changed.
- An object should be able to notify other objects without having to make any assumption about who these objects are. This implies that these objects should not be tightly coupled.

As with most solutions, the challenge is in the details, and the observer pattern is no exception to this. Even though the logical relationship illustrated in Figure 5.4 states that the subject is observed by the observer, this is a bit misleading when you implement the pattern. More correctly, when the observer wants to express its interest in observing, it *registers* with the subject. Then whenever a change occurs in the subject's state, it notifies the observer of the change. Furthermore, when the observer no longer has an interest in observing the subject, the it *unregisters* from the subject. These steps are known as observer *registration*, *notification*, and *unregistration*, respectively [38]. Most frameworks implement registration and notification by way of callbacks.

An example of a subject notifying observers is depicted in Figure 5.5. In order to register with the subject, the observer invokes the *Register* method of the subject, passing itself as argument. The subject receives this reference and stores it in order to notify the observer when a change in state occurs. A common place to store the observer instances is in a container dedicated to this purpose. When a state change occurs (*AuthorChanged*), the subject retrieves all the observers stored in its associated container by invoking the *GetObservers* method. The subject then iterates through the retrieved observers, calling the *Notify* method, which notifies the observer of the change.

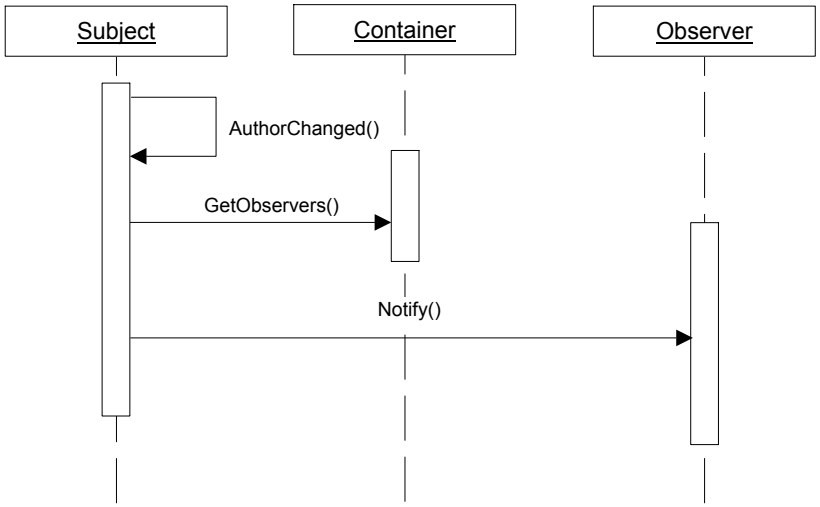

Figure 5.5: Observer and subject relationship.

The approach for unregistering an observer is similar to the one used for registering.

Many frameworks provide different ways of realizing the observer pattern. However, as we target for C# 2.0 we concentrate on those features provided by .NET that we employ in order to implement the observer pattern. In .NET you can take advantage of delegates and events, which provide a new and powerful means of implementing the observer pattern. In short, a delegate is the type-safe object-oriented equivalent of a function pointer. A delegate instance holds a reference to an instance or class method, and permits anonymous invocation of the bound method. An event is a special construct that is declared on a class in order to expose state changes to interested objects at run-time. An event represents a formal abstraction of the registration, unregistration and notification methods that were mentioned previously. Delegates can be used in conjunction with events, which encompasses that delegates can be registered with specific events at run-time. When an event is raised, all registered delegates are invoked so that they receive notification of the event.

In .NET the observer pattern is also known as the *Event Pattern*. In general, this pattern is expressed as formal naming conventions for delegates, events, and related methods involved in the event notification process. It is recommended that all frameworks that utilize events and delegates adopt this pattern. An example of using the *Event Pattern* in C# 2.0 is given in Listing 5.1.

```
1 public class BlogEntry {
2
3   // Declare a delegate for the event
4   public delegate void AuthorChangedHandler(object sender,
```

```
5                 AuthorChangedEventArgs e);
6     // Declare the event using the delegate
7     public event AuthorChangedHandler AuthorChanged;
8
9     Author _author;
10    public Author Author
11    {
12      set
13      {
14        _author = value;
15
16        // fire the event
17        OnAuthorChanged();
18      }
19    }
20
21    // Method to fire event delegate with a proper name
22    protected void OnAuthorChanged()
23    {
24      AuthorChanged(this,
25                    new AuthorChangedEventArgs(_author));
26    }
27  }
28
29  // Specialized class for the AuthorChanged event
30  public class AuthorChangedEventArgs:EventArgs {
31
32    private Author _author;
33
34    public PriceChangedEventArgs(Author author)
35    {
36      _author = author;
37    }
38    public Author Author
39    {
40      get { return _author; }
41    }
42  }
```

Listing 5.1: *Event Pattern* example.

A subject only needs to expose an event, nothing more is required. However, the observer must create a specific delegate instance and register it with the subject's event. In addition, an observer has to employ a delegate instance of

the type specified by the event, otherwise the registration fails – in the example given in Listing 5.1, this type is the one specified at line 7. During the creation of the delegate instance, the observer passes the name of the method, instance or static, that is notified by the subject to the delegate. After the delegate is bound to the method, it may be registered with the subject's event. The delegate may be unregistered from the event in the same manner. Finally, subjects provide notification to observers by invocation of the event.

5.5 Concluding Remarks

In this chapter we have described the most apparent characteristics of the applied patterns in our solution. We gave examples of their usefulness and applicability, as well as their relation to VDO. The background on design patterns was given in order to manifest what the foundation of VDO's design is based on.

III

Design & Realization

6 Specification of Requirements

As covered in Section 2.2 on page 34 we would like to support the application developer in the communication with an RDBMS by automatically generate classes to wrap and ease this communication. In the following section we give our requirements for such a solution as seen from a user perspective. As VDO is aimed at application developers, the requirements are established on the basis of our understanding of that user perspective together with descriptions of how the expected programming experience should be.

At this point it should be evident that we are going to generate a specialized API, which helps the application developer when communicating with a RDBMS. Some of our requirements repeat parts of the definitions for the chosen design patterns in order to clarify how we find those parts to be of relevance in the design of VDO, thus giving a more thorough understanding of how the chosen patterns fit into our needs.

6.1 Requirements for the Generated Classes

The requirements we have for the generated classes are sorted into the following different categories.

1. User Oriented Solution.
2. Automatic Class Generation.
3. CRUD Operations.
4. Impedance Mismatch Concerns.
5. Unique Data.

6. Concurrency Concerns.

7. Observable Changes.

In the following sections we will give a thorough explanation for each of the categories.

6.1.1 User Oriented Solution

As a first and very important requirement we find it to be crucial that VDO is aimed at application developers as they are the users of VDO. We find that VDO should be user oriented instead of being query oriented, thus the application developer should not feel the need to think in both object oriented and relational ways, but instead be able to concentrate on the object oriented world only. This should be done without hiding the fact that used data is saved in an external data storage, thus explicit storage operations should be implemented in an intuitive and simple way to let the application developer control when to save or delete data from external data storage.

It is imperative that the generated classes only have accessible methods for needed functionality, and hide methods used internally for implementation purposes in order to not revealing irrelevant details or confuse the user and to maintain the guidelines for API design as covered in Chapter 4 on page 43. Further, to assist the application developer it is of significance that accessible methods and properties are described and documented wisely to support the development process, e.g. describing behavior and possible exceptions.

To simplify the usage of VDO and methods that communicate with the RDBMS the handling of database connections should be taken care of by VDO, not by the developer. Consequently, the developer should only have to define the database connection once in a single place before using the auto-generated API, subsequently he should not worry about maintaining the database connection.

6.1.2 Automatic Class Generation

The process of generating the API classes should be automated in setting up the needed classes such that the developer can focus on developing the application. This implies that the developer should only have to define how to access the database schema in the RDBMS, i.e. provide connection parameters such as hostname, username and so on, and then let the generation tool take care of everything else.

Some of our requirements result in the need of saving extra information in the RDBMS. It is imperative that the original schema remains untouched in regard to existing tables (and possible views, triggers, etc.) so that existing applications can continue to use the RDBMS simultaneously with a new application using the

generated API. Therefore it is significant that the required information for VDO to work is saved independently from the clients using the data.

In conclusion, this means that the procedure for class generation should both make necessary modifications to the database schema and generate classes for communicating with the database schema in question.

6.1.3 CRUD Operations

As the generated classes are supposed to replace the need for direct communication with the RDBMS they should make available the most common data operations needed when using an RDBMS as a data storage - the CRUD operations. To leave no room for errors in the communication with the RDBMS it is of importance that queries are performed implicitly through methods on the classes and objects during run-time, thus hiding the communication.

In VDO we would like to join **create** and **update** into a logical *save* operation to keep it more simple for the developer to use. This way he does not have to think of whether the data object has been saved prior to the present state. To ensure consistency the developer is only allowed to save if no other client has saved a version of the data object which is newer than the one at hand, otherwise the operation should result in an error. This is explained in greater detail in Section 6.1.6 on page 72.

With regards to creating a new object, the user has to use a constructor which takes all the required values as parameters. These values could be either primitive values or another data object as a result of a foreign key relationship. When the new data object is created it can be saved without the application developer having to perform any special operation even though it is being saved as a new row in the corresponding table.

The **delete** operation should remove the corresponding data row from the RDBMS. If another client has saved a newer version of the corresponding row in the RDBMS the delete operation should be canceled and the there should be served an exception. If another client has deleted the row then no modification to the database is necessary as both of the involved clients (the client which has already deleted the row and the client trying to delete) are both agreeing that the row should be removed from the RDBMS, thus there is no need to raise an error.

Prior to deleting the data object at hand there should be performed a local *cascading delete* with the data objects, i.e. every data object which has a reference to the data object at question should be deleted from the RDBMS first as they cannot exist in the tables when the referential integrity is broken [44, ch. 6]. Though this could be handled by the RDBMS we have chosen this solution to ensure that the local data objects are in fact updated according to the

corresponding tables.

After the data object has been deleted it behaves like a newly created object, thus it should be possible to save the object afterwards resulting in a new row in the corresponding table with a new identity. This is due to the fact that no matter which data object the application developer accesses the same operations should be available without any precautions needed to be taken.

The **read** operation should be implemented such that it is possible to make a simple search on any of the columns in the table. In this context a simple search should be perceived as an equivalent to the SQL query shown in Listing 6.1 where there is only given one parameter which the resulting rows should conform to. It should not be possible to combine several search criteria into one read as this would bring an unneeded amount of complexity to the specialized API. Further, in this context a read operation is considered complex if results should conform to more than one search parameter.

```
1 SELECT * FROM authors WHERE name='Anders Hejlsberg'
```

Listing 6.1: A simple SQL search query.

The result for such a simple search should a collection of those data objects that match a given search criteria, e.g. a collection with all data objects with a given zip code. The read operation should take a search criteria that is used to read those data objects with an exact match to the given value.

If the *read* operation is performed on a primary key column which is the only primary key column then the result should be the one data object corresponding to this specific primary key instead of a collection with at most one data object matching the search criteria. If the requested primary key does not exists the application developer should be served with an exception.

As this type of operation supports input from the end-user of the developed applications it necessary that it is aware of injection attacks, thus implementing prevention of injection attacks directly into the generated classes.

The CRUD operations that work on all rows in the table should be static on the generated class (i.e. the *read* methods) and methods that work on a single row in the table should be available as instance methods at each instantiated data object (e.g. the *save* method).

6.1.4 Impedance Mismatch Concerns

For each table in the used database schema there should be generated exactly one class to both wrap any communication with this table and to represent each row in the table as a data object during run-time, cf. the *Active Record* design pattern as covered in Section 5.1 on page 56.

To solve those problems put forward in Section 2.1.2 on page 9 we want the accessible data on the data objects to be strongly typed in a way so the developer can use the built-in data types in C# so the developer will not have to think about converting the values. Through the data objects it should be possible to access data from the row, which the data object is modeling.

The accessible data should take special care of those data types from the RDBMS that have no direct equivalent type, e.g. `nvarchar`. These non-equivalent types should be mapped to the C# type, which has the nearest fit and the corresponding properties should be equipped with functionality to perform checks during run-time to ensure that restrictions like size limitations are not violated. Furthermore, the possibility of nullable types in C# should be utilized in order to make a correct mapping from data rows that contain null as values. As nullable is directly supported in the C# primitive types this would further let the developer use the built-in types without having to think in ways of the database.

The supported database types should be:

- `boolean`: In SQL there is no `boolean` primitive but instead a `bit` primitive which is basically the same with 0 for `false` and 1 for `true`. This is mapped to the `boolean` type in C#.
- `datetime`: Fortunately the `datetime` primitive from SQL is also found in C# so this i just a direct mapping to `datetime` in C#.
- `numerics`: We would like to support any numeric values in SQL and map it to its corresponding primitive in C#. Fortunately every numeric type from SQL has an equivalent type in C#, e.g. the SQL type `smallint` maps to the `int16` type in C#.
- `strings`: String values in SQL and in C# vary as we have mentioned several times through this thesis. Every of these string types from SQL should be accessible as a C# `string` type and checked during run-time whenever being set by the application developer to ensure that restrictions from the SQL types are not violated.

To simplify foreign key relations from the RDBMS in the specialized API the primitive value of any foreign key should be hidden from the application developer and instead the user should have direct access to the data object representing the data row in question. As a foreign key is just another way of saying that a certain column in a table *references* a primary key column in another table. To translate this into the terms of C# it would result in a direct reference to another data object at run-time, thus we also use *references* to explain when one data object has a direct reference to another data object as a result of a foreign key relation in the corresponding table. Further, we also use *referenced by* to explain when a

data object is referenced by another data object, as *referenced by* is the opposite of *references*.

At any given moment the application developer should have the ability of reading a timestamp of the latest modification made to the RDBMS during runtime as he could be interested in this particular information. As we do not want to hide the fact that we use an external data storage we find it to be useful to know the exact time when the current data object was saved to the used RDBMS.

6.1.5 Unique Data

In a running application there should be no duplicate data objects, i.e. any row from the RDBMS are at most represented by one data object in the running program, thus if the application developer requests a data object that is already present in the identity map the data object from the identity map should be returned instead of creating an identical copy.

This is also the case for collections with search results, which means that two search requests with equal search criteria should result in the same collection and not just two identical collections that contain the same data objects.

The reason for this is that we want these collections to be automatically updated (described in the following section) and thus keep the amount of collections as subject to updates as low as possible. Further, we find it unnecessary to create a new collection if there already exists one that matches the search criteria. A final argument for this behavior is that if we were to create a new collection every time a search was performed we would have to poll the RDBMS instead of returning the collection with already loaded data objects.

To ensure that the returned collections with data objects always conform to the search criteria they should be *read-only* to prevent the application developer from altering the content.

6.1.6 Concurrency & Synchronization Concerns

As we established in Section 2.2 on page 34 we want to implement a lightweight concurrency model with considerations to the options available when using an RDBMS for data storage. Our lightweight concurrency model has two principal aspects to it, which we elaborate on in the following.

The first aspect is to prevent any VDO-based application from deleting or overwriting data in the RDBMS if it was retrieved at a time prior to the time of the most current data found in the RDBMS. This should result in a behavior which resembles two concurrent transactions with conflict serializability [44, ch. 15].

The second aspect is that the retrieved data objects are automatically kept at a state that represents the present state of data in the RDBMS such that changes made by another application are reflected automatically without having

the application developer to manually perform these updates. This automation should be handled by a *synchronization agent*, which works in concert with the identity map to keep every loaded data object synchronized by polling the RDBMS for changes at frequent intervals and thus updating the data objects accordingly to the polled data.

If a data object that is subject for update has unsaved local modifications, then these local modifications should not be overwritten but instead the data object should be flagged so that it cannot be saved (i.e. overwrite the newer row in the RDBMS) at a later point. If the application developer tries to save a flagged data object then he is served with an exception and has to take action to resolve this conflict.

If a data object gets updated to reflect changes in the RDBMS, then it should be possible for the developer to be informed about the changes by way of events. Moreover, the developer should have the option to adjust a synchronization interval value on each generated class that controls the time interval between each synchronization with the database.

Furthermore, collections that represent search results should always conform to the original search criteria given and be updated whenever its contained data objects change such that they always consist of data objects that match the search criteria. This means that if a data object no longer matches the search criteria for a collection, regardless whether it is modified and saved locally or updated due to another clients modification, then it is removed from the collection, and if a data object fulfills the search criteria for a collection then it is added if it was not a member beforehand.

6.1.7 Observable Changes

The developer should have the ability to be informed of any changes to the data objects, e.g. when a data object has been saved to or deleted from the database or when a data property on the data object has changed.

In order to keep the programming burden for the developer as low as possible it should be possible to be notified of these events as soon as they happen, i.e. the developer should be able to subscribe to events that represent changes in the same manner as in the event pattern, described earlier in the report.

This is in keeping with our concurrency concerns, which means that when a local data object is updated due to modifications made by another client the application developer is able to act upon this.

As mentioned earlier the collections that contain data objects from a read operation should be updated automatically according to the RDBMS used as back-end. Whenever there is a change in the count of elements, e.g. when a new data object is added to the collection, the application developer should also have the opportunity to act on this, thus an event should be fired from a collection whenever a data object is added or removed.

7 Design by Task Scenario

Section 4.2 on page 45 stated that the requirements for the API should take form of use cases and that we should code to the interface of the API definition early. This gives rise to taking a design approach that is in keeping with those practices.

Our approach to this is to communicate the requirements specified in Chapter 6 on page 67 in a tangible manner such that we achieve a sound design for the classes that are generated and the supporting parts of VDO that the user is exposed to. By having a design for the parts the user is exposed to entails that we have a solid foundation, when constructing the class generator and supporting classes in the VDO framework.

We implement our design approach by creating a set of *task scenarios* that identify threads of usage for the API that we construct. A task scenario is somewhat similar to a use case known from [31], however it deviates in that it is comprised of some other components that we hold are appropriate when designing an API such as VDO. Details about the components that constitute a task scenario are described in Section 7.2 on page 77.

The task scenarios root in the different areas of concerns laid out in the requirements in Chapter 6 on page 67, which means that we define the task scenarios accordingly. By defining scenarios of usage we are able to systematically elicit functional and operational requirements for the generated classes together with supporting classes in the VDO framework. Thus, the scenarios jointly constitute the possibilities and characteristics of generated classes.

In order to design the parts the user is exposed to we make use of an existing database schema that serves as a *design example* that is used in connection with the task scenarios. This database schema is presented in Section 7.1 on the following page. By using a design example gives us the advantage of having a concrete guide to design classes and specify behavior from, because each task scenario is rooted in the database schema.

However, though we use a concrete design example it does not impose obstructions to identify general interface patterns and characteristics of the parts that should be exposed to the user. This is due to the database schema is selected on the grounds that it is representative and sufficiently complete to showcase the requirements demanded in Chapter 6 on page 67 and to generalize from when building the class generator.

The final result that originates from the design example is subject for being generalized into patterns of interfaces that eventually constitute any generated class' structure.

7.1 Design Example

The design example that we employ to construct the API of generated classes is the database schema *WebLog* depicted on Figure 7.1. This schema sets the stage for the task scenarios that are being presented in the following sections. The schema is chosen on the basis that it has the necessary composition in order to manifest the requirements that the generated classes have to fulfill. The task scenarios are rooted in the tables of *WebLog*. It is from these tables that we continuously develop the design the part of the VDO framework that is exposed to the user. The final result should give us a design that we can generalize from when constructing the VDO framework.

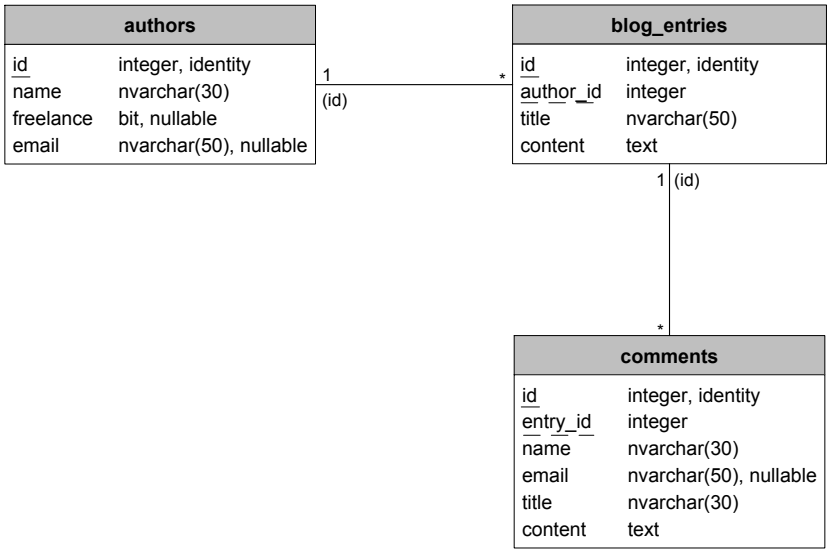

Figure 7.1: Tables in *WebLog*.

The tables in Figure 7.1 are employed in connection with a simple application that is hosted on a web server. The table *authors* represents persons that are affiliated with a web site. Each of these person can publish blog entries that are

stored in the *blog_entries* table. After having published a blog entry, the author can get feedback on it as a series of remarks made by visitors of the web site, these are stored in the *comments* table.

The table *authors* has some columns that impose restrictions on how the corresponding generated class' behavior and interface are defined. The column *freelance* has the data type `bit` and is nullable. This means that the value of the *freelance* column in a row may be unknown. Furthermore, the columns *name* and *email* in *authors* have both the type `nvarchar` with a character length restriction, which implies that the corresponding class should take measures to ensure it complies with this. The *id* column of *authors* has the type `integer` and is *identity*, which means that the RDBMS seeds unique *id* values by way of auto-incrementation, i.e. the generated class should not provide a means to set the value of *id*.

Tables *blog_entries* and *comments* have some similar characteristics that resemble those of *authors*, however as indicated in Figure 7.1 they both participate in a foreign key relationship, which implies that an appropriate modeling has to be done in the corresponding classes in order to reflect relationships among tables.

7.2 Task Scenarios

In the coming sections we employ task scenarios as a means to continuously building up the interface for the parts of VDO that are exposed to the user. We presuppose that the user has generated a DLL file from the schema in 7.1 on the facing page, and concentrate on behavior and interfaces of generated code.

The level of abstraction we use for describing task scenarios is narrative, therefore the scenarios may include a fair amount of information, which gives rise to employ a template to systematically describing scenarios in a common format. The template is composed of a series of components that together constitute a task scenario. The components are individually elaborated on in the following itemized list:

- **Task Scenario:** A short description that explains the relevance of the particular task scenario. It includes matters like what prompts the task scenario.
- **Classes:** VDO classes involved in the task scenario that the user is exposed to. We do not list classes from C#, even though these may appear code examples.
- **Methods:** Methods, if any, involved that are relevant to the VDO classes.
- **Interface:** Here we specify how we find a task scenario best represented in VDO, that is we argument for different aspects of the interface that is a

result of the task. This comprises parts the user is exposed to, as well as choices we take with regards to accessibility, naming of interface components and so on.

Moreover, we also illustrate interface code that we *extract* from the task scenario in this part. The code listed resembles C#, but it is prototypical and does not compile, nor does it strictly conform to the syntax of C#. Its principal purpose is to illustrate and assist rather than being syntactically correct.

- **Pre-conditions:** A list of conditions, if any, that have to to hold before commencing the task. We would like emphasize that a pre-condition in the context of a task scenario is not on the same footing as a method pre-condition as we operate on a higher level of abstraction.

- **Post-conditions:** A list of conditions, if any, that have to hold after the task.

- **Exceptions:** A list of exceptions, if any, that may be thrown. We do not specify the interface for exceptions, because we find it to be sufficient to only know the name and which type of error message is provided.

  In some cases a violation of a task scenario pre-condition can cause an exception to be thrown. Idealistically one ought to assume that pre-conditions are fulfilled, that is the user has the responsibility to ensure this. However, in order to define when and what exceptions can be thrown entails that we do not preclude this part.

- **Code Example:** Here we give a small code example that demonstrates the interface we specify for the task scenario. The code listed in the example is syntactically correct.

Some task scenarios involve more material if they are general enough, then we try to avoid repeating already made statements or arguments in other task scenarios. Furthermore, some task scenarios do not list all of the components in the aforementioned template, which is due to that some subjects do not apply in a given task scenario.

Before presenting task scenarios we would like to point out that we have chosen to present code examples that exemplify a task scenario at the end of each scenario, this is done on the grounds that code examples are based on the design decisions that are previous to the example.

7.2.1 Initialize VDO

- **Description:** This task scenario represents a fundamental task in VDO, which is to initialize it such that a connection to the database can be established.

- **Classes:** `VDOEngine`, `DataProvider`.

- **Interface:** The requirement needed in order to initialize VDO involves one task from the user's side. The user is exposed to is a static access class named `VDOEngine`, for which there exists a static property named `DataProvider`. The task required in order to initialize `VDO` is to set an instance of a provider specific class that implements the interface `DataProvider` to the static property. The particulars about the interface `DataProvider` is deferred until Chapter 8 on page 105.

  The interface that is deduced from this task scenario is depicted in Listing 7.1.

```
1 public static class VDOEngine
2 {
3  public static DataProvider DataProvider { set; }
4 }
```

Listing 7.1: Interface of the `VDOEngine`.

- **Pre-conditions:** The following conditions have to hold for all task scenarios:

  ▷ **Pr.1**: The generated DLL and the VDO framework DLL have to be referenced and imported in the IDE.

  ▷ **Pr.2**: A RDBMS to be available and running.

- **Post-conditions:** The following condition has to hold for all task scenarios:

  ▷ **Po.1**: The specified `DataProvider` is employed, whenever an interaction is made with the RDBMS.

- **Code Example:** The example in Listing 7.2 showcases the necessary step required in order to initialize VDO.

```
1 VDOEngine.DataProvider = new MSSQL2005DataProvider("host", "
    database", "username", "password");
```

Listing 7.2: Initialize VDO.

7.2.2 Create a data object

- **Description:** This task scenario represents one of the most basic things that a user can do in VDO, namely creating a new instance of a generated class.

  After generating classes from the database schema, the user should be able to create a data object by way of a constructor that has parameters that reflect data types and constraints of the columns in the corresponding table.

  It is significant that it is as intuitive as possible for the user to carry out the instantiation of a data object in concert with specifying a references relationship.

  In this task scenario we use the table *blog_entries* in Figure 7.1 on page 76 as basis for interface and example code.

- **Classes:** `BlogEntry`, `Author`, `ValueRequiredException`, `StringLengthException`.

- **Methods:** `BlogEntry` constructor.

- **Interface:** The reason that we employ a constructor with parameters in order to instantiate a data object rather than using a empty constructor and setters is that we at compile-time can ensure that all necessary values are supplied to the insert SQL statement.

  Other important parts to emphasize are that we wish to adhere to naming conventions in the platform, which means that we transform the names in the corresponding table schema in a way such that parameter names in a constructor reflect the naming convention in C#, yet have a close resemblance to the original names in the table schema. An example of this is where a column is named *User_Rating*, for which the parameter is named `userRating`. Having meaningful names for parameters contributes to the documentation value of the API.

  Likewise, the name of a generated class should be the corresponding table name in singular form with the initial letter in upper-case. This means that the table *blog_entries* is being transformed to `BlogEntry`.

  Furthermore, as you can see in 7.3 an `Author` instance is the first parameter in the constructor of `BlogEntry`. This choice is a consequence of the fact that we wish to model a references relationship as a reference to the data object modeling the referenced row.

  We maintain that providing a reference to a data object in order to establish a references relationship is more in keeping with the object-oriented mindset than providing a value type that corresponds to the *author_id* column in the *blog_entries* table. This choice gives us a significant advantage, which

is ensuring that the value of *author_id* is valid, thus sparing the user from the possibility of exceptional processing.

The interface that is extracted from the task scenario is given in Listing 7.3.

```
1 public BlogEntry(Author author,string title,string content)
```

Listing 7.3: Interface for constructor part of `BlogEntry`.

- **Pre-conditions:**
  - ▷ **Pr.1**: Parameters of the constructor are in accordance with the schema of a table the data object is modeling. That is, they reflect data types and constraints of corresponding columns. There are two special cases that have an impact on a parameters' existence and type, when the corresponding column is:
    - ∗ **Pr.1a**: An identity. In this case no parameter is provided.
    - ∗ **Pr.1b**: A foreign key. In this case the corresponding parameter is a reference type as we require that the user provides a data object that represents the references relationship as an argument in the constructor.
  - ▷ **Pr.2**: An argument provided in the constructor can only take the value null if its associated column in the corresponding table is nullable.

    Note: This requirement constitutes two cases that are handled differently:

    1. The first case is where a SQL data type maps to a value type in C#. This type of mapping provides the opportunity to check at compile-time whether a provided argument is valid, as there is a syntactical difference between nullable and non-nullable value types, e.g. `int?` and `int`.

    2. The second case is a SQL data type that is mapped to a reference type in C#, for instance from the SQL type `nvarchar` to the C# type `string`. This type of mapping does not allow us to determine at compile-time whether the provided argument is valid, because a `string` can reference `null`.
       As we do not wish to introduce new types for the user implies that the validity of the argument is determined at run-time.
  - ▷ **Pr.3**: For any parameter in the constructor that is of type `string` the provided argument's length is not greater than the corresponding column's length restriction.

- **Exceptions:**
  - ▷ `ValueRequiredException`: If **Pr.2** does not hold and the constructor argument is a reference type, then an exception is thrown at run-time. The exception contains a string property that describes the error message in detail.
  - ▷ `StringLengthException`: If **Pr.3** does not hold, then there is thrown an exception at run-time indicating the error in detail by providing a string property that describes it.
- **Code Example:** The following example in Listing 7.4 serves as purpose to exemplify how a data object is created. As seen at line 1, the constructor of an `Author` instance allows for taking `null` as argument, because the corresponding columns *email* and *freelance* in the table *authors* are nullable.

  Moreover, line 2 demonstrates what is required of the user in order to establish a relationship between a `BlogEntry` instance and an `Author` instance.

  The data objects in Listing 7.4 are still subject to be saved in the database, this is elaborated on in Section 7.2.5 on page 89.

```
1 Author author = new Author(name, null, null);
2 BlogEntry blogEntry = new BlogEntry(author, title, content);
```

Listing 7.4: Creation of data objects.

7.2.3 Read a field of a data object

- **Description:** This task scenario encompasses an important requirement, which comprises the ability to read columns that a data object models in a way that is lucid to the user. This implies foreign key and primary key columns that form a part of a relationship to another table as well as regular columns.

  Once more we employ the table *blog_entries* in Figure 7.1 on page 76 as basis for interface and example code.

- **Classes:** `BlogEntry`, `Author`, `VDOCollection<Comment>`.

- **Interface:** The task required of a user in order to read a field value should only consist of reading the value of a property in a data object that corresponds to the field's column. However, there is an exception to this in that there is no corresponding property for a foreign key column, as we consider that an intuitive way of representing a references relationship is with a property that is a reference to the data object modeling the referenced row, as done with constructor parameters.

We would like to emphasize that we retain a consistency between naming and ordering of parameter names and property names, the only difference is that property names have their initial letter in upper-case according to the platform convention.

As a data object can model a row that is referenced by rows in other tables, we maintain that a simple and comprehensible approach to represent this in a manner that is object-oriented is by way of providing the user with strongly typed collection properties, where each of these contains data objects that associate to a specific table.

These strongly typed collections are read-only and of type `VDOCollection<T>`, where `T` is generated class from table T. It is not our intention to get into the particulars about the interface of `VDOCollection<T>` other than stating that it should implement `IEnumerable<T>` to support the `foreach` statement, as well as providing an indexer to get an element at a specified index, a `Count` property to get the number of elements contained in the collection, the method `IndexOf` for determining the index of a specific element, and the method `Contains` to determine whether an element is in the collection. As stated in Section 6.1.7 on page 73 it should be possible to subscribe to changes in collections. We defer the discussion about this until Section 7.2.9 on page 99.

The naming of a collection property should be from the name of the class parameter of the collection in plural. For instance, a `BlogEntry` instance contains a property `Comments` of type `VDOCollection<Comment>`.

The resulting interface that is extracted from the task scenario is illustrated in Listing 7.5. The vigilant reader may have noticed that not all properties have a set part, the reason for this is elaborated on in Section 7.2.4 on page 86, where we discuss the particulars that come into play when assigning values to properties.

```
1 public int Id { get; }
2 public string Title { get; set; }
3 public string Content { get; set; }
4 public Author Author { get; set; }
5 public VDOCollection<Comment> Comments {get;}
```

Listing 7.5: Interface for properties of `BlogEntry`.

- **Pre-conditions:**
  - ▷ **Pr.1**: The data object's `Save` method is invoked prior to this task scenario.
  - ▷ **Pr.2**: In accordance with the schema of a row's table, which a data object models there exist corresponding properties, apart from foreign

key columns that are represented as a properties that are data objects that model referenced rows.

Additionally, for a primary key column in a row that a data object models, which is referenced by rows in table *C*, there exists a strongly typed `VDOCollection<C>` property, where `C` is a generated class from table *C*, that contains data objects that model these rows.

- **Post-conditions:**
  - ▷ **Po.1a**: If **Pr.1** holds, then for all properties that have a corresponding column the following applies: the value that is read from the property reflects the last fetched value of the corresponding row's column in the associated table.
  - ▷ **Po.1b**: If **Pr.1** does not hold and a property's corresponding column is:
    - ∗ identity, then the value that is read from the property defaults to 0.
    - ∗ not identity, then the value that is read from the property is equivalent to a value that either was assigned to the property or provided by the property's associated constructor parameter.
  - ▷ **Po.3**: For every property of type `VDOCollection<C>` where `C` is a class generated from table *C*, the following applies:
    - ∗ For each row in table *C* that has a references relationship to the data object's associated row there is a corresponding instantiated data object in the strongly typed collection. If no such row exists in *C*, then the strongly typed collection is empty.
      The motive for having an empty collection rather than returning `null` is due to the best practice in Section 4.3.2 on page 51 that recommends that we avoid returning values that demand exceptional processing.
    - ∗ `VDOCollection<C>` is automatically kept at a state that reflects the present state of references relationships that data objects of type `C` share with the property's data object.

- **Code Example:** The example in Listing 7.6 depicts a simple scenario that demonstrates three different kinds of reads that can be done on a data object.

```
1 Console.Write("Blog entry \"{0}\" is owned by author: {1}",
2                blogEntry.Title, blogEntry.Author.Name);
3
4 // Listing row data
5 foreach(Comment c in blogEntry.Comments)
```

```
6 {
7  Console.WriteLine(c.Title);
8  Console.WriteLine(c.Content);
9 }
```

Listing 7.6: Reading properties of data objects.

The first argument to the `Write` method at line 1 shows how the *title* column in *blog_entries* is wrapped as the `Title` property of a `BlogEntry`. The second argument at line 2 is a bit more interesting in that we have an `Author` instance as a property instead of having the *author_id* as a property. The loop at line 3 demonstrates when a user wants to read all comments that belong to a `BlogEntry` instance, which in effect means rows from the table *comments* that have a references relationship to the row that the `BlogEntry` instance represents.

7.2.4 Change a field of a data object

- **Description:** This task scenario is relevant when a user wants to change the value of a column in a row that a data object represents, which effectively is analogous to altering the value of a property and saving the data object.

  For the time being we only concentrate on changes made on properties and defer the details about saving the data object's state in the RDBMS until Section 7.2.5 on page 89.

  As the user may be interested in notifying other objects about changes to the value of a property, the alteration of its value should cause an event to be published to observers that have an interest in change notifications of the property. This applies to changes that are made on the value of a property locally, as well as changes made by other applications on the associated column's value.

  For this task scenario we employ the table *blog_entries* as the foundation for interface and example code. This is due to the fact that *blog_entries* participates in two types of relationships, it references the table *authors* and is referenced by the table *comments*. This entails special treatment of some properties, which we discuss in the following.

- **Classes:** `BlogEntry`, `Author`, `VDOCollection<Comment>`, `ValueRequiredException`, `StringLengthException`, `TitleChangedEventArgs`, `ContentChangedEventArgs`, `AuthorChangedEventArgs`.

- **Interface:** We consider that an easily understood way of accomplishing the task of changing a column value in a row is to change it through the set part of the matching property in the data object. By taking this approach we ensure type-safety. Another way of changing fields could have been by providing a update method that takes two arguments, the name of a column and a value. However, this is considered by us to be error-prone as the user may inadvertently misspell the column name or provide a column name that has another data type.

  Furthermore, this task scenario also encompasses the possibility for changing the value of a row's foreign key column such that the row participates in a new relationship with a different row in the referenced table. As we do not expose foreign key columns as properties, the task of specifying a new relationship is a matter of using the property that models the referenced table as a means in order to change the relationship.

  The interface that applies to properties in this task scenario is given in Listing 7.7. It is evident from the listing that the properties `Id` and `Comments`

do not have a set part.

The reason for not having a set part on `Id` is that its corresponding column is a primary key that is identity, which means that we let the RDBMS take care of ensuring the uniqueness of the identity value.

The incentive for not a having set part for `Comments` is because we maintain that introducing a set part is more of an impediment than an advantage for the user. The way we introduce a references relationship in VDO is by passing a data object that models the referenced row in the constructor of the data object that references the row, e.g. passing an `Author` instance as an argument in the `BlogEntry` constructor.

```
1 public int Id { get; }
2 public string Title { get; set; }
3 public string Content { get; set; }
4 public Author Author { get; set; }
5 public VDOCollection<Comment> Comments { get; }
```

Listing 7.7: Interface for the properties of `BlogEntry`.

Lastly, we maintain that it is important that the naming of event arguments, events and delegates adhere to C# naming convention. Conforming to this should make it easier for the user to recognize the different parts of a data object that comprise the event pattern in C#. Listing 7.8 demonstrates event related code for the `Author` property of `BlogEntry`. It should be noted that `EventArgs` at line 3 in the Listing 7.8 originates in C#.

```
1 public delegate void AuthorChangedHandler(object sender,
     AuthorChangedEventArgs e);
2 public event AuthorChangedHandler AuthorChanged;
3 public class AuthorChangedEventArgs : EventArgs
4 {
5   public Author Author { get; }
6 }
```

Listing 7.8: Interface for event related code for the `Author` property.

All properties in a data object with a set part have event related code that follows the same naming and structure as the one given in Listing 7.8. This effectively means that we preclude the possibility to subscribe to changes on the `Comments` collection through an event in the `BlogEntry` instance. The incentive for this choice is elaborated on in Section 7.2.9 on page 99, where we discuss registering to changes in a `VDOCollection`.

- **Pre-conditions:**
  - ▷ **Pr.1**: A property can only take the value `null` if the column in the corresponding table is nullable.

- ▷ **Pr.2**: A property of type `string` leads to the requirement that the property can only take string values of a length that is not greater than the corresponding column's length restriction.

- **Post-conditions:**
  - ▷ **Po.1**: After changing a property `<P>` an event named `<P>Changed` with an argument of type `<P>ChangedEventArgs` is published to all observers that have subscribed on changes to the `P` property. The event argument `<P>ChangedEventArgs` has a property that is identical to the `P` property, except for missing the set part.

- **Exceptions:**
  - ▷ `ValueRequiredException`: If **Pr.1** does not hold, then an exception is thrown at run-time stating that a value is required. The exception contains a property that describes the error message.
  - ▷ `StringLengthException`: If **Pr.2** does not hold, then there is thrown an exception at run-time indicating that the length of the string is invalid by providing a property that has a descriptive error message.

- **Code Example:** The code example in Listing 7.9 demonstrates this task scenario in its entirety.

```
1 Author author = new Author("Dave Matthews", "dm@vdo.tk");
2
3 BlogEntry blogEntry = BlogEntry.ReadById(4);
4 blogEntry.Author = author;
5 blogEntry.Title = "Observational Affairs in VDO are Deft!";
6 blogEntry.Content = null;
```

Listing 7.9: Adjusting properties of a `BlogEntry` instance.

As illustrated on line 4, the task of changing the references relationship is a matter of assigning an `Author` instance to the `Author` property of `BlogEntry`. Subsequent to the assignment on line 4, an event is published to observers with an appropriate argument.

The assignment to `Title` at Line 5 causes a `StringLengthException` to be thrown as the corresponding column imposes a restriction on the string length of 30 characters. Moreover, the assignment at line 6 causes a `ValueRequiredException` to be thrown as the corresponding column is not nullable.

```
1 public class Observer
2 {
3  public void OnAuthorChangedHandler(object sender,
4                 BlogEntry.AuthorChangedEventArgs e)
```

```
 5  {
 6   int id = ((BlogEntry)sender).Id;
 7   Console.WriteLine("Blog entry with id {0} changed author
        to {1}", id, e.Author.Name);
 8   bool? free = e.Author.Freelance;
 9   if (free.HasValue ? free.Value : false)
10   {
11    Console.WriteLine("No interest in freelance entries.");
12    ((BlogEntry)sender).AuthorChanged -=
         OnAuthorChangedHandler;
13   }
14  }
15  static void Main(string[] args)
16  {
17   VDOEngine.DataProvider = new MSSQL2005DataProvider("
        localhost", "weblog", "wl", "wl");
18
19   BlogEntry blogEntry = BlogEntry.ReadById(1);
20   Observer obi = new Observer();
21   blogEntry.AuthorChanged += obi.OnAuthorChangedHandler;
22
23   Author author = new Author("Beck", true, "beck@vdo.tk");
24   blogEntry.Author = author;
25  }
26 }
```

Listing 7.10: (Un)Subscribing to changes in a data object.

The code in Listing 7.10 exemplifies how the event pattern from C# is integrated into a `BlogEntry`. An important thing to notice is how subscription and unsubscription to an event are done at line 21 and 12, respectively.

7.2.5 Save a data object

- **Description:** After creating or altering a data object, the user should be able to save it to the database. It is of paramount significance that the save operation requires as little involvement as possible from the user's part. This is especially true if the save operation entails a corresponding insert or update operation in the RDBMS.

  When a data object is saved to the RDBMS it may be of great utility to be notified subsequently to the save operation. The user may be interested in keeping objects responsible for displaying data informed about this. Thus, we hold that it is expedient that the user can subscribe to an event that elucidates a save operation on a data object.

  In this task scenario we employ the table *comments* as basis for interface and example code.

- **Classes:** `Comment`, `BlogEntry`, `OutOfSyncException`, `StateChangedEventArgs`.
- **Methods:** `Save`.
- **Interface:** The interface for a `Save` method has some important aspects to it that sets the scene for an elaboration. The choice of the name stems from the fact that a method should clearly communicate what it does. It saves data to the database, which is clear given the context of VDO: an object-oriented wrapper of a database. Thus, we omit a part of the method, which indicates that data gets saved to a database.

  Moreover, the name is chosen to be only `Save` regardless of the save operation causes an insert or update SQL statement in the database back-end. We maintain that it is not necessary to provide methods for each of these operations, `Insert` and `Update`, or having the method named `SaveOrUpdate`. The user should not be worried about matters that can be handled in the implementation.

  The interface that we extract from this task scenario with regards to the `Save` method is given in Listing 7.11.

```
1 public void Save()
```

Listing 7.11: Interface for the `Save` instance method of `Comment`.

The interface to event related code for a data object that originates from this task scenario is given in Listing 7.12. We adhere to the naming convention in C# for event arguments, events and delegates. However, as we operate with events that apply on object level we introduce the naming prefix `State` in order to emphasize a higher level of notification. The property `Change` of `StateChangedEventArgs` is an enumeration, `StateChangeEvent`, that is used as an indication for object level events. The enumeration value `Deleted` at line 3 is unconnected with this task scenario, however it applies to the task scenario in Section 7.2.6 on page 92, which employs the same event related code given in Listing 7.12.

```
1 public delegate void StateChangedHandler(object sender,
     StateChangedEventArgs e);
2 public event StateChangedHandler StateChanged;
3 public enum StateChangeEvent { Deleted, Saved };
4 public class StateChangedEventArgs : EventArgs
5 {
6  public StateChangeEvent Change { get; }
7 }
```

Listing 7.12: Interface for event related code for a data object.

- **Pre-conditions:**
  - ▷ **Pr.1**: If the data object associates to a row in the corresponding table, then the row's fields must not have been altered since the data object was last synchronized with the row's fields.
- **Post-conditions:**
  - ▷ **Po.1a**: If the data object does not associate to a row in the corresponding table, then a new row with valid data is *inserted* in the corresponding table.
  - ▷ **Po.1b**: If the data object associates to a row in the corresponding table, then the row is *updated* with valid data.
  - ▷ **Po.2**: Subsequent to saving a data object, the time its associated row was last modified is stored in the data object.
  - ▷ **Po.3**: A property that maps to a primary key column that is identity has a unique value that differs from the value of equivalent properties for data objects of same type.
  - ▷ **Po.4**: If data object `Parent` referenced by data object `Child` does not exist in *TableOfParent*, then data object `Parent` is automatically saved to *TableOfParent* prior to saving data object `Child` to *TableOfChild*.
  - ▷ **Po.5**: After invoking `Save`, an event named `StateChanged` with an argument of type `StateChangedEventArgs` is published to all observers that have subscribed on save notifications from the data object. The argument `StateChangedEventArgs` has an enumerated property of type `StateChangeEvent` named `Change`, which has the value `StateChangeEvent.Saved`.
- **Exceptions:**
  - ▷ `OutOfSyncException`: If **Pr.1** does not hold, then an instance of this exception is thrown. The exception contains a string property, which indicates the error message that the data object is out of sync with its associated row.
- **Code Example:** The code in Listing 7.9 illustrates the task required in order to save newly created data objects that form a part of a relationship. The only thing required executing this task is to invoke the `Save` method of the data object `comment` at line 5. The save operation performed on `comment` propagates to its parent `blogEntry`, which also propagates a save operation to its parent `author`.

```
1  Author author = new Author(authorName, email);
2  BlogEntry blogEntry = new BlogEntry(author, title, content);
3  Comment comment = new Comment(blogEntry, authorName, email,
4                                "Re: " + title, content);
5  comment.Save();
6  comment.Title = Regex.Replace(comment.Title,"Re: ","FYI: ");
7  try {
8   comment.Save();
9  } catch (OutOfSyncException e) {
10  // performing exceptional processing.
11 }
```

Listing 7.13: Saving data objects.

The invocation of `Save` at line 7 may cause an `OutOfSyncException` to be thrown. This entails synchronizing the state of the data object with the corresponding row's state. The particulars with regards to synchronizing a data object are described in Section 7.2.7 on page 94.

7.2.6 Delete a data object

- **Description:** This task scenario comes about when the user wishes to delete a data object, which analogously corresponds to deleting the data object's associated row.

  As we integrate the observer pattern and automatic synchronization in classes that are generated from tables, we maintain that it is of great value to the user if he can be notified when a data object's associated row is deleted. Thus, this task scenario also involves delete notifications.

  For this task scenario we employ the table *authors* as the basis for interface and example code.

- **Classes:** `Author`, `StateChangedEventArgs`.

- **Methods:** `Delete`.

- **Interface:** There are a few naming considerations we would like to go into details about with regards to the `Delete` method. Our choice of naming is a consequence of the fact that we only perform one type of deletion, which is to cause a row in the RDBMS to be deleted. As a data object wraps a row, the naming `Delete` is preferred over the naming `DeleteFromDB`, as this choice is consistent with the naming of the `Save` method.

  Moreover, as we do not wish to render a data object useless upon a delete operation, the naming `Destroy` is considered to be improper and misleading.

  The interface that originates in this task scenario is given in Listing 7.14. We omit interfaces for event related code owing to that we already presented it in Listing 7.12 in Section 7.2.5 on page 89.

```
1 public void Delete()
```

Listing 7.14: Interface for the `Delete` instance method a `Author` class.

- **Pre-conditions:**
  - ▷ **Pr.1**: Prior to invoking `Delete` it must hold that no modification has been performed on the row that the data object models since the last synchronization with the row.
- **Post-conditions:**
  - ▷ **Po.1**: Subsequent to invoking the `Delete` method, the associated row in the corresponding table is deleted.
  - ▷ **Po.2**: The value of the property, which maps to the primary key column that is identity is undefined.
  - ▷ **Po.3**: A cascading delete is performed on the data object `Parent`, which effectively means that `Delete` is invoked on each `Child` data object that references `Parent`, and the referenced by `VDOCollection<Child>` property of `Parent` has zero count in elements.
  - ▷ **Po.4**: After invoking the `Delete` method, an event named `StateChanged` with an argument of type `StateChangedEventArgs` is published to all observers that have subscribed on delete notifications from the data object. The argument `StateChangedEventArgs` has an enumerated property of type `StateChangeEvent` named `Change`, which has the value `StateChangeEvent.Deleted`.
- **Exceptions:**
  - ▷ `OutOfSyncException`: If **Pr.1** is violated, then a `OutOfSyncException` is thrown, containing an error message property that indicates that the data object is out of sync with its associated row.
- **Code Example:** The example in Listing 7.15 demonstrates which task is required of the user, when he wishes to delete a matching row from the RDBMS. After having invoked `Delete` at line 24, all rows in the table *blog_entries* that reference the row `author` models are deleted from the RDBMS. Furthermore, each row in the table *comments* that reference any of those rows in *blog_entries* are deleted from *comments*. As a result of invoking `Delete` an event is published.

```
1 public class Observer
2 {
3  public void OnStateChangedHandler(object sender,
4                   Author.StateChangedEventArgs e)
```

```
5   {
6    if (e.Change.Equals(Author.StateChangeEvent.Deleted))
7   {
8    // Remove data from the GUI.
9   }
10  else if (e.Change.Equals(Author.StateChangeEvent.Saved))
11  {
12   // Inform the user that data was saved.
13  }
14 }
15
16 static void Main(string[] args)
17 {
18   VDOEngine.DataProvider = new MSSQL2005DataProvider("
        localhost", "weblog", "wl", "wl");
19
20   Author author = Author.ReadById(6);
21   Observer obi = new Observer();
22
23   author.StateChanged += obi.OnStateChangedHandler;
24   author.Delete();
25   Console.WriteLine("No. of blog entries: {0}",
26   author.BlogEntries.Count);
27   author.Save();
28  }
29 }
```

Listing 7.15: Deleting a row from the *authors* table in the RDBMS.

Line 27 entails that a new row is inserted in the table *authors* and that the property `Id` of `author` has a unique value different from 6, cf. line 20.

7.2.7 Synchronize a data object

- **Description:** This task scenario is due to a need from the user to manually synchronize the state of a data object with values from the associated row in the corresponding table.

  Here we assume a setting where other users may have altered the values in the row through other applications. In such a setting we think it is significant that the user has the ability to adjust to the last change that has happened to a row of his interest.

  The table used for interface and example code in this task scenario is *comments*.

- **Classes:** `Comment`.

- **Methods:** `Refresh`.

- **Interface:** A thing that we would like to emphasize is the naming of the method. We exercised great care in selecting the name for the method in order to have a close connection between naming and method behavior. We maintain that the name `Refresh` has a stronger semantic meaning than the name `Update`, given what the method does. The name `Update` has another meaning in the realm of databases, which is to update data in the database.

  Candidates for naming that also are strong in semantic meaning are `Poll-Newest` and `Synchronize`. However, given that we have a method named `Save` we select what we find most supplementing and lucid: `Refresh`. One advantage in the name is that it somewhat communicates that the method has to be invoked each time the user wishes to synchronize, whereas the two other names can arguably be understood as a having a continuing effect.

  The interface that stems from the task scenario is given in Listing 7.16.

```
1 public void Refresh()
```

Listing 7.16: Interface for the `Refresh` instance method of `Comment`.

- **Post-conditions:**

  - ▷ **Po.1**: After invoking the `Refresh` method, the state of the data object should have the same values as the matching row at the time when synchronization was performed. The time the row was last modified is stored in the data object.
  - ▷ **Po.2**: If the corresponding row has been deleted in the RDBMS since the last synchronization, then the data object's state is equivalent to the state caused by invoking the `Delete` method of the data object, as described in Section 7.2.6 on page 92.

- **Code Example:** The code in Listing 7.17 for this task scenario exemplifies the act of manual synchronization of a `Comment` instance with data from its associated row in the RDBMS.

```
1 comment.Refresh();
2 comment.Title = titleInput;
3 try {
4  comment.Save();
5 } catch (OutOfSyncException e) {
6   // Alert of and adjust to changes in the RBDMS.
7   // Code to handle this goes here.
8 }
```

Listing 7.17: Synchronizing a data object with newest row data.

7.2.8 Retrieve data objects by criteria

- **Description:** This task scenario is of relevance when a user wants to fetch data objects that match a given criteria.

  As stated in Section 6.1.3 on page 69, the read operation should enable a simple search on any of the columns that constitute a table on the basis of a column value criteria, as well provide a means to fetch all rows from a table. Furthermore, it was also stated that the read operation should return a single data object if a search is performed on a primary key and the corresponding table only constitutes one primary key.

  This set the stage for specifying an interface for the read operation that accommodates the requirements with regards to search criteria in concert with our concerns about type-safety and injection attacks. Furthermore, as consistency and simplicity should be a characterizing aspect of VDO it necessitates some careful considerations in order to orchestrate a read operation that accommodates its requirements while being intuitive.

  The table *blog_entries* is used as basis for interface and example code that is connected to this use.

- **Classes:** `BlogEntry`, `Author`, `VDOCollection<BlogEntry>`.

- **Methods:** `ReadAll`, `ReadById`, `ReadByTitle`, `ReadByContent`.

- **Interface:** There are different approaches that can be taken in order to compose the read operation for a data object.

  Our first idea was to provide the user with a class method named `ReadBlogEntry` that takes two parameters that correspond to a column name and a column value. The method could then be overloaded where the first parameter was fixed to be of type string and the second parameter varied according to the data types of the columns in the table schema, e.g. for `BlogEntry` we would have the method overloaded like `ReadBlogEntry(string columnName, int fieldValue)` and `ReadBlogEntry(string columnName, string fieldValue)`. However, taking this approach entails that we cannot ensure the same behavior for the same arguments. The problem arises when it comes to the column name. If we let column name's associated parameter be a string or an enumeration constant gives the user the possibility to specify a column name or enumeration constant as a first argument, for which the second argument may be of the incorrect data type.

  Given the problems linked to the `ReadBlogEntry` method we avoid ambiguous overloading and take a different approach by providing a series of read methods that differ slightly in naming. The requirements for the read operation has lead us to 3 different types of class methods.

The simplest of these is the class method named `ReadAll` that takes no arguments. It returns all rows in a table as a strongly typed `VDOCollection` of data objects. The method is named `ReadAll` because given that it is a class method it should be evident for the user through its usage that it returns all rows in a table as a collection of data objects. Furthermore, the naming of the method also roots in the fact that all data fetched from the RDBMS is *read* into memory in the application.

The second type of read method is one that involves a column value criteria, i.e. it has a single parameter. We find the format `ReadBy<X>`, where `<X>` is a property name that associates to a column name, to be the most easily understood naming composition. Furthermore, the name of the method's parameter is rooted in the name of the associated property, apart from the initial letter in lower-case.

The parameter of a `ReadBy<X>` should have a data type that is compatible with the corresponding column's data type. This entails that the user can only invoke the method if its parameter is bound to an argument with correct data type. In the case when a parameter's corresponding column is a foreign key a read method is not provided, because the parameter of that method would be a reference to a data object. Having a reference to the data object effectively means that we already have access to all the data objects which reference to this data object.

Lastly, `ReadBy<X>` returns a `VDOCollection` of data objects that match the column value criteria.

The third type of read method is a special case of `ReadBy<X>`, where `<X>` associates to the only primary key in a table. In this case, the method returns a single data object. The motive for returning a single data object was put forward in Section 6.1.3 on page 69.

The resulting interface we deduce from the task scenario is given in Listing 7.18.

```
1 public static BlogEntry ReadById(int id)
2 public static VDOCollection<BlogEntry> ReadByTitle(string
     title)
3 public static VDOCollection<BlogEntry> ReadByContent(string
     content)
4 public static VDOCollection<BlogEntry> ReadAll()
```

Listing 7.18: Interface for read class methods of `BlogEntry`.

- **Post-conditions:**
  - ▷ **Pr.1**: If table C contains a single primary key PK, then the parameter `pk` of `ReadByPK` is bound to a value for which there exists a row where

the value of *PK* uniquely matches the bound value.

- **Post-conditions:**
  - ▷ **Po.1**: The `ReadAll` class method returns a strongly typed `VDOCollection`, which contains data objects of type `C`. Each item in the collection corresponds uniquely to a row in table *C*. If *C* is empty, then the returned collection is empty.
  - ▷ **Po.2**: If table *C* contains a single primary key *PK*, then `ReadByPK` returns a data object of type `C` that corresponds to a row for which the value of *PK* uniquely matches the argument bound to parameter `pk`.
  - ▷ **Po.3**: If table *C* contains more than one primary key, then `ReadBy<X>` returns a strongly typed `VDOCollection`, which contains data objects of type `C`. Each item in the collection uniquely matches a row in the *C* table that satisfies the argument bound to parameter `x`. If no match is found in *C*, then the returned collection is empty.
  - ▷ **Po.4**: A strongly typed `VDOCollection` that is returned as a result of invoking `ReadBy<X>` or `ReadAll` is automatically kept at a state that reflects the search criteria, as described in Section 6.1.6 on page 72. Furthermore, multiple invocations of a `Read` method, which return collections, with identical search criteria results in a reference to the same collection.
- `ItemNotFoundException`: If **Pr.1** does not hold, then an instance of this exception is thrown. The exception contains a error message property, which indicates that there does not exist a row, which contains a column value that matches the provided argument to `ReadBy<X>`.
- **Code Example:** The example in Listing 7.19 showcases some scenarios that involve obtaining `BlogEntry` instances by way of read methods.

```
1  VDOCollection<BlogEntry> entries = BlogEntry.ReadAll();
2  Console.WriteLine("Listing all blog entries");
3  foreach(BlogEntry be in entries)
4  {
5   Console.WriteLine(be.Title);
6   Console.WriteLine(be.Content);
7  }
8
9  BlogEntry blogEntry = BlogEntry.ReadById(4);
10 entries = BlogEntry.ReadByTitle(blogEntry.Title);
11 Console.Write("Entries with similar title as blog entry");
12 Console.WriteLine(" with id {0}:", blogEntry.Id);
```

```
13
14 foreach(BlogEntry be in entries)
15 {
16  Console.WriteLine(be.Id);
17 }
```

Listing 7.19: Data objects fetched on the basis of criteria.

7.2.9 Subscribe to changes in a collection

- **Description:** This task scenario is relevant when a user wishes to be notified of when a data object is either removed or added to a collection. This is due to the requirement in Section 6.1.7, which requires that collections provided by the VDO framework should be observable.

  The user has different ways of obtaining a collection, i.e. by way of a read method or through a referenced by property in a data object. In both cases the collection's content corresponds to a search criteria. In the case where it is obtained by way of a read method the search criteria is explicit. Conversely, if a collection is obtained through a property then the search criteria is implicit as the collection reflects the references relationship that other data objects share with the property's data object.

  In spite of how a user obtains a collection it is subject for changes as data objects can be removed from or added to it as described in Section 6.1.6 on page 72.

- **Classes:** `VDOCollection<T>` where `T` is generated from table T in the RDBMS, `StateChangedEventArgs`.

- **Interface:**. The task required of the user to subscribe to changes in a collection should be similar to subscribing to changes on save and delete notifications on a data object.

  We stated in Section 7.2.4 on page 86 that we preclude the possibility to subscribe to changes in a collection that belongs to a data object, e.g. the `Comments` collection in a `BlogEntry` data object. This design decision is rooted in the fact that there is a semantic disparity between having a `blogEntry.CommentsChanged` event and a `blogEntry.Comments.StateChanged` event. We hold that the latter of these two is most comprehensible, namely because `Comments` is the same object though the number of elements in it changes. If we took the first of these as our approach, then the interface exposed to the user would signal that the `Comments` property can change to reference a different collection object, which is not possible (i.e. a missing set part) as it is the same object that only changes in the count of elements.

```
1 public delegate void StateChangedHandler(object sender,
     StateChangedEventArgs e);
2 public event StateChangedHandler StateChanged;
3 public enum StateChangeEvent { ItemAdded, ItemRemoved };
4 public class StateChangedEventArgs : EventArgs
5 {
6  public StateChangeEvent Change { get; }
7  public T Item { get; }
8 }
```

Listing 7.20: Interface for event related code of a `VDOCollection`.

Listing 7.20 illustrates the interface that we extract from this task scenario. As stated at line 3 we employ an enumeration `StateChangeEvent` to indicate the type of change. We follow the same naming pattern as we did for events that apply on an object level in order to be consistent. In this setting we also use an enumeration property named `Change` in StateChangedEventArgs. However, as we operate on a collection level we introduce an additional property in `StateChangedEventArgs` named `Item`, cf. line 7.

We let the data object of type `T` that is the cause of the event notification be available as an property in order to give the user a possibility of responding to a change in a specific manner. This assists the user with regards to not having to search a collection for a data object that is either missing or recently added.

Keeping a user interface updated to changes is of great importance in many applications, which gives rise to presume that this composition is of great utility in the context of keeping a user interface updated, because the objects that are responsible for displaying data get a reference to a data object and can act accordingly. Moreover, this implies that there is no need to have logic for regularly polling for changes.

- **Pre-conditions:**
  - ▷ **Pr.1**: A `VDOCollection<T>`, where `T` is a class generated from table *T*, conforms to a search criteria whether this is explicit or implicit, and is updated whenever data objects are added to or removed from it according to the search criteria.
- **Post-conditions:**
  - ▷ **Po.1**: Subsequent to registering with `StateChanged` the user is notified when a data object is removed from or added to a collection.
- **Code Example:** The code in Listing 7.21 exemplifies what is needed in order to keep track of changes in a collection that is either obtained by way of a `Read` method or through a property in a data object.

```
1 VDOCollection<Author> authors = Author.ReadAll();
2 authors.StateChanged += AuthorsChangeHandler;
3
4 BlogEntry blogEntry = BlogEntry.ReadById(4);
5 blogEntry.Comments.StateChanged += CommentsChangeHandler;
```

Listing 7.21: Examples of subscribing to changes in a `VDOCollection`.

7.2.10 Change the sync interval of a data object

- **Description:** This task scenario is very simple, however it is relevant due to a possible need from the user to being able to adjust the time interval between each synchronization of data objects with row data in the RDBMS.

- **Classes:** All VDO classes that are generated from tables in a RDBMS.

- **Interface:** The only thing required of the user in order to control the synchronization interval value on each generated class is to assign a value to a static property. We find the naming `SyncInterval` to be appropriate and informative. We use the type `double` to indicate time units in milliseconds. The interface for the property is illustrated in Listing 7.22.

```
1 double static SyncInterval { get; set; }
```

Listing 7.22: Interface for the sync interval property.

- **Pre-conditions:**

  ▷ **Pr.1**: Prior to setting the `SyncInterval` property of a generated class it has a default value of 30000 milliseconds.

- **Post-conditions:**

  ▷ **Po.1**: Subsequent to setting the `SyncInterval` property of a generated class its value is equal to or larger than 5000 milliseconds.

- **Code Example:** The example in 7.23 demonstrates how a user can change the synchronization frequency for all data objects of type Comment.

```
1 Comment.SyncInterval = 15000;
```

Listing 7.23: Changing the sync interval for instances of a `Comment`.

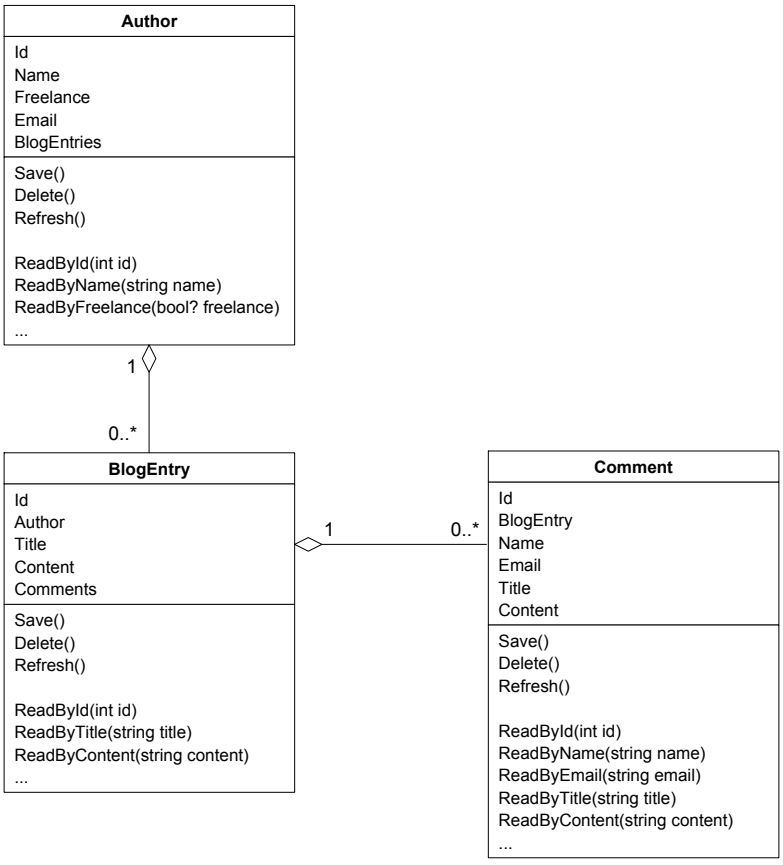

Figure 7.2: Classes generated from *WebLog*.

7.3 Generated Classes

Subsequent to generating classes from the database schema in Figure 7.1 on page 76 the result should look like what is depicted in Figure 7.2 on the preceding page. It is for illustrative purposes that we elude some components of a class, this especially applies to event related code.

As illustrated in Figure 7.2 on the facing page we represent relationships by way of reference types in C#. Likewise the database schema depicted in Figure 7.1 on page 76, an author can publish zero or more blog entries, for which each can have zero or more comments attached.

However, the classes depicted need to have supporting classes in the VDO framework in order to accommodate the requirements specified in Chapter 6 on page 67 and communicate with the RDBMS. The realization of this is what we bring about in the next chapter.

8 Realization of the VDO framework

In this chapter we give an explanation of the realization of VDO with regards to both the static API and the specialized API that is generated on the basis of a database schema. This serves both as realization of the requirements in Chapter 6 on page 67 and the design specified in Chapter 7 on page 75.

It is noteworthy that we separate the description of the VDO class generator from the rest of the VDO framework as the class generator is not needed during run-time. An overview over the class generator can be found in Chapter 9 on page 135.

8.1 The Static API

As advanced in Chapter 2.2 on page 34 it is our objective to create a run-time environment (referred to as the static API) that takes care of the database communication and serves as support for the specialized API generated by the VDO compiler.

To reduce the amount of code that needs to be generated we put as much functionality as possible into the static API. The static API consists of classes that the application developer uses in applications, e.g. VDOCollection, and internal classes with generic functionality that are used for implementation purposes and special classes that take measures against the impedance mismatch concerns we specified in Section 2.1.1 on page 8.

In the following sections we examine different groups of classes in the VDO API and describe how each of these are needed in order to make our specialized API viable.

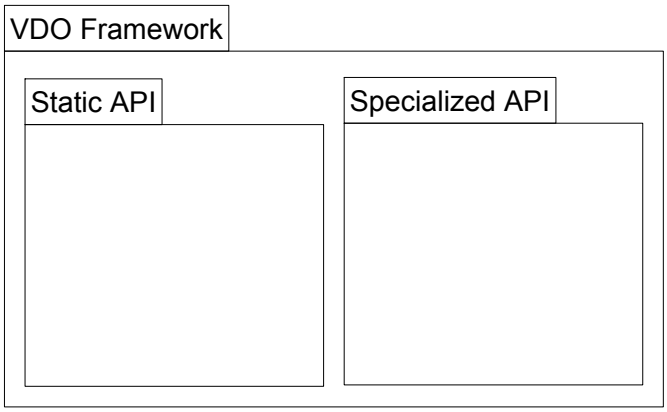

Figure 8.1: Overview of the VDO framework.

8.1.1 Data Service

The `DataService` namespace of the VDO framework is comprised of classes that are needed for direct communication with the RDBMS, intended for usage by the classes in the generated API.

The most important class in this namespace is the `VDOEngine` class as this is the foundation for causing the specialized API to require as few arguments as possible in order to communicate with the RDBMS. `VDOEngine` is responsible for maintaining connections to the RDBMS and serving these connections for the specialized API during run-time. The `VDOEngine` is constructed as a static class, which in our implementation is very similar to the singleton pattern [23], thus the application developer only has to initialize the connection to the RDBMS once in the application that employs the specialized API.

```
1 DataProvider dataProvider = new MSSQL2005DataProvider("host", "
     database", "username", "password");
2 VDOEngine.DataProvider = dataProvider;
```

Listing 8.1: Initializing a connection to *Microsoft SQL Server 2005*.

To initialize an RDBMS connection the application developer has to use an implementation of the abstract `DataProvider` class. Currently we have only made a single implementation into the `MSSQL2005DataProvider` class for *Microsoft SQL Server 2005* due to the fact that our VDO compiler currently only provides support for this RDBMS. Listing 8.1 shows how the application developer is supposed to initialize the connection with the RDBMS before using the rest of the VDO framework and the specialized API generated by the VDO compiler. The classes in the DataService namespace are depicted in Figure 8.2 on the facing page.

Though we only provide support for one platform at the given moment the usage of an abstract class as the defined class for `VDOEngine` makes it straightforward to support other RDBMS systems in the future as long as the VDO compiler

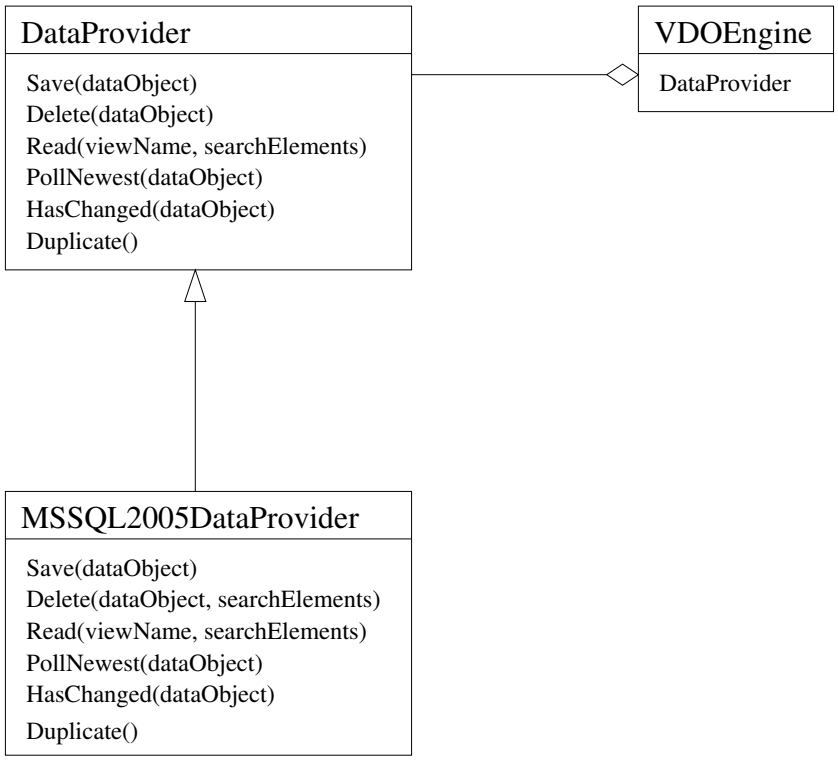

Figure 8.2: The classes in `DataService`.

is able to generate the specialized API from other RDBMS'.

In the following sections we describe `VDOEngine` and `DataProvider`.

8.1.1.1 VDOEngine

The `VDOEngine` is a very simple class due to the fact that it is only responsible for handling connections to the RDBMS. As the each class in the specialized API is subject to have a synchronization agent running in a thread and due to the fact that a `DbDataReader` in ADO.NET cannot handle multiple queries at the same time we have chosen to use a pool of database connections inside the `VDOEngine` to prevent bottleneck situations.

```
1 public static class VDOEngine {
2   public static DataProvider DataProvider { internal get; set; }
3 }
```

Listing 8.2: Interface for the `VDOEngine`.

When the `DataProvider` property of `VDOEngine` is set the local pool is be filled with the right amount of connections. This amount is hardcoded to 15 at the time of this writing, but could be set to any number reasonable to have connections for both the synchronization agents and standard search operations.

With a local counter it is managed to iterate through this pool each time a method inside the VDO framework requests a database connection.

8.1.1.2 DataProvider

The `DataProvider` is our own abstraction layer so we can prevent the VDO framework from being hard coupled with a certain database. We have made methods for those operations that we support in this version of the framework.

```
1 public abstract class DataProvider {
2   protected internal abstract DbDataReader Save(AbstractRecord
       dataObject);
3   protected internal abstract void Delete(AbstractRecord
       dataObject);
4   protected internal abstract DbDataReader Read(string viewName,
       DBColumnCollection searchElements);
5   protected internal abstract DbDataReader PollNewest(
       AbstractRecord dataObject);
6   protected internal abstract DbDataReader PollAfter(string
       viewName, DateTime timestamp);
7   protected internal abstract bool HasChanged(AbstractRecord
       dataObject);
8 }
```

Listing 8.3: The interface for the abstract class `DataProvider`.

In Listing 8.3 we can see the interface of the `DataProvider` class, which should be implemented in classes specialized for communicating with a certain database. In the following list we explain the required behavior and internal usage for each of the methods:

- `Save`: This method is used for saving a given data object into a corresponding table in the RDBMS. If the data object is not present in the table (i.e. if the data object has been deleted from the table or has never been saved before) it is inserted, otherwise the corresponding row in the table gets updated. The returning `DbDataReader` holds the newest timestamp after the save operation has been executed. Moreover it also contains auto generated identity values (in some RDBMS known as *auto increment*) if the data object has just been inserted in the table.

- `Delete`: This method is used to delete a given data object's corresponding row in the RDBMS.

- `Read`: This method is used in every read operation performed on the specialized API. The returning `DbDataReader` should contain every row from a given view (views are explained thoroughly in Section 8.2.6 on page 130, but basically the view contains rows from a table joined together with the timestamp of the last save operation).

- `PollNewest:` This method is used to get the most current data for a given data object, thus the returning `DbDataReader` contains this information. The method is used to refresh data in the data object (i.e. inside the `Refresh` method on data objects).

- `PollAfter:` This method is used to get all information saved in a table subsequent to the time of the given timestamp. The method is used by the synchronization agent to read and update information for local data objects.

For each of the methods it is important to notice that they only operate on the RDBMS and do not alter data objects, thus the functionality for updating data objects according to the executed operation should be implemented inside data objects.

8.1.2 Database Specific Types

In order to take care of meta data from a database schema during run-time we use classes made for the mapping from the primitive data types we support in SQL to the corresponding data types in the C#. At run-time these classes are used to wrap values and have the possibility to perform checks of values, e.g. whether the length of a given string value is inside the allowed range according to the corresponding column's declaration.

In the generated classes there is a private instance of one of these type classes for each of the columns in the corresponding table, representing values and meta data of columns.

The `Types` namespace consists of several classes, which only are used internally in the specialized API. These have a great impact on safe queries as checks and security measures can be implemented with effect on the specialized API without the need of performing modifications to the code generator, i.e. not changing the generated code for a given database schema.

In the following we outline the type classes in the `Types` namespace:

- `DBTypeColumn:` This is the abstract class, which all the other type classes inherit from. The most generic functionality for the type classes is implemented on this class.

- `DBBooleanColumn:` The `bit` type in the RDBMS is represented during run-time as a boolean with this class.

- `DBDateColumn:` The `datetime` type in the RDBMS is represented in this class during run-time.

- `DBNumberColumn:` Any numeric value type in the RDBMS is represented with this class during run-time.

- `DBStringColumn`: Any text type in the RDBMS is represented with this class during run-time. If there are any restrictions to the length of the string in the RDBMS this class ensures that the application is not able to violate the restriction.

```
1 public class DBNumberColumn<T> : DBTypeColumn {
2   public DBNumberColumn(string columnName, DbType columnType,
        bool isNullable, byte? precision, int? scale, int? length,
        bool isPrimaryKey, bool isForeignKey, bool isIdentity);
3   public T Value { get; set;}
4   protected internal override string SQLValue { get; }
5 }
```

Listing 8.4: Interface for the `DBNumberColumn` class.

Instead of having one class for every supported primitive type from SQL we have created a class for each category of supported types and used generics in these classes. In Listing 8.4 we have shown the interface for the `DBNumberColumn` where the generic `T` is a C# numeric type (e.g. `int`). It is important to observe that the constructor parameters of type class correspond to meta data for columns in the RDMBS.

```
1 public abstract class DBTypeColumn {
2   protected DBTypeColumn(string columnName, DbType columnType,
        bool isDBNullable, byte? precision, int? scale, int? length,
         bool isPrimaryKey, bool isForeignKey, bool isIdentity);
3   ...
4   protected internal string SQLIdentifier { get; }
5   protected internal SqlParameter SQLParameter { get; }
6 }
```

Listing 8.5: Interface for the abstract `DbTypeColumn` class.

The `DBNumberColumn` specializes the abstract `DbTypeColumn` class, which has the interface illustrated in Listing 8.5. To keep focus on the most important in this listing we have not shown trivial properties matching the constructor parameters, which all have a `get` part but no `set` part as they model the meta information and thus not subject to change during run-time. The two properties shown in Listing 8.5 are used internally when creating the SQL queries for database communication:

- **SQLIdentifier:** This property is used directly in the SQL query as an identifier for the value given by the `SQLParameter` instead of using the value directly. The main reason for taking this approach is to prevent SQL injections as the values are interpreted only as values and not as a part of the SQL query.

- **SQLParameter:** This property is used when creating the SQL queries as the `SQLParameter` consists of a value for the `SQLIdentifier`. The `SQLParameter` is used when executing a query in the `DataProvider` so identifiers can be translated into the corresponding value. Basically, the `SQLParameter` is used by us as a wrapper for the columns type, identifier and corresponding value.

To better understand how the `SQLIdentifier` is used in the SQL queries we exemplify with a simple query in Listing 8.6 as it is given to the database connection together with the `SQLParameter` holding the value for the identifier. In this example the identifier is `@name`.

```
1 SELECT * FROM authors WHERE name=@name;
```

Listing 8.6: A sample query with an identifier.

As aforementioned, type classes are used internally in generated classes as private variables. In Listing 8.7 we give an example of how a `DBStringColumn` is used to model the `name` column from a table. When the private variable is instantiated the meta data gets *saved* in the type object through the constructor. The application developer can manipulate the `DBStringColumn`'s `Value` through an encapsulating property named after the corresponding column.

```
1  private DBStringColumn _name = new DBStringColumn("name", DbType.
       String, false, null, null, 30, false, false, false);
2  ...
3  public string Name
4  {
5    get
6    {
7      return this._name.Value;
8    }
9    set
10   {
11     if (this._name.Value != value)
12     {
13       try
14       {
15         this._name.Value = value;
16       }
17       catch (ValueRequiredException ex)
18       {
19         throw new ValueRequiredException("property", "Name");
20       }
21       catch (StringLengthException ex)
22       {
23         throw new StringLengthException(ex, "Name");
24       }
25       this.IsDirty = true;
```

```
26          OnNameChanged(new NameChangedEventArgs(this.Name));
27        }
28      }
29  }
```

Listing 8.7: Usage of a `DBStringColumn` and a property to model the `name` column from a table.

The property in Listing 8.7 on the preceding page uses a property directly on the used `DbTypeColumn` to both get and set the value. In the *try-catch* block we can see how the value is set unless an error occurs. If this is the case then the exception thrown from the `DbTypeColumn` in question is catched and re-thrown. The reason for catching and throwing is because we want the origin of the exception to be the data object and not the `DbTypeColumn` used for the property at question. This further hides the implementation details from the application developer as he is should not know about the usage of the `DbTypeColumn` inside a data object.

```
1  private string _value;
2  ...
3  private bool isTooLong(string value)
4  {
5   return value.Length > (int)Length;
6  }
7  public string Value
8  {
9   get { return _value; }
10  set
11  {
12   if (!IsDBNullable && value == null)
13   {
14    throw new ValueRequiredException();
15   }
16   else if (!IsDBNullable && value != null)
17   {
18    if (isTooLong(value))
19    {
20     throw new StringLengthException((int)Length, value.Length);
21    }
22   }
23   _value = value;
24  }
25 }
```

Listing 8.8: Implementation of the `Value` property on `DBStringColumn`.

In Listing 8.8 we illustrate how the `Value` property is implemented inside the `DBStringColumn`. As depicted, we perform checks to ensure that `null` is only allowed if the corresponding column allows it, and moreover, to ensure that the given string value does not exceed length limitations.

If other types of checks would be necessary they should be implemented in the same way as the checks shown in Listing 8.8 on the facing page.

8.1.3 Exceptions

In connection with the application developer using the specialized API our general approach is to hide as much of the implementation details as possible. This is not to hide the fact that the VDO framework is using an RDBMS for data storage, but instead to avoid that the implementation has a negative impact on the API and ease the usage of the API. Therefore we introduce specialized exceptions to the VDO framework (either from the static or the specialized API) with a specific error message, rather than letting the implementation "reveal itself" or using more general exceptions thrown from the RDBMS.

Moreover, we would also like to prepare the use of the generated API with different types of RDBMS as these can return different types of exceptions. By handling every exception inside the VDO framework and inside the generated API we are able to make this usage uniform regardless of which RDBMS is being used.

As important are the exceptions thrown as a result of violating restrictions specified in the database schema, e.g. when an application tries to set a string property on a data object with a length that is longer than the corresponding `nvarchar` column allows.

The `Exception` namespace holds every exception that the specialized API and the VDO framework can throw at run-time. To give a better overview of the possible errors the following list is compiled:

- `ItemNotFoundException`: This exception is thrown if a read operation that returns a single data object from the RDBMS is not able to find it in the corresponding table.
- `OutOfSyncException`: This exception is thrown if a data object is older than the most recently saved corresponding row in the RDBMS. This exception can be thrown both when trying to save or delete the data object from the RDBMS.
- `StringLengthException`: This exception is thrown if a string property is set with a string longer than the allowed length in the associated column.
- `ValueRequiredException`: This exception is thrown if the developer tries to assign `null` when the corresponding column in the RDBMS does not allow `null`.
- `NotConnectedException`: Exception thrown when trying to use database related operations in the VDO framework, for instance `ReadAll`, before the `DataProvider` property of `VDOEngine` has been set.

8.1.4 Structure of Active Record

As previously stated we would like to generate classes based on the *Active Record* design pattern. To let the class generator focus on generating the part of the classes that are schema specific we put the most generic functionality in the abstract class `AbstractRecord`, which all generated classes inherit. Thus, the abstract class can be viewed as a shell for the classes are generated.

In the following sections we give a walk-through of the functionality implemented in the `AbstractRecord` class and explain what underlying cause we have for the different functionality of our realization of the pattern.

8.1.4.1 State Properties

During run-time we need to keep track of the state of data objects to ensure that we do not need to save a data object to the RDBMS if no modifications are made on it, or to use an SQL insert query if the data object has not been saved previously, and likewise an SQL update query if it has been saved before.

- `IsNew`: This property informs the VDO framework whether the data object has been saved to the RDBMS yet, thus the `DataProvider` is able to use either an `insert` or an `update` query inside the `Save` method. When creating a new data object this property is set to `true`. When a data object is deleted from the RDBMS this property is set to true such that it signals that it is possible to save the data object as a new row.

- `IsDirty`: Whenever the value of a public property is changed then this property is set to `true` as it is no longer equal to the row it is modeling. When the data object is saved or refreshed the property is set to `false` as it is once again equal to the row it is modeling. Further, if this property is set to `true` then the data object is not be updated by the synchronization agent as this would overwrite local modifications.

- `IsOutOfSync`: This property is set to `true` by the synchronization agent if the data object is subject for update, but has local modifications that have not yet been saved to the RDBMS, i.e. the `IsDirty` property is `true`. The property is set to `false` by the `Refresh` method as it overwrites local modifications with the most recent data in the mapped row.

- `ModifiedTime`: This property is at all time holding a timestamp telling when the current data object was last saved to the RDBMS. When a data object is saved to the database it is updated according to the time of saving.

Of these four state properties only the `ModifiedTime` property is available to the developer as we do not find it necessary for the developer to know about these internal stages for the data object, as they are only present for implementation purposes.

8.1.4.2 Database connectivity

In order to hide implementation details from the application developer we have chosen not to expose the interface of the `DataProvider`. This is possible as C# allows for giving internal access for certain assemblies. This is possible when using `internal` in combination with the attribute `InternalsVisibleTo` with a strong-named assembly, which is the name of our framework, i.e. "VDO". However, as we cannot know what the name of a generated assembly is going to be, as it is named after the database schema it is generated upon, we have solved this in another way.

```
1 protected static DbDataReader DBRead(string viewName,
     DBColumnCollection searchElements);
2 protected static DbDataReader DBPollAfter(string viewName,
     DateTime timestamp);
3 protected DbDataReader DBSave();
4 protected void DBDelete();
```

Listing 8.9: The interface given access to the database connection.

As the specialized API consists of classes specializing the `AbstractRecord` class and due to the fact that inheriting classes can use `protected` methods without the application developer having access to these methods we have made methods to wrap the functionality from `DataProvider` which should be used by the specialized API. As the `AbstractRecord` class resides in a well known assembly space we can give access to the internal `DataProvider` on the `VDOEngine` without exposing it to the application developer.

The different wrapper methods on the `AbstractRecord` class are shown in Listing 8.9. In Listing 8.10 we have shown how the `DBSave` method is implemented on the `AbstractRecord` class in order to wrap the communication with the RDBMS. The rest of the protected methods shown in Listing 8.10 are implemented a similar way.

```
1 protected DbDataReader DBSave() {
2   DataProvider dataProvider = VDOEngine.DataProvider;
3   return dataProvider.Save(this);
4 }
```

Listing 8.10: The protected `DBSave` method on the `AbstractRecord` class.

8.1.4.3 Match Elements

When a local data object is being saved we would like to be able to check if it matches the search criteria for any of the collections with search results held by the identity map. As this should be done automatically and without the need for database communication a method is implemented on the `AbstractRecord` class to check if a data object conforms to some given criteria.

```
internal bool MatchElements(DBColumnCollection searchElements) {
  bool result = false;
  bool equals = false;

  foreach (DBTypeColumn searchElement in searchElements)
  {
    equals = false;
    foreach (DBTypeColumn column in ColumnElements)
    {
      if (column.GetHashCode() == searchElement.GetHashCode())
      {
        equals = true;
        break;
      }
    }
    if (!equals)
    {
      result = false;
      break;
    }
    result = true;
  }
  return result;
}
```

Listing 8.11: The `MatchElements` method.

Thus, when a data object is being added to a collection it is possible to ensure that the data object is in fact satisfying the search criteria. Listing 8.11 depicts how the `MatchElements` method is implemented. Further, when the state of a data object changes as a result of being saved to the database the collection is able to check whether it still complies to the search criteria by calling this method.

8.1.4.4 Meta data

At run-time we need to be able to access the meta data for the row modeled by the data object. As explained in Section 8.1.2 on page 109 some of this is represented during run-time in the various instances of `DBTypeColumn`, but we also require to know the name of the corresponding table (for insert and updates) and view (for reading the timestamp together with the data for the row) when interacting with the RDBMS.

```
protected internal abstract String TableName { get; }
protected internal abstract String ViewName { get; }
```

Listing 8.12: The interface for `TableName` and `ViewName` in the `AbstractRecord` class.

To enable this information we have defined two abstract properties to be implemented in the generated classes. Listing 8.12 shows the interface for these

two properties. The properties are used from the DataProvider to identify which tables to use in the SQL queries.

8.1.4.5 Events

On the AbstractRecord class we provide a StateChanged event, which resembles changes made to the data object, i.e. when the object is saved to or deleted from the RDBMS or when the synchronization agent updates it according to changes in the RDBMS. This is done in the same manner as specified in Chapter 7.2.5 on page 89.

8.1.5 DBColumnCollection

The DBColumnCollection is used for implementation purposes in the VDO framework to group the database specific types. This grouping is used both to group the DbTypeColumn instances which form the primary key and the rest of the columns from the table. Further, we use DBColumnCollection internally in the VDOCollection to manage the ancillary search criteria as this gives us the ability to reuse the different specializations of DbTypeColumn and thus the functionality of SQLParameter and SQLIdentifier to keep the executed queries at a safe state.

```
1  public override int GetHashCode() {
2    string result = "";
3
4    if (_elements.Count == 0)
5    {
6      result = "++empty++";
7    }
8    else
9    {
10     foreach (DBTypeColumn element in _elements)
11     {
12       result += "+" + element.ColumnName + "=" + element.SQLValue
             ;
13     }
14   }
15   return result.GetHashCode();
16 }
```

Listing 8.13: The GetHashCode on a DBColumnCollection.

As we use the DBColumnCollection in both VDOCollection to keep track of search criteria and in the data objects to group the primary keys, and they both have to be managed in the identity map, we found it to be most evident to implement functionality into this class that applies for both. In the identity map both the data objects and the VDOCollection holding search results are saved in

two hash tables as they both need to generate a unique hash value according to the primary keys and search criteria, respectively.

The method for getting a hash value is shown in Listing 8.13 on the preceding page, in which we see that we iterate through each `DbTypeColumn` element in the `DBColumnCollection` and concatenate both the name of the corresponding column and the value. Finally we return the hash value for the generated string and as the `GetHashCode` method on a string is guaranteed to be unique by C# we have guarantee that we return a unique hash code for both data objects and collections with search results.

8.1.6 The VDO Collection

Whenever the specialized API returns a collection with search results, either directly from a search method or as a property on a data object it is a strongly typed collection such that the application developer has knowledge about the content type of the collection. As this collection should also deal with the search criteria initially used when retrieving the data objects from the RDBMS we cannot use a standard collection from the C# library. Instead we have created a specialized `VDOCollection` with the functionality we need to take care of the search criteria together with the data objects fulfilling the search criteria.

To the application developer the `VDOCollection` works similar to any other collection except that it is read-only together with the fact that it is possible to subscribe to events when data objects are added to or removed from the collection by the synchronization agent due to changes in the corresponding table.

```
1  public abstract class VDOCollection<T> : IEnumerable<T>
2          where T : AbstractRecord
3  {
4    protected List<T> _elements = new List<T>();
5    protected DBColumnCollection _searchElements = new
        DBColumnCollection();
6    ...
7    protected internal void Add(T element);
8
9    // Read-only collection specific functionality
10   public int Count;
11   public int IndexOf(T dataObject);
12   public bool Contains(T dataObject);
13   public T this[int i];
14   IEnumerator<T> IEnumerable<T>.GetEnumerator();
15   IEnumerator IEnumerable.GetEnumerator();
16 }
```

Listing 8.14: Interface for `VDOCollection`.

Listing 8.14 shows the interface for the `VDOCollection`. The reason for showing two of the internal variables in this listing and the reason why they are

`protected` instead of `private` is fully intentional as we once again would like to hide as many implementation details as possible, but need to do it in a way quite unusual. As we need to be able to set the `DBColumnCollection` containing the search criteria, but do not want the developer to be exposed to this implementation detail we have made the `VDOCollection` abstract. We see two purposes for making the `VDOCollection` abstract:

1. We do not want the application developer to create instances of the `VDOCollection` as they should only consist of data objects fulfilling some given search criteria. By making the class abstract we cannot prevent the application developer from implementing this class, but we have made it difficult to do so, thus the application developer will probably not use an implementation of `VDOCollection`.

2. Giving the internal part of the specialized API access to alter the elements in the collection. Though it is possible in C# to give access for a certain namespace or assembly we cannot know what the name of the generated assembly is as it is named after the database schema it is generated from. By generating an internal implementation of the `VDOCollection` we have bypassed these possible restrictions.

We have shown a generated implementation of `VDOCollection` in Listing 8.15. It should be evident that this implementation gives internal access to create an instance with a `DBColumnCollection` consisting of the search criteria, as well as to adding elements to the collection. Further, the internal hiding of the `Add` method ensures that the `VDOCollection` is *read-only* for the application developer.

```
1  internal class WebLogVDOCollection<T> : VDOCollection<T>
2          where T : AbstractRecord
3  {
4    internal WebLogVDOCollection(DBColumnCollection searchElements)
5    {
6      this._searchElements = searchElements;
7    }
8    protected internal void Add(T element)
9    {
10     base.Add(element);
11   }
12 }
```

Listing 8.15: An implementation of `VDOCollection` for the *WebLog* database schema.

If we look back at the last part of the code shown in Listing 8.14 on the preceding page, then we see some collection specific details. These enable `VDOCollection` to be usable as a collection for the application developer. The most

important feature is the ability to use a `foreach` statement. To use this mechanism `VDOCollection` implements the `IEnumerable` interface, which basically ensures that the class implements the method `GetEnumerator`. As generics are used in our collection we had to implement it two times as shown in Listing 8.16. As there is an internal `_elements` list with data objects we use the `GetEnumerator` it implements as there is no difference in the elements inside this list and the elements visible to the application developer.

```
1 IEnumerator<T> IEnumerable<T>.GetEnumerator()
2 {
3   return _elements.GetEnumerator();
4 }
5 IEnumerator IEnumerable.GetEnumerator()
6 {
7   return _elements.GetEnumerator();
8 }
```

Listing 8.16: Implementation of `GetEnumerator()` on `VDOCollection`.

Of other collection specific features we have implemented a `Code` property, an `IndexOf` and a `IndexOf` method, and an indexer (`this[int i]`), all known from most other collections in C# to let the application developer use the `VDOCollection` in a manner he is accustomed to, apart from that it is read-only.

When a data object is added to the collection through `Add` a check is made to determine if it fulfills the search criteria for the collection and is only added to the internal list if the criteria are fulfilled. When the data object gets added the `VDOCollection` subscribes to the event fired whenever the data object changes state (i.e. when the object gets saved or removed from the RDBMS). Whenever the state changes for a object the collection checks if it does still fulfill the search criteria and removes it if it does not.

When the content of the `VDOCollection` changes (i.e. a data object is added or removed) it uses the `StateChanged` event to inform subscribers about this change with an appropriate enumeration value for the `Change` property in the `StateChangedEventArgs`.

8.1.7 Identity Map

As covered in Section 5.2 on page 57 we find identity map to be of great usefulness in VDO. Both to ensure that we do not have two data objects modeling the same row in the RDBMS, but also to return the requested data object without having to communicate with the database. Furthermore, the collections with search results are also handled by the identity map so that already loaded search results are returned without communicating with the RDBMS.

When a new object is created or read from the RDBMS it is added to the local identity map, and removed when it is deleted. Figure 8.3 on the facing page demonstrates how a class method and identity map in concert load a data object

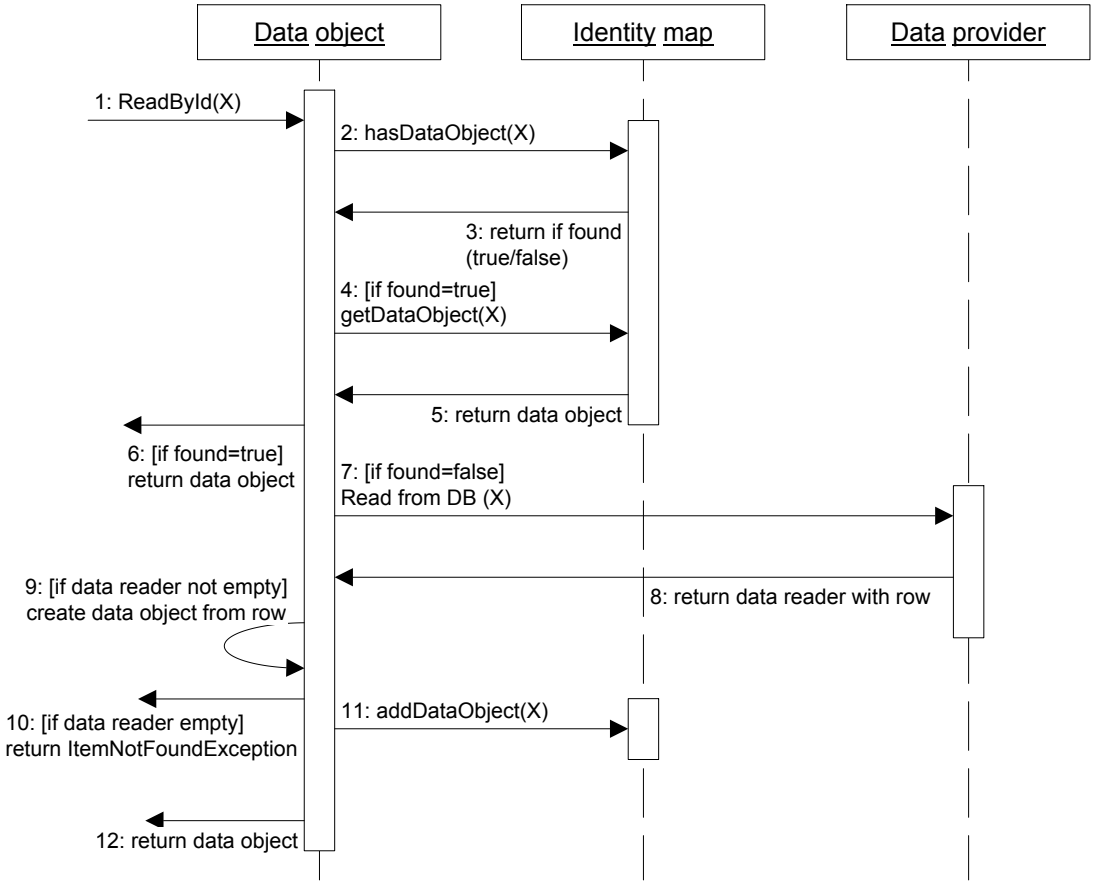

Figure 8.3: Sequence diagram for the VDO identity map.

from either the identity map or the database. Internally in the identity map the data objects and collections are saved in two separate `Hashtable` instances according to their primary keys and search criteria respectively.

When a new data object is added to the identity map it is also added to collections in the identity map for which the data object fulfills a search criteria. Furthermore, the identity map subscribes to an event that is raised whenever the data object changes state and removes it from the identity map if the data object is deleted from the RDBMS.

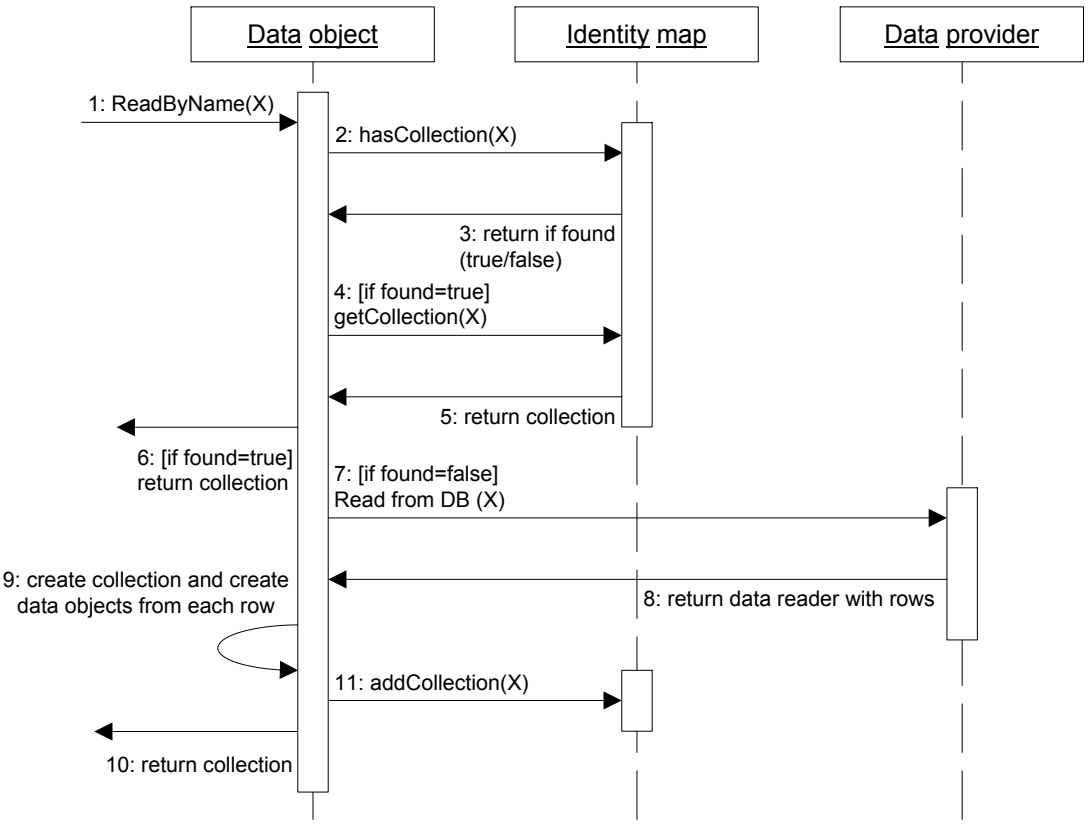

Figure 8.4: Sequence diagram for the VDO identity map with collection.

Figure 8.5 on page 130 depicts how the identity map is involved when searching for data objects fulfilling a certain search criteria. Due to the fact that the search criteria get grouped in a `DBColumnCollection` instance we can employ the hash code from this class in order to check whether there exists a `VDOCollection` for a particular search criteria.

Due to the nature of the `Hashtable` used for organization of the objects and collections inside the identity map, the hash value for a saved object in the `Hashtable` (both data objects and collections) cannot be directly available on the data objects and collections as the key when adding them to the lists. This is because when we check whether a collection for some given search criteria exists

we only give a `DBColumnCollection` containing these search criteria.

When checking the `Hashtable` not only does the hash value have to be the same, but also the object from which the hash value has its origin. Thus, when checking the `Hashtable` whether a collection for a given search criteria exists would return `false` if we had just checked directly with the `DBColumnCollection` even though the collection would exist and the collection in question returns the exact same hash value.

To work around this problem we have chosen to use the hash value generated from the hash value from `DBColumnCollection` (which also is what generated the hash value for both the data objects and the collections) as shown in Listing 8.17. We can do this as everything in C# is objects, thus we use the hash value as our index key and not the object itself. This way every object in the `Hashtable` is indexed according to a `integer`'s hash value, which entails the ability to find the right data object or collection according to this hash value. The behavior we have strived to fulfill is very similar to the behavior of `HashMap` known from Java.

```
1 public void addCollection(VDOCollection<T> collection)
2 {
3   objectCollections.Add(collection.GetHashCode(), collection);
4 }
5 public bool hasCollection(DBColumnCollection searchElements)
6 {
7   bool result = objectCollections.ContainsKey(searchElements.
       GetHashCode());
8   return result;
9 }
```

Listing 8.17: How we use the `Hashtable` inside the `IdentityMap`.

8.2 The Specialized API

With the knowledge of how our the static VDO works we are set commencing the description of the specialized API that the VDO compiler outputs.

In the following sections we outline the basics of the specialized API and how it interacts with the static API, described in the previous sections. For ease of exposition we describe the specialized API by way of examples of code from classes in the generated API.

8.2.1 Properties

As established in Chapter 7.2 on page 77 there should be properties on the generated classes to get and set certain values. For every column in the modeled table there should be a corresponding public property to let the developer alter this in an intuitive way. If the property is modeling a foreign key column then it should use the class modeling the referencing foreign key's table.

For any property reflecting an identity column (also known as *auto-increment* in some RDBMS) there is no set part as the value for this column is seeded by the RDBMS upon insertion.

```
1  private DBNumberColumn<int> _authorId = new DBNumberColumn<int>("
       author_id", DbType.Int32, false, 10, 0, null, false, true,
       false);
2  private Author _author;
3  ...
4  public Author Author
5  {
6    get
7    {
8      if (_author == null)
9      {
10       _author = Author.ReadById(_authorId.Value);
11     }
12     return this._author;
13   }
14   set
15   {
16     if (value == null)
17     {
18       throw new ValueRequiredException("property", "Author");
19     }
20     if (_author != value)
21     {
22       _author = value;
23       this._authorId.Value = value.Id;
24       this.IsDirty = true;
25       OnAuthorChanged(new AuthorChangedEventArgs(this.Author));
26     }
27   }
28 }
```

Listing 8.18: The public `Author` property on the generated `BlogEntry` class.

The most important feature of the public properties is the fact that they are lazy loaded, as covered in Section 5.3. Listing 8.18 illustrates how this is implemented as a check at line 8 through 11. If it is the first time the current property is being requested, then it is loaded through the appropriate method on the referencing class, i.e. `Author.ReadById` in the given example. Collections modeling *referenced by* are implemented in the same way, i.e. with a check to see if it has already been loaded and if not, then its loaded before returning the corresponding result.

In the `set` part of the property in Listing 8.18 we can further see how the value to be set is first checking whether it is null before actually setting the private variable. Next the `IsDirty` property is set to `true` to prevent it from being overwritten with updated data by the synchronization agent. Finally an event is raised to inform subscribers that this property has changed.

8.2.2 Synchronization Agent

The synchronization agent is one of the central parts of the VDO framework as it is responsible for polling the RDBMS for changes made by other clients at regular intervals. The synchronization agent starts to run as a static thread as soon as the first data object is put into the identity map, i.e. when either a newly created data object is saved to the RDBMS or when a data object is read from it.

To lower the data processing involved only rows altered since the last time of reading are read from the database by saving the latest read timestamp and use this when polling. Listing 8.19 exemplifies a sample query of how a SQL query for polling looks like. The reason for using the view and not reading directly from a table is the fact that we do not have access to time of modifications nor information of whether any rows have been deleted in the table. This is explained in more detail in Section 8.2.6 on page 130.

```
1 SELECT * FROM vdo_authors_actions_view
2 WHERE modified_time > @lastReadTimestamp
```

Listing 8.19: Sample SQL query for polling the RDBMS.

When the synchronization agent has got rows altered since the last polling it does the following for each of them:

- Checks if the corresponding data object is loaded locally. If so then it is updated with the newly read data. If the data object has local modifications the `IsDirty` property is set to true and no update takes place. If the row has been deleted the local data object is set to deleted too despite of if it has local modifications.

- Checks if the corresponding data object conforms with the search criteria for any of the collections in the identity map. If this is the case the data object is added to both the identity map (if not present already) and the collections it conforms with.

It is important to observe that only data objects already present in the identity map are updated. A row from the RDBMS is only created as a new local data object if it conforms with the search criteria for collections in the identity map, thus only modifications that have impact on the local data objects and search results are taken into account.

In order for the application developer to be able to change the interval between these synchronization executions we have a public property on the generated classes to set this interval.

8.2.3 The Event Pattern

A central part of VDO is the integration of the observer pattern in order to enable a notification mechanism of changes in the database.

In order to enable the application developer to observe changes made to the data object we take advantage of the event pattern, which is implemented in the generated classes. Listing 8.20 illustrates code that we generate for handling events for a property, in this example the `Freelance` property of the `Author` class.

```
1  public delegate void FreelanceChangedHandler(object sender,
       FreelanceChangedEventArgs e);
2  public event FreelanceChangedHandler FreelanceChanged;
3  protected void OnFreelanceChanged(FreelanceChangedEventArgs e)
4  {
5    FreelanceChangedHandler handler = FreelanceChanged;
6    if (handler != null)
7    {
8      handler(this, e);
9    }
10 }
11 public class FreelanceChangedEventArgs : EventArgs
12 {
13   private bool? _freelance;
14   public FreelanceChangedEventArgs(bool? freelance)
15   {
16     _freelance = freelance;
17   }
18   public bool? Freelance { get { return _freelance; } }
19 }
```

Listing 8.20: Example of how event related code is generated.

For each of the public properties that have a set part in the generated classes there is created a public event, which is used for subscription on changes. When a property is being set the last thing to be done is to call the corresponding method for handling the right event.

In the example in Listing 8.20 this is a call to `OnFreelanceChanged` inside the `Freelance` property. Inside this method we employ the `FreelanceChanged` event and serve the data object itself and the new value of the property in question.

With this construction inside the body of the properties the application developer can subscribe to changes made in only those properties of interest to him. These events are raised no matter if the change is made locally or by the synchronization agent due to changes in the RDBMS.

It should be noted that inside `OnFreelanceChanged` we make a temporary copy of an event in order to avoid the possibility of a race condition if the last subscriber performs an unsubscription subsequently after the `null` check and prior to the event is raised.

8.2.4 Static Methods

As previously covered in Section 7.2 on page 77 the generated API should feature functionality for retrieving data objects based on criteria defined by the developer. This is elaborated in Section 7.2.8 on page 96.

This functionality should be available as static methods on the generated classes as they are not coupled with a instances.

For each of the columns in the table being modeled there should be generated a method for searching according to this particular column, though not if the column is a foreign key column. Further, there should be a method for fetching every row from the corresponding table. Each of these read methods should return a strongly typed `VDOCollection` consisting of data objects conforming with a given search criteria.

```
1  public static VDOCollection<BlogEntry> ReadByTitle(string title)
2  {
3    DBColumnCollection searchElements = new DBColumnCollection();
4    VDOCollection<BlogEntry> searchResult;
5
6    DBStringColumn _title = new DBStringColumn("title", DbType.
       String, false, null, null, 50, false, false, false);
7    _title.Value = title;
8    searchElements.AddElement(_title);
9
10   if (_identityMap.hasCollection(searchElements))
11   {
12     searchResult = _identityMap.getCollection(searchElements);
13   }
14   else
15   {
16     searchResult = BlogEntry.ReadByHelper(searchElements);
17     _identityMap.addCollection(searchResult);
18   }
19   return searchResult;
20 }
```

Listing 8.21: The `ReadByTitle` method on the `BlogEntry` class.

As the only real difference in the behavior of these methods for reading in the data objects is the composition of search criteria they resemble the example code in Listing 8.21, which is the generated method for reading in `BlogEntry` data objects according to their title in the RDBMS. The `searchElements` variable is holding the search criteria and is used to first check if there exists a collection in the local identity map, which resembles the criteria and secondly, if no collection is found in the identity map, then the helper method `ReadByHelper` fetches the data objects from the RDBMS. For the `ReadAll` method the `searchElements` variable is an empty collection of search criteria as every data object (and thus the corresponding row) conforms with a non-existing criteria which should be

met, i.e. no SQL `WHERE` clause.

The `ReadByHelper` method for the `BlogEntry` class is shown in Listing 8.22 where we can see how this method takes care of the specific details of first communicating with the RDBMS through the `AbstractRecord` wrapper method `DBRead` and further how the data objects are being constructed and put into the specialized `VDOCollection` to be returned. Moreover, we can see how the specialized `VDOCollection` is getting its search criteria as a constructor parameter at line 4.

```
1  private static WebLogVDOCollection<BlogEntry> ReadByHelper(
       DBColumnCollection searchElements)
2  {
3    DbDataReader dataReader = AbstractRecord.DBRead(_viewName,
       searchElements);
4    WebLogVDOCollection<BlogEntry> elements = new
       WebLogVDOCollection<BlogEntry>(searchElements);
5    while (dataReader.Read())
6    {
7      BlogEntry element = new BlogEntry(
8        (int)dataReader["id"],
9        (int)dataReader["author_id"],
10       (string)dataReader["title"],
11       (dataReader["content"] == System.DBNull.Value) ? null :
12       (string)dataReader["content"],
13       (dataReader["modified_time"] == System.DBNull.Value) ? null
           : (DateTime?)dataReader["modified_time"]);
14     if (!_identityMap.hasDataObject(element))
15     {
16       _identityMap.addDataObject(element);
17       elements.Add(element);
18     }
19     else
20     {
21       elements.Add(_identityMap.getDataObject(element.
           PrimaryKeyElements));
22     }
23   }
24   dataReader.Close();
25   return elements;
26 }
```

Listing 8.22: The `ReadByHelper` method on the `BlogEntry` class.

As each generated class differs with regards to the columns in the corresponding table and thereby also in the parameters for the constructor the `ReadByHelper` method cannot be implemented in a generic way. The vigilant reader may have noticed that the possible `null` values from the RDBMS are handled in a special way to convert from the way the `null` is represented in the result from the database into how `null` is treated in C#. Of course, this check is only performed for columns that are `nullable` in the RDBMS.

As a final stage for the `ReadByHelper` method the newly constructed data objects are put into the collection and the local identity map.

8.2.5 Instance Methods

According to the *Active Record* design pattern each data object should have methods for altering the RDBMS according to the data object in question, i.e. methods for saving and deleting. In VDO we have also chosen to have a `Refresh` method as covered in Section 7.2.7 on page 94, which is also placed as an instance method on the data objects.

```
1  public new void Save() {
2    base.Save();
3    if (IsDirty  IsNew)
4    {
5      DbDataReader dataReader = base.DBSave();
6      if (dataReader.Read())
7      {
8        if (IsNew)
9        {
10         Id = (int)dataReader["id"];
11         _identityMap.addDataObject(this);
12       }
13       ModifiedTime = (DateTime)dataReader["modified_time"];
14       IsNew = IsOutOfSync = IsDirty = false;
15     }
16     dataReader.Close();
17     OnStateChanged(new StateChangedEventArgs(StateChangeEvent.
          Saved));
18   }
19 }
```

Listing 8.23: The generated `Save` method on a class in the specialized API.

It is inside these instance methods that the state properties are mostly used and set according to the state of the object. Listing 8.23 shows the `Save` method that is generated for the `BlogEntry` class. As shown in the listing the data object is only saved to the RDBMS if it is either new (i.e. it has not been saved to the database) or if it is dirty (i.e. if any of the properties of the data object has been altered since the last time it was saved). If none of these two properties are true then there is no need for saving the data object to the RDBMS, because it does not change the content of the matching row.

Conversely, if the data object is new then the generated identity is extracted (if any) from the returned `DbDataReader` and the data object is added to the identity map prior to extracting and setting the timestamp of the data object. Next, the three state variables are set to `false` as the data object after a save is neither new, out of sync, or dirty. Lastly, the save method raises an event such

that those objects, which have subscribed to changes in the data object are able to react upon it.

Basically these instance methods alter the data object according to the database communication in those manners like setting the modified timestamp and controlling the state properties. The reason that the generated `Save` starts by calling the `base.Save` method is because in this way we could implement general functionality in the `AbstractRecord` class instead of generating the same code repeatedly.

The only thing happening in the `AbstractRecord.Save` method is to make a cascading save, i.e. first saving those objects which are used as references and translated into foreign keys in the table, as it is needed that they have a explicit identity in the RDBMS in order to be able to use them as foreign keys.

8.2.6 Modified Database Schema

In order for the VDO framework to be able to discover changes made by other clients without having to read every row from the table at question it is necessary to extend the database schema such that extra information can be saved for every row. This technique is often referred to as RIP tables, as they most frequently are used for keeping track of those rows being removed from the tables.

In VDO we do not settle with information about removed rows as it is just as important to be able to check if a row has been changed in the table since the last time of reading. This leads us to generate what we call an *action table* as it holds information about the latest modifications (i.e. actions) performed on the table at question. Figure 8.5 demonstrates the *authors* table together with the corresponding action table; *vdo_authors_actions*.

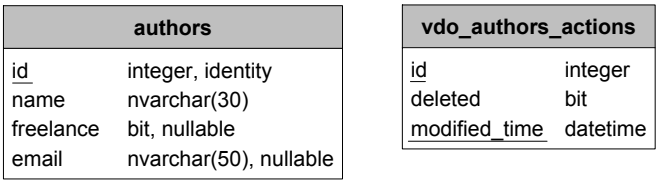

Figure 8.5: Diagram with a table and the corresponding generated action table.

The reason for not having a foreign key to the original table is because of the fact that we would like to be able to get information about deleted rows, thus a foreign key would not be a possibility. In the generated action table there is a column named `deleted` for setting whether an action is a delete action, a column named `modified_time` with the timestamp for the action and a column for each of the primary keys from the original table to make it possible to map the action to the proper row.

As one of our key criteria is to be able to use the VDO framework together with existing clients it is important that the action table is kept updated without interaction from the user, thus it is updated by the RDBMS. For VDO we have chosen to use triggers for this updating as shown in Figure 8.6 where a sequence diagram illustrates what is performed in order to update the action table when a row is inserted into the *authors* table.

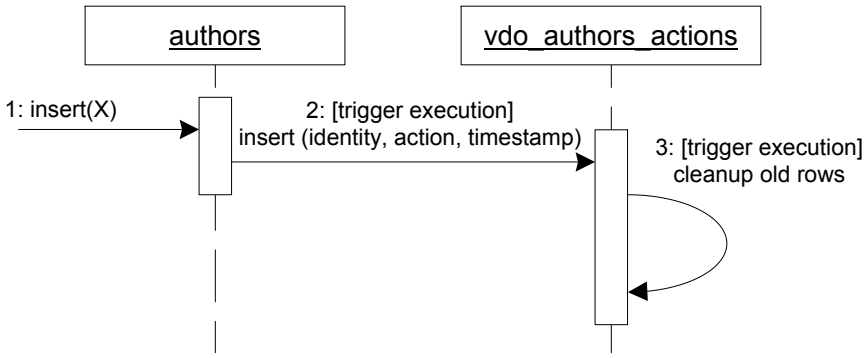

Figure 8.6: Sequence describing the insert SQL statement.

Whenever an `insert`, `update`, or `delete` is performed a trigger is executed to insert information about the operation into the action table. In the example in Figure 8.6 the action parameter is `false` as it should inform whether the row has been deleted. When a row gets inserted into the action table a trigger on this table will cleanup the action table by deleting any rows older than the just inserted. The reason for the cleanup in the action table is due to the fact that we are not interested in any history for the actions, but only want to know about the most current action and to counteract a overfilled table.

In order to be able to search in the action table we add an `index` to the columns, which is equivalent to the primary key columns from the original table as we have to search based on these columns. By adding indexes we are ensured that the RDBMS is able to search according to these columns in a more efficient way than if we have not added the keys, which is important as there can be performed many requests due to the integrated synchronization agent that polls the RDBMS for changes.

It should be evident that the action table is kept updated no matter which client is modifying the table, thus an application using the VDO framework is able to coexist with an application using another way of communicating with the RDBMS.

When polling the RDBMS for data we have to receive information from both the table in question and the action table in order to both read the actions that were performed most recently and to read the data from the rows. The most

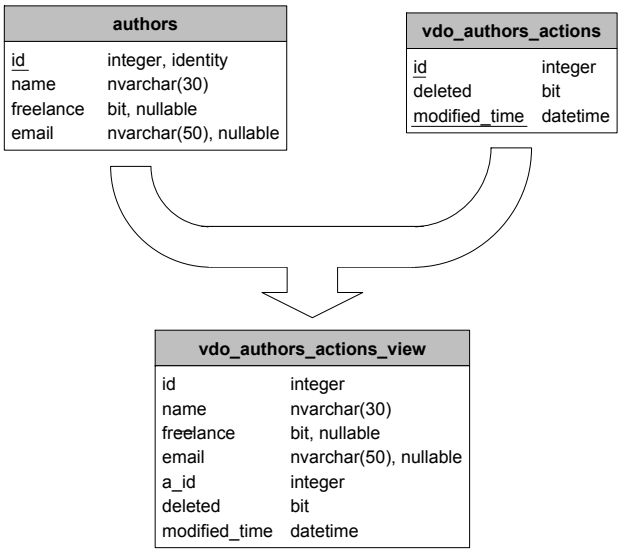

Figure 8.7: The generated view for the *authors* table.

straightforward way to do this is by joining the tables and as we request this information frequently a *view* is generated, from which we can select the rows of relevance. When the synchronization agent needs to get information from the RDBMS it only needs to request those rows from the view with a timestamp later than the one most recently read at the last sync.

Figure 8.7 depicts the view for the *authors* table and its corresponding action table. Listing 8.24 shows the SQL query employed to generate the view as the figure does not explain every aspect of the `join` operation.

```
1 SELECT authors.email AS email, vdo_authors_actions.id AS a_id,
     authors.id AS id, authors.name AS name, vdo_authors_actions.
     modified_time, vdo_authors_actions.deleted
2 FROM (authors FULL OUTER JOIN vdo_authors_actions ON authors.id =
      vdo_authors_actions.id)
3 WHERE (authors.id = vdo_authors_actions.id) OR (authors.id IS
     NULL OR vdo_authors_actions.id IS NULL );
```

Listing 8.24: The query used in the view from Figure 8.7 on the facing page.

The reason for including the `id` column from both the table in question and the corresponding action table is due to the fact that when a row is removed from a table it results in a `null` value when joining the two tables. Furthermore, if a row has not been saved subsequent to creation of the VDO triggers, then there is no equivalent row in the action table, thus resulting in a `null` value in the `id` column in the action table. Therefore we have to use both of the included `id` columns when polling the RDBMS for data as shown in Listing 8.25 to ensure that we get the data for the rows in question.

```
1 SELECT * FROM vdo_authors_actions_view WHERE a_id=1 OR id=1;
```

Listing 8.25: A sample query reading data from the view.

9 The VDO Compiler

During the previous Chapter we have covered what is generated into the specialized API and how it interacts with the static API, together named the *VDO framework*. In this chapter we describe in short how the VDO compiler works from input to end result.

The VDO compiler is a class generator, which generates the classes for a specialized API reflecting a given database schema. In Chapter 6 on page 67 we stated that this compilation process should work automatically such that the user has to supply the connection information for the VDO compiler to be able to read a database schema.

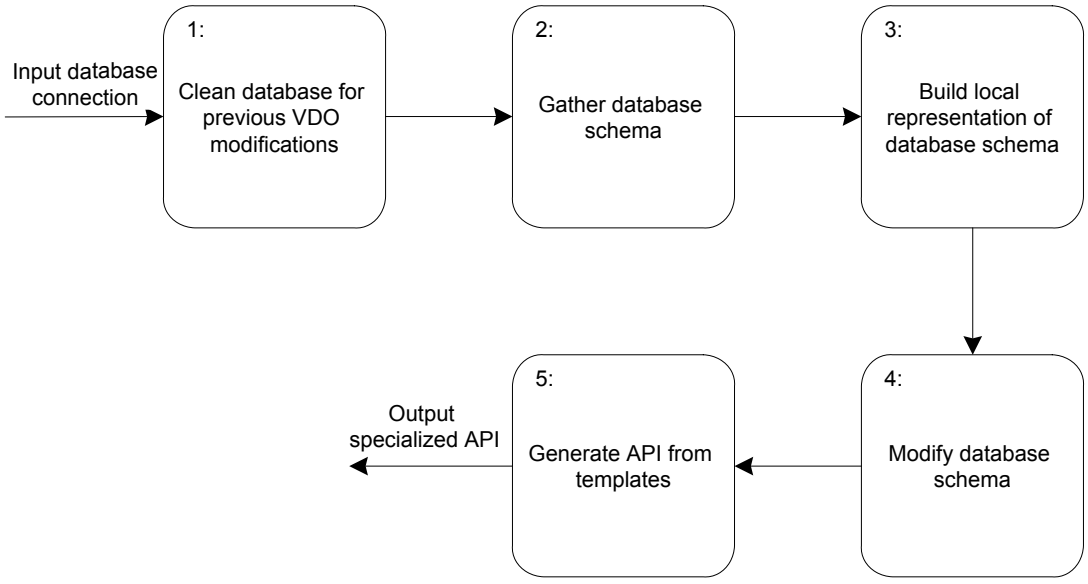

Figure 9.1: Flow of the VDO compiler.

Figure 9.1 illustrates the flow of the VDO compiler. In the following list we describe the main steps involved when generating code. It is important to

understand that as our focus has not been on the architecture of the compiler means that we do not go in detail about the compilation process, but only provide an overall description of the process. If the reader is interested in the internal structure of the compiler, then we refer him to the released source code on the VDO web site mentioned in the preface of this paper.

1. The VDO compiler starts by cleaning up the database schema for previous modifications (the modifications made are described in item 4). As every modification to the database schema has the prefix *vdo_* we can use this to detect what to cleanup from the database schema.

2. When the database schema in question has been cleaned we gather the database schema information, i.e. for every table, column, and type. Alongside this information gathering we ensure that the database schema conforms with our requirements, e.g. that no unsupported SQL types are used in the table.

3. From the gathered information about the database schema we create an internal representation to use in the rest of the compilation process. This internal representation includes meta data available from the database schema.

4. To make preparations for the database schema to work in cooperation with the VDO framework the compiler generates the action tables, necessary triggers, and views for the framework described in Section 8.2.6 on page 130. Every modification is generated with the prefix *vdo_* to ease a later removal of these modifications.

5. The final stage of the VDO compiler is the part which generates the classed to model the database schema for the specialized API and compiles these classes by way of the `CodeDOM` library in .NET. This procedure is done through templates and uses the previously gathered meta data.

After this 5-step process the VDO compiler ends by outputting the compiled API into a usable DLL file.

As we can see the needed input from the user is very limited as everything is taken care of by the VDO compiler.

IV Evaluation

10 Achievements

This chapter serves as purpose to provide an overview of accomplishments that we have achieved with VDO. This encompasses solutions to the problems outlined in Section 2.2 on page 34 and objectives put forward in Chapter 3 on page 39.

The principal characteristic of VDO is that it addresses a range of problems in a *combined* way that, to our knowledge and overview, is not accomplished by any other solution we have encountered.

Using VDO as an interface to a database entails a series of qualitative benefits. Our principal goal was to serve an application developer by enhancing productivity by reducing unnecessary reproduction of effort. This is realized by constructing a code generator that outputs a specialized API according to a provided database schema and its interdependent relations.

Our realization of this approach eliminates problems that are connected with safe queries, i.e. malformed SQL syntax, misspellings, SQL injection attacks and data type mismatch problems. Furthermore, this approach increases maintainability, because as a database schema changes throughout the lifetime of an application the developer can accommodate this by running the VDO code generator and produce a modified domain model of the database schema. The task of running the code generator only consists of providing connection parameters, because our goal in this regard was minimal effort and interaction from the user.

When the application is rebuilt with the modified DLL, compile-time errors such as the following are generated:

- *Cannot resolve symbol "X"* in the case when column x or table x is removed.
- *Cannot convert source type "int" to target type "smallint"* in the case of changing the data type of a column from `int` to `smallint`.
- *Constructor "Customer" has 3 parameters but is invoked with 2 arguments* in the case when a new column is added to a table.

This means that the specialized API gives compile-time support for the developer that ensures that database access code and database schema are consistent.

Figure 10.1: The IDE prompting the developer.

Secondly, as the VDO encapsulates a database schema entails that the IDE has information about the correspondence with the database schema. This gives opportunity for the developer to rely on IDE assistance in remembering things like names or data types of columns. An example of this is depicted in Figure 10.1.

The specialized API that is generated from the database schema combines different design patterns. Our realization of *Observer* in concert with *Identity Map* and *Synchronization Agent* has proven to be a beneficial composition, because this also has an impact on reducing unnecessary reproduction of effort.

Given the fact that the specialized API is automatically generated entails that the application developer does not have to implement three things that work in concert. We solved the uniqueness problem of representing a row as a unique data object in the programming environment. By exploiting this fact made it possible to employ a synchronization agent for each generated class in order to keep data objects and collections (search results as well as properties) of these consistent with changes in the database. In extension to this we automatically integrated the event pattern as means of informing about local changes and changes in the database.

This is maintained by us to contribute substantially in reducing coding efforts, because the developer is served with a solution that frees him of implementing a polling mechanism together with observer functionality as well as ensuring uniqueness. Furthermore, our integration of these components means that if structural changes such as adding new columns and tables are performed on the database schema, then the developer is not required to implement additional observer functionality and adapt the polling mechanism. It is just a matter of regenerating the domain model.

We hold that any application from our target audience that is currently operating with SQL strings could benefit from VDO.

Furthermore, we envisage that VDO may fit well in with applications that are being developed using the extreme programming methodology (XP) [49] as these are characterized of being subject to constant changes during the development process. Iterative customer feedback is reflected in iterative changes to the application and database schema, which in effect can cause a large amount of time being spent on keeping existing SQL code consistent with the changes. Using VDO in a setting such as XP may also result in that developers become more ready to perform changes to the database schema in order to comply with the needs of the customer.

A part of our objectives was to provide a solution that is user-friendly. Using the best practices outlined in Chapter 4 on page 43 during the design phase had a positive effect on us as designers, because it gave us a reference to frame our discussions. We exercised great care and thought in designing the parts of VDO that are exposed to a developer in a user-friendly and intuitive way by applying best practices from Section 4 on page 43 and drawing on own experience from usability theory. This does not replace a usability test in the field, but as we see ourselves as a constituting part of the target audience gives some justification to design choices that are based on own reasoning and experience with regards to usability.

In the next section we discuss some perhaps questionable design choices that have come to our attention after having designed VDO.

10.1 Design Decisions Reconsidered

In this section we examine selected parts of VDO that are subject to a reconsideration with regards to design.

In Section 6.1.3 of Chapter 6 on page 67 we stated that a read operation performed on a primary key that does not exist should result in serving an exception to the developer. Furthermore, the same read operation should return a *single* data object. This requirement gives rise to questioning the consistency of `ReadBy<X>` methods.

The design choice was rooted in the relational world, because a primary represents something that is unique, thus the incentive for returning a single data object. This is a consequence of bringing an entity from the relational world into the object-oriented world. If we weigh the pros and cons of returning a single object against a collection, then we have the following:

- **Single Data Object**: This signals uniqueness and is in accordance with the definition of a primary key. However it entails that we bring an exception

into existence as well as being inconsistent with regards to the interface of the other read methods.

- **Collection**: Here we do not require an exception because we could return an empty list and be in accordance with the interface of the other types of read methods. But taking this approach signals multiplicity and thus not in keeping with the relational view.

Basically this means that we have violated the best practice in Chapter 4 on page 43 that recommends that you should avoid return values that demand exceptional processing. The approach that we have taken forces the developer to use exceptions for control flow as a consequence of being "faithful" towards the relational world.

Another design issue worth emphasizing is that the read methods in VDO do not scale in the number and type of search criteria. Complex read operations was not an objective that we had when designing VDO. However, as a higher level of expressiveness with regards to search criteria in VDO is desirable in the future makes it relevant to mention the scalability of read methods.

If we were to implement the SQL logical connectives `and`, `or` and `not` together with comparison operators would incur a large amount of methods together with decreased readability to mention a few issues. A simple example that shows two possible ways of fetching persons with the same name and age by way of read methods is:

```
... ReadByNameAndAge(name, age);
... ReadByNameAge(name, age, Conjunction.And);
```

It is evident that this is not a good approach in order to enable a higher degree of expressiveness. Matters get worse when there is a need for a read method that is equivalent to the SQL statement of the form depicted in Listing 10.1.

```
SELECT ... WHERE name=@name OR (age < @age AND ...);
```

Therefore read methods are not subject for an extension in this way. They work in regard to simple searches with a single parameter.

10.2 Limitations

As satisfied as we are with our VDO solution we are aware that it has some flaws, which we cover in this section. It is noteworthy that some of the outlined problems are not specific for the VDO framework but more general problems with interacting with an RDBMS

As previously described in Section 8.1.5 on page 117 the `DBColumnCollection` is used to group the primary keys inside the data objects and is further responsible

for generating a hash code based on these primary keys. Unfortunately this behavior results in a non-constant hash value for a data object as it is generated upon the column name and the value for the column. As the value for identity columns are delivered by the RDBMS upon insertions and reset when deleted the value can be changing for this particular property throughout the lifetime of a data object as it can first be created, saved, and then deleted.

If the application developer puts the data object into a local hash table prior to and subsequent to saving he is able to insert the same data object twice. Further, any data object which has not yet been saved will have the same hash value, thus it is no longer unique for the particular data object as expected. Unfortunately this is a result of the implementation, thus we have not been able to hide our implementation details as good as we should with regards to the hash value.

When we generate the specialized API we do not allow foreign keys to a table that has more than one primary key column as this would be very difficult to interpret into object-oriented semantics as we see it.

In relation to database communication there are some issues as well. If any `SqlException` is thrown by the database connection during run-time, then these are not caught and re-thrown at the moment. As we see it there could be two reasons for getting an `SqlException` together with VDO:

- If the used username for the RDBMS does not has the required privileges, e.g. to execute `insert` queries.
- If the database schema has been altered since the time of generating the specialized API.

We have not taken any precautions against these two problems, which entails that the application developer is not provided with any safety, because the VDO framework does not produce exceptions if any of these two scenarios occurs.

When reading in data objects from the RDBMS it is not possible to specify more than one search criteria, which is a large restriction compared to using SQL directly for reading in data from the database. Neither is it possible to use joins between the tables to search by more criteria, thus VDO is not able to fill the needs when anything else than a very basic search feature is necessary.

As we have only implemented and tested Microsoft SQL Server 2005 we do not have a full outline whether we are tightly coupled with this implemented RDBMS, though we are certain that there are areas of the VDO framework that are based on functionality only available in conjunction with this database. Though, we believe that the functionality needed in an implementation of the `DataProvider`

is possible with most popular RDBMS available, thus it is possible to remove the coupling between VDO and Microsoft SQL Server 2005.

11 Questions Answered

In Chapter 3 on page 39 we expressed an interest in examining a series of aspects that are in relation to the design and realization of VDO. These aspects constituted a part of our overall objectives as they were engendered by a curiosity about applicability and consequences of our chosen API approach.

11.1 Active Record & Safe Queries

We defined safe queries as being comprised of the three constituents *query vulnerability*, *query correctness* and *type checking* in Section 2.1.1 on page 8. In the following we evaluate how well the chosen design pattern *Active Record* accommodates the concerns that we associate with safe queries.

Query vulnerabilities in applications that employ a database as back-end are often caused by weak input validation and dynamic construction of SQL statements that do not use type-safe parameters. A malicious user may exploit this security breach in order to execute SQL injection attacks that for instance grant access to sensitive information or erase data in tables. Our integration of *Active Record* into VDO has in this regard proven to be good design choice, because the pattern works well in concert with taking measures against injection attacks. The interface of a data object that a developer is exposed to is strongly typed, thus constrains input to prevent SQL injection. This narrows the possibilities of performing SQL injections on input that has type `string`. Input to a data object that is subject to be shipped to the database is bound to a type-safe `SqlParameter`, which causes input to be treated as a literal value and not as executable code by the database.

Query correctness consists of ensuring that generated SQL queries are valid in regard to representations of columns and tables that are used in the program-

ming language correspond to the database schema and that queries shipped to the database have correct syntax. Our approach to *Active Record* is based on mapping strategy where a table corresponds to a class, and columns correspond to constructor parameters and properties. This combined with the fact that the corresponding names used in the schema are encapsulated in a data object makes it possible to ensure query correctness. However, query correctness is not a consequence of using *Active Record*, but rather the result of design and implementation techniques. Automatic class generation and the fact that the interface of a data object does not allow for SQL strings is the combination that ensures query correctness.

Type checking was defined by us to ensure validity and correspondence between data types used in the database schema and data types used in the programming language. In this connection *Active Record* has proven to support us in ensuring type checking, namely because properties and method parameters of generated classes encapsulate private type classes, e.g. `DBNumberColumn<byte?>`. This setting makes it possible perform type checking on data types that the developer is accustomed with, while retaining the possibility of enforcing that the developer adheres to SQL data type declarations, for instance a column of type `nvarchar(50)` that is nullable.

In conclusion, the integration of *Active Record* into VDO has facilitated us in accomplishing safe queries. However it should be noted that it does not enforce safe queries by definition according to what we have learned from [21]. The reader may recall from Section 5.1 on page 56 that the description of *Active Record* stated that the pattern was *typically* characterized of having methods that did different things. This engenders a certain scope of freedom to influence the realization of the pattern by our own design preferences.

11.2 Integration of Observer & Synchronization

We wanted to investigate how far it would be possible to integrate the observer pattern together with a mechanism for automatic synchronization to serve a lightweight concurrency model and to reduce the workload for the developer as to keeping the local data consistent with the RDBMS in a multi client environment.

To be able to read changes from the RDBMS without having a huge overhead we needed to make it possible only to read in data which have been changed by other clients of the database schema since our last reading. In order to do that we needed to introduce a timestamp that should tell the time of the last modification made to the row at question. We also had to keep track of which rows are removed from the RDBMS to be able to update the data objects modeling these rows as the only alternative would be to read in every row from the table and check whether

any rows are missing according to the local data objects.

As we wanted our VDO framework to work together with other clients entailed that we could not alter the database schema with regards to the original tables as this could possibly break those other clients. Thus, we would have to place both the timestamp and the tracking of deleted rows outside the original table. Further, as VDO should be able to know if a non-VDO client updated the tables this information should be saved without special interaction from the client.

For our solution to be able to keep track of changes and especially deleted rows we introduced an *action table* for each of the original tables to hold timestamp information and whether the row at question were deleted or just altered. As this information should be saved independently of the client we chose to use triggers in the RDBMS to keep the action table updated automatically.

With the modified database schema we could implement a synchronization agent that continuously polls the RDBMS for changes and updates the corresponding local data objects accordingly to changes performed by other clients. Though, for the synchronization agent to be able to update the local objects required access to the loaded data objects, i.e. some sort of cache containing these data objects. In VDO this local cache is our *Identity Map* which also ensures uniqueness for the data objects such that the synchronization agent at most should update one local data object according to a changed row in the table.

As covered in Section 5.4 on page 60 we have chosen to use the *Event Pattern* which is an integrated part of C# which is similar the *Observer Pattern*, yet more flexible to use and integrate.

Whenever a data object is updated by the synchronization agent it uses the event pattern to inform objects that have subscribed to changes. The application developer can both subscribe to changes for specific properties or more generally as to when the data object gets updated, thus he is automatically informed when the data object gets updated by the synchronization agent.

Further, if any search results are present in the running application these are updated as well by the synchronization agent to reflect changes made to the data in the RDBMS both in regards to if data objects should be added or removed respective to the alterations done by the other clients of the database schema. As with the data objects the application developer has the possibility to subscribe to these changes and be informed when the content of search results changes.

With these findings in mind we can say that it is to a high degree possible to reduce the workload for keeping local data consistent with corresponding data in an RDBMS by taking advantage of a synchronization agent together with the *Event Pattern* in automatic code generation from a database schema. Though, for the synchronization agent to be able to detect every alteration of the data in the RDBMS with a minimum overhead modifications to the database schema are deemed necessary.

11.3 C# Influence on Design

In the development of VDO there are some language features of C# that have influenced the design of the specialized API in a positive way with regards to reaching some of our goals. The most apparent and important of these features have been:

- **Internals**: The fact that it is possible to make `internal` parts visible by way of the attribute `InternalsVisibleTo` in conjunction with a strong-named assembly has had a great impact on VDO as whole. This has enabled us in hiding parts of the API that are not relevant for the application developer, such as direct access to the database and implementation techniques like DBTypeColumn. If this was not possible then we either had to expose the developer for parts of the static API or include the static part in the generated DLL file, which would increase the file size. More importantly, by separating the static part from the generated part entails that if you wish to distribute a new version of a specialized API then you only have to distribute just what is changed.

  Furthermore, using internals makes it possible for us to eliminate errors in the static part without requiring the application developer to regenerate the specialized API.

- **Nullables**: This language feature has proven to be particularly useful, because it resulted in that we were able to map SQL data types that allow null to primitive types in C#. In Java this would only have worked if all SQL data types were mapped to reference types, because as of now Java does not support nullable primitive types.

- **Event Pattern**: Our integration of the observer pattern in C# has enabled us in designing the interface exposed to the user in a a non-intrusive manner by de-coupling the observer from the subject. Furthermore, a great benefit of using the event pattern is that delegates have the ability to refer any method that conforms to the same signature. Effectively this entails that any class can act as observer on a data object despite of the interfaces it implements or classes it specializes.

  We could have employed interfaces like `IObserver` and `IObservable` to accomplish a publish/subscribe mechanism, but the event pattern completely eliminates the coupling between the involved parts.

- **Properties**: We maintain that properties have contributed by adding a certain degree of elegance and intuitiveness to the design of the specialized part of VDO. This especially applies to our way of modeling foreign key relationships. Our requirement was that the specialized part of VDO should

correspond to a provided database schema and its interdependent relations. Properties that are references and referenced by effectively model implicit joins.

However, we could have accomplished the same functionality by using get and set methods known from Java, but in our view it makes more structural sense in mapping from columns to properties than mapping to methods.

V

Closure

12 Conclusion

In this thesis we set out to create an API that should assist an application developer by automating the task of both communicating with an RDBMS in a manner that accommodates safe queries and represents data from the RDBMS for easy manipulation.

The automation part was prompted by a series of issues that are connected with database access from an object-oriented environment, which we outlined in Section 2 on page 7.

The principal incentive for the automation was to enhance productivity by reducing unnecessary reproduction of effort and to provide a user-friendly way of representing data contained in an RDBMS.

We have shown that by using well-known design patterns and good practices for developing APIs it is possible to reduce the workload of the application developer while maintaining an intuitive framework which wraps the communication with an RDBMS and at the same time keeping the loaded data automatically updated.

Conclusively we maintain that even though the VDO framework has got some limitations with regard to the searching ability the overall result of our solution shows that it is possible to keep loaded data and search results updated according to changes made to the RDBMS by other clients.

References

[1] ADO.NET. http://msdn.microsoft.com/data/learning/adonet/.
[cited at p. 4]

[2] C#. http://msdn2.microsoft.com/en-us/vcsharp/aa336809.aspx.
[cited at p. iv]

[3] Java API specifications. http://java.sun.com/reference/api/.
[cited at p. 10, 11]

[4] Java Data Objects. http://java.sun.com/products/jdo/.
[cited at p. 5, 21]

[5] Microsoft SQL Server 2005. http://www.microsoft.com/sql/default.mspx.
[cited at p. iv]

[6] Microsoft Visual Studio 2005. http://www.microsoft.com/sql/default.mspx.
[cited at p. iv]

[7] MySQL database manual. http://www.mysql.org/doc/.
[cited at p. 10, 11]

[8] .NET Framework. http://msdn2.microsoft.com/en-us/library/x9t6k3aa(VS.80).aspx.
[cited at p. iv]

[9] OpenJava. http://www.csg.is.titech.ac.jp/openjava/.
[cited at p. 21]

[10] PostgreSQL database manual. http://www.postgresql.org/docs/manuals/.
[cited at p. 10, 11]

[11] Sun's Java Database Connectivity (JDBC) interface. http://java.sun.com/products/jdbc/.
[cited at p. 4]

[12] M. P. Atkinson and R. Morrison. Orthogonally Persistent Object Systems. *VLDB Journal*, 4(3):319–401, 1995.
[cited at p. 5]

[13] Joshua Bloch. How to design a good api and why it matters. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 506–507, New York, NY, USA, 2006. ACM Press.
[cited at p. 43, 44]

[14] Joshua Bloch. How To Design A Good API and Why it Matters - Slides. http://lcsd05.cs.tamu.edu/slides/keynote.pdf, February 2007.
[cited at p. 43, 44, 45]

[15] Joshua Bloch. How To Design A Good API and Why it Matters - Video. http://video.google.com/videoplay?docid=-3733345136856180693, February 2007.
[cited at p. 43]

[16] Frans Bouma. Why a cache in an O/R mapper doesn't make it fetch data faster. http://csharpfeeds.com/post.aspx?id=1593, August 2006.
[cited at p. 59]

[17] Gregory T. Buehrer, Bruce W. Weide, and Paolo A. G. Silvilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. *ACM*, SEM, september 2005.
[cited at p. 12]

[18] William R. Cook and Siddhartha Rai. Safe query objects: statically typed objects as remotely executable queries. http://webcourse.cs.technion.ac.il/236800/Winter2006-2007/ho/WCFiles/SafeQueryObjects.ppt. PowerPoint slides.
[cited at p. 21]

[19] William R. Cook and Siddhartha Rai. Safe query objects: statically typed objects as remotely executable queries. [41], pages 97–106.
[cited at p. 5, 21, 22, 25, 26]

[20] Microsoft Corporation. The LINQ Project. http://msdn.microsoft.com/data/ref/linq/, November 2006.
[cited at p. 5]

[21] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 1th. edition, 2002.
[cited at p. 35, 55, 56, 57, 58, 59, 60, 146]

[22] Steve Friedl. SQL Injection Attacks by Example. http://www.unixwiz.net/techtips/sql-injection.html.
[cited at p. 12]

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object Oriented Software*. Addison-Wesley, 1st. edition, 1995.
[cited at p. 35, 55, 60, 61, 106]

[24] William G. J. Halfond, Alessandro Orso, and Panagiotis Manolios. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In *ICSE*, pages 175–185. ACM, 2006.
[cited at p. 12]

[25] Rune Hammerskov, Jakob Andersen, and Lars Nielsen. Amigo - An Object Relational Query Language. http://www.cs.aau.dk/library, June 2006. Report is unpublished but available through AAU.
[cited at p. 5]

[26] Rune Hammerskov, Jakob Andersen, and Lars Nielsen. Language Integrated Persistence. http://www.cs.aau.dk/library, January 2006. Report is unpublished but available through AAU.
[cited at p. 5]

[27] Alex Henning Johannesen and Jacob Volstrup Pedersen. An Exploration of the Impedance Mismatch. http://www.cs.aau.dk/library, January 2007. Report is unpublished but available through AAU.
[cited at p. iii, 5, 7, 8, 13, 32, 37]

[28] W. Keller. Mapping Objects to Tables – A Pattern Language, 1997.
[cited at p. 5]

[29] Günter Kniesel. Encapsulation = Visibility + Accessibility. http://citeseer.ist.psu.edu/kniesel96encapsulation.html.
[cited at p. 51]

[30] David Maier. Representing Database Programs as Objects. In François Bancilhon and Peter Buneman, editors, *DBPL*, pages 377–386. ACM Press / Addison-Wesley, 1987.
[cited at p. 4]

[31] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Object Oriented Analysis & Design*. Marko, 2000.
[cited at p. 45, 75]

[32] Russell A. McClure and Ingolf H. Krüger. SQL DOM: compile time checking of dynamic SQL statements. [41], pages 88–96.
[cited at p. 5, 27]

[33] Oracle. Oracle Toplink. http://www.oracle.com/technology/products/ias/toplink/index.html, November 2006.
[cited at p. 5]

[34] Agile Data Home Page. Techniques for Successful Evolutionary/Agile Database Development. http://www.agiledata.org/, April 2007.
[cited at p. 4]

[35] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.
[cited at p. 48]

[36] D. L. Parnas. A technique for software module specification with examples. *Commun. ACM*, 15(5):330–336, 1972.
[cited at p. 48]

[37] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2000.
[cited at p. 45]

[38] Doug Purdy and Jeffrey Richter. Exploring the Observer Design Pattern. http://msdn2.microsoft.com/en-us/library/ms954621.aspx, January 2002.
[cited at p. 61]

[39] LLC Red Hat Middleware. Hibernate Reference Documentation, version 3.2.0ga. http://www.hibernate.org/hib_docs/v3/reference/en/html/, November 2006.
[cited at p. 14, 16, 17]

[40] LLC Red Hat Middleware. Relational Persistence for Java and .NET. http://www.hibernate.org/, November 2006.
[cited at p. 5]

[41] Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors. *27th International Conference on Software Engineering (ICSE 2005), 15-21 May 2005, St. Louis, Missouri, USA*. ACM, 2005.
[cited at p. 156, 158]

[42] Jeff Rubin. *Handbook of Usability Testing*. Wiley, 1994.
[cited at p. 45]

[43] Robert W. Sebesta. *Concepts of Programming Languages*. Addison Wesley, 6th. edition, 2004.
[cited at p. 32]

[44] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Book Company, 5th. edition, 2006.
[cited at p. 3, 4, 9, 37, 69, 72]

[45] Steven Smith. Caching in O/R Mappers and Data Layers. http://aspadvice.com/blogs/ssmith/archive/2006/09/01/Caching-in-O_2F0O_R-Mappers-and-Data-Layers.aspx#22311, September 2006.
[cited at p. 59]

[46] Michael Spivey. *An Introduction to Logic Programming through Prolog*. Prentice Hall, 1996. Available for free through http://spivey.oriel.ox.ac.uk/mike/logic/index.html.
[cited at p. 4]

[47] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. Advanced Programming Techniques. The MIT Press, 2nd edition edition, march 1994.
[cited at p. 4]

[48] Michael Thomsen. Persistent Storage of OO-models in Relational Databases. 1998.
[cited at p. 57]

[49] Don Wells. Extreme Programming: A gentle introduction. http://www.extremeprogramming.org/, February 2006.
[cited at p. 141]

[50] Wikipedia. Business intelligence. http://en.wikipedia.org/wiki/Business_intelligence.
[cited at p. 37]

[51] Wikipedia. Object-relational Impedance Mismatch. http://en.wikipedia.org/wiki/Object-Relational_impedance_mismatch, April 2007.
[cited at p. 4]