Faculty of Engineering and Science
Aalborg University

Department of Computer Science

TITLE:

Persistent language extensions and constructs for Java 1.5

SUBTITLE:

Achieving to solve a subset of the impedance mismatch problem.

PROJECT PERIOD:

Dat5, Cis3,
Sep 1st - Dec 20th 2005

PROJECT GROUP:

d530a

GROUP MEMBERS:

Rolf Njor Jensen
Thomas Møller
Peter Sönder

SUPERVISOR:

Per Madsen

COPIES: 5
PAGES: 85

Abstract:

This report has been developed at the Faculty of Engineering and Science at Aalborg University as a Computer Science project.
The purpose of this report is to shed some light on the impedance mismatch through an analysis of existing solutions. This analysis will then be basis for a prototype that implements a part of a new language construct for the Java programming language.
One of the ideas behind the new language construct is that it through a Java-like syntax will make it easier to express SQL queries. This way of querying will also make the mapping between the object-oriented model and the relational model transparent for the system developer.
The solution handles some known problems to existing object-relational mapping like static type checking, type mismatch, and optimization.

It's nice to be important, but it's more important to be nice

Scooter, Move Your Ass!, And The Beat Goes On! - 1995.

Preface

Prerequisites

Prerequisites to this report is intermediate knowledge of Java 1.5, SQL, object-oriented programming and modeling, and the relational model.

Reading notes

During the course of this report, a considerable number of acronyms is introduced. The first time an acronym is presented, it is written in full, followed by the shorthand form in parenthesis: Three Letter Abbreviation (TLA). Subsequent uses of the acronym is the shorthand form. At the beginning of this report there is a complete list of the acronyms used in this report.

Typographical notes

In the rest of the report, text that is source code of some sort will be printed with this font: "`System.out.println(''Hello World!'');`" or using a figure as shown in Figure 1.

```
1 System.out.println("Hello World!");
```

Figure 1: Example of source code.

Common examples

In Appendix A there is a description of a small set of relations, which will be used as a common example throughout the report.

Contents

List of Acronyms

The American National Standards Institute (ANSI)

The Institute's mission is to enhance both the global competitiveness of U.S. business and the U.S. quality of life by promoting and facilitating voluntary consensus standards and conformity assessment systems, and safeguarding their integrity. *19*

Application Programming Interface (API)

An application programming interface (API) is the interface that a computer system or application provides in order to allow requests for service to be made of it by other computer programs, and/or to allow data to be exchanged between them. *10, 11*, *16*, *21, 22*, *24*, *29*, *49, 50*

Binary Large OBject (BLOB)

The BLOB data type is from SQL, denoting a data type that contains a large binary value. The maximum size of the type varies from RDBMS to RDBMS *36*

Call Level Interface (CLI)

Interface to a RDBMS that uses text strings containing queries in the SQL syntax to perform operations on the database. *1*, *5*, *10, 11*, *19*, *39*, *61*

Database Management System (DBMS)

A collection of programs used to store, modify, and retrieve information from a database. *21*

Extended Backus Naur Form (EBNF)

All EBNF constructs can be expressed in plain Backus Naur Form (BNF) using extra productions. EBNF is more readable and succinct than BNF. *57*

Enterprise Java Beans (EJB)

Enterprise JavaBeans (EJB) technology is the server-side component architecture for the Java 2 Platform, Enterprise Edition (J2EE) platform. EJB

technology enables rapid and simplified development of distributed, transactional, secure and portable applications based on Java technology[14]. *24*

Explicit Query Execution (EQE)

The queries to be performed on the RDBMS are stated in SQL within the application, either as Embedded SQL or CLI. *5*

ExactVM (EVM)

Derived from the classic Java Virtual Machine (JVM) to support research into main-memory space management and garbage collection running enterprise servers under Solaris. *16*

HotSpot (HotSpot)

HotSpot is the primary Java Virtual Machine (JVM) for desktops and servers produced by Sun Microsystems[36]. It features techniques such as just-in-time compilation designed to improve performance. *16*

Hibernate Query Language (HQL)

HQL is an object query language used by the Hibernate framework[19]. *14–16*, *39*

The International Organization for Standardization (ISO)

The mission of ISO is to promote the development of standardization and related activities in the world. *19*

Java Compiler (JavaC)

The Java Compiler tool reads class and interface definitions (source files), written in the Java programming language, and compiles them into byte code class files[25], which can then be executed on a JVM. *19*, *24*, *34*, *51, 52*, *56*

Java DataBase Connectivity (JDBC)

JDBC is an API for Java that wraps CLI's for several different databases. JDBC is maintained by Sun Microsystems. *vii*, *1*, *5*, *10–12*, *19, 20*, *25*, *61*

Java Data Objects (JDO)

JDO is a framework for managing persistence within an EJB environment. *24*

Java Data Objects Query Language (JDOQL)

The query language used by JDO. *39*

Java Virtual Machine (JVM)

The Java Virtual Machine (or JVM) is a virtual machine, thus the name, that executes Java byte code. *16, 17*, *45*

Language Integrated Query (LINQ)

Language integrated query (LINQ) is a Microsoft project that aims to add a native querying syntax to C# and VB.Net. The feature can be applied to both XML and relational database data sources. *24*

Open DataBase Connectivity (ODBC)

A standard for an API that wraps CLI's for several different databases, providing a common interface regardless of the underlying RDBMS. ODBC is a standard co-opted by Microsoft from the SQL Access group consortium. *1*, *5*

Object-Relational (OR)

A term used to describe a mapping between a relational and an object-oriented data model. *13*, *24*, *63*

Plain Old Java Queries (POJQ)

A project[7] aiming to implement Native Queries[11]. *39*

Query By Example (QBE)

A way of querying where a prototype is specified, and the result of the query contains elements that match the given prototype. *22*

Relational Database Management System (RDBMS)

Software systems that manage and store data organized by the relational data model. See Abraham Silberschatz *et al*[2] for more verbose explanation. *1*, *3–9*, *11*, *16*, *35, 36*, *49*, *69*

Structured Query Language (SQL)

SQL is a widely used language to query RDBMS. Provides constructs for insert, update and deleting information in a RDBMS. *1*, *3–5*, *9–12*, *14*, *16–19*, *24, 25*, *39*, *42–44*, *46*, *51–59*, *62*

Three Letter Abbreviation (TLA)

A meta-acronym. As the name suggests, used to describe acronyms of length three. *v*

eXtensible Markup Language (XML)

XML is a metalanguage written in Standard Generalized Markup Language (SGML), which is defined by ISO standard 8879. *13*, *15*, *17*, *33*, *62*

Chapter 1

Introduction

1.1 Setting the scene: "The Impedance mismatch"

This is the scene: on one hand, Relational Database Management System (RDBMS) interfacable through the Structured Query Language (SQL), and on the other hand, applications developed with widely adopted statically typed object oriented languages such as Java[23] and C#[12]. The problem we will be addressing in this report is related to the problem of integrating databases and programming languages.

The problem has been dubbed *"The impedance mismatch"* by Maier[28] in 1990, and lately has been investigated extensively by Cook and Ibrahim in a as yet unpublished article named *"Integrating Programming Languages & Databases: What's the problem?"*[9]. The term *"The impedance mismatch"* stems from electrical engineering and is used to express how easily two systems are connected.

The most widely used method to connect programming language like Java to RDBMS are currently Call Level Interface (CLI) like JDBC and Open DataBase Connectivity (ODBC). Chosen mostly because they provide access to the databases superior abilities to query data, support multiple simultaneous accesses to data (concurrency), bulk data manipulation, etc. However, use of CLI lacks several properties - static checking of queries, resistance to injection attacks, type mismatch, etc.

Some work has been done in investigating the problems facing a solution to the impedance mismatch problem. And even though there are many existing (partial) solutions, all solutions seem to be deficient in some manner.

1.2 Outline

This report is divided in three chapters in addition to this. Starting with the *Analysis*, known criteria to evaluate solutions to the impedance mismatch problem are described, and a number of existing solutions are reviewed. Exploring their weak and strong points, a number of new criteria are identified. Prioritizing all of the cri-

teria, we identify a subset of the impedance mismatch problem we want to solve, and list selected design goals to a new solution.

Chapter 3 - *Designing a solution* - proceeds by developing the foundations for a solution to the problem. During the design phase, a rudimentary prototype has been developed, and Chapter 4 - *Implementing a compiler prototype* - describes the envisioned compilation process and the actual implementation. Chapter 5 concludes the project, and points to possible directions of future work.

Chapter 2

Analysis

In the following chapter we will try to lay out the problems - partly as they are described in already published work. From these problems we will identify a set of criteria from which a solution to the impedance mismatch problem can be evaluated.

We will also review existing solutions and approaches to integrating programming languages and databases. This review is conducted to identify weak and strong points of the existing solutions. These insights are then used to extend the set of criteria to evaluate a solution to the impedance mismatch problem.

In the end of the chapter we will precisely identify a subset of the problem of the integration, that we will try to solve - embodying the "Problem Statement".

2.1 Identifying the issues

In this section, we will present a list of criteria to evaluate some particular solution to the impedance mismatch problem. The presented criteria in this section stem partly from those presented by Cook and Ibrahim[9]. In some cases however, we identify the issues a little differently, and in these cases, it will explicitly be stated as such. Although the descriptions of the issues may seem a bit verbose, it is so to make the chapter readable without prior knowledge to the referenced work concerning the impedance mismatch. Some knowledge regarding the relational data model, SQL, and object-oriented programming in Java is, however, required.

2.1.1 Static Checking

The first issue to present is the ability to check code on compile time, rather than having it generate errors at run time. We refer to this as static checking. Two aspects are discussed in the following.

One area of static checking is static typing. Basically static typing is about whether or not it is possible to make static type checking of the interactions between the programming language and the RDBMS. In essence this means that if

static type checking is present, it is possible at compile time to check whether there are any type-related errors in the queries that are passed from the application to the RDBMS runtime.

Another area is static semantic checking. Static semantic checking is whether or not it is possible to check references to database tables from the programming language at compile time.

An example of the lack of static type and semantic checking and the problems this presents are presented in Figure 2.1. Imagine that we have set up a database connection named *conn* and execute the SQL statement (line 2 and 3). The example is based upon the common example in Appendix A. The example contains two semantic errors, and one type error. In line 2, in the select statement, the table *person* in the from clause is misspelled. In the same statement, a variable from the relation (dep_id) with the type integer is compared to a textual value. Both errors will not be exposed before run time. In line 6, the application tries to retrieve an integer from the result set - but the result set contains strings. This problem will remain concealed until run time too.

```
1 Statement stmt = conn.createStatement();
2 String sqlQuery = "SELECT fname FROM parson " +
3                   "WHERE dep_id = 'Peter'";
4 ResultSet rs = stmt.executeQuery(sqlQuery);
5
6 while ( rs.next() ) {
7   int departmentId = rs.getInteger("fname").intValue();
8 }
```

Figure 2.1: Demonstration of missing static checking

One solution to this problem could be to use SQLJ[35] or SQL DOM[29] to add static type check to the development process. SQLJ is described in Section 2.2.5. SQL DOM uses a object-oriented model to build the queries that are possible to make - which can be verbose in large systems.

Recent work[17] has proved it possible to solve some of the static type checking problems regarding dynamically generated queries. Although able to check certain type properties, the analysis does not currently cover query parameters or type-checking use of returned values.

Cook and Ibrahim[9] state that the static check criteria apply to all the other criteria, as an extra dimension.

2.1.2 Interface styles

By interface styles we mean how the system developer interacts with the persistent storage. In existing solutions this is a broad spectrum, ranging from orthogonal

persistence (described in Section 2.1.2.1) to Explicit Query Execution (EQE) (described in Section 2.1.2.2).

Besides the issues that arise with each extreme, another issue applies to the interface style: "Which paradigm?" Orthogonal persistence uses the syntax and semantics of the "host" language - be it Java, C#, SML, PASCAL, PROLOG or likewise. EQE uses the syntax and semantics of the language inherent to the RDBMS - SQL.

2.1.2.1 Orthogonal persistence

There does not exist a general definition of orthogonal persistence. We will define orthogonal persistence as Malcolm P. Atkinson[5] defines it. He uses three properties that must apply for orthogonal persistence to work. These are described in the following:

- *Orthogonality* - The persistence facilities must be available for all data, irrespective of their type, class, size or any other property.
- *Completeness or Transitivity* - If some data structure is preserved, then everything that is needed to use that data correctly must be preserved with it, for the same lifetime.
- *Persistence Independence* - The source and byte codes should not require any changes to operate on long-lived data. Furthermore, the semantics of the language must not change as the result of using persistence.

2.1.2.2 Explicit Query Execution

EQE comes in two flavors. One is Embedded SQL and the other is CLI. Both styles enable bulk data manipulation (updating many tuples with one SQL command), explicit optimization, and fine-grained control with queries.

Embedded SQL (or Embedded Queries as it is sometimes referred to), is SQL statements that are embedded into the host language, amending the syntax. The statements are present at call time, effectively enabling static type checking, but disabling dynamic queries - i.e. the ability to build queries dynamically during application execution.

CLI is a quite prevalent solution. Examples of CLI are JDBC and ODBC. SQL statements are sent to the RDBMS as textual, uninterpreted strings at runtime. CLI enables dynamic queries, etc. On the other hand, CLI suffers from being subject to injection attacks (having a user inject potentially harmful SQL statements when building queries), static type checking is not possible - with the possibility of type errors causing the application to misbehave at runtime.

Figure 2.2 shows an example of how SQL injection can occur. The problem occurs if a malicious user sets `fname` to some arbitrary SQL - say for instance `;drop table person;`. In this case the injected code would drop the person table which is definitely not the intention from the developer.

```
1 public String updatePerson(int id, String fname){
2   return String sql = "UPDATE person SET " +
3                       "fname = '" + fname + "' " +
4                       "WHERE id = " + id.toString();
5 }
```

Figure 2.2: Example of SQL injection.

2.1.3 Type mismatch issues

The type system of a programming language usually does not correspond to the types represented in databases. Even different databases can have dissimilar implementation of data types, since there is no absolute definition of the precision of numeric types[9]. The length of strings also varies between databases and programming languages, which again lead to incompatibility between the two worlds.

Another issue is the interpretation of null values. Programming languages and databases has a shared keyword *null* but it is interpreted different between the two. In programming languages a reference can be a null reference, which means that it has no target and therefore no value. In databases null values are seen as unknown values. This difference in semantics needs to be taken into consideration when dealing with null values.

Because the types between databases and programming languages are not directly comparable, we need to consider how data is mapped between the two worlds. We refer to this as data mapping.

2.1.3.1 Data mapping

There are two aspects of data mapping between databases and programming languages. One is how to map primitive types where the main concern are to preserve the precision of the data types. Another is how to map object types to the database. An issue that is somewhat more complex than handling primitive types.

Primitive types Considerations on how primitive types from the programming language can be stored in the database without the loss of precision needs to be made. Integers and reals are an example of this. Depending on which RDBMS and on which programming language chosen, there are slightly different value domains for the common types (integers, longs, etc), and some primitive types might exist in one but not the other - the varchar type is an example.

Object types Whereas primitive types have a fairly straight forward mapping of types between programming language and the database, object types has not. An object type can be anything from a rather simple string to an object with multiple references of many to one, one to many or many to many. Another problem with

object types is how to compare two types. Ambler[3] has written extensively on the matter, considering also how inheritance is mapped from object-oriented to relational, and how relationships among objects can be modeled.

2.1.3.2 Interpretation of null values

As we have mentioned earlier having a null value from a RDBMS and comparing it with a null value from an object-oriented language is not straight forward. First we need to look at how the language and the database perceives the word null.

Programming languages such as Java perceives null as a null reference. In Java a reference variable holds a null value until it is assigned an object. To illustrate this, we have Figure 2.3. In line 2 a variable `person` is declared but not assigned any value. It therefore holds a null reference. In line 3 a new object of type `Person` is assigned the variable `person` which result in `person` no longer being a null reference.

In other words a null value in Java is an *undefined value*.

```
1 ...
2 Person person;
3 person = new Person();
4 ...
```

Figure 2.3: Assignment in Java.

Databases uses `null` values in another manner. Null values in a database are interpreted as *unknown* values and therefore are treated differently than in programming languages. In a database aggregates etc. can be performed on null values - having the RDBMS simply omitting null values. In a programming language this will typically result in a null pointer error.

Since null values in a database is an unknown value a comparison will yield an unknown answer - a null result. As a consequence hereof in a database `age = null` will always return a null, even if `age` in fact is null. In contrast with programming languages `x == null` will return true if `x` is a null reference.

2.1.4 Reuse

2.1.4.1 Query parameters

Query parameters are parts of the query that are parametrized, with the parameter ultimately not known before runtime of the application. An example could be a filter - a string matching parameter. The query here would then depend on one or more input values.

2.1.4.2 Dynamic queries

Sometimes parameterized queries alone simply does not cut it - this is where dynamic queries come into hand. Different filters have different structures, and instead of simply creating different parameter values, dynamic queries are used instead.

Like parametrized queries, dynamic queries are built at run time. An example would be that several filters may be added, tuples ordered, joins added, etc. at run time. A wide spread use of dynamic queries appear in web applications, where the dynamic query is constructed depending on user input.

2.1.4.3 Modular queries

Composition and decomposition of programming constructs are basic properties of most high-level languages - and a solution to the impedance mismatch problem should support this.

However, (de)composition of queries may have severe impact on optimization. One of the strengths of the RDBMS is specifically optimization of the query based on extensive knowledge of data topology, current CPU load, known indexes, etc. This optimization becomes less doable if the queries are only sent in small chunks to the database with many round trips.

2.1.5 Concurrency

Support for concurrency in databases respectively programming languages are modeled distinctively different.

2.1.5.1 Transactions

In the database, concurrency is modeled with the concept of transactions. Several clients may compete for access to shared resources - the relations - grouping one or more interactions with the RDBMS into transactions. These transactions are then by the RDBMS guaranteed to fulfill the ACID properties (as defined by Silberschatz *et al*[2]):

Atomicity Either all operations of the transaction are reflected properly in the database, or none are.

Consistency Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

Isolation Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_1 and T_2, it appears to T_1 that either T_2 finished execution before T_1 started, or T_2 started execution after T_1 finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

Durability After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

2.1.5.2 Threading

A programming language such as Java has the concept of threading - which can be compared to the databases transactions. A thread is a single sequential flow of control within a program. This means that a system developer can use threads to isolate tasks.

Each thread is a sequential flow of control within the same program. This means that each thread can run independently from the other threads, but at the same time.

As with transactions, a thread has a beginning, then a sequence of operations followed by an ending. Threading makes it possible to run more than one instance of an application at a time.

All the threads of a Java program operate on the same memory, and exclusive access to resources is achieved using synchronization locks on objects and classes.

2.1.5.3 Evaluating concurrency

The model of concurrency in programming languages like i.e. Java, does not guarantee any fulfillment of the ACID properties. Many solutions to the impedance mismatch problem provide some way of starting, committing and aborting transactions. The problem is, however, that two executing threads may use the same connection to a database, and interleaving interactions to the database. The database sees only one connection with one transaction, but within the running application this may be two distinct flows of execution, and the ACID properties no longer hold.

2.1.6 Optimization

As mentioned before, one of the tasks that a RDBMS is particularly good at, is optimization of query execution. This involves rearranging query operations, choosing which indexes should be used to select particular tuples, etc.[16]

In the following, we divide the issues by search and navigational issues - responding largely to the `WHERE` and `SELECT` parts of an SQL query. Lastly, we consider optimization issues in data manipulation (`INSERT, UPDATE, DELETE`).

2.1.6.1 Optimizing search - Criteria shipping

Query optimization is performed by most RDBMS. It is based upon the databases knowledge about how data is stored, available memory etc. This knowledge is not available to the programming language. Therefore a mechanism to gather criteria from a query is needed, so that it can be passed to the RDBMS at runtime.

2.1.6.2 Optimizing navigation

Prefetching related objects When translating the relational data model to the object model, foreign keys in tuples become references between objects. When querying the database, it is an issue how the referenced objects get loaded. Following the references blindly may cause loading of a very large object graph, although it is never needed by the application. However, if no prefetching is done the result may be that the database is queried many times - reducing performance[6].

There are several degrees of solution present for this issue. In the most extreme, CLI forces the system developer to define how much data is loaded - giving the system developer explicit control. Some object relational tools do not enable navigational prefetching at all, some enable global settings, and some even allow for query specific settings, although these seem cumbersome to specify.

2.1.6.3 Bulk data manipulation

A general trait of the optimization issues is whether queries are grouped and shipped to the database engine in large portions.

This also applies to data manipulation operations - i.e. `INSERT`, `UPDATE`, `DELETE` statements. In SQL it is possible to define data manipulation to a set of tuples defined by some criteria - e.g. moving all persons from department 1 to department 2: `UPDATE person SET dep_id = 2 WHERE dep_id = 1`.

Performing the same operation by loading all the objects into the application and then modifying the department id (dep_id) on each member and finally updating each object in the database greatly degrades performance.

2.2 Reviewing Existing Solutions

In the following section we will present a review of several existing solutions to the impedance mismatch problem. Each of the solutions we review, are representatives of different approaches to solving the impedance mismatch problem.

Each of the reviews will start with a brief introduction of the solution. Then we will outline how a solution to the common example in Appendix A would look like using this particular solution. Specifically with respect to setting up the environment, how to store objects, and finally how to perform queries. Ultimately, there will be a listing of the weak and strong points of the solution - either evaluated by the criterion already put forth - or given that neither of them are applicable, introducing a new criteria.

2.2.1 JDBC

Java DataBase Connectivity (JDBC) is a number of Java classes and interfaces that specify an Application Programming Interface (API) that a *driver* for a spe-

cific RDBMS should implement. The provided API enables sending queries to the database, and retrieving results from the queries.

JDBC is a CLI, which means that communication with the database is uninterpreted strings. There is no automatic mapping and conversion between objects and tuples, but instead it is the system developer that is responsible for the mapping.

2.2.1.1 Setting up the environment

Setting up a JDBC connection requires two steps. One is to load the driver and the other is to make the actual connection. Figure 2.4 illustrates a connection to a MySQL[31] database called *somedb*.

```
1 Class.forName("com.mysql.jdbc.Driver");
2 Connection con = DriverManager.getConnection(
3         "jdbc:mysql://localhost:3306/somedb",
4         "user",
5         "password");
```

Figure 2.4: Setup of JDBC connection.

2.2.1.2 Storing Objects

Once the connection has been established, it is time to utilize it. Retrieving objects from the database is done using a SQL statement directly embedded in the source. Figure 2.5 illustrates this.

```
1 Statement stmt = con.createStatement();
2 String fname = "Peter";
3 stmt.executeQuery(
4          "INSERT INTO person (id, fname, lname, dep_id)
5             VALUES (’’, ’" + fname + "’, ’Madsen’, ’1’);");
```

Figure 2.5: Storing objects with JDBC.

2.2.1.3 Querying

The difference between storing and querying is that a query returns a ResultSet. A ResultSet is a table of data representing the database result set. It provides getter methods for retrieving columns values. An example of this is illustrated in Figure 2.6 where we invoke two getter methods called `getString()` and `getInt()`.

```
1 Statement stmt = con.createStatement();
2 ResultSet rs = stmt.executeQuery(
3       "SELECT * FROM person
4           WHERE fname = 'Peter' AND dep_id = 1");
5 while (rs.next()) {
6     String fname = rs.getString("fname");
7     int depId = rs.getInt("dep_id");
8     ...
9 }
```

Figure 2.6: Querying objects with JDBC.

2.2.1.4 Evaluation of JDBC

JDBC implements SQL as uninterpreted strings in the programming language. This means that there is no static checking on types nor is it possible to check for spelling errors in column names etc.

The actual mapping of objects to tables must be done the system developer, and can be a tedious task if the database schema changes on a regular basis.

Optimization and bulk data manipulation can be done with fine-grained control, as the system developer has the power of SQL right at hand. JDBC allows SQL statements to be grouped together into a single transaction and can thereby fulfill the ACID properties. The transaction is controlled by the `Connection` object which means directly implemented by the system developer.

Pros

- Support for transactions through the JDBC driver.
- Support for bulk data manipulation and optimization.
- Explicit declaration of SQL statements - full support for SQL statement.

Cons

- Tedious to implement persistence.
- No support for static type checking or typos in the SQL statements.
- No direct mapping between the relational data model and the object-oriented model.
- Explicit declaration of SQL statements.
- Vulnerable to script injection.

2.2.2 Hibernate

Hibernate is a framework, which somewhat automates the task of mapping Java objects to a relational database. Hibernate handles one-to-one, one-to-many and many-to-many relations. These relations are defined in eXtensible Markup Language (XML) files called mapping files, thus not completely eliminating the tedious task of mapping between the object-oriented paradigm and the relational database.

2.2.2.1 Setting up the environment

Hibernate uses mapping files, that are XML files written by hand. This leaves room for mistakes that can compromise the correct mapping between the database and the programming language.

These mapping files contains all information about Object-Relational (OR) mapping such as primary key, foreign key and database attribute.

To illustrate this mapping, we use the database example from Appendix A. In this example (see Figure 2.7) it is easy to see that it is still possible to make typos in i.e. table names. The example shows what is needed to connect to a database using Hibernate.

```
1  <hibernate−mapping>
2    <class name="Person" table="person">
3      <id name="id" column="person\id">
4        <generator class="increment"/>
5      </id>
6      <property name="fname"/>
7      <property name="lname"/>
8      <property name="dep_id"/>
9    </class>
10 </hibernate−mapping>
```

Figure 2.7: Hibernate mapping file.

Once the mapping files exist, Hibernate delivers many tools for manipulating the database. The Hibernate framework include tools for:

- generating Java classes from existing mapping files.
- generating mapping files from existing database tables.
- generating database tables from existing mapping files.

As you can see, one of the tools have support for generating mapping files from an existing database, thus eliminating the hand written error that might occur.

Hibernate uses mapping files to directly describe which columns belong to what table, and what table is related to what database - in case more than one database is in use.

The Hibernate framework is situated between the application and database. It creates a tier between the two. This is illustrated on Figure 2.8.

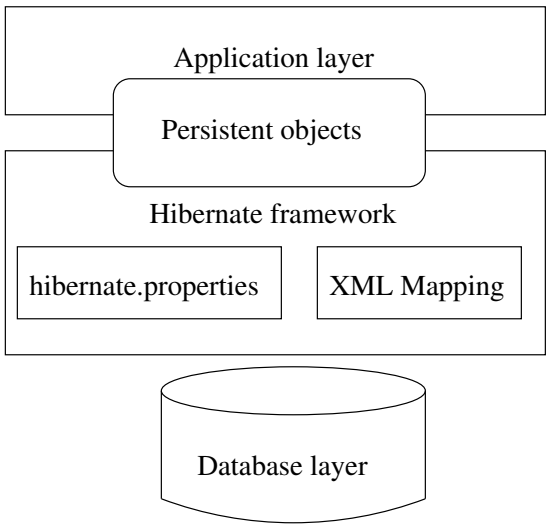

Figure 2.8: Hibernate architecture (source: hibernate.org).

2.2.2.2 Storing objects

Hibernate offers a persistence manager that handles storing and retrieving objects from the database. In the Hibernate framework this is called a *session*. Each session contains a transaction, where it is possible to make a rollback or commit as known from the database world. When the persistence manager, or session, is set up, objects can be stored in the database, as shown in Figure 2.9.

From this example it is easy to see that once the object should be persisted, it is done using the `session.save(person)`. Finally, the entire transaction need to be committed as if it was a database. This is done using `tx.commit()`.

2.2.2.3 Querying

Hibernate implements its own query language called Hibernate Query Language (HQL)[20] that is an object-oriented query language. Furthermore SQL native to the used database can be used.

A Hibernate session has a `createQuery` method, that can be used to query the database. An example of a query in HQL can be seen in Figure 2.10.

```
1  Session session = HibernateUtil.currentSession();
2  Transaction tx = session.beginTransaction();
3
4  //Create our example object
5  Person person = new Person();
6  person.setFName("Peter");
7  person.setLName("Madsen");
8  person.setDepId(1);
9
10 session.save(person);
11 tx.commit();
12 HibernateUtil.closeSession();
```

Figure 2.9: Storing objects in Hibernate

```
1  Query query = session.createQuery(
2    "SELECT p FROM person AS p WHERE p.lname = :lname");
3
4  query.setCharacter("lname", 'Madsen');
5
6  for (Iterator it = query.iterate(); it.hasNext();) {
7    Person p = (Person) it.next();
8    //Handle your query result here...
9  }
```

Figure 2.10: Retrieving objects in Hibernate.

Furthermore, HQL can handle filters through a `createFilter` method on session. Aggregate functions can be handled through scalar results. Scalar results are considered ”scalar” since they are not entities in persistent storage.

2.2.2.4 Evaluation of Hibernate

Problems with mismatch between the type system of the database and the programming language, is still existing but can be handled by the user through the XML mapping files. This is a manual task, since types of the database does not match the types of the programming language.

Hibernate supports transactions, though this has to be implemented manually by the system developer. Hence concurrency can be obtained, but the system developer needs knowledge about the underlying database system. One of Hibernates key features is a cache architecture, that can be used in a cluster environment to enhance performance.

Last Hibernate offers a more automated implementation of tedious database mapping, though manual labor is still needed.

Pros

- The system developer is relieved of writing trivial methods to save objects to persistent storage, and to marshall primitive values from tuples into objects.
- Handles data mapping (once the mapping files are in place, and only when they are correct).
- The RDBMS does not need to be present at compile-time.
- Optimization through caching.
- HQL is closer to the object-oriented paradigm than SQL.

Cons

- Although HQL is closer to the object-oriented paradigm, it is still influenced strongly by SQL.
- A somewhat manual mapping between database and programming language.
- No static typing.
- A lot of work is needed to set up the tool.

2.2.3 PJama

PJama[33] was developed by Sun Microsystems[36] and Glasgow University in the years 1996-99 after which it was aborted. PJama was an experiment to fully implement orthogonal persistence in the language Java. Unfortunately for the PJama team, they choose to use ExactVM (EVM) instead of HotSpot (HotSpot), which, at the time was a new and sophisticated Java Virtual Machine (JVM). EVM was abandoned in the summer of 1999, which also meant that the PJama project was abandoned.

2.2.3.1 Setting up the environment

Although PJama has been abandoned, it is still possible to get it to work. The latest version is 1.65 and was updated in October 1999[4].

PJama implements persistence independence (see Section 2.1.2.1) to such a degree that there is no change in semantics - the only thing needed is very few lines of code that explicitly call the PJama API to set up the application.

2.2.3.2 Storing objects

Due to the nature of orthogonal persistence, there is no need to explicitly store objects.

2.2.3.3 Querying objects

As with storing, there is no need for explicitly querying.

2.2.3.4 Evaluation of PJama

Although PJama has been abandoned it has not all been in vain. PJama has had some great achievements, which will we discuss briefly here.

First of all, PJama has been a *proof by case* even though it never was completed. The PJama team have shown the world that it is possible to implement a language that is orthogonal persistent. They made a demo that showed the ability to stop and start a Swing[23] demonstration - preserving the interactive state[5].

As already mentioned, they achieved persistence independence. The crew also tested scalability with up to 10 Gbytes of data and more than 300 million objects - and recall that this was in the early 1999. They even supported virtually any changes to the definitions of classes (also known as schema evolution).

Transactions is an area of PJama that is considered tricky and hard to implement. Since storing objects is done automatically by the JVM it is necessary for the system to guarantee that a transaction does not overlap checkpoints, where the JVM stores objects. If this happens, the system could end up in a inconsistent state. This problem, is yet to be solved by the developers of PJama.

Pros

- Schema evolution.
- Transparent persistence.

Cons

- Changed the JVM.
- Lack of transactions.
- The power of relational databases is missing.

2.2.4 Cω

Cω is a experimental version of the C# programming language and is developed by Microsoft Research[30]. While some of the features of Cω have been incorporated into C# version 3, Cω remains a development language.

Cω amends C# with the aim to implement natural language abstractions working with relational data and XML documents[15]. Cω offers type check on SQL queries on the same level as on types in the programming language, given that SQL syntax is amended to the rest of the language - it is integrated into the host language.

2.2.4.1 Setting up the environment

When dealing with a Cω project that uses SQL, the database that the application has to work on has to be present. From the database, the Cω compiler infers the necessary information to be able to make static checking of the code.

2.2.4.2 Storing objects

Figure 2.11 shows how a tuple is inserted into a relation. As you can see, the SQL syntax of the example is part of the language on equal terms with the object-oriented syntax. Semantic checks and type checks are done at compile time.

```
1 public class StorePerson {
2   static void Main() {
3     int n = INSERT id = "", fname="Peter" INTO DB.Person;
4     ...
5   }
6 }
```

Figure 2.11: Storing objects with Cω.

2.2.4.3 Querying

Figure 2.12 shows an example of a query. Like the insert, the query is part of the language. The result of the query is a resultset - an iterator, with structs containing the variables of the resulting tuples. This also shows that there is no conversion from tuples to objects and vice versa.

```
1  public class RetrievePerson {
2    static void Main() {
3        ...
4        res = SELECT * FROM DB.Person WHERE fname=="Peter";
5        foreach(row in res) {
6          Person p = new Person();
7          p.setFName(row.fname);
8          ...
9        }
10   }
11 }
```

Figure 2.12: Retrieving objects with Cω.

2.2.4.4 Evaluation of Cω

Since Cω implements SQL directly in the programming language, search optimization and bulk data manipulation can be done with just as fine-grained control as with CLI.

A strong aspect of Cω is the static checking system, eliminating many potential run time errors. Moreover, the type mismatch issues are handled by the compiler.

Cω has built-in language constructions for transaction handling, thus making concurrency programming possible with the ACID properties intact.

What Cω does not do, is converting tuples to objects, references to collections, and vice-versa. This task is still the responsibility of the developer.

Pros

- Support static type checking on SQL queries.
- Support for concurrency integrated by language constructions.

Cons

- No mapping between the relational data model and the object-oriented model.
- The original object-oriented language is extended with at new paradigm given that SQL syntax is integrated directly into the language - this gives the programmer more expressiveness, but also a larger language to master.

2.2.5 SQLJ

SQLJ is a set of programming extensions invented by large corporations including Oracle[32], Sun Microsystems[36] and IBM[22]. It is a The International Organization for Standardization (ISO) and The American National Standards Institute (ANSI) standard. It allows a programmer using Java to embed statement that provide access to a SQL database. SQLJ defines the embedding of static SQL statements in a Java program. An expression written in SQLJ is usually more compact and readable than an equivalent expression using JDBC.

A program written with SQLJ cannot be compiled by Java Compiler (JavaC), since the syntax is not valid Java syntax. The SQLJ uses a translator to create valid SQL statements. It uses `#sql` to identified the presence of SQL statements. It uses `:` to indicate the presence of Java attributes inside the SQL statement.

2.2.5.1 Setting up the environment

SQLJ consists of two components: the translator and the runtime libraries. So, in order to get SQLJ to work, you need to get the following:

- Oracle SQLJ translator.
- Oracle JDBC driver.

- Oracle database.

As you might have noticed, SQLJ uses Oracles translator, JDBC driver and database. It is also have used IBM's database called DB2. At the time of writing there does not exist any support for other databases like for instance MySQL[31] or PostgreSQL[34].

Creating access to the database is done as you do with JDBC. This is illustrated in Figure 2.13.

```
1 Oracle.connect(
2   "jdbc:oracle:thin:@localhost:1521:persijdb",
3   "user",
4   "password"
5 );
```

Figure 2.13: Initialization of SQLJ

2.2.5.2 Storing objects

Once more we return to our example database from Appendix A. Once more, we are trying to insert a person into the database. This is illustrated in Figure 2.14.

```
1 String fname = "Peter";
2 #sql {
3   INSERT INTO person (fname) VALUES (:fname)
4 };
```

Figure 2.14: Storing data with SQLJ

2.2.5.3 Querying

Retrieving objects is done in a similar fashion. This is shown in Figure 2.15.

2.2.5.4 Evaluation of SQLJ

One good thing about SQLJ is the support for static type checking. This means that typos can be caught on compile time rather than on runtime as it does with regular JDBC.

Pros

- Supports static type checking.
- Transactions through Java threading.

```
1  //Declare an instance of the SelRowIterator
2  SelRowIter result = null;
3  String fname = "Peter";
4  #sql result = {
5      SELECT * FROM person WHERE fname = :fname
6  };
7
8  //Populate the iterator and process all rows returned
9  while(result.next()) {
10   fname = result.fname();
11   ...
12 }
```

Figure 2.15: Retrieving data with SQLJ

Cons

- No support for dynamic queries.
- No support for system developer to optimize the queries. This is done by SQLJ on compile time.

2.2.6 db4objects

db4o[13] is an object-oriented database. It comes in a version for C# and in a version for Java. In the newest version of db4o, version 5, Safe Query Objects[10] are implemented according to "Native Queries for Persistent Objects, a design whitepaper"[11]. In the following, examples are in Java 1.5 syntax, but equivalent examples are possible in C#.

2.2.6.1 Setting up the environment

The db4o API does not require the programmer to mark the classes as persistent, nor does it require any mapping files.

The Database Management System (DBMS) in db4o can operate in a "local" mode accessing a database file, but not requiring a database daemon, or in client/server mode. The code in Figure 2.16 sets up access to the database file.

2.2.6.2 Storing Objects

Figure 2.17 is a small example showing how to store objects in the database using the db4o API. Using Appendix A as a starting point, we are creating a new person with the name "Peter" and storing the object in the database.

```
1 private Starter(){
2   new File(dbfile).delete();
3   db = Db4o.openFile(dbfile);
4   try {
5     //database interaction here...
6   } finally {
7     db.close();
8   }
9 }
```

Figure 2.16: Initialization and utilization of db4o.

```
1 private void storeObjects() {
2   Person person = new Person();
3   person.setFName("Peter");
4   db.set(person);
5 }
```

Figure 2.17: Storing objects in db4o.

2.2.6.3 Querying

Queries can be undertaken in three ways in db4o. One is through Query By Example (QBE), another is Query API and the third is by using Native Queries.

Since db4o uses an object-oriented database, Native Queries become interesting. Since there is no mapping between the database and the programming language, Native Queries is to be able to express the queries to the database using the normal syntax and semantics of the host language. Figure 2.18 shows a simple example of a native query.

```
1 private void retrieveObjects() {
2   ObjectSet person = db.query(
3     new Predicate<Person>() {
4     public boolean match(Person person)
5       return person.getFName().equals("Peter");
6     }
7   );
8 }
```

Figure 2.18: Using native queries in db4o.

`db.query(Predicate predicate)` is the method that does the actual querying. It returns an instance of `ObjectSet` which basically is an iterator. Predicate

is an abstract class which includes the signature `public abstract boolean match (ExtentType candidate)`; which (according to the semantics of an abstract member) must be implemented in the actual parameter. Consequently `new Predicate<...> {...}` is written in the `query` method.

The purpose of the match method is to evaluate whether the actual parameter candidate is to be included in the set of objects being returned - if so, to return true, and if not, to return false.

This way, the implementation of the match method can use the instance of the object to test whether the desired criteria are fulfilled.

Given that all the tests are expressed in the host language, the queries are subject to the same level of type checking as the rest of the application. Moreover, the queries are not subject to script injection problems.

2.2.6.4 Evaluation of db4o

Queries in db4o are expressed in natural Java syntax. Because of this, database calls does not break with the object-oriented paradigm. On the other hand, because of the nature of the match-method, we loose the relational power like aggregate functions (`MAX, MIN` etc.) and the cost of this is perhaps a decreasing efficiency.

Notice, although we state that there is little or no room for optimization, we have not been able to find some concluding evidence for this statement, but taking into account that db4o's focus is on persisting objects and not directly on performance. If the purpose of the application is to request and manipulate primitive values instead of objects most of the time, then db4o is not the best choice for the application, if optimization is in focus.

Pros

- Supports static typing.
- Uses the programming languages syntax (Java or C#).

Cons

- Not orthogonal persistence.
- No or little room for optimization.
- Concurrency is handled manual.

2.2.7 Other Solutions

The solutions presented so far, are all exponents for some particular flavor of the wide range of solutions to the impedance mismatch problem. In the following section we present some of the other solutions we have encountered during our research.

2.2.7.1 Object-Relational Mapping Frameworks

- TopLink[37] is an OR mapping package for Java. It is a commercial product owned Oracle in 2003.
- Enterprise Java Beans (EJB) is an API in the Enterprise Edition of the Java 2 Platform. Among other things, EJB provide persistence.
- Java Data Objects (JDO) is a specification of Java object persistence. It is made by Sun Microsystems and is also implemented by others including JPOX[26] and Castor[8].
- iBATIS[21] is a Data Mapper framework for Java and .NET applications.
- Language Integrated Query (LINQ)[27] is a set of extensions to the .NET Framework that encompass language-integrated query, among other.
- Another project group in Aalborg University is currenctly also looking at the impedance mismatch problem - using C# as their choice of language.

2.3 Discussion

In this section we will summarize the criteria used throughout the analysis. Going through the existing solutions we have also found some new criteria, which we will also discuss. This discussion then leads to the problem statement in Section 2.4.

2.3.1 Conclusion on review

When going through the reviews we found some new criteria worth looking at. They are listed in the following.

Build process focuses on how cumbersome the introduction of the framework or language extension becomes. For instance take Hibernate (see Section 2.2.2). Setting up Hibernate can be a tedious task where mapping is done manually, but once the mapping is done, the framework ease the task of interacting with the database. Also, how complicated the compilations process is.

Tool support applies to whether or not there exists tools to support the development. For instance, working with SQLJ (see Section 2.2.5) Java code that is not compilable with JavaC is created. This adds another step to the *build process* - here we need a translator that translate SQLJ to compilable Java code.

Language alteration can be seen as language extension. Take Cω (see Section 2.2.4). Here the developer needs to learn a whole new syntax, since SQL is written directly inside the code. In fact, Cω and SQLJ adds elements from another paradigm - the declarative paradigm. This greatly increases the developers learning curve.

Schema evolution is an expression of how well changes in the object model is transferred to the underlying database schema. Although PJama never was implemented fully, it handles this whereas a JDBC solution does not handle this at all. All changes made in the object model will need to be modified in the SQL statements.

2.3.2 Summarizing criteria

In Table 2.1 we have summarized all the criteria we found during analysis including a little description of each of the criteria.

Name	Description
Static checking	Having static checking is a huge advantage for the system developer. Static checking enables the ability to check code at compile time, type as well as semantics. This greatly decreases the risk of encountering run time errors.
Interface style	Interacting with the database either through SQL or the host language is something that need to be considered. Some existing solutions to the impedance mismatch problem breaks with the object-oriented paradigm thus ending up as a multi paradigm language. The level of persistence should also be taking into account.
Type mismatch issue	Either the problem of matching the type system from the programming language to the database is handled directly by the system developer (i.e. in JDBC) or it is handled by the framework or language extension (i.e. with Hibernate once it is up and running).
Reuse	To what extent it is possible to reuse parts of the query in different contexts.
Concurrency	To what extent the application supports currency and to what degree it supports the ACID properties.

continued on next page

continued from previous page

Optimization	Whether or not the system developer should be able to do optimization. This could be through prefetching related objects.
Build process	Introducing either a new framework or language extension the build process cycle can vary greatly. PJama has little or no extra build steps whereas SQLJ adds the need to run a pre-compiler
Tool support	To what extent there is tools that support the framework or language extension.
Language alteration	How many, if any, changes are there made to the host language. In Cω another paradigm is introduced. PJama does not add any new syntax.
Schema evolution	To what extent changes in the database schema is handled by the framework or language extension.

Table 2.1: Summarizing criteria from the analysis.

2.3.3 Prioritizing criteria

Now that all the criteria has been described and we have looked at the criteria in context to different frameworks and language extensions, it is now time to prioritize them.

2.3.3.1 Criteria in focus

Static checking The ability to perform static checking is desirable. Having the power to catch errors at compile time instead of at run time would greatly increase the stability of programs, and must be seen as an important feature.

Type mismatch issue Another great concern is the issue regarding mapping objects between a programming language and a database. Having the language or framework handling this possible mismatch would relieve the developer of a tedious work task. Along with static checking this increases the robustness of database interaction.

Language alteration Having a framework or language extension that is intuitive and easy to use is also something we would like to have. This makes it more easy for the system developer to use the new functionality. If it is possible to keep it completely within the host language (i.e. like PJama) this makes it even more simple to use and learn to use.

Reuse Having the power of dynamic queries is another aspect, that seems important in interacting with a database. This makes it easier to extend and change existing queries.

Interface style Letting the system developer interact with the database in a natural way is preferable. This means that the extent of our changes to the host language must be homogeneous with the values and underlying thoughts.

Build process The build process should be kept as simple as possible. Adding new steps to the build process could influence the appearance of the language in a negative way. Simplicity seems to be a key aspect.

Optimization - criteria shipping and search Separating dynamic criteria from the query's static form, and implementing mechanisms to support navigational prefetching will be in focus.

2.3.3.2 Criteria not in focus

Optimization - bulk data manipulation Executing bulk data manipulation using selection predicates will not be in focus.

Schema evolution This is not in our scope. How is changes in our object model or in the database schema to be handled? There are different approaches to this question in the reviewed solutions but it does not have a straightforward answer. Although this is an important feature, we leave this is up to future work.

Concurrency Although any solution striving to solve the impedance mismatch surely should handle concurrency problems, we choose not to do so - for know.

Neither in the sense that between loading objects from persistent storage and storing them again, they might be altered, and the state of the objects loaded into the application no longer are identical.

Nor in the sense that the concurrency constructs in programming languages do not match the concurrency constructs in databases - and that having more than one executing thread in an environment with one or more connections to a database can give non-deterministic behavior.

However, providing support for concurrency is not in focus due to the limited scope of the project.

Tool support Once an idea has been designed and realized, it would be time to look at tool support. Tool support will therefore not be in focus in the project.

2.4 Problem statement

The goal of the project is to demonstrate how persistence using a relational database can be integrated into a statically typed object-oriented language, while keeping the following language design goals in mind:

Assimilation The support for persistence should be integrated in the host language in such a fashion that new language syntax and semantics conform with the paradigm of the host language, and that there are not introduced any new key abstractions.

Partial persistence The support for persistence should not be orthogonal, but rather require the system developer to decide which elements that are to be persisted.

Statically checked Queries to the database should be statically checked.

Minimal alteration The syntactic addition to the object-oriented host language should be as small as possible, to ease the acquisition of the required skills to use the language.

Performance Queries are to be passed to the database engine in such a fashion that the database engine is able to optimize queries.

2.4.1 Method

A solution to the impedance mismatch solution potentially consists of parts residing in different parts of an application environment. Some parts might be language modifications (syntax and semantics) and some might be parts of a new API.

While the ultimate goal of this project is to design a solution to the impedance mismatch problem satisfying all of the criteria found to be relevant, the ambition for this project is a little less ambitious.

During the course of the rest of this report, we wish to develop a foundation for a solution, exploring different solution approaches to different parts of a solution. Some of the work is done by investigating existing work, and some is supported by implementing prototypes. The implementation process is undertaken to encounter problems and expose hard-to-solve areas. The goal of the prototypes is therefore not to produce a usable solution, but rather to assist the design process.

Developing the solution, we divide the solution into different parts. The first part, identifying persistable types, is a necessity dictated by the partial persistence design goal from the problem statement. From this part, we proceed by designing the following parts:

Mapping the object-oriented model to the relational model When dealing with relational databases and object-oriented programming languages, we need to identify how mapping between the two abstraction models is handled.

Querying Some way of expressing queries that retrieve objects based on selection predicates

Data manipulating How are objects stored in the database? How are they deleted?

When developing each of the parts, we consider different approaches. The approaches either: Originate from existing solutions mentioned in Section 2.2, are described in existing literature, or are principal approaches to the solution.

Each of the approaches are evaluated either with regard to the criteria set forth in Chapter 2, or to the inherent properties of the approaches.

The foremost task however, is to reduce the scope of the problem.

Chapter 3

Designing a solution

3.1 Introduction

In this chapter a solution to a subset of the problems proposed in Section 2.4 is designed. The solution is an extension the programming language Java. The first section presents the method by which the solution is designed. Due to the limited time frame of the project, the scope of the problem is reduced (Section 3.2). Then we will have a look at how we can identify the persistable types (Section 3.3). After this, we will be looking at the mapping from objects to relations (Section 3.4) and querying (Section 3.5). Finally we will be discussing how we can manipulate data (Section 3.6).

3.2 Focus of the solution proposal

In this section we will be reducing scope of the problem defined in Section 2.4. We will be doing this by making some choices based on the focus of the project.

3.2.1 Extend an existing language, or build a new one?

Given that we want to solve the impedance mismatch problem for a statically typed object-oriented language, we might choose to either extend an existing language, or design a new language.

Although designing a new language would provide with the possibility to define all the semantics of object behavior upon creation, deletion, etc., the task of defining a whole language is to big a task given the scope of the project. Given that the focus of the project is to provide a solution to the impedance mismatch problem, and not present a better incarnation of an object-oriented language, it seems more relevant to focus on the parts concerned to persistence.

Both Java[23] and C#[12] seem to be good candidates as languages to extend. They are statically typed, well-documented, and widely used languages.

We have decided to use the programming language Java as the language, we want to extend. Java was the choice, because this is the language the project group has the most experience with. It could just as easily have been C#, but since no member of the team has any experience with C# we choose Java.

3.2.2 Working with legacy database schema's or not?

To reduce the scope of the solution, as a first approach, we do not work with legacy database schema's. Although it is indeed a valid aspect of the problem whether a solution is able to operate on a data schema that is defined separately from the given application, the starting point for our solution will be in a database schema that is generated from the type system (e.g. the classes) in the application.

This approach gives some advantages regarding simplifying the compilation process. Given that the database schema can be inferred from the type system, the actual database does not need to be present at compile time. Moreover, mapping issues between the object-oriented model and the relational model are simplified.

3.2.3 Naming the baby

Not wanting to keep the reader in undue suspense, we reveal the name of our language without further ado: *PersiJ* .

3.3 Identifying persistable types

Given that we identified that the solution should support partial persistence, there needs to be some way to distinguish non-persistable types from persistable types.

If the classes are marked as persistent through the original syntax of the language, it is possible to make runtime reflection as to whether they are persistent or not, and the compiler can verify that the marking is correct.

In the following we sketch different approaches, presenting advantages and disadvantages of each approach. Our examples are based on the class Model, which may extend some other class, as sketched in Figure 3.1.

```
1 //extends modifier optional (denoted by square brackets)
2 public class Model [extends SomeClass] {
3     // class body
4 }
```

Figure 3.1: A simple model class.

3.3.1 Inheriting from a superclass

The first, and maybe somewhat naive approach just extends a superclass common to all classes that are to be persistable. Figure 3.2 illustrates this.

```
1 public class Model extends PersistentRoot {
2     // class body
3 }
```

Figure 3.2: Extending a common superclass.

This approach, although simple, has several drawbacks: Since Java only allows a single class to be inherited, this approach makes it impossible for the Model class to inherit from another class. If there at some point arises a need for sub classing Model, the subclasses will also be persistent - which is not necessarily the intention. The semantics of inheriting to and from persistable classes is discussed in more detail in Section 3.4.2.1.

3.3.1.1 Semantics of persistent members when inheriting

In Java, the semantics of inheritance are well-defined. Introducing persistable classes, it becomes unclear how inheritance is handled.

3.3.2 Implementing an interface

Another approach is to make an interface - in this case `Persistable`. The class or classes that need to be persistent should then implement this interface, as Java allows for multiple interfaces. This is actually the same approach as Java uses with the `Serializable` interface (which marks the class as being able to be serialized - i.e. to disk). Figure 3.3 illustrates this idea which is also known as a *marker* interface[23].

```
1 public class Model implements Persistable {
2     // class body
3 }
```

Figure 3.3: Implementing an interface.

3.3.3 Using meta data

Another approach is to decorate the source code with some meta-information. This can be done by e.g. using describing files (maybe even using XML), or by using the Java 1.5 annotations[24].

By using external files to list the persistable types, the information is removed from the elements that it concerns. The information that is needed for the system developer to understand the semantics of the type, is split to several locations. In effect, it becomes more cumbersome for the system developer to understand the code or there must exist some tool to support this.

Java annotations places the information next to the class and type definition. Moreover, the information is present without breaking the syntax of Java, enabling compilation by a standard JavaC. Figure 3.4 shows this idea.

```
1 // annotations definition
2 public @interface Persistable {
3   ...
4 }
5
6 // class using Persistable annotation
7 public @Persistable class Person {
8   ...
9 }
```

Figure 3.4: Marking a type persistent with an annotation.

3.3.4 Introducing new syntax

The last approach is to introduce some new syntax to prescribe when a type is persistable. Figure 3.5 shows how this syntax might be. The placement of the new syntax element, the keyword `persistable`, is in the class declaration.

```
1 public class persistable Person {
2   // class body
3 }
```

Figure 3.5: Marking a type persistent with a new syntax.

3.3.5 Conclusion

Extending a common superclass is discarded because that inhibits persistable classes from inheriting from other classes given the single inheritance of Java. Using meta data represented in external files decreases the system developers readability.

We are now left with three alternatives. Implementing a common empty, interface (i.e. with no method signatures), use meta data represented in form of annotations or extending the existing syntax.

At this point of the design process, neither of the alternatives are chosen. Depending of the style of the rest of the design, either of the alternatives might be chosen, to achieve the most coherent solution.

3.4 Mapping from objects to relations

As stated previously, as a first approach PersiJ focuses on working with database schema's inferred from the object model. This means that mapping relational models to objects models is not covered for now. Moreover, since Java does not have multiple inheritance, only single inheritance is considered. Ambler[3] has written in some detail about techniques to solve problems relating to mapping objects to relations. In the following we present different approaches to the issues concerning the mapping are briefly described. If any of the approaches is favorable over the others, it is explained why so.

The basic (and somewhat naive approach) is: One object maps to one tuple, objects of the same class belong to the same relation, and object members become attributes in the relation. While this approach provides a good starting point, it does need to be altered when following issues are taken into account: How do we links objects with their persistable counterparts? (*shadow information*). Determining which object members to persist (*Persistable object members*). Dealing with *type mismatch*. Object members might be reference(s) to other object(s) (*modeling references*), inheritance (*modeling inheritance*).

3.4.1 Shadow information

No matter which approach is taken to map the objects to relations, some information needs to be present in the objects - namely the primary keys that identify the tuples they become in the RDBMS. This information is dubbed *"shadow information"*.

3.4.2 Persistable object members

Given that only a subset of the classes are persistable, naturally not all members of an object can be persisted. Only the members that belong to a type that is marked persistable (as discussed in Section 3.3) can be persisted. The state of other members is not persisted. Exceptions to this rule is the types in the `java.lang` package that wrap around the primitive types, and the type `String`.

3.4.2.1 Dealing with inheritance

How about inheritance? - Lets consider the different cases that may occur:

Non-persistable class inheriting from non-persistable class No change to Java 1.5 syntax and semantics.

Non-persistable class inheriting from persistable class The class inherits members as it would do if the super-class had been non-persistable. The class is not treated as a persistable class, and can not be neither retrieved nor inserted into persistent storage.

Persistable class inheriting from non-persistable class If any of the inherited members are of a persistable type, they are persisted. Other members that are of a type that is not persistable, are not persisted.

Persistable class inheriting from persistable class Any inherited members of a persistable type, become persistable.

For the two latter cases, the Section 3.4.5 describe in detail how the inheritance structure is mapped to the relational model.

3.4.3 Type mismatch

Java supports some primitive types, namely `char`, `int`, `boolean`, `byte`, `double`, `float`, `long`, and `short`. These do, to some degree map directly to RDBMS data types, and in general, object members that are primitive types, map to an attribute in a relation. The domain range of the primitive types vary from RDBMS to RDBMS. Until a concrete implementation of PersiJ is designed, it is not considered how to achieve optimal mapping of the primitive types.

Achieving an optimal mapping depends on several aspects:

- The conversion from the primitive type in Java to the primitive type in the database must be able to be performed in such a fashion that the whole domain range of the Java primitive can be represented in the database primitive. That is, there must be no loss of information.

- The receiving primitive type in the database must have as small a domain range as possible - resembling the Java type as closely as possible. As an example: If all the primitive types are mapped to Binary Large OBject (BLOB) variables in the database, the RDBMS can not use indexes on the variables, nor maintain knowledge about data topology, etc.

3.4.4 Modeling relationships

Relationships between objects are (normally) expressed either by an object member being a reference to another object, or the member being a collection of references to other objects. The relationships can be uni- or bi-directional depending on whether the relationship can be traversed just one way, or both ways.

Previously, we stated that only object members that belong to types that are marked persistable, should be persisted. Having a one-to-many relation is modeled using a collection of references to other objects. One solution could be recognizing

some of the standard collection types in Java (`ArrayList`, `Vector`, etc.) as reference collections, and model these as persistent relationships. Another solution is to provide a new type (implementing `java.lang.Collection`) that the system developer can use to model persistent relationships between classes. As we will see later, this approach enables some other desirable properties.

Figure 3.6 shows an example of (reflective) relationships - one-to-one, one-to-many and many-to-many, using a special collection type.

```
1  public class Person {
2      //one-to-one relationship
3      private Person father;
4
5      //one-to-many relationship
6      private PersistableCollection<Person> children;
7
8      //many-to-many relationship
9      private PersistableCollection<Person> siblings;
10 }
```

Figure 3.6: How references are normally expressed object-oriented.

Relationships can be modeled with two different strategies in the relational model - foreign keys and associative relations.

Foreign keys can be used to cross-reference relations. In one-to-one the foreign keys must be present in one of the two relations - which one does not matter. In one-to-many the key must be present in the *one* relation to the *many* relation. This way the foreign key will know which of the many objects it is related to.

Many-to-many can be implemented by adding more attributes to each relation thus ensuring that the mapping is preserved. This, on the other hand, is not optimal, as the many-to-many is in fact less than many - as we have defined the maximum number through the number of attributes added. Implementing many-to-many can instead be done by associative relations. The idea is to implement an extra relation between the many-to-many, so there arise a one-to-many on both sides of this new relation. This way, we can model one-to-many between the to relations using the extra new relation.

3.4.5 Modeling inheritance

Encountering inheritance, the initial approach of *one object equals one tuple* needs to be reconsidered. In the following, the classes of Figure 3.7 are used as common example. All of the strategies of mapping inheritance to relations, has one thing in common. The base class of all other classes (`Object` in Java) is not considered the base class, and instead the next classes in the global inheritance hierarchy becomes the roots of a number of inheritance hierarchies. This alteration of the

perception of the global inheritance hierarchy is done for several reasons: First, Javas `Object` class does not contain any members that are reasonable to persist. Second, if the whole type system is one global inheritance hierarchy, many of the modeling schemes are invalidated.

```
1  public abstract class Person {
2      private String name;
3  }
4
5  public class Customer extends Person {
6      private Vector<Preference> preferences;
7  }
8
9  public class Employee extends Person {
10     private Real salary;
11 }
12
13
14 // Class added later:
15 public class Executive extends Employee {
16     private Real bonus;
17 }
```

Figure 3.7: Classes with inheritance.

Ambler[3] suggests different approaches, as we will discuss.

Map hierarchy to a single relation Each hierarchy corresponds to one relation, with tuples for all the object members. If there are alterations to the class hierarchy, attributes are removed or added to the relation.

Map each concrete class to a relation Each object becomes one tuple in one relation. The classes in Figure 3.7 become a relation for `Customer` and `Employee` classes, with their inherited members appearing in each relation because `Person` is declared abstract - they must both contain the attribute `name`. An extension of the inheritance hierarchy with a new sub-class (like `Executive`) requires the addition of a relation. Other alterations to the hierarchy will require adding, altering or removing attributes in the affected relations.

Map each class to its own relation Each class corresponds to a relation. Each object then becomes one tuple in each relation from its class until the base class. The classes in Figure 3.7 become a relation for each class. Extending the inheritance hierarchy requires adding a relation, and other alterations require only altering one relation.

Generic mapping A number of relations contain tuples that describe the type system: the classes, their members, inheritance relations and attributes. Finally, there is one relation containing tuples with all the values. Altering the inheritance hierarchy requires updates to the describing relations. This scheme is quite flexible, but is disfavored compared to the former schemes. This scheme does not scale very well, and queries quickly become very ineffective compared with the former schemes.

Each of the strategies have their own strengths and weaknesses - varying with the most characteristic operations that some application might perform. Rather than choosing one over the other, it may be beneficial to the let the system developer indicate which strategy should be used.

3.5 Querying

This section concerns querying - how it is possible to obtain information from persistent storage based on selection predicates.

In the review of existing solutions in the previous chapter, we have encountered different mechanisms of specifying the query. Revisiting each of these approaches again, we consider if any of them are appropriate for PersiJ.

Since one of the design goals of PersiJ is having static checking, approaches based on CLI and special SQL-like query languages (i.e. HQL) represented as strings are not appropriate for PersiJ.

Cω implements static checking but breaks with the object-oriented paradigm. Queries are simply written in a variation of SQL. One of the proposed criteria concerns language alteration being minimal. Another concerns assimilation of the new solution into the host language. Since SQL is not object-oriented, and since it would be a quite voluminous expansion of the Java syntax, this approach is also disfavored.

In db4o, and other approaches, it is possible to programmatically build a query by adding constraints and other elements to some data structure representing a query. Although obeying the object-oriented paradigm and enabling static checking, these approaches typically suffer from being very verbose. They simply require too much typing from the system developer to achieve even simple queries.

As described earlier (Section 2.2.6) the design white-paper "Native queries for persistent objects" by William R. Cook and Carl Rosenberger[11][1] presents a way of expressing queries using the syntax and semantics of either Java or C#. While presenting how to write queries corresponding to SQL `SELECT` statements with `WHERE` clauses, the ability to express aggregated queries is not present. While the white-paper makes elegant use of delegates (anonymous methods) in C#, the

[1] As of August 2005, a project was launched on java.net by Wesley Biggs: Plain Old Java Queries (POJQ) aiming to implement Native Queries for native Java, and translating to Java Data Objects Query Language (JDOQL). At the time of writing, no significant headway had been made.

equivalent Java solution involves using anonymous classes in a much less elegant fashion.

The querying part of PersiJ is based on Native Queries, but including syntax alterations. In the following, we will proceed by starting with describing the Native Queries approach, and then describe how it can be altered to fit the requirements of PersiJ.

3.5.1 Native Queries

Figure 3.8 (from [11]) shows the abstract class `Predicate`, with the abstract method `match`. The query method being invoked in lines 7-15, takes an instance of the Predicate class as parameter. The instance is here provided as an anonymous implementation.

```
1  // abstract Predicate class
2  public abstract class Predicate <ExtentType> {
3      public <ExtentType> Predicate () {}
4      public abstract boolean match (ExtentType candidate);
5  }
6  //Invocation of a query using an anonymous specification of the Predicate class
7  List <Student> students = database.query <Student> (
8      new Predicate <Student> () {
9          public boolean match(Student student) {
10             return (
11                     student.getAge() < 20 &&
12                     student.getName().contains("f")
13                     );
14         }
15     });
```

Figure 3.8: The abstract query class, and an anonymous implementation of same.

The idea is that the match method is applied on every candidate object, and if the match method evaluates to true, the candidate object is included in the result set. The intention is that the compiler performs analysis of the match method, translating it to an expression native to the underlying database system. If the compilation is not able to translate the match method, it is always possible to fall back to a default behavior - instantiating every candidate object, and invoking the match method. However, the default behavior is of course undesirable due to the high performance penalty.

3.5.2 Fitting Native Queries to PersiJ

Although Native Queries provides a way of expressing selection predicates through the syntax and semantics of the host language, the required implementation of the

abstract method is quite inelegant.

The essence of the Native Queries approach is the implementation of the match method, and this is the part we want to keep. The `db.query` requires some `db` object, and as mentioned, the anonymous implementation of the abstract `Predicate` class is quite verbose.

Since there is a lack of anonymous methods in Java [2] , we choose to extend the syntax of Java with a new language construct.

3.5.3 New language construct: Existing

The language construction must be able to express the selection predicate, and indicate whether or not the candidate is to be included in the result set. Figure 3.9 shows an example of this syntax construction.

```
1 PersistableCollection<Person> results = existing(Person p) {
2     include p;
3 }
```

Figure 3.9: The existing language extension

The `existing` keyword indicates that the following block contains a retrieval of existing objects from persistent storage. Within the block, the system developer should be able to express the predicate used for selection. Just like the match method takes the candidate object as parameter, the existing block does the same. By letting the identifier be specified in a "formal parameter list"-like way, the type of the objects that are to be retrieved is also specified.

The basic semantics (as observed by the system developer) is that the code within the existing block is evaluated once for each candidate object. As a first approach, the selection predicate can be expressed with all the language elements of Java.

3.5.3.1 Include keyword

Within the block, the system developer has to indicate whether or not the current candidate object is to be included in the result set. A first approach might be to use the `return` keyword, returning either true/false or the candidate object identifier/null. In Java in general however, `return` indicates that the execution flow exits from the current scope, which is not the behavior of the existing block. Therefore we opt to introduce a new keyword - `include`.

[2] C# has anonymous methods, called delegates

3.5.3.2 Informal semantics

Developing the example a bit further, Figure 3.10 shows an existing block using a selection predicate. The intuitive meaning of the example is that if the name of a candidate object is "Peter", the object should be included in the result set. To be able to translate this to SQL, we will try to informally describe the semantics of the existing block.

```
1 PersistableCollection<Person> = existing(Person p) {
2     if(p.getName().equals("Peter")) {
3         include p;
4     }
5 }
```

Figure 3.10: An existing block with a selection predicate.

Comparing primitives A comparison of primitives can be translated directly to a corresponding part of a `WHERE` clause in SQL.

Method invocations The `equals()` is a method invocation, with the intention of determining whether the object having the method invoked, and the object passed in the actual parameter are equal. For some types, this behavior can be translated directly to an equality comparison. This direct translation applies to the encapsulating primitive types (`Integer`, `Double`, `Boolean`, etc) and to the `String` type.

For an invocation of `equals()` or any other method on all other types, the methods have to be investigated in a recursive manner, ending with comparison operations on primitive members.

During the recursive investigation, a traversal of a reference to another persistable type leads to a join in the SQL statement.

There are of course some limitations. All the references that appear on the traversal, must be of a persistable type. If they are not of a persistable type, they have no persistent representation.

Identifiers from enclosing scope In Figure 3.10, the parameter to the query is expressed statically. One of the goals of PersiJ is however to enable dynamic parameters. An extension of the existing block therefore becomes that identifiers from the enclosing scope of the existing block become in-scope for the existing block, so the code in Figure 3.11 is possible.

The operations that concern the identifiers that are not dereferenced in some manner by the candidate object, can be evaluated once, resulting in primitive values that can be used as parameters to the SQL query.

```
1  public PersistableCollection<Person>
2                          someOrdinaryMethod(String name, Integer minimumAge) {
3      // the identifiers name and minimumAge are in scope for the method, and
4      // therefore referenceable from whithin the existing block
5      PersistableCollection<Person> matchingPeople = existing(Person p) {
6          if(p.getAge() > minimumAge && p.getName().equals(name)) {
7              include p;
8          }
9      }
10     return matchingPeople;
11 }
```

Figure 3.11: Existing block using external identifier.

3.5.3.3 Return type

The return type of the existing block has to be some kind of collection that contains references to the objects that satisfy the selection predicate. The `PersistableCollection` type may well hold the references. Using a (to PersiJ) custom collections type also enables navigational prefetching, letting the `PersistableCollection` dynamically determine when to load objects from persistent storage.

3.5.3.4 Limitations

Some limitations to the expressiveness of the code within the existing block does exist. One of them is that the methods that are invoked must be free of side effects. If they have side effects, the behavior is undefined. Another is the issue of non-termination. Since iteration constructs (`for`, `while`) are allowed, they can express non-termination - which can not be expressed in SQL.

3.5.4 New language construct: Constructed

One of the powers of SQL and shortcomings of some of the reviewed solutions is the ability to use aggregates. Take the following simple example. In Appendix A we have a table called *person*. If we just need to see how many persons that is contained within that table, we could use the `existing` block to select all persons to a list, and thereafter counting them from within the programming language. From an optimization point of view, this is highly inefficient.

In a system not using SQL you would simply use the SQL `COUNT(x)`, where *x* is the element you want counted. By doing this you are creating new data, not directly present in the database.

Expressing aggregate queries should be possible in PersiJ. The following describes an idea for expressing aggregated queries that is still in a very early stage.

The language is expanded with one more language construct - similar to the existing block. The code within the block is translated to SQL, Figure 3.12 gives an example. The keyword chosen to denote the block, is `constructed`, the intuitive meaning being that the value returned from the block is "constructed" from existing data. The example shows how a count of the Person objects that are persisted is obtained.

```
1 Integer numPersons = constructed<Integer> {
2   PersistableCollection<Person> allPeople = existing(Person p) {
3     include p;
4   }
5   return allPeople.size();
6 }
```

Figure 3.12: Example of the syntax for the constructed block.

The design of this language construct is still in a very preliminary state, and apart from an idea about being able to have some methods on `PersistableCollection` that are recognized as expressing `min`, `max`, `count` and so forth, the specifications remain vague at the time of writing.

3.5.5 Discussion

The discussed language extensions for expressing queries are still in a preliminary stage. Much work needs to be done regarding formalizing grammars and formal semantics. Although they are still at this quite raw stage, they do possess some quite desirable properties: The syntax used to express selection predicates is not very verbose. Moreover, it is object-oriented, and very close to the semantics of Java. A major disadvantage is however that since the `existing` and `constructed` blocks are not methods on a class or an object, they break with the object-oriented paradigm. Apart from formally defining semantics and grammars, it is still an open question whether the expressiveness of the selection predicates is equal to its counterpart in SQL.

3.6 Manipulating data

This section concerns how inserting, updating and deleting objects from the persistent storage can be expressed in PersiJ. In common we denote these operations as manipulating data.

3.6.1 Implicit or explicit management?

A choice has to be made as to whether inserting, updating and deleting objects from persistent storage is to be transparent for the system developer, or whether it

requires explicit interaction from the system developer.

As a first approach, we consider the transparent option. We quickly recognize that there are several problems that arise as a consequence of this choice.

3.6.2 Transparent manipulation

Transparent manipulation is the same approach as the orthogonally persistent system PJama (see Section 2.2.3) - to the system developer there is no difference between the memory-management model, and the way persistent objects are treated regarding when they are persisted, and when they are deleted.

In this case, inserting and updating is solely managed by the run-time environment. This means that when you as system developer change an object you cannot tell the application when these changes should be made persistent. There are several issues to be considered in this scenario.

When are objects persisted? - How does the run-time environment guarantee that instances of persistable types are persisted upon application termination?

Deleting works in a way similar to garbage collection of in-memory objects. If all references to a persistent object cease to exist, the object is also removed from the persistent storage. Figure 3.13 shows an example where a collection of objects is deleted from the database. That is, if there are no other references to the objects.

```
1 // Retrieving all persons called Peter
2 PersistableCollection<Person> persons = existing(Person p) {
3   if(p.getFName().equals("Peter") {
4     include p;
5   }
6 }
7
8 // Deleting all persons called Peter
9 persons = null;
```

Figure 3.13: Deleting a group of persistent objects by nulling all references to them.

Deleting an object is not necessarily an easy task. Moreover, as long as there are references to the persistent objects in memory, the objects themselves remain in memory. While this approach may work for small (or maybe even larger applications) that terminate frequently, applications that are not necessarily designed for termination might face having to retain an ever-increasing number of objects in memory.

The transparent approach means that the whole run-time environment (in the case of Java, the JVM) needs to be altered to implement the new semantics of persistent object manipulation. Altering the JVM does not lie within the limited scope of this project - and even if it did, the PJama project showed us that this is a

task that is not easy.

3.6.3 Explicit manipulation

Providing the system developer the means of explicitly determining when to manipulate data can be done in several ways. In the following we will start by determining which operations that are needed to be able to manipulate objects. Then we will outline some approaches to providing these operations, and determine which of them is most viable.

As stated in Section 2.3, the scope of the current project does not include enabling bulk data manipulation based on selection predicates.

3.6.3.1 Necessary operations

SQL provides the operations `INSERT`, `UPDATE` and `DELETE`. Before proceeding with determining how these will map to explicit operations in the programming language, the semantics of the operations is shortly described:

`INSERT` inserts a new tuple in a relation - generating an error if unique constraints on the relation are violated.

`UPDATE` updates a relation, assigning the specified values tuples to all the tuples satisfying a selection predicate - enabling bulk data manipulation.

`DELETE` deletes tuples from a relation. The tuples to delete are specified with a selection predicate.

Supporting the `DELETE` operation is straightforward when the selection predicate is based only on the primary keys of the tuples. As mentioned in Section 3.4, the primary keys are stored in the persistent objects as shadow information. Some way of indicating the references to the objects that are to be deleted, is therefore the straightforward mapping. For future reference, we call this operation *unPersist*.

Considering `UPDATE` and `INSERT`: Taking an alternate view than the SQL operations, one might consider the typical situations when a system developer wants to store an object. Given a persistable object, the system developer can have two intentions:

1. Saving the object as-is, generating new tuples in the database if it has not been persisted previously, or updating the existing tuples if the object has been persisted previously. For future reference, we call this operation *persist*.

2. Regardless of whether the object has been persisted previously, store a new instance of the object in the database, inserting new tuples as needed. For future reference, we call this operation *persistAsNew*.

3.6.3.2 Session object

Similar to Hibernate, one approach could be to have a session object, encapsulating a session with a database. The operations could then be methods on the objects, taking as parameter either a single reference to an object, or an `Iterable` type containing references to the objects to perform the operation on. Figure 3.14 shows an example of how this might be.

```
1  public void someMethod(Person p, Vector<Person> persons, Session s) {
2      // persisting:
3      s.persist(p);
4      s.persist(persons);
5
6      // persisting as new:
7      s.persistAsNew(p);
8      s.persistAsNew(persons);
9
10     //deleting:
11     s.unPersist(p);
12     s.unPersist(persons);
13 }
```

Figure 3.14: Manipulation operations on a session object.

3.6.3.3 Common methods on persistable objects

Another approach would be to let all persistable objects have three common methods implementing the manipulating operations. Figure 3.15 shows an example.

In favor of this approach compared with the session object, is that to perform the operations, there does not need to be other objects than the ones that are to be manipulated.

However, this approach presents some challenges. To the system developer it is not apparent where the implementation of the operations reside. While the methods might be injected by a pre-compiler, the implementation of the methods is not available at compile time. If the persistable classes are indicated using a marker interface, the method signatures could be included in the interface. Unfortunately, the interface does not make it clear to the system developer where the implementations come from since it is empty.

3.6.3.4 Using the references type

The references type introduced in Section 3.4.4 might also be used. The operations would then be methods on the collection type. The methods could then be overloaded, to accept either no arguments (the semantics then being that the operation

```
1  public void someMethod(Person p, Vector<Person> persons) {
2      // persisting:
3      p.persist();
4      for(Person tmpP : persons) {
5          tmpP.persist();
6      }
7
8      // persisting as new:
9      p.persistAsNew();
10     for(Person tmpP : persons) {
11         tmpP.persistAsNew();
12     }
13
14     //deleting:
15     p.unPersist();
16     for(Person tmpP : persons) {
17         tmpP.unPersist();
18     }
19 }
```

Figure 3.15: Manipulation operations directly on persistable classes.

be performed on all elements of the collection) or to accept one reference as parameter - performing the operation on the referenced object (if the object is present in the collection). An example is shown in Figure 3.16.

```
1  public void someMethod(Person p, PersistableCollection<Person> persons) {
2      // persisting:
3      p.persist();
4      persons.persist();
5
6      // persisting as new:
7      p.persistAsNew();
8      persons.persistAsNew();
9
10     //deleting:
11     p.unPersist();
12     persons.unPersist();
13 }
```

Figure 3.16: Manipulation operations on the collections type.

Disadvantages of this approach is the increased complexity of the collections type, and that the collections object that contains the reference(s) that need to be manipulated has to be present to be able to perform the manipulation.

There are several advantages. The syntax is less verbose than the two preceding approaches. Using the references collection it is also able to ship `unPersist` operations as bulk data manipulation to the RDBMS. It is not a mystery to the system developer where the implementation comes from.

3.6.4 Conclusion

Each of the three latter approaches may be viable. However we are inclined towards the last approach - using the references type to support data manipulation, due to the fact that there is no need to inject new code in code maintained by the system developer. Moreover, the smaller an API required to operate PersiJ, the easier it is for the system developer to use the framework. If the session-style object can be avoided all together, the API required may be smaller.

3.7 Summary

Throughout this chapter, alternatives to foundations for a solution to the impedance mismatch problem have been discussed. This section is a summary of the conclusion of each of these discussions:

Identifying persistable types Three viable alternatives to mark up persistable types are identified: Using a marker interface, using annotations, or introducing a new class modifier - modifying the syntax of Java.

Mapping objects to relations There are different aspects of the mapping - which object members to persist, how to model inheritance, and how to model relations. Some information that need not be seen by the system developer is needed to relate the objects to their counterpart tuples - shadow information. Several viable schemes of modeling inheritance are described, and it might be beneficial to let the system developer indicate which scheme is appropriate. The discussion of how to model references between objects leads to the introduction of a special collections class that can handle the references - `PersistableCollection`. The mapping of references to the relational model is achieved by using foreign keys and associative relations.

Querying Discussing existing approaches compared with criteria from the analysis leads to the introduction of two new language constructs. The `existing` block is used to retrieve objects of a persistable type from the database, based on selection predicates expressed within the block. The syntax and semantics of the contents of the block has a close resemblance to ordinary Java. Criteria shipping is achieved by letting the scope of the context of the block be accessible from within the block. The other language construct, the `constructed` block, is used to obtain aggregated queries. Like the `existing` block, the syntax and semantics bare a close resemblance

to Java. The return type of the blocks is the `PersistableCollection`, and it is speculated that the class can be used to implement some scheme of navigational prefetching.

Data manipulation Discarding the use of run-time environment managed persistence, we proceed by identifying three operations that need be provided to support data manipulation. Three viable ways of exposing these operations are discussed, but simplicity of the provided API, less verbosity and the ability to perform the operations on many objects at once leads to the choice of using the class `PersistableCollection` to expose the operations.

Thus, a foundation for PersiJ is laid out.

Chapter 4

Implementing a compiler prototype

4.1 Introduction

The purpose of the prototype is to render further development and implementation of the compiler probable. We seek to find new non-trivial aspects of the compiler implentation and thier solution. In our case this is the implentation of the persistence functionality of our language. An area that seem difficult or at least require further investigation is the mapping between the Java-like object-oriented constructs to SQL that is used by the database. Congruence between the semantics of the querying constructs and the relational world is especially interesting, and will be the main focus of the prototype.

The rest of this chapter will be organized as follows. First we will give a description of our thoughts regarding the compilation process (Section 4.2). Then we proceed to describe the actual implentation of the *existing* construct (Section 4.3). Finally we summerize the chapter with a description of lessons learned with the prototype (Section 4.4).

4.2 Envisioning the compilation process

With this section we would like to demonstrate our thoughts on the compilation process. The implementation of the prototype compiler could be approached in several ways. We could write a compiler from scratch or implement a pre-compiler utilizing the already existing JavaC. We have chosen to implement the prototype compiler as a pre-compiler as this has some obvious advantages. With this approach we can concentrate on implementing our language construct, and leave the rest of the Java compilation, to the JavaC.

The compilation process of the pre-compiler is done in four steps. First we scan through the source code, recognizing every language constructs that is introduced by PersiJ. These are replaced by Java dummy code. This step also handles

checking validity of the syntax.

Secondly we compile the Java code, that is now free from PersiJ code, with the JavaC. The now compiled classes can be loaded and provide reflective access to the type system. Furthermore semantic verification of the rest of the Java code is assured.

Next we translate the PersiJ code to Java code. In this step semantic verification of the PersiJ blocks is performed. Last the PersiJ code now compiled into Java code is injected into the code we had as starting point and then it is compiled using the JavaC.

Steps in the compilation process are transparent for the system developer. The output, if no errors are encountered, is the compiled Java byte code. In Section 4.2.6 we will briefly describe how optimization can be performed before compiling the final Java byte code.

4.2.1 Steps in the pre-compiler

There are a few steps in the pre-compiler. They are summarized in the following:

- Parse through the code and do a syntactical check. Extract the PersiJ code and inject compilable Java dummy code.
- Compile the Java source code (without PersiJ code) - because this gives us the Java type system.
- Once the type system is present it is time to translate the PersiJ code. The code is first validated from the type system (syntactical check), and then translated to valid Java code and injected into the source files again.
- The source code is now containing only valid Java code and can be compiled with JavaC.

The translation of PersiJ code to SQL has three main steps, verification as described in Section 4.2.3, code generation as described in Section 4.2.4 and compiling as described in Section 4.2.5.

4.2.2 Identifying PersiJ code and type system generation

The first task is to identify the places in the source code where PersiJ code is present. Once identified, it must be replaced with temporary dummy code. The idea is, as just mentioned, to create intermediate source code that can be compiled by JavaC.

If the code fails to compile, the error must lie someplace outside of the PersiJ code, as we guarantee that the injected code can compile. If no errors exist, the process continues.

All classes are then compiled with JavaC - thus giving us reflective access to the type system of our program.

4.2.3 Syntactic and semantic verification

Now that all classes have been compiled, we can use the information present in the type system to verify the PersiJ code. Tasks of the verification includes identifying the `persistable` classes and the blocks containing `existing` and `constructed`.

4.2.3.1 Persistable

When the keyword `persistable` is encountered, the class in which it is present, need to be verified before moving on. This verification consist of syntactic and contextual analysis as with any other Java compilation. Also we need to tag the class persistable (i.e. introducing class members, annotations, etc.) thus enabling us to remove the `persistable` modifier, rendering the class to be valid Java 1.5 syntax. If the verification does not reveal any flaws or errors, the creation of the database schema is possible. Otherwise the compilation process is terminated and an error code is printed to the user.

4.2.3.2 Existing and constructed

Both the `existing` and `constructed` blocks are verified in a similar fashion, since their basic construction are alike. We need to perform syntactic verification on the `existing` and `constructed` blocks. The syntax within `existing` and `constructed` blocks is a subset of the Java 1.5 syntax, thus leaving us the responsibility to check it's validity.

Another aspect of verifying the two blocks, is to check whether all classes that is to be used in the database interaction is marked persistable. If they are not, we cannot guarantee that they are persisted in the database and therefore should halt the compilation process. This is where we need the type system, as mentioned in Section 4.2.1 - that is, we need to check the types in the blocks with the compiled persistable classes.

All the variables sent to an `existing` or `constructed` block are evaluated when the block scope is opened. This is also known as criteria shipping (see Section 2.1.6.1).

If the verification completes without errors, it is possible to generate the database connection and the SQL statements needed to replace the blocks.

4.2.3.3 Data manipulation constructs

In Chapter 3 we decided that the system developer explicit should state when he needs to (un)persist an object. Here we use two methods - `persist()` to persist the object in the database (overwriting if the object is already present) and `persistAsNew()` (always creating a new object). When deleting an object - new or old - the method used is `unPersist()`.

Data manipulating constructs are bound to the `PersistableCollection` described in Section 3.5.3.3.

4.2.4 Code generation

After the verification is completed, it is time to generate code. First the `persistable` classes should be used to generate the database schema and then the `existing` and `constructed` blocks should be injected into Java code again replacing the previous injected dummy code.

4.2.4.1 Persistable

Classes marked persistable, should be mapped to the database. So besides having to generate usual Java byte code, a mechanism to transform the structure of the class into a form that is usable in the database is needed. We refer to this as mapping objects to relations and this is described in Section 3.4.

When creating the database, all tables and attributes are made with lower case. This is done to make it more simplified when transforming the PersiJ source code such as existing and constructed to usable SQL statements.

4.2.4.2 Existing

Once the information in the persistable classes has been translated into a database schema, it is time to look at the `existing` block. This task is a bit more complex than with the persistable classes. First, we need to understand the semantics of the code contained within the `existing` block. For instance, take a look at Figure 4.1.

```
1 PersistableCollection<Person> persons = existing(Person p){
2   if(p.getFName().equals("Peter"){
3     include p;
4   }
5 }
```

Figure 4.1: Example of the existing block.

In order to generate valid Java and SQL, we must first analyze each part of the existing block.

The first thing the compiler encounters is the reserved word `if`. The `if` statement needs to be analyzed - and in this case it can be done quite easily, as there only exists one predicate: `p.getFName().equals("Peter")`. Translating this to words we would say the *we need all persons called "Peter"*. The *person* comes from the class definition in the existing block (see Section 4.3 for more information). The SQL statement we are looking for is `SELECT * FROM person WHERE fname = "Peter"` (notice that person and fname are all in lower case).

Figure 4.2 shows a special case, where the `if` statement is omitted. The `if` statement is allowed absent if we are querying all objects of a certain type.

```
1 PersistableCollection<Person> persons = existing(Person p){
2   include p;
3 }
```

Figure 4.2: Example of existing block without if-statement.

Translated to English this would give us: *we need all persons*. This is a more simple query than the one before, because the no predicate is present - we are simply retrieving all objects from the database. In SQL we write `SELECT * FROM person`.

4.2.4.3 Constructed

Although the constructed and existing blocks may seem alike, the constructed block is treated a bit differently. This is because with the constructed block we are *constructing* new elements which are not present in the database - or at least not directly. For instance, take a look at Figure 4.3.

```
1 Integer numPersons = constructed<Integer> {
2     PersistableCollection<Person> persons = existing(Person p) {
3         if(p.getFName().equals("Peter")){
4             include p;
5         }
6     }
7     return p.size();
8 }
```

Figure 4.3: Example of the constructed block.

As you might notice there is a existing block present inside the constructed. The idea is that we can reuse the basic ideas from the existing block, and add the power of aggregates (see Section 3.5.4). Explained in plain English the idea in Figure 4.3 would be: *we need to count the number of persons in the database called "Peter"* thus the SQL statement we are looking for is `SELECT count(fname) FROM person WHERE fname = "Peter"`.

The difference between the constructed and existing block is the `return p.size()` (in this case). In order to catch this we need to analyze the entire block before constructing the SQL statements.

4.2.4.4 Data manipulation constructs

Data manipulating features are bound to the `PersistableCollection` (Section 3.5.3.3).

4.2.5 Compiling

When the verification process is completed, the creation of the database schema and all necessary SQL statements is possible. SQL statements is to be injected in their respective places in the source code.

Afterward it is time to compile the entire application. The compilation can, once more, be done with JavaC. Recall that all classes in the project at this state is regular Java classes. It is unlikely that any errors occurs at this state, as the code (regular Java code) was compiled once without problems. The code injected by the PersiJC is compilable thus should not give any problems.

4.2.6 Optimization

The produced code from the pre-compiler could be optimized through manual optimization of the generated SQL statements. A way of enabling this type of optimization, could be to allow interfering with the compilation process. If the compiler could be called with a flag, that prevented the last step of compiling Java code with the injected SQL code, the system developer could force the output of the PersiJ compiler to be a Java source code. Manual alterations could then be made to the SQL statements, though this is not suggestible, as one loses the advantage of the checking that the PersiJ compiler offers.

4.3 Actual implementation

As mentioned previously in this chapter, a problematic aspect of the compilation process, is the translation of PersiJ code to SQL statements and Java. With this in mind, we have implemented a prototype pre-compiler that is able to recognize a small part of the `existing` block construct. The output of this translation is only a SQL statement and in Section 4.3.2 we have an example of this output. But first, in Section 4.3.1, we present the grammar for the existing block which is the grammar that the actual prototype can recognize.

4.3.1 Grammar

The following paragraphs will describe how to read the grammar. A *?* means that the element should be present zero or one time. Reading a * means that the element is present zero to many times and finally when reading a + the element should be present at least once but can be many.

```
<ExistingBlock>             ::=  existing ( <ObjectDeclaration> )
                                 { (<IfStatement> | <IncludeStatement>) }
<ObjectDeclaration>         ::=  <ClassName> <identifier>
<IfStatement>               ::=  if ( <Expression> )
                                 { <IncludeStatement> }
<IncludeStatement>          ::=  include <identifier> ;
<Expression>                ::=  <ConditionalOrExpression>
<ConditionalOrExpression>   ::=  <ConditionalAndExpression>
                                 ( || <ConditionalAndExpression> )*
<ConditionalAndExpression>  ::=  <RelationalExpression>
                                 ( && <RelationalExpression> )*
<RelationalExpression>      ::=  <PrimaryExpression>
                                 (<RelationalOperator>
                                 <PrimaryExpression>)?
<PrimaryExpression>         ::=  [!]  (<MethodCall> | <FieldName>
                                 | <ExpresionParenthical>)
<ExpresionParenthical>      ::=  ( <Expression> )
<FieldName>                 ::=  <identifier> ( .  <identifier> )+
<MethodCall>                ::=  <identifier> ( .  <Method> )+
<Method>                    ::=  <MethodName> ( (<Argument>)?  )
<MethodName>                ::=  <identifier>
<Argument>                  ::=  <String> | <Integer>
<RelationalOperator>        ::=  != | == | <
                                 | > | <= | >=
<Bool>                      ::=  true | false
<ClassName>                 ::=  <identifier>
<String>                    ::=  " (<digit> | <char>)* "
<identifier>                ::=  <digit> ((<digit>)+ | (<char>)+)*
<digit>                     ::=  [0-9]
<char>                      ::=  [a-z, A-Z]
```

Table 4.1: Grammar for existing block.

Reserved words (i.e. `existing` and `include`) are written in bold font face. The complete grammar is presented in Extended Backus Naur Form (EBNF) in Table 4.1.

4.3.2 Example code

We will now present the PersiJ code that our compiler actually can translate into SQL. For instance, it is possible to retrieve all elements from the database. It is possible to recognized the code in Figure 4.4 Notice that PersiJC in its current form cannot return anything. The output from this piece of code will be `SELECT * FROM person`.

```
1 existing(Person p) {
2   include p;
3 }
```

Figure 4.4: Existing block example: Select whole table.

This example is not all that exciting - on the other hand, when we add a predicate to the block it becomes more interesting. For instance, we could *get all per-*

sons called "Peter". Figure 4.5 shows this. The SQL generated here is `SELECT * FROM person WHERE fname = "Peter".`

```
1 existing(Person p) {
2   if{p.getFName().equals("Peter") {
3     include p;
4   }
5 }
```

Figure 4.5: Existing block example: Select all persons called "Peter".

It is also possible to use the *and/or* construction. Figure 4.6 shows this idea with an *and* (&&). This would produce the SQL statement `SELECT * FROM person WHERE (fname = "Peter" AND lname = "Madsen").`

```
1 existing(Person p){
2   if((p.getFName().equals("Peter") && p.getLName().equals("Madsen"))){
3     include p;
4   }
5 }
```

Figure 4.6: Existing block example: Select with where clause

It is also possible to extend this with parenthesis - as many as you like. It is also possible to have an arbitrary number of and/or after each other. Figure 4.7 shows this example. Here it is possible to see that we have used many parenthesis and even a *not* (line 3). This means that is is possible to create a negated boolean expression. The SQL statement generated here is `SELECT * FROM person WHERE (fname = "Peter" AND lname = "Madsen") OR (...) OR (((fname <> "Rolf")))`

```
1 existing(Person p){
2   if(((p.getFName().equals("Peter") && p.getLName().equals("Madsen")))
3     || (...) || (((!p.getFName().equals("Rolf"))))){
4       include p;
5   }
6 }
```

Figure 4.7: Existing block example: Arbitrary number of predicates.

4.4 Discussion

In the beginning of this chapter we stated that we would implement a prototype. The function of this prototype was to see how trivial the translation of PersiJ code into SQL. This turned out to be a non-trivial translation - and the compiler cannot even do anything useful other than the translation.

The prototype has demonstrated that evaluation of method calls to SQL is not as easy as we have laid it out in this chapter. We have in this chapter translated method calls to English, and from English to SQL - which is straight forward.

Our prototype only supports the `equals()` method on strings. This means that boolean operators such as == and != have not been implemented. Also, if the `equals()` is not side-free (a system developer has changed the function of this) we cannot do anything, as our interpretation of equals is as the same as the Java specification - comparing two strings.

We argue that at some point, the equality testing within the `equals()` method will consist of primitive types - strings included - compared with other primitive types. If the compiler can understand the semantics of the method and identify the compared primitives, we will be able to map these to the `WHERE` clause in the SQL statement.

Chapter 5

Conclusion

5.1 Conclusion

The analysis of the impedance mismatch problem led to a listing of a number of criteria to evaluate attempts to solve these problems. Though there are many attempts at solutions, all seem to be deficient in some way or another with regard to the proposed criteria. By prioritizing these criteria, a problem statement was formulated, giving some design goals for a new solution to the impedance mismatch problem.

Using these design goals as a foundation for the design process, a sketch of a new solution to the impedance mismatch problem (PersiJ) has been evolved. The solution is only sketched, with many parts still having several potential forms. This is intentional - the purpose being to asses that the design goals could be met.
But how does the proposed solution match with the design goals?

Assimilation The proposed mechanisms for identifying persistable classes, mapping data and manipulating data all fit with the object-oriented paradigm. While the new language constructs for querying do not conform with the object-oriented paradigm, the syntax and semantics used to express the predicates used to evaluate the selection, uses the syntax of the host language, and a semantics that bears close resemblance.

Partial persistence PersiJ allows for indication of persistable types, distinguishing these from the rest.

Statically checked Expressing predicates using a syntax and semantics that is part of the language enables static checking to be performed using a preprocessor.

Minimal alteration A total of three new keywords have been introduced and one new data type.

Performance Due to the fact that the `existing` and `constructed` blocks intentionally are compiled to CLI code using JDBC, the queries are expressed

in SQL, and the database is at liberty to optimize. The compilation from `existing` and `constructed` has only been investigated to a very rudimentary stage, and it remains to provide a final proof-of-concept. Whether the `existing` and `constructed` blocks can be composed and the resulting queries shipped as one to the database, remains unanswered.

We believe that PersiJ provides viable foundations for further development of a solution to the impedance mismatch problem showing some interesting characteristics.

5.2 Future work

The implementation of a rudimentary prototype of a compiler for PersiJ has been commenced to uncover problematic areas. This process has showed that the compilation of the code within the querying constructs as far from trivial. More work needs to be done, specifying a formal semantics of the `existing` and `constructed` blocks to be able to proceed with the development of a working compiler.

Although PersiJ is in conformance with the design goals, much work has to be done before PersiJ is a working solution. The envisioned compilation process needs to be implemented.

Some language design work also remains. Although concurrency was disregarded during the design of PersiJ, concurrent programming is a requirement if PersiJ is to become a solution fitted for real-world problem solving. One of the biggest challenges is to be able to guarantee ACID properties using the concurrency constructs of programming languages. Concurrency constructs in programming languages is an open question, and maybe recent work with Concurrent Haskell by Tom Harris *et al*[18] on transactional memory could prove useful.

New to Java 1.5 is (among other things) annotations. These could be used in the persistable classes as described in Section 3.3.3. An addition to the example Figure 3.4 could be to use annotations to give more meaning to the classes. This meta data could then be used in the translation from classes to database schemas. An example showing this additional information is shown in Figure 5.1. Although we have not been focusing on annotations, they could prove to be a powerful tool. This is at least what the team behind Hibernate think. They are currently working on implementing annotations into their framework thus eliminating the need for the XML mapping files[19]. The use of annotations could maybe even be taken further - using the Annotations Processing Tool[1] to mark up query blocks.

We have chosen not to focus on legacy database schemas. Perhaps we could use some ideas as those Hibernate is using. Recall from Section 2.2.2 that Hibernate can generate the mapping files from the database. Perhaps we could use the database to generate the skeleton of a class - then only leaving the implementation to the system developer.

Schema evolution during the development phase and subsequent maintenance

```
1  public @Persistable class Person {
2      private String fname;
3      private String lname;
4
5      @unique
6      private String socialSecurityNumber;
7
8      @onDeleteCasade
9      private Department department;
10
11     @onDeleteRestrict
12     private PersistableCollection<Project> project;
13
14     // getter and setters
15 }
```

Figure 5.1: Example of how annotations could be used.

of an application is hardly a rare event. Some mechanism of supporting schema evolution may prove interesting work.

Although we have looked a bit at navigational prefetching, this could be studied in greater depth. As Philip A. Bernstein *et al*[6] states, then navigational prefetching is an area of great importance, because this is a place where performance can be gained.

According to William R. Cook and Ali H. Ibrahim[9], bulk manipulation is not efficient in neither OR nor Java-based persistent programming languages. Therefore, looking to how PersiJ could support bulk manipulations could also be an area of further investigation.

Bibliography

[1] Annotations processing tool. http://java.sun.com/j2se/1.5.0/docs/guide/apt/index.html.

[2] ABRAHAM SILBERSCHATZ, H. F. K., AND SUDARSHAN, S. *Database System Concepts*, fourth edition ed. McGraw-Hill, 2002.

[3] AMBLER, S. W. *Agile Database Techniques: Effective Strategies for the Agile Software Developer*. Wiley, John & Sons, Incorporated, 2003.

[4] ATKINSON, M., AND JORDAN, M. Orthogonal persistence for the java platform - draft specification, jun 24 1999.

[5] ATKINSON, M. P. Persistence and java - a balancing act. In *Proceedings of the International Symposium on Objects and Databases* (London, UK, 2001), Springer-Verlag, pp. 1–31.

[6] BERNSTEIN, P. A., PAL, S., AND SHUTT, D. Context-based prefetch for implementing objects on relations. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases* (San Francisco, CA, USA, 1999), Morgan Kaufmann Publishers Inc., pp. 327–338.

[7] BIGGS, W. Plain old java queries. https://pojq.dev.java.net.

[8] The castor project. http://www.castor.org/.

[9] COOK, W. R., AND IBRAHIM, A. H. Integrating programming languages & databases: What's the problem? Submitted for publication may 2005, accessed November 2005 at http://www.cs.utexas.edu/users/wcook/Drafts/2005/PLDBProblem.pdf, 2005.

[10] COOK, W. R., AND RAI, S. Safe query objects: statically typed objects as remotely executable queries. In *ICSE '05: Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ACM Press, pp. 97–106.

[11] COOK, W. R., AND ROSENBERGER, C. Native queries for persistent objects. http://www.cs.utexas.edu/users/wcook/papers/NativeQueries/NativeQueries8-23-05.pdf, August 2005.

[12] Visual c# developer center. http://msdn.microsoft.com/vcsharp/programming/language/.

[13] db4objects. http://www.db4o.com/.

[14] Enterprise javabeans (ejb). http://java.sun.com/products/ejb/.

[15] GAVIN BIERMAN, ERIK MEIJER, W. S. The essence of data access in cω. In *Lecture Notes in Computer Science* (August 2005), vol. 3586, pp. 287–311.

[16] GAWECKI, A., AND MATTHES, F. Integrating query and program optimization using persistent CPS representations. In *Fully Integrated Data Environments*, M. P. Atkinson and R. Welland, Eds., ESPRIT Basic Research Series. 2000, pp. 496–501.

[17] GOULD, C., SU, Z., AND DEVANBU, P. Static checking of dynamically generated queries in database applications. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 645–654.

[18] HARRIS, T., MARLOW, S., PEYTON-JONES, S., AND HERLIHY, M. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2005), ACM Press, pp. 48–60.

[19] Hibernate. http://www.hibernate.org/.

[20] Hibernate query language. http://www.hibernate.org/hib_docs/reference/en/html/queryhql.html.

[21] ibatis. http://ibatis.apache.org/.

[22] Ibm. http://www.ibm.com/.

[23] JAMES GOSLING, BILL JOY, G. S., AND BRACHA, G. *The Java™ Language Specification, 3rd Edition*. Addison Wesley Professional, 2005.

[24] Java annotations. http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

[25] Java programming language compiler. http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/javac.html.

[26] Jpox java persistent objects. http://www.jpox.org/.

[27] .net language integrated query. http://msdn.microsoft.com/netframework/future/linq/.

[28] MAIER, D. Representing database programs as objects. 377–386.

[29] MCCLURE, R. A., AND KRÜGER, I. H. Sql dom: compile time checking of dynamic sql statements. In *ICSE '05: Proceedings of the 27th international conference on Software engineering* (New York, NY, USA, 2005), ACM Press, pp. 88–96.

[30] Microsoft research. `http://research.microsoft.com/`.

[31] Mysql.com. `http://www.mysql.com/`.

[32] Oracle corporation. `http://www.oracle.com`.

[33] Pjama: Orthogonal persistence for the java platform (opj). `http://research.sun.com/forest/opj.main.html`.

[34] Postgresql. `http://www.postgresql.org/`.

[35] Sql-java (sqlj). `http://www.sqlj.org/`.

[36] Sun microsystems. `http://www.sun.com/`.

[37] Oracle toplink. `http://www.oracle.com/technology/products/ias/toplink/`.

Appendix A

Example of database and Java mapping

This appendix is dedicated to describe the relational database used throughout the report. The database in question consists of three tables:

- person
- department
- project

There exists two additional tables. One is the relationship between *person* and *project* and is a many-to-many relation. In order to model this in a RDBMS we need one table to explain this relation, thus the *project_person* table. The other is *person* and *departments* called *deptments_person*.

Figure A.1 illustrates the relations of all tables in our example database.

A.1 Description of the database

Each person is constructed of four attributes. There is an id (`id`) that identifies each person uniquely. This id is used in the relation to the project he is working on - thus one person can be related to one project and only one - but since this id is not related directly to the project table but to a project-person table, this means that each person can be working on more than one project. The person also has a first name (`fname`) and a last name (`lname`). Finally each person has a relation to a department. This goes, as with the project, through a table that makes it possible to implement a many-to-many relation in the database. Figure A.2 shows the Java source code needed to implement the person class.

Each project consist of an id (`id`) which identifies each project uniquely. This id is the one used in the project-person relation. The project also can be identified by a name (`name`). Finally each project is related to one department (`dep_id`). Figure A.3 shows the Java code that implements the project class.

The final table, departments, has an id (`id`), like the two others, to identify each department uniquely. This id is referenced by the project table (`dep_id`) as one project is related to one department, but one department can have many projects. Each department has also a name (`dep_name`). Finally each department is related to a person (`dep_id`) through the person-department table, because many persons can be working in many departments - thus another many-to-many. Figure A.4 shows the Java code.

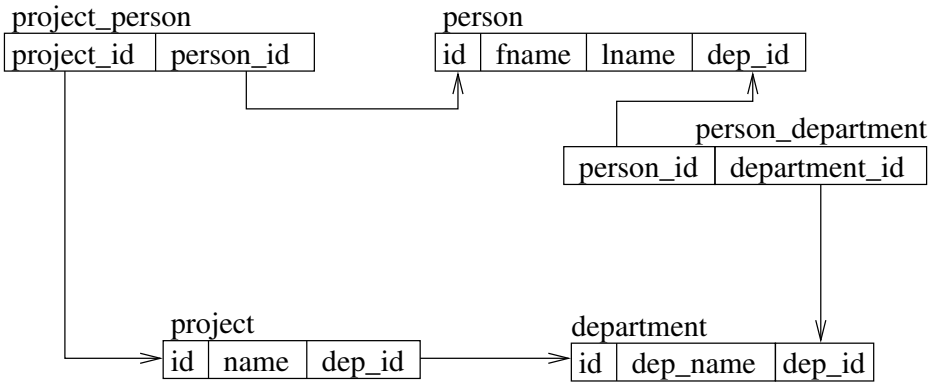

Figure A.1: Example database.

A.2 Mapping database to Java code

Mapping the relational database to the object oriented programming language Java would give us three classes, *Person* (see Figure A.2), *Department* (see) and *Project* (see Figure A.4).

```
1  package model;
2  public class Person{
3      private int id;
4      private String fname;
5      private String lname;
6      private int depId;
7      private ArrayList<int> projectId;
8
9      //getter and setters
10 }
```

Figure A.2: Java code for the class Person.

```
1  package model;
2  public class Department{
3      private int id;
4      private String depName;
5
6      //getter and setters
7  }
```

Figure A.3: Java code for the class Department.

```
1  package model;
2
3  public class Project{
4    private int id;
5    private String name;
6    private int depId;
7    private ArrayList<int> personId;
8
9    //getter and setters
10 }
11
```

Figure A.4: Java code for the class Project.