Shinji Takasaka

Survey of Persistence Approaches

December 2005

Department of Computer and Systems Sciences
Master Thesis at Royal Institute of Technology/Stockholm University in collaboration with Swiss Federal Institute of Technology Zurich - ETH

Supervisor at DSV: Maria Bergholtz
Supervisor at ETH Zurich: Stephanie Balzer and Prof. Dr. Bertrand Meyer
Examiner: TBD

Master thesis at the department of Computer and Systems Sciences at Stockholm University (corresponds to 20 credits for each author)

Contents

List of Figures

List of Tables

Summary. The majority of modern object-oriented programming languages lack native support for object persistence. Transparent object persistence increases productivity by allowing programmers to concentrate on implementing the business logic. In this report we present the features that an ideal transparent persistence approach should support. We also assess six existing persistence approaches by comparing them according to a set of criteria. We believe that the features presented here are a step forward towards the construction of the ideal transparent object persistence approach.

Acknowledgement. Without the contribution of many people this work would not have been possible. It is a great pleasure to acknowledge all the contributions of each of you who helped me in this journey.

First, I would like to thank Stephanie Balzer, my supervisor at ETH Zurich, for her unconditional help, constant support, enthusiasm, and new ideas. She was always there to give advice and to listen to my ideas. She taught me how to write academic reports and made me a better English writer.

A special thanks goes to Bertrand Meyer for providing the funding that covered the transport expenses from Sweden to Switzerland. Without this financial support it would not have been possible to work on this dissertation.

Thanks to Maria Bergholtz, my supervisor at KTH/SU, for her help, ideas and for reviewing my work.

Besides my supervisors, I would like to thank Mick Jordan and Tony Printezis from Sun Microsystems Inc. for helping me with PJama.

I would like to thank Johan Häggbom and Gabriella Sebardt for their friendship, kindness, and for helping me to make my life in Sweden easier and fun. And also I would like to thank Camila Rodrigues for helping me to make my life in Switzerland easier and enjoyable.

I also would like to thank Patrick Eugster and Martin Stoffel for giving me insightful comments and reviewing my work; and Marco Aurélio Amaral Henriques, João Frederico Meyer and Eduardo Henrique Lins for trusting my capabilities and for their recommendation.

And I would like to thank Yoshimi Takasaka and Sakuko Takasaka, my parents, for their unconditional love and support.

❖ 1

Motivation

1.1 Background

Since the invention of the first programmable computer by Zuse in 1941 storing data has been a major concern. During the Second World War, paper was in short supply in Germany. So Zuse, instead of using paper tape or punched cards, used old movie films to store data. Since then a number of programming paradigms and persistence mechanisms and storage media were developed. At present, the object-oriented paradigm for programming languages and the relational model for storing data in databases are the dominant techniques used when building applications.

In common object-oriented languages the lifetime of objects is limited by the duration of program execution (James Gosling 1996) (Meyer 1992) (Microsoft 2002). However, persistence capable objects are desired in many problem domains for developing complex software systems since persistent objects produce simpler and clearer designs. Handling such persistent objects should be transparent in the programmer's perspective. Transparent object persistence allows programmers to concentrate on implementing the business logic since with transparent persistence programmers need not to know additional mechanisms for persisting objects other than the programming language. Thus, programmers can be more productive.

1.2 Problem

Widely used object-oriented programming languages do still not provide native support for object persistence or provide only basic persistence mechanisms. However, most modern applications require more elaborated persistence mechanisms. Due to the lack of persistence mechanisms in today's programming languages the use of database management systems for storing

objects is common. The mapping activity required to overcome the problems resulting from the use of those two different paradigms demands a lot of effort, especially when inheritance is involved, and is a potential source of errors.

1.3 Objectives

In this report we propose a set of features that the ideal transparent object persistence approach should support.

1.4 Purpose

The intents of the work presented here are:

1. Define a set of criteria that will be used for comparing the persistence approaches.
2. Compare the approaches using the criteria defined in 1.
3. Define a set of features that the ideal transparent object persistence approach should support.

1.5 Method

In order to fulfill the objectives presented in the previous section we start by examining previous work (Jordan 2004; Atkinson, Bancilhon, DeWitt, Dittrich, Maier, and Zdonik 1990) carried out in similar fields. This research work will serve to define the preliminary set of criteria that we use for comparing the persistence approaches. Then we analyze and evaluate those preliminary criteria that result in the final set of the assessment criteria.

We implement a sample application using each persistence approach to guide us during the assessment phase. This application should be simple, but complex enough to help us in the judgment.

Based on the persistence approaches' evaluation we delineate an outline of the features that an ideal persistence mechanism should support.

1.6 Limitation

Top-down and bottom-up are two well-known approaches to design a software application. The top-down approach is application centric (programming language) while the bottom-up approach is data centric (database).

Usually, in industry, the bottom-up approach is used because new applications are built ”on top of” existing databases while in academia the top-down approaches are preferred since there is no need to deal with legacy data.

The solution presented in this report is based on the top-down approach. Therefore, only aspects which affect the top-down approach are considered.

1.7 Thesis Outline

This report starts by introducing the world of persistence in order to give the background needed to be able to understand our work. In chapter 4 we describe each criterion used for judging and compare the persistence approaches. Chapter 5 describes each persistence approach that are compared in this report. Chapter 6 provides a detailed assessment of each persistence approach. In chapter 7 we present the features that an ideal persistence approach should support by analyzing the assessment results and using the knowledge gained during the research. In chapter 8, finally, we draw the conclusions of our work and discuss future work.

❖ 2

Understanding Persistence

In this chapter we give an introduction to persistence and present the most commonly used persistence approaches.

Modern *object-oriented* programming languages such as Eiffel (Meyer 1992), Java (James Gosling 1996) and C# (Microsoft 2002) provide the programmer a number of services that are extremely useful for developing robust, reliable and secure enterprise applications. However, these programming languages are not able to maintain the object state beyond a single program execution in a satisfactory way.

Before proceeding, we would like to introduce the definition of the terminology used in this report.

- *Persistence* is the ability of data to outlive the lifetime of the process that created it.[1].
- *Object persistence* is the concept of persistence applied to the object model.
- *Relaxed transparent persistence* refers to the ability to persist data without the programmers knowing *how* it is achieved.
- Traditionally, programmers are used to the ”read, compute and write” model. This model leads programmers to try to distinguish transient and persistent objects. But if programmers apply object-orientation to model a system then they have only to concern about the relative lifetime of objects. The *strict transparent persistence* refers to the ability to persist data without the programmers knowing *how* it is achieved and *what* is persisted. When strict transparent persistence is supported there is no need for distinguishing transient and persistent objects. This can be achieved persisting the entire computation state.

[1] Persistence is also defined in the literature as the ability to support data values as long as they are needed, be it for a short time or long time. (Atkinson and Morrison 1995)

According to the above definition when we store data on a disk or in a database we are persisting data. When we are using an object-oriented programming language and we store objects we are persisting objects.

For the rest of this report when we refer to transparent persistence, we refer to the ability to hide how data are stored, i.e. relaxed transparent persistence.

In a typical application, we find two types of data (Atkinson, Bailey, Chisholm, Cockshott, and Morrison 1983):

- *short-term data*, also referred as *transient data*, are data that programming languages create, manipulate, and then discard automatically at the end of program execution.
- *long-term data*, also referred as *persistent data*, are data that the programming languages either store in *file systems* or handle over an external *database management system* (DBMS) for future manipulation or sharing data among other applications. DBMSs provide persistence, data sharing and retrieval services that most object-oriented programming languages fail to support.

In 1978, Atkinson proposed *orthogonal persistence* (Atkinson 1978) to provide integration between programming language and DBMS mechanism. Programming languages that support orthogonal persistence typically provide all the features usually found in DBMSs such as data storage, transaction management, concurrency control and indexing. This results in a platform where application programmers are not aware of the existence of a database. Programming languages that support orthogonal persistence provide to the application programmer a transparent persistent computational environment. Atkinson and Morrison, on their article entitled ”Orthogonally Persistent Object Systems” (Atkinson and Morrison 1995), define *orthogonal persistence* as being a language-independent model of persistence based on the following three principles:

- *Type orthogonality* - every object, independent of its type, has the same right to persist.
- *Persistence by reachability* - the reachability from the root of persistence determines the lifespan of each object.
- *Persistence independence* - the introduction of persistence cannot introduce semantic changes to the source code.

2.1 Existing Persistence Approaches

We present in this section the most relevant existing persistence techniques at present.

2.1.1 Serialization

Serialization is a technique where objects are encoded into a binary representation. Object persistence is achieved by saving the serialized objects in files or databases. Basically, the persistence process using serialization is based on the following steps:

1. Encode object into a binary representation;
2. Save the binary representation in a file or in a database;

While the retrieving process is based on the following steps:

1. Read the binary representation from a file or a database;
2. Decode the binary representation to get the object.

Serialization is the simplest mechanism to achieve persistence. It provides neither object query mechanism nor transaction support (section 4.5). Furthermore, it does not implement persistence independence (to be discussed in section 4.2) and is not scalable (Evans 1999).

2.1.2 Relational Database

Storing data in a *relational database management systems* (RDBMS) (Codd 1970) is by far the most commonly used technique to manage persistent data. Simplicity, scalability, transaction support, query support, query optimization, and security are some of the reason for its popularity (Bergholt and et al. 1998).

While the relational model is based on mathematical principles, the object model is based on software engineering principles. When both models are used in the same application - the object model for the programming language and the relational model for the database - we face a problem called ”object-relational impedance mismatch” or simply *”impedance mismatch”*.

Due to the difference of the underlying paradigms, both models do not work smoothly together. For instance, in the object paradigm you deal with objects that are linked through references (unidirectional) while in the relational paradigm you deal with relations and constraints (bidirectional).

To overcome the impedance mismatch, objects are usually mapped to relational databases. The process of mapping objects to relational database can

be complex, especially if inheritance is involved, and can be a source of bugs when not carefully carried out. Often, programmers are distracted from their actual task due to difficulties in understanding and managing mappings. This is not satisfactory because, ideally, programmers should primarily concentrate on building business logic and not on building infrastructure.

In a typical database application 30 percent of the code is related to the tasks for moving and organizing data to and from external storage(IBM Research Centre San Jose 1978). A lot of effort is spent by programmers to translate between different paradigms used by the program languages and database management systems.

Automated *object-relational mapping* (O-R mapping or simply ORM) tools are being developed to simplify the mapping process and reduce development time. It generates all the SQL code needed to store objects resulting in less error-prone and more portable systems. The major advantage of using ORM tools is that application programmers do not have to be aware of the schema of the relational database since they just work with objects. The disadvantage is that the extra layer may have a negative impact on performance. (Ambler 2005)

2.1.3 Object-Oriented Database

An *Object-oriented database management system* (OODBMS) is a DBMS based on an object-oriented data model (Harrington 2000). When using OODBMS the problem of the impedance mismatch does not exist because both, the programming language and the database management system, use the same data model. Thus, the resulting applications have less code and are easier to maintain.

When the first OODBMSs were introduced in the market the consensus was that it would take over the RDBMS market in little time, but what we see now is a consolidated market for RDBMS and OODBMS being used only in niche markets. There are a number of reasons for not adopting OODBMS, but the main reasons are satisfactory performance, lack of an *Object Query Language* (OQL) standard, and migration costs.

When compared to RDBMS, OODBMSs are still less efficient and less scalable. As a result, most of the new enterprise applications still uses RDBMS because they demand performance and scalability.

2.1.4 Object-Relational Database

With the arrival of client-server and internet-based applications a new challenge has emerged. These kind of applications demand supporting many si-

multaneous request to the system and consequently to the underlying relational database. As said before, the dominant model for building applications is the object model. From these facts, the interest on object-relational database management systems (ORDBMS) has emerged.

According to Stonebraker (Stonebraker and Moore 1996) an ORDBMS is a DBMS that combines the best of the relational and object-oriented DBMSs, i.e the mathematical foundations of the relational model and the complex and extensible data types of the object model. ORDBMSs support complex objects and have a good object query system.

2.1.5 Prevalence

Prevalence is a recent technique to implement object persistence. It uses serialization to store all commands which change the state of an object. It is based on transaction journals (also referred as logging) and snapshots (also known as checkpoint) concepts (Elmasri and Navathe 2003) used in many database management systems.

Since every transaction is logged, when a crash occurs to the system, it is possible to reestablish the previous system state by executing all stored commands against the last taken snapshot.

Prevalence mostly relies on the serialization mechanism present in the programming language to store snapshots and persistent objects. It does not use any external storage. Instead, it persists all classes reachable from the *prevalent system*. The prevalent system is a single class used as object container.

The major drawback of this approach is that the amount of objects that the system can handle is constrained by the size of the main memory.

2.1.6 Persistent Programming Languages (PPL)

Persistent programming languages are programming languages that comply with the three principles of orthogonal persistence (type orthogonality, persistence by reachability and persistence independence) (Atkinson, Bailey, Chisholm, Cockshott, and Morrison 1983) shortly introduced at the beginning of this chapter. The goal of PPLs is to unify data models and type systems.

One example of PPL is PJama, one of the persistence approaches that we assess in this report.

❖ 3

Test Application Explained

During our assessment phase we implement a simple application to help us in the judgment of the persistence approaches. In figure 3.1 the UML diagram of the application is presented.

We have implemented this application for each of the persistence approaches compared in this report. The actual source code is found in the Appendix A.

The application is a simple banking system. An *account* has an *account type*, belongs to a *branch* and has a *customer*. One branch may have zero, one or many *employees*. The application is not intended to reflect any real application.

We used the application to test some of the criteria presented in the chapter 4 such as persistence by reachability, transaction support and query language.

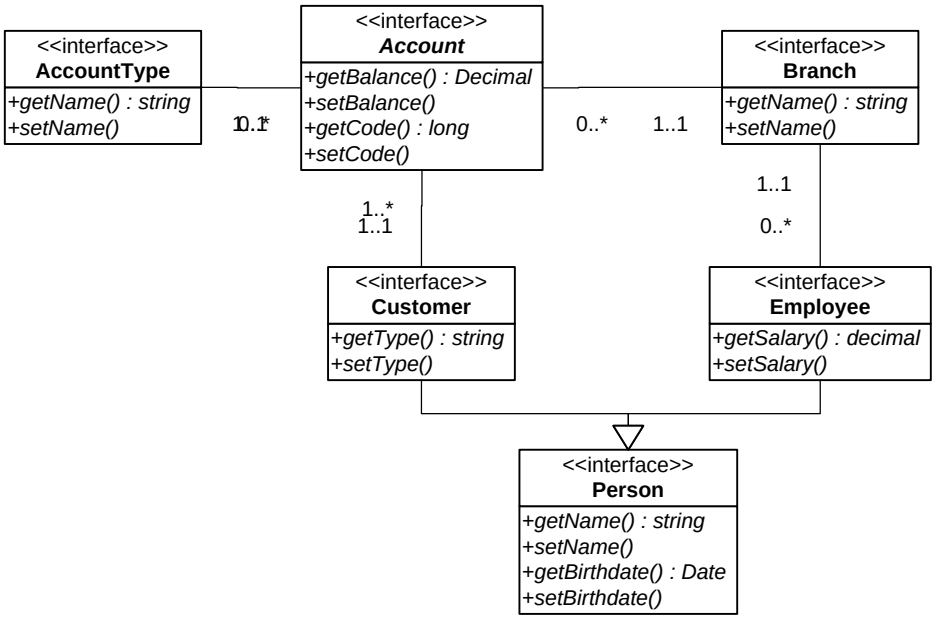

Fig. 3.1. Banking System UML diagram

❖ 4

Assessment Criteria

This chapter describes the criteria and methodology used for assessing the persistence approaches.

We assume that the three principles of Persistent Programming Languages (PPL) are the basic criteria used for comparing the persistence approaches (i.e, type orthogonality, persistence independence and reachability). The criteria presented here are based on the object-oriented model.

In order to find the criteria used in our work we have studied some works conducted in the related field. The preliminary set of criteria mostly, but not only, rely on the work by Mick Jordan where he compares persistence mechanisms for Java based on a set of criteria (Jordan 2004). We also rely on the ”Object-Oriented Database System Manifesto” by Atkinson et al., where they describe features that an OODBMS must implement (Atkinson, Bancilhon, DeWitt, Dittrich, Maier, and Zdonik 1990), and other works. After this investigation we selected the most relevant criteria, in our understanding, out of the preliminary set of criteria.

We use the following criteria for the assessment of the persistence approaches:

- Type Orthogonality
- Persistence Independence
- Persistence by Reachability
- Integrity Checking Support
- Transaction Support
- Reusability
- Query Language
- Schema Evolution Support
- Performance
- Scalability
- Usability

In the next sections we discuss each assessment criteria in more detail.

4.1 Type Orthogonality

The principle of type orthogonality states that all data have the same rights to persist irrespective of their type. Although orthogonality is highly desirable in a programming language most of them fail to completely support orthogonality. In most cases orthogonality is not supported because it is complex to implement and usually results in performance degradation (Jordan and Atkinson 2000).

4.2 Persistence Independence

Persistence should be possible to be achieved without any source code change, including annotations used as special comments. Annotations are metadata that programmers add to the source code to guide the compiler during the compilation process.

Another aspect closely related to persistence independence is the use of special development tools, such as special compilers or pre or post-processors, to create new classes that provide persistence. The extra step introduced in the compilation process is unsatisfactory because it reduces productivity by perturbating the normal program building process and does not allow the use of diagnostics, and debugging tolls because the output does not correspond to the source that the programmer is working with.

4.3 Persistence by Reachability

A persistence mechanism should provide a way to determine if an object is to be persisted or not. Usually it is provided mechanism to indicate the *roots of persistence*. The root of persistence is the object from where all objects reachable from it are persisted. When the root of persistence is indicated, it is necessary that all transient objects, at commit time [1], are reachable from the persistent object should also be persistent.

4.4 Integrity Checking Support

The intents of integrity checking is to provide means to guarantee accuracy and consistency of data in persistent applications.

[1] Commit time is the time when a set of database operations are made permanent

In databases, integrity checking is provided by enforcing *integrity constraints* and *type checking*, while in programming languages integrity checking is provided by type checking. The difference between the type system used in databases and in programming languages is that the later is easily extended while extending in databases is very hard, most of the time impossible.

Most relational databases guarantees accuracy and consistency by providing the following implicit and explicit integrity constraint types:

- An attribute can be declared as NOT NULL, which specifies weather null values are allowed or not.
- An attribute can be declared as UNIQUE, which specifies if the values should be unique or not in the relation.
- One or more attributes can be declared as PRIMARY KEY. A key identifies one and only one tuple in a relation. It is common in a relation to have more than one key. The keys in this case are called *candidate keys*. Generally one of the candidate keys is elected as the *primary key* of the relation. *Entity integrity constraint* says that primary keys cannot have null values.
- An attribute can be declared as FOREIGN KEY. A foreign key in an attribute guarantees that the value in this attribute is found as the primary key of another relation. This type of constraint is called *referential integrity* constraint and is used for maintaining consistency among tuples in two relations.

Besides these constraint types, it is possible to use TRIGGERS and ASSERTIONS when a more elaborated check is needed. Triggers and assertions are dynamic in nature, allowing run-time calculation of constraint values (Elmasri and Navathe 2003).

The type checking mechanisms present in programming languages perform static checking, before program execution. Consequently, static checking provides some level of safety within the application. Also, static checking impacts on efficiency because type checking code is not needed during run-time.

An unification of integrity constraint from the data model and type checking from the object model is desired in order to have the best of these two models, i.e. the richness of expression of data models and the completeness of protection required in static form.

4.5 Transaction Support

Transaction is a logical unit of database processing treated in a consistent and reliable way. Each logical operation works independently from other transactions. Databases should support four basic properties in order to guarantee

data integrity: atomicity, consistency preservation, isolation and durability. Usually, those properties are called ACID properties in the literature. (Elmasri and Navathe 2003)

Atomicity guarantees that either all tasks in a transaction are performed or not performed at all. Therefore when a transaction fails the DBMS should be able to rollback and reestablish the state of the last commit point.

Consistency is preserved when the database is in a legal state before and after the execution. A consistent transaction takes the database from one consistent state to another.

Isolation guarantees that each transaction is executed isolated from all other transactions. This means that when transactions are executed concurrently any transaction should not be interfered by any other operation until its completion. Most of the time this property is relaxed because the locks used to guarantee isolation would degrade performance.

Durability guarantees that the transactions committed successfully will be persisted in the database and the changes are not lost by any system failure. Usually durability is implemented by writing every transaction into a log [2] before commits and when a failure occurs the database is restated by reading all transactions from the log.

4.6 Reusability

The reusability principle is defined as the ability to execute either in persistent or in conventional execution contexts without changing code. Code developed for conventional context should be possible to be uses in persistent context and code written for persistent context should be possible to be used in conventional context.

This is closely related to another criterion namely, persistence independence, since changes to source code or bytecode may make this code not reusable in other context.

4.7 Query Language

Once the objects are persisted the approach should allow retrieving the objects. Object retrieval is an important feature of a persistence framework. Next we describe some of the several types of existing query mechanisms.

- *SQL* is a declarative language used for creating, modify and retrieve data from RDBMSs. SQL is standardized by the American National Standards

[2] Also called DBMS journal in literature.

Institute (ANSI) and the International Organization for Standardization (ISO). The success of relational databases in the industry relies on the existence of a standardized query language (SQL). Users can then switch databases without changing the program code. In practice most of the major commercial products implement different subsets of the SQL language but the user has to take care to use only features that are standardized, then the conversion process would be simpler and easier (Elmasri and Navathe 2003).

- *Object Query Language* (OQL) is a query language used for creating, modifying and retrieving objects from object-oriented databases. Although a number of efforts were carried out in this field no winning standard was elaborated. In the 1990s a group of OODBMS vendors founded the Object Data Management Group (ODMG). Their main goal was to establish a set of specifications in order to increase portability among different products from different vendors. Their work finished in 2001 with the release of the third version of the ODMG specification (Cattel and Barry 2000) which was used as the basis for the Java Data Objects (JDO) specification. The current situation is that almost every product has its own OQL language, such as Hibernate Query Language, EJB-QL, TopLink Expressions, JDOQL, etc., making it hard to learn.
- *Query By Example* (QBE) is a query language used to perform simple queries based on an example provided by the user (Zloof 1977). It is widely used when information is retrieved based on some attributes whose value are collected via a form-like interface. It is easier to use than other formal query languages, such as SQL and OQL, because the actual query is generated by the system.
- *Simple Object Database Access* (SODA) is an object query mechanism based on object graphs. It is possible to reach every objects associated to the root persistent object by traversing the graph (Grehan 2005).
- *Native Query* (Cook and Rosenberger 2005) was developed to use the programming language itself as the query language. The commonly used query mechanisms, namely SQL and OQL, are expressed using strings. Such query strings are undesired in modern integrated development environments (IDE) because IDEs do not check embedded strings for errors while queries expressed using the programming language are checked. Another advantage of native queries is that the programmer deals only with the programming language which means that one does not need to learn several different query dialects.

4.8 Performance

In this report we focus on the mechanisms that each approach provides to increase performance. When external databases are used as underlying object storage it is important to keep the number of database queries at a minimum in order to improve performance. Lazy reading, indexing and provision of a cache are the most commonly used techniques to improve performance.

The lazy reading approach is typically used when a table with several attributes exists and only some of them are required. One example of this situation is when you need to show a list of persons where the user will select one of them to get more detailed information. In this case only some attributes, for example name, date of birth and nationality, would be retrieved from the database and only when the user selects the person the system would retrieve the complete details for that particular person.

Another commonly used technique to improve performance is caching. It keeps copies of the entities in memory reducing the time to access data. Caching is extremely useful when no other application changes the database. But when two or more applications change the database, very efficient cache invalidation algorithms should be implemented in order to keep good performance.

4.9 Schema Evolution Support

Schema evolution refers to the evolution of the data schema. Most RDBMSs support schema evolution but it is not natively supported by most object-oriented languages. To overcome this problem most of the persistence approaches that deal with RDBMS have external tools that create new classes which reflect the changes introduced in the schema. Therefore, the programmer does not have to take care of the schema evolution.

4.10 Scalability

Scalability is a relative concept that is hard to define precisely. We understand scalability as the ability to increase the capability of the system as the requirements increase. We assess scalability by investigating the architecture of the persistence approaches and checking if persistence approaches offer support for the following scalability dimensions:

- Storage: How well the approach can deal with increase in data size requirement.

- Processing capability: How well processing speed can be scaled.
- Workload: How well the approach can deal with increase in the number of simultaneous client requirement.

Usually strategies such as clustering and caching are used to improve the three dimensions of scalability presented.

4.11 Usability

The usability is a dominant factor for deciding in favor of a certain persistence approach. But what is exactly meant by usability? According to the International Standards Organization (ISO) usability is:

The extent to which a product can be used by specified user to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.

In our context we assess usability by considering the following questions:

- Efficiency - How many steps programmers have to perform during development and deployment of the application?
- Learnability - How easy is to learn using the persistence approach?

❖ 5

The Persistence Approaches Compared

In this chapter we present each persistence approaches in detail.

5.1 Hibernate 3.0

Hibernate, an open-source Object-Relational mapping (ORM) framework, provides object persistence services for the Java language. It uses relational databases to store data and aims at solving the problem of impedance mismatch by mapping the object-oriented domain model of the application to the traditional relational model of the database.

Hibernate defines the details of a persistent class using external XML files. Every persistent-capable class must at least specify the corresponding relation in the relational database but it is possible to include details like which attributes in a class are to be persisted. With the introduction of annotations in Java 5.0 it will be possible to express the associations directly in the source code, hence a external file describing the mapping will not be necessary anymore.

Hibernate uses external relational databases as underlying storage. As consequence, it does not use the object identifier that the VM provides, because the identity provided by the VM cannot be guaranteed to be the same by Hibernate since objects are marshaled/unmarshaled to/from database. Hibernate controls object persistence using identities and states. Hibernate accepts identities generated by its own generation mechanism from the external datastore or assigned by the programmer.

An object may be in one of these different states:

- transient - The object does not exists in the underlying datastore.
- persistent - The object exists in the underlying datastore.
- detached - The object was retrieved from the datastore. Therefore the object exists in the underlying datastore, but Hibernate does not guarantee

that the object identity (Java identity used internally) is equivalent to the one it had before it was stored.

Having two-level caching architecture, Hibernate allows using third-party cache. This is plugged in the second-level cache. The first-level cache is the Hibernate Session. Concurrent access to the database by an external application is allowed but in this case the programmer should guarantee that changes made in cache are written back to the database and vice-versa.

Hibernate is widely adopted by the industry because it is open-source and adapts to any development process, whether starting from scratch or from a legacy database. For the industry it is essential that such tools are provided since most of the new applications have to deal with legacy databases.

5.2 TopLink 10*g*

TopLink is a tool that integrates Java applications to databases providing support for transparent persistence. We also categorize it as an Object-Relational Mapping tool (like Hibernate). Both products aim at having developers focused on implementing business logic rather than building basic support for object persistence.

TopLink uses an external relational database as underlying datastore and connects to them through JDBC [1] drivers. A set of XML mapping files (mapping metadata descriptors) define how objects are stored in the underlying datastore and another XML connection metadata file is used for describing the database connection parameters.

The SQL statements are dynamically generated based on the mapping descriptors. As a consequence, programmers do not have to change nor recompile the application's source code.

TopLink manages object identity using two distinct techniques:

- *Native Sequencing* - Identities are generated by the underlying database management system.
- *Sequence Tables* - Identities are managed by TopLink using a special table where the next identities for relations are stored. This technique allows preallocating a range (defined by the preallocation size) of identity numbers in memory. Therefore TopLink does not have to query the database for the next identity, improving performance.

[1] Java DataBase Connectivity

5.3 Java Data Objects 2.0 (JDO)

JDO is a specification that endeavors to achieve transparent persistence of Java objects. The work carried out by ODMG group on the binding between the object model and the Java programming language was used as basis for the JDO specification. The latest version of the specification was approved in early 2005 and is under development (Group 2005) at the time of writing this document.

The second major objective of the JDO architecture is to enable storing objects across a wide range of underlying data storage technologies, such as relational databases, object-oriented databases, XML databases and others.

JDO defines three different forms of identity:

- Application identity - managed by the application and based on some fields (often called primary-key) of the class. This type of identity is commonly used in RDBMSs.
- Datastore identity - managed by the datastore and untied to any field of an object. This type of identity is commonly used in OODBMSs.
- Nondurable identity - managed by the JDO implementation to guarantee uniqueness in the VM but not in the datastore. This type of identity is commonly used where performance is a priority concern.

JDO also defines the following required life cycle states:

- Transient
- Persistent-new
- Persistent-dirty
- Hollow
- Persistent-clean
- Persistent-deleted
- Persistent-new-deleted
- Detached-clean
- Detached-dirty

And defines the following optional life cycle state:

- Persistent-nontransactional
- Persistent-nontransactional-dirty
- Transient-clean
- Transient-dirty

Please refer to the JDO 2.0 specification document (Group 2005) to get a complete description of the life cycle states mentioned above.

JDO uses external XML file, specified by the specification, to define mapping when using relational database as underlying datastore. Mapping is done by specifying associations from classes and interfaces to relations, and class attributes to relation attributes. As in Hibernate, with the introduction of annotations in Java 5.0 it will be possible to express the associations directly in the source code.

Besides the relational model the mapping between the objects and the model used in the datastore is not standardized by JDO specification.

JDO also uses bytecode post-processing, called *enhancement* in the JDO specification, to modify persistence-capable classes and make them able to run in the JDO environment. It is necessary since persistence-capable classes must implement specific interfaces defined by JDO specification.

The JDO implementation used during the assessment phase is JPOX 1.1, an open-source JDO specification compliant implementation.

5.4 Orthogonal Persistence for the Java Platform (OPJ)

Orthogonal Persistence for the Java Platform (OPJ) (Jordan and Atkinson 2000) is a specification that extends the Java Language Specification (JLS), providing orthogonal persistence to the language. OPJ aims to provide a fault-tolerant environment, achieved by supporting persistent threads and by periodically checkpointing the state of the system to stable memory [2].

Systems implementing the OPJ specification periodically capture the computational state in an operation called *checkpoint*. The computational state snapshot is known as *suspended computation* and consists of the durable representation of the computation. A *persistent store*, usually a file on a disk, is used for storing the suspended computation. The suspended computation may be *resumed* in the future on the same or even on a different machine than that suspended the application. This results in applications that are initialized once and ”just keeps on going” (Jordan and Atkinson 2000)

OPJ does not specifies query facilities, leaving each particular implementation decide how to retrieve persistent objects.

Pjama (Atkinson and Jordan 2000) (Lewis, Mathiske, and Gafter 2000) is a particular implementation of the OPJ specification developed by the same team responsible for the OPJ specification.

[2] Usually it is used hard disks.

5.5 Prevayler

Prevayler is an open-source framework that provides prevalence services, presented in section 2.1.5, for the Java. It was implemented by the person behind the idea of the prevalence layer.

There are many other implementations of the idea of prevalence in other programming languages such as Bamboo for .NET (Bamboo.Prevalence 2005) and Florypa for Smalltalk (Beck, Mee, Soares, and Wuestefeld. 2005).

A typical application using a prevalence layer has three main components:

- The *prevalent system* used as a container for business objects. The system also handles snapshot.
- *Business Object* [3] classes.
- *Transactions* (or commands) are classes that modify the business objects and are logged using serialization.

In prevalent systems, a dedicated thread takes snapshots in intervals defined by the programmer. Programmers are aware of the existence of the *snapshot* mechanism, also known as *checkpoint*, and have to handle it explicitly.

The prevalent system class and the business classes have to implement the `Serializable` interface while command classes have to implement transaction specific interfaces, `TransactionWithQuery` or `Transaction`.

For each command a new class has to be created.

5.6 Db4o 4.6

An open-source object-oriented database, *db4o* is implemented for Java and C#.NET platforms. Since it stores any object natively it does not have the problem of impedance mismatch.

db4o has been used in some niche areas, such as embedded systems, in which features like no administration, small footprint and reliability are critical.

db4o does not use an external database as underlying datastore but binary files, known as `ObjectContainer` in the db4o language, created using the programming language. Pre-compiling, post-compiling or enhancement processes are not needed in order to have persistence capable objects.

Two types of identities are provided by db4o:

[3] Business objects are objects that abstract the entities present in the domain of the application. For example, an application for a telecommunications company would work with concepts such as Customer, Phone Call and so on.

- *Internal IDs* - every persistent object receives an internal ID. Uniqueness is guaranteed within one ObjectContainer even when the connection to the database is closed and reopened. The internal ids will change when an object is moved from one container to another.
- *Unique Universal IDs* (UUIDs) - This type of identity guarantees uniqueness across containers.

In db4o, objects are moved from one container to another when the defragmentation process takes place. During this process all indexes are recreated resulting in a faster and smaller database.

❖ 6

Assessing the Persistence Approaches

In this chapter we discuss the assessment of the persistence approaches based on the assessment criteria discussed earlier in chapter 4.

6.1 Type Orthogonality

In order to support type orthogonality, an object persistence approach has to provide the right to every type to persist, without no exception. Transparent persistence refers to the ability to persist data without the programmers knowing how it is achieved. A persistence approach that complies with the type orthogonality principle also complies with the transparent persistence principle.

6.1.1 Hibernate

Hibernate does not implement type orthogonality. Its main goal is to serve as a bridge between the object world of programming languages and the relational world of the datastores. One counter example of type orthogonality is that it is not possible to persist the class `Thread`.

6.1.2 TopLink

TopLink also does not try to implement orthogonality. It aims at providing transparent persistence as hibernate does. The proof of non-orthogonality is that it is not possible to persist the class `Thread`.

6.1.3 JDO 2.0

JDO is a specification for transparent object persistence for Java and does not implement orthogonal persistence. Again, the proof of non-orthogonality is that it is not possible to persist the class `Thread`.

6.1.4 PJama

PJama supports orthogonal persistence. It uses a modified *virtual machine* [1] that allows all classes defined in the Java language specification, including `Thread` and `Class` classes, to persist.

6.1.5 Prevayler

Prevayler does not intend to provide orthogonality. It was designed to achieve object persistence transparently and fault-tolerance. It is not possible to persist the `Thread` class, evidencing that Prevayler is not type orthogonal.

6.1.6 Db4o

Db4o does not support type orthogonality. Db4o has types that are not possible to persist.

6.2 Persistence Independence

An approach supports persistence independence when the introduction of persistence does not require source code change. The source code used in a persistent context should be reusable in a non-persistent context.

We also assess if the normal application build process has to be modified by a pre-compiler, post-compiler or enhancer.

6.2.1 Hibernate

Hibernate's persistent classes do not have to implement interfaces or extend from a persistent superclass. The only requirement for a persistent class used in Hibernate is a no-argument constructor. Hibernate allows any POJO [2] object to persist.

In business classes, Hibernate requires indicating the root of persistent objects and demarcating the transaction scope when operations to the database are not auto-committed. Otherwise, code can be totally persistence independent.

[1] A virtual machine isolates the application from the computer operating system, increasing portability. A program written for a virtual machine can, virtually, run in any computer platform.

[2] Plain Old Java Object (POJO) refers to pure Java objects, objects that do not implement or extend any framework specific interface or class.

6.2.2 TopLink

TopLink, as Hibernate, requires demarcating the transaction scope and indicating the root of persistent objects. Any POJO objects are eligible to persist. These facts evidence that TopLink is not completely persistence independent.

6.2.3 JDO 2.0

Persistent classes in JDO do not have to fulfill any extra requirement. Any POJO objects are eligible to persist. However, JDO uses a post-compilation process called *JDO Enhancer*. In the enhancement process, pure Java classes are modified to implement contracts present in the javax.jdo.PersistenceCapable interface. The enhancer uses information, present in a XML file, that describes the persistence properties of pure Java classes. Therefore, JDO is not totally persistence independent.

6.2.4 PJama

PJama is a persistence independent platform. It does not require that persistent classes implement any kind of interface nor extend persistent super classes. The only requirement is to indicate the root of persistence when no thread is active.

6.2.5 Prevayler

Prevayler violates persistence independence principle because persistent classes have to implement the Serializable interface and commands have to implement either Transaction [3] or TransactionWithQuery [4] interfaces.

The use of Prevayler as persistence mechanism does not impact on the normal build process of an application.

6.2.6 Db4o

When using db4o, code can be completely persistence-independent except for one requirement, db4o requires defining the root of the persistent objects.

Regarding application building process, db4o does not affect the normal process of application building.

[3] Nothing is returned as result of the command execution
[4] An object is returned as result of the command execution

6.3 Persistence by Reachability

A persistence approach supports persistence by reachability if all objects reachable from the persistent root object are also persisted.

6.3.1 Hibernate

Hibernate is able to persist all objects referenced from a root of persistence. If it is needed to persist objects using reachability, the programmer have to indicate that in the mapping document. Here is an example:

```
<one-to-many name="branch"
   column="branch_id"
   not-null="false"
   cascade="persist,delete-orphan"/>
```

The example above states that when the object is persisted the `branch` associated to it should be also persisted. Hibernate does not automatically delete unreferenced child objects unless the association is mapped with `cascade = "delete-orphan"`.

There are a number of methods, present in the `Session` interface, used for indicating the roots of persistence in Hibernate. See below some of them.

1. `Serialiable save(Object obj);`
2. `void save(Object obj, Serializable id);`
3. `void saveOrUpdate(Object obj);`

The method 1 first assigns a generated identifier to the given object and then persists it. The method 2 persists the given object, assigning the given identity. Method 3 either save or update the object instance depending on the identifier property.

6.3.2 TopLink

TopLink is able to persist all objects referenced from a root of persistence.

TopLink uses the following methods, present in `UnitOfWork` interface, to register the root of persistence:

1. `Vector registerAllObjects(Collection objs);`
2. `Vector registerAllObjects(Vector objs);`
3. `Object registerObject(Object domainObject);`

All methods presented above registers the root of persistence that will be persisted on commit. The clones of the given objects are returned. Programmers have to edit the returned clone objects because it is not allowed to edit the original objects in the transaction.

6.3.3 JDO 2.0

JDO supports persistence by reachability. In order to persist objects, programmers have to use one of the following methods defined in the JDO specification:

- `void makePersistent (Object obj);`
- `void makePersistentAll (Object[] objs);`
- `void makePersistentAll (Collection objs);`

All these methods assign an object identifier to the root object and change its state to *persistent-new*. Any object reachable from the root object will become conditionally persistent, that is, the state of the objects will become persistent, transitively, until commit time. And at commit time the reachability algorithm will run again checking if the conditionally persistent object are still reachable from the root object, reverting to transient state the objects that are not reachable anymore.

6.3.4 PJama

PJama persists every object reachable from the root of persistence. When programmers want not to persist certain objects in the graph the attributes referencing these objects should be declared with the `transient` modifier. See section 6.2.4 for the conflicts between the use of `transient` modifier in JOS and OPJ.

6.3.5 Prevayler

Prevayler relies on the object serialization mechanism provided by the programming language used to implement it. The JOS mechanism supports persistence by reachability, consequently Prevayler also supports reachability.

6.3.6 Db4o

Db4o supports persistence by reachability. It also uses a method to indicate the root of persistence. The method, found in the `ObjectContainer` class, is the following:

- `void set(Object obj);`

At commit time, db4o persists all objects reachable from the root of persistence objects.

6.4 Integrity Checking Support

Integrity constraints and type checking guarantee application reliability. We investigate and assess each persistence approach in terms of integrity checking support.

6.4.1 Hibernate

Hibernate allows defining all types of constraints present in the relational model, i.e key, null value, uniqueness, entity integrity and referential integrity (also referred to as foreign keys). From the programming language it uses the type checking mechanism.

6.4.2 TopLink

Similarly to Hibernate TopLink allows defining all types of integrity constraints present in the relational model. It also uses the type checking mechanism of the programming language.

6.4.3 JDO 2.0

JDO allows using primary keys and not null constraints in class attributes. In order to mimic referential integrity JDO uses the association capability of the object model. It is evident that the referential integrity expressed in the relational model is more powerful than using simply associations. It also uses the type checking mechanism of Java.

6.4.4 PJama

PJama provides no extra integrity checking mechanism besides the ones provided by the Java programming language, i.e mimics referential integrity through associations and type checking mechanisms. It worths remembering once more that the referencing mechanism, through attributes, used in the object model is less expressive and less powerful than the referential integrity mechanism used in the relational model. There is a research group that is implementing a system, called NightCap, which aims at providing PJama with means to express and verify precise semantic properties of classes (Collet and Vignola 2000). To achieve application correctness and data consistency NightCap uses assertions (precondition, postconditions and invariants) (Meyer 1997). The dedicated assertion language used is OCL-J, a mapping from the *Object Constraint Language* (OCL) (Clark and Warmer 2002) to Java. OCL is part of the UML standard (Group 2004).

6.4.5 Prevayler

Prevayler just provides the integrity checking mechanism used in the Java programming language. It uses the associations to mimic referential integrity and the type checking mechanism of the programming language.

6.4.6 Db4o

As in Prevayler, db4o relies on the integrity checking mechanism provided by the programming language. It uses the associations of the object model to mimic referential integrity and the type checking mechanism of the programming language.

6.5 Reusability

We assess reusability by investigating particular requirements that an approach needs to perform in order to persist an object. Every persistence particularity introduced in the source code hinders reusability, so code has to be modified to use with other persistence approaches. The reusability criteria is closely related to the persistence independence criteria.

6.5.1 Hibernate

Hibernate requires a no-argument constructor for persistent classes. However, it does not require to implement a specific interface nor extending a persistent super class, making Hibernate's persistent classes reusable in different contexts (persistent or transient) and persistence approaches.

Business objects used in Hibernate are required to provide the root of the persistent objects. When operations against the database are not autocommitted, it is also needed to demarcate the scope of the transaction.

6.5.2 TopLink

TopLink uses POJO objects as persistent classes. There is no need to implement specific interfaces or extend persistent super classes. Consequently, persistent classes used in TopLink can be reused with other persistence approaches.

In most cases, TopLink requires demarcating the limits of transaction and indicating the root of persistent objects.

6.5.3 JDO 2.0

JDO requires using an enhancer but since it is also standardized, bytecode enhanced using one JDO implementation can be used with other implementations. The standardization brings bytecode portability between different JDO implementations.

Enhanced bytecodes also works in a non-JDO environment but there is a small performance penalty. The overhead introduced by enhancement is about 11.5% (Jordan 2004).

JDO requires demarcating the limits of the transaction and also indicating the roots of persistence.

6.5.4 PJama

Code written for PJama can be reused in a normal Java virtual machine, except for the `transient` modifier.

The modifier `transient` has to be worked around because it was misused in the Java Object Serialization (JOS) specification. In the JOS context the modifier `transient` means that the attributes marked with this keyword have its reachability cut but also mean that it receives a custom serialization by the class implementation. Two additional sub-modifiers, `storage` and `serial`, are introduced to solve the problem. The following combinations are possible:

- `transient` - attribute is interpreted as transient by PJama and JOS.
- `transient storage` - attribute is interpreted as transient by PJama.
- `transient serial` - attribute is interpreted as transient by JOS.
- `transient storage serial` - same as `transient`, attribute is interpreted as transient by PJama and JOS.

One great advantage of PJama in terms of reusability is that it implements the OPJ specification. A standard specification guarantees reusability among systems using it.

In PJama transactions are initialized when the system starts up and are closed when it terminates.

6.5.5 Prevayler

Prevayler fails to meet the reusability criterion since it requires handling of persistent objects. For example, every persistent class has to implement the `Serializable` interface because Prevayler uses serialization for persisting objects. Also, every command that changes the state of an object has to implement either `TransactionWithQuery` or `Transaction` interface (see section

5.5). Moreover, the programmer has to add every persistent object to the prevalent system. It is not possible, as this can be done in Hibernate, TopLink, PJama or db4o, to solely indicate the root of persistence.

6.5.6 Db4o

Business objects used in db4o are not required to implement interfaces nor have to extend persistent super classes. The demarcation of transaction limits is optional since after every `commit` a new transaction is started. In general terms, db4o source code is reusable.

6.6 Performance

In this report we do not measure performance using benchmark tools. Instead, we investigate mechanisms adopted by persistence approaches that aim at improving performance such as caching, lazy loading and indexing (see section 4.8).

6.6.1 Hibernate

Hibernate uses an external storage (database) which makes data retrieval slower than mechanisms that limit persistence capability to the main-memory size but has the advantage of scalability.

Hibernate provides two different object caches, usually referenced as first and second level caches. Hibernate provides the first level (transaction level) cache that is associated with the `Session` object, while third-parties provide the second level (cluster or VM level) cache in form of pluggable caches and that are associated with the `SessionFactory`.

Hibernate uses some other strategies for diminishing the number of database round-trips:

- SQL statements are only executed when actually needed.
- Unmodified objects are never updated.
- Using outer join fetching.
- Lazy object loading.

6.6.2 TopLink

TopLink provides a caching mechanism. There was a problem with the caching mechanism until the version 9 of TopLink because the changes made by third-party applications was not reflected in the cached objects. But in the version 10 this problem was solved by introducing a cache invalidation method.

TopLink also allows lazy loading. This improves performance because it does not retrieve all referenced objects by the main object.

6.6.3 JDO 2.0

The JDO 2.0 specification also allows the use of two level caches as in Hibernate. The first one, on transaction level, is associated with the `PersistanceManager` and the second, on virtual machine level, is associated with the `PersistanceManagerFactory`.

6.6.4 PJama

Performance was not a primary concern when PJama was developed. The intent of PJama was to introduce orthogonal persistence into the Java programming language. A considerable overhead, 15 to 20% in average, is added with the provision of orthogonal persitence (Jordan 2004).

6.6.5 Prevayler

Prevayler has no mechanism intended for performance improvement. All persistent objects have to fit in the main memory, so every change to the object state is very fast, but the drawback is that only a limited number of objects can exist in the application.

6.6.6 Db4o

Db4o provides the following mechanisms aimed at improving performance:

- Object caching to reduce database round-trip.
- Field indexing to improve query performance.

6.7 Scalability

Scalability is a relative measurement. We understand scalability as the ability of the system to grow in one or more dimensions in response to increasing requirements. The dimensions that we investigate in this report are: storage, processing capability and workload (see section 4.10).

6.7.1 Hibernate

Hibernate has a two level cache architecture, as already mentioned in section 6.6.1. The first level cache is implemented by Hibernate and the second level is attached to the application in form of a plug-in. This architecture, with pluggable cache, allows third-parties specialized in scalability to plug their product into Hibernate. The separation of concerns makes it possible to build an optimal system.

One plug-in that enhances Hibernate's scalability is GigaSpaces (GigaSpaces 2005). It is a grid based distributed caching system. The advantage of using such kind of caches is that the cache size is not limited to the capacity of the main memory, instead it can grow, virtually, unlimited.

It is also possible to use Hibernate in a cluster of computers, increasing processing and workload capabilities.

6.7.2 TopLink

TopLink supports grid computing, resulting in scalability in all three dimensions previously mentioned. TopLink can also be used in a computer cluster, providing load balancing, workload scalability and processing capability.

6.7.3 JDO 2.0

The JDO specification does not explicitly demand that the implementation should be scalable. Each particular implementation should provide its own solution for scalability. But design solutions such as cache, present in the JDO specification, favor scalability.

6.7.4 PJama

PJama, as implemented today, has a limitation on data storage size. If the data size reaches its limit it is not possible to create persistent objects anymore.

In terms of workload and processing capability, it also does not scale because each data store is controlled by one virtual machine, not allowing concurrent access.

6.7.5 Prevayler

Prevayler scores low in terms of scalability. The storage size is limited to the size of the main memory, or up to 4GB in computers with a 32-bit architecture. Although 64-bit computers are becoming popular, it is still very expensive to buy large amount to RAM compared to disk memory.

Prevayler does not allow concurrency, so it is not scalable at all in terms of processing capability and workload.

6.7.6 Db4o

The maximum data store size that db4o can handle is 254 GB. Although it is large enough for many applications, it cannot be used for systems that demand high storage space such as banking and geographical information systems.

Concerning workload and processing capability, db4o has a simple client/server version where multiple clients can perform operation against the database. It is one step further if compared to Prevayler or PJama (single VM) but not as scalable as TopLink or Hibernate which use external database systems.

6.8 Transaction Support

We investigate transaction support by assessing how the persistence approaches support the ACID properties presented in section 4.5.

6.8.1 Hibernate

Hibernate supports all four ACID properties of transactions. For some of the properties such as durability Hibernate relies on the underlying database system.

6.8.2 TopLink

TopLink also supports all properties of transactions. Programmers explicitly start and close transactions, allowing grate control of the persistent object.

6.8.3 JDO 2.0

When using JDO, programmers have to start and terminate transactions explicitly. Every operation to the database should be done inside a transaction.

6.8.4 PJama

PJama supports transactions, the chain transaction model, i.e. a transaction starts implicitly when the application starts or resumes execution. PJama does not support programmable rollback. It has one level rollback (the last checkpoint) and the only way to rollback is to exit the application with an error.

6.8.5 Prevayler

Prevayler relies on serialization for persisting objects, inheriting all the deficiencies present in JOS. Atomicity is not supported. And it also does not support isolation as it does not prevent other threads from accessing the serialized objects. It lacks a controlled rollback mechanism.

6.8.6 Db4o

As in PJama db4o implicitly starts a transaction when the system opens a container and the transaction is closed when the container is closed. But unlike PJama, db4o allows programmable rollback. It is possible to use commit and rollback whenever they are needed.

6.9 Usability

We assess usability of the persistence approaches by investigating the number of steps users have to perform when developing and deploying applications. We also assess learnability by investigating how many configuration files programmers have to use. The reason to count the number of metadata files is that they hinder learnability (see section 4.11).

6.9.1 Hibernate

Hibernate does not use an enhancer, a pre-compiler or a post-compiler. So it is possible to have the bytecode of the application just after the compilation, a one step process.

Hibernate uses two configuration files, one used for configuring the framework and accessing the database and the other for mapping the object attributes to the table fields. Having distinct files makes the maintenance task more difficult. With Hibernate 3 it is possible to use annotations but one still has to deal with mapping. Programmers have to put the configuration files in the classpath of the application.

6.9.2 TopLink

TopLink uses a one step process to generate the bytecode of the application. It does not use any other tool besides the compiler.

When it comes to configuration, TopLink uses two metadata files.

- project.xml - configures the connection to the database.

- sessions.xml - describes the mapping between classes and tables, attributes and columns.

These files should be in the classpath of the application.

6.9.3 JDO 2.0

JDO has a set of mapping metadata files that are used by the enhancer when enhancing the bytecodes and during the application execution. The mapping file contains metadata which map object attributes to relation attributes.

JDO hinders usability because it uses an enhancer to force all persistent and persistent-aware classes to follow the JDO interface.

6.9.4 PJama

PJama does not make use of external metadata files or additional steps in the building process. A PJama programmer does not have to learn anything besides the Java programming language, which is a tremendous advantage in terms of learnability.

PJama requires the use of a special *virtual machine*. Beside the use of the virtual machine no other changes to the development process are needed. It is possible to use any compiler during the development of the program and no pre-compiling or post-compiling process is needed.

6.9.5 Prevayler

Prevayler uses no external metadata or additional tools to build the application. In terms of learnability, programmers have to learn the required methods present in the interfaces provided by Prevayler.

6.9.6 Db4o

Db4o uses no external metadata files to describe mappings because it does not use an external data store. Every operation is performed within the domain of the programming language. Db4o uses the standard compiler provided by the programming language and no other type of compiler.

6.10 Schema Evolution Support

We investigate schema evolution support by assessing how the persistence approaches deal with changes to the class description.

6.10.1 Hibernate

As long as Hibernate uses relational databases as storage, it faces the problems that the relational model has. This means if a change in the schema has to be introduced in an application the source code and the database schema have to be modified, and usually the data from the old schema has to be migrated to the new one.

Hibernate has two tools that support schema evolution:

- Middlegen (Middlegen 2005) - is a tool that automatically generates mapping metadata files based on the relational schema model. When a change in the schema is introduced, the database schema should be modified before running Middlegen.
- hbm2java (Hibernate 2005)- is a tool that automatically generates Java source code based on the mapping metadata files.

Besides these two tools, Hibernate provides no schema evolution support.

6.10.2 TopLink

TopLink provides a tool to generate Java source code based on an existing database schema. In order to have the new Java model class the database schema should be modified before running the tool.

6.10.3 JDO 2.0

The JDO standard provides no mechanism to support schema evolution. Eventually, a particular implementation of the JDO standard can provide appropriate tools but they are not required by the JDO 2.0 specification.

6.10.4 PJama

Although the OPJ specification does not provide a standard for supporting schema evolution, PJama provides a standalone command line tool to deal with evolution (Dmitriev 2001). It also uses an additional API when persistent object conversion is required. Currently, evolution of single classes, class hierarchies and persistent objects are supported. Persistent classes can be replaced by their new versions and persistent objects can be converted to conform to the new definition of their classes. A *conversion class* is needed when the conversion is not possible automatically.

6.10.5 Prevayler

Prevayler provides no special support for schema evolution besides the ones provided by JOS.

6.10.6 Db4o

Db4o supports some types of schema evolution as:

- Adding and removing attributes from a class.
- Renaming attributes in a class.
- Changing attribute types.

6.11 Query Language

In here, we investigate how the persistence approaches support querying the persistent objects.

6.11.1 Hibernate

Hibernate provides three ways of querying persistent objects.

- Hibernate Query Language (HQL) - is an object query language. Queries are expressed in terms of classes and attributes. Strings are used for expressing the query statements. See below the source code fragment of the method that retrieves a `Customer` having a `name` as parameter.

```
private Customer getCustomerByName(String name) {

  Session session = HibernateUtil.currentSession();

  Transaction tx = session.beginTransaction();
  Customer c = (Customer)session.createQuery(
    "FROM ch.ethz.se.Customer as customer " +
    "WHERE customer.name = ?").setString(0, name).uniqueResult();
  tx.commit();

  HibernateUtil.closeSession();

  return c;
}
```

- Criteria - is an interface that allows programmers to build queries dynamically rather than using query strings. See below a source code fragment of a query that uses `Criteria`. The method below retrieves a list of `Customer` having a `name` as parameter. The string ”name” present in the code fragment refers to the attribute name of the class `Customer`.

```
  private List getCustomersByName(String name) {
    List customers = session.createCriteria(Customer.class).
      add(Restrictions.eq("name", name)).list();
    return customers;
}
```

- Native SQL

6.11.2 TopLink

TopLink provides several options for query selection:

- TopLink expressions - is an object query language where the queries are expressed in terms of classes and attributes. Strings are used to express the query statements. See below the source code fragment of the method that retrieves a `Customer` having a `name` as parameter.

```
private Customer getCustomerByName(String name) {
    UnitOfWork uow = session.acquireUnitOfWork();

    ExpressionBuilder eb = new ExpressionBuilder();
    Expression exp = eb.get("name").equal(name);
    Vector customers = session.readAllObjects(Customer.class, exp);

    Customer c = null;
    for (Iterator itr = customers.iterator(); itr.hasNext();) {
      c = (Customer) itr.next();
    }

    uow.commitAndResume ();
    return c;
}
```

- EJB QL - TokLink can also query objects using the standard object query language used in EJB [5].

[5] Enterprise Java Bean

- Native SQL
- Stored procedures
- QBE - See below the source code fragment of the method that retrieves a `Customer` having a `name` as parameter.

```
private Customer getCustomerByName(String name) {
  ReadObjectQuery query = new ReadObjectQuery();
  Customer customer = new Customer();
  customer.setName(name);
  query.setExampleObject(customer);

  Customer result = (Customer) session.executeQuery(query);

  return result;
}
```

6.11.3 JDO 2.0

The JDO 2.0 specification provides two ways of retrieving persistent objects:

- JDOQL - is an object based query mechanism which allows to express the query in terms of classes and attributes. See below a fragment of the method that retrieves a `Customer` having a `name` as parameter.

```
public Customer getCustomerByName(String name) {
  Transaction tx = pm.currentTransaction();
  tx.begin();

  Query query = pm.newQuery(
      "SELECT FROM ch.ethz.se.jpox.persistent.Customer " +
      "WHERE name == \"" + name + "\"");
  query.setUnique(true);
  Customer customer = (Customer) query.execute();

  tx.commit();

  return customer;
}
```

- Native SQL

6.11.4 PJama

PJama provides no mechanism to query persistent objects. It access a persistent object by navigating the object tree.

6.11.5 Prevayler

Prevayler also provides no mechanism to query persistent objects. It access a persistent object by navigating the object tree.

6.11.6 Db4o

db4o can query persistent objects using the following mechanisms:

- QBE - See below an example of QBE being used fro retrieving a `Customer` having a `name` as parameter.

```
public Customer getCustomerByName(String name)
{
  Customer proto = new Customer(); //Instanciates a prototype object used as exar
  proto.Name = name; //assigning the given name to the prototype
  Customer customer = (Customer) db.get(proto).next();
  db.commit();

  return customer;
}
```

- SODA - See below a method that uses SODA for retrieving a list of `Employee` having a `Branch` as parameter.

```
public ArrayList getEmployeesByBranch(Branch branch)
{
  Query query = db.query();
  query.constrain(typeof(Employee)); //From the Employee object
  query.descend("workingBranch").constrain(branch.Name); //descend to the branch
  ObjectSet os = query.execute();
  ArrayList list = new ArrayList();
  while (os.hasNext())
  {
    list.Add((Employee) os.next());
  }
  db.commit();
  return list;
}
```

- Native Queries - This query mechanism will be supported in version 5.0, it was not supported in the version, 4.6, with which we worked with during the assessment phase. See below an example of a query using native query.

```
public IList getCustomerByName(string name) {
  IList <Customer> customers = database.Query <Customer> (
    delegate(Customer customer) {
      return customer.Name.Contains(name);
    }
  );
  return customer;
}
```

The method above retrieves a list of `Customer` having a `name` as parameter..

6.12 Assessment Summary

	Type Orthogonality	**Persistence Independence**	**Persistence by Reachability**
Hibernate	- No	- No	+ Yes
TopLink	- No	- No	+ Yes
JDO	- No	- No	+ Yes
PJama	+ Yes	+ Yes	+ Yes
Prevayler	- No	- No	+ Yes
Db4o	- No	- No	+ Yes

Table 6.1. Persistence Approaches Assessment Summary 1

	Integrity Checking	**Reusability**	**Performance**
Hibernate	+ Type checking + Integrity constraint	- No argument constructor - Demarcating transactions	+ Caching
TopLink	+ Type checking + Integrity constraint	- No argument constructor - Demarcating transactions	+ Caching
JDO	+ Type checking + Integrity constraint	- Demarcate transaction + Standard	+ Caching
PJama	+ Type checking	+ Standard	- No
Prevayler	+ Type checking	- Implement interface	+ All objects in memory
DB4o	+ Type checking		+ Caching +Indexing

Table 6.2. Persistence Approaches Assessment Summary 2

	Scalability	Transaction	Usability
Hibernate	+ Caching + Clustering + Griding	+ Yes	+ 1 step compilation - metadata files
TopLink	+ Caching + Clustering + Griding	+ Yes	+ 1 step compilation - metadata files
JDO	+ Caching	+ Yes	- enhancer - metadata file
PJama	- No concurrent access to data storage - Limited data storage size	- No	- non standard virtual machine + optimal learnability
Prevayler	- Limited data storage size - No concurrent access to data storage	- No	+ 1 step compilation
DB4o	- Limited data storage size	+ Yes	+ 1 step compilation

Table 6.3. Persistence Approaches Assessment Summary 3

	Schema Evolution	Query Language
Hibernate	+ Yes	+ Yes(HQL, Native SQL, Criteria)
TopLink	+ Yes	+ Yes(TokLink Expression, EJB QL, Native SQL, Store procedure, QBE)
JDO	- No	+ Yes(JDOQL, Native SQL)
PJama	++ Yes	- No
Prevayler	+ Yes	- No
DB4o	+ Yes	+ Yes(QBE, SODA, Native Queries)

Table 6.4. Persistence Approaches Assessment Summary 4

❖ 7

The recommended and the ideal Persistence Approach

In the first part of this chapter we analyze the assessment presented in the previous chapter and indicate the best persistence approach according to the criteria presented in chapter 4. And in the second part we discuss the features of an ideal persistence approach.

7.1 Evaluating Existing Object Persistence Approaches

We start the evaluation by presenting the positive and negative aspects of each persistence approach (see section 6.12 for the assessment summary).

7.1.1 Hibernate 3.0

Positive

+ Supports persistence by reachability
+ Supports type checking
+ Supports integrity constraints
+ Supports caching
+ Supports clustering and griding
+ Supports transactions
+ One step compilation
+ Partially supports schema evolution (datastore-schema evolution)
+ Provides objects retrieval mechanism

Negative

- Does not support type orthogonality
- Does not completely support persistence independence
- Requires no-argument constructor
- Requires transaction demarcation
- Uses metadata file

7.1.2 TopLink 10*g*

Positive

+ Supports persistence by reachability
+ Supports type checking
+ Supports integrity constraints
+ Supports caching
+ Supports clustering and griding
+ Supports transactions
+ One step compilation
+ Partially supports schema evolution (datastore-schema evolution)
+ Provides objects retrieval mechanism

Negative

- Does not support type orthogonality
- Does not completely support persistence independence
- Requires no-argument constructor or a factory that instantiates a persistent object
- Requires transaction demarcation
- Uses metadata file

7.1.3 JDO 2.0

Positive

+ Supports persistence by reachability
+ Supports type checking
+ Supports integrity constraints
+ Supports caching
+ It is a standard
+ Supports transactions
+ Provides objects retrieval mechanism

Negative

- Does not support type orthogonality
- Does not completely support persistence independence
- Requires transaction demarcation
- Uses metadata file
- Uses bytecode enhancer
- Does not support schema evolution (Support is implementation specific)

7.1.4 PJama 1.6.5

Positive

+ Supports type orthogonality
+ Supports persistence independence
+ Supports persistence by reachability
+ Supports type checking
+ Allows reusability (implements OPJ standard)
+ Optimal learnability. There is no need to learn anything else than the programming language
+ Supports schema evolution (object model evolution)

Negative

- Does not support performance enhancement
- Does not completely support transactions
- Does not allows concurrent access to datastore
- Have limited datastore size
- Uses non standard virtual machine
- Does not provide objects retrieval mechanism

7.1.5 Prevayler

Positive

+ Supports persistence by reachability
+ Supports type checking
+ All persistent objects are in main memory
+ One step compilation
+ Partially supports schema evolution

Negative

- Does not support type orthogonality
- Does not completely support persistence independence
- Persistent objects must implement the `Serializable` interface
- Does not allow concurrent access to datastore
- Has limited datastore size
- Does not completely implement transactions
- Does not provide objects retrieval mechanisms

7.1.6 Db4o

Positive

+ Supports persistence by reachability
+ Supports reusability
+ Supports type checking
+ Supports caching
+ Supports indexing
+ Supports transaction
+ One step compilation
+ Partially supports schema evolution
+ Provides objects retrieval mechanism

Negative

- Does not support type orthogonality
- Does not completely support persistence independence
- Has limited datastore size (254GB)

With the data collected in our assessment several interpretations are possible. When we assign the same weight to each criterion, i.e just count the positive aspects and subtract the negative aspects, Db4o is the winner followed by, TopLink and Hibernate, PJama, JDO, and Prevayler in this order. JDO appears in the lower part of this ranking because the specification does not require the implementation of some features that Hibernate and TopLink provide, such as schema evolution support.

But when we focus on orthogonal persistence support, PJama is the winner. PJama is the only persistence approach compared in our work that supports the three principles of orthogonal persistence: type orthogonality, persistence by reachability and persistence independence.

7.2 Features of an Ideal Transparent Object Persistence Approach

Next we present the features that an ideal transparent object persistence approach should have. But before proceeding we would like to provide some terminology conventions (Bradner 1997).

- *MUST* means that the feature is an absolute requirement.
- *SHOULD* means that there may have a valid reason to ignore the feature. Understanding the implications before discarding the feature is crucial. Take the feature as a recommendation.

7.2.1 Type Orthogonality

Type orthogonality is not needed to achieve transparent persistence. Db4o, Hibernate and TopLink are some example of approaches that supports transparent persistence without supporting type orthogonality. Type orthogonality SHOULD be implemented.

But if the strict version of transparent persistence (see chapter 2) is required, then type orthogonality MUST be implemented. It is not possible to persist the entire computational state without the chance to persist all types.

7.2.2 Persistence by Reachability

Persistence by reachability and type orthogonality lead to a model where programmers do not have to distinguish between transient and persistent objects. Instead, the entire computational state is persisted and programmers have not to worry about relative lifetime of objects. But when persistence by reachability does not come with type orthogonality, programmers should be aware of transient and persistent objects. In this case, programmers should specify the root of persistence in order to persist objects. The concept of persistence by reachability is now well-understood in the object community and is extensively used by garbage-collectors. Every object not reachable from the root of persistence is automatically garbage-collected, freeing main memory. Persistence by reachability MUST be implemented.

7.2.3 Persistence Independence

The introduction of persistence should not impact the semantic of the programming language. We give a counter-example to illustrate the benefits of persistence independence. Imagine that the programming language requires two distinct classes, one used in memory (transient) and one used for persisting. This leads to the existence of two different versions for the same class, hindering reusability, consistency and reliability. Persistence independence MUST be implemented.

7.2.4 Integrity Checking Support

Software reliability is a primary concern in an era where software applications are present everywhere in the entire civil structure. Banking, defense and medical systems are some areas where people mostly rely on software applications. And where failure can cause serious damage, even death.

Currently, the research community is trying to use Design by Contract (Meyer 1997) to support integrity checking. (Balzer 2005) and (Collet and Vignola 2000), in distinct work, introduce the use of assertions in persistent programming languages. Support for integrity checking MUST be implemented.

7.2.5 Query Language

The success of the relational model for persistence relies heavily on the provision of a standardized and simple data retrieval mechanisms. Although (Jordan 2004) claims that the query languages is not part of the scope of persistence, we strongly believe that the provision of a standardized object retrieval mechanism would lend wings to the consolidation of object persistence approaches. Object retrieval mechanisms MUST be provided.

7.2.6 Transaction Support

Client-server and internet applications are present in all organizations. These kind of applications allow concurrent access to the application, demanding a good mechanism to control concurrency and integrity. Transaction is the mean to guarantee consistency and concurrency. Most relational databases support concurrency and transactions very well and to replace them the object persistence approach should support concurrency and transaction too. Transaction and concurrency MUST be supported.

7.2.7 Performance and Scalability

With the popularization of internet, millions or even billions of users can potentially access a single system at the same time. To to able to handle all the connections a high-performant and scalable application is needed. Performance and scalability are not a required feature for a object persistence approaches but are highly recommend.

Performance and scalability SHOULD be a primary concern.

7.2.8 Schema Evolution

Currently, requirements of applications are in constant change. This is because the competitive market demands new products and services. To cope with the presented scenario the persistent approach should provide mechanisms that allow the required changes. Franconi et al. (Franconi, Grandi, and Mandreoli 2000) present an interesting approach to the problem of schema evolution and would be a good start for further research in this field.

Schema evolution SHOULD be supported.

7.2.9 Usability

Simplicity is a key word when it comes to usability. A simple and well designed system conducts to a system that is easier to use and to learn. An easy to use tool is always desired but designers should not favor usability in detriment of the MUST features. Usability SHOULD be implemented.

❖ 8

Future Work and Conclusions

We start this last chapter by reviewing our work and then we continue by presenting our thoughts about future work. Finally, we present our conclusions.

This report addressed aspects of persistence relevant to transparent object persistence. Our goals were:

1. To assess existing persistence approaches based on a set of criteria.
2. To elaborate the set of features an ideal transparent object persistence approach should implement.

We elaborated the set of assessment criteria by investigating and analyzing work conducted in related fields, and then choosing the most relevant aspects to transparent object persistence.

The next step was to create a simple application to implement and test all the persistence approaches compared in this report. Refer to chapter 3 to get a detailed description of the simple banking system that we used. The implementation and testing phases gave us important insights regarding each persistence approach.

The assessment of the persistence approaches was based on specifications, documents provided by each persistence approach and the sample application implementation. The result of the assessment showed that Db4o is the best persistence approach when the same weights were given to each criterion. However, in terms of orthogonal persistence, PJama is the best approach, complying with all the three principles (type orthogonality, persistence by reachability and persistence independence).

The ideal object persistence approach MUST, according to our vision, implement the following features: persistence independence, persistence by reachability, integrity checking, transaction and query language. It SHOULD support type orthogonality, reusability, performance, scalability, schema evolution and usability.

This report ends by stating the required features that an object persistence approach should implement. The next steps towards the construction of an ideal transparent object persistence framework would be a series of studies to explore and define the best way to implement each required feature described in this report (see section 7.2).

It is also necessary to conduct a deeper study concerning type orthogonality in order to clarify the resulting advantages and disadvantages.

Applications dealing with complex, long-term data are very common today. Many of these applications need a better approach for achieving object persistence. We believe that the assessment and the features presented in this report are a step further towards the construction of an ideal transparent object persistence approach.

References

[Ambler 2005] Ambler, S. W. (2005, June 21). The design of a robust persistence layer for relational databases. White paper, Ambysoft Inc.

[Atkinson and Jordan 2000] Atkinson, M. and M. Jordan (2000, June). A review of the rationale and architectures of pjama: a durable, flexible, evolvable and scalable orthogonally persistent programming platform. Technical Report SMLI TR-2000-90, Sun Microsystems Laboratories.

[Atkinson 1978] Atkinson, M. P. (1978). Programming languages and databases. In S. B. Yao (Ed.), *Fourth International Conference on Very Large Data Bases, September 13-15, 1978, West Berlin, Germany*, pp. 408–419. IEEE Computer Society.

[Atkinson, Bailey, Chisholm, Cockshott, and Morrison 1983] Atkinson, M. P., P. J. Bailey, K. Chisholm, W. P. Cockshott, and R. Morrison (1983). An approach to persistent programming. *Comput. J. 26*(4), 360–365.

[Atkinson, Bancilhon, DeWitt, Dittrich, Maier, and Zdonik 1990] Atkinson, M. P., F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. B. Zdonik (1990). The object-oriented database system manifesto. In H. Garcia-Molina and H. V. Jagadish (Eds.), *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, May 23-25, 1990*, pp. 395. ACM Press.

[Atkinson and Morrison 1995] Atkinson, M. P. and R. Morrison (1995). Orthogonally persistent object systems. *VLDB Journal 4*(3), 319–401.

[Balzer 2005] Balzer, S. (2005). Contracted persistent object programming.

[Bamboo.Prevalence 2005] Bamboo.Prevalence (2005, November). http://bbooprevalence.sourceforge.net/. http://bbooprevalence.sourceforge.net/.

[Beck, Mee, Soares, and Wuestefeld. 2005] Beck, K., R. Mee, H. Soares, and K. Wuestefeld. (2005, August). http://www.prevayler.org/wiki.jsp?topic=florypa.

[Bergholt and et al. 1998] Bergholt, L. and et al. (1998). Database management systems: Relational, object-relational, and object-oriented data models.

[Bradner 1997] Bradner, S. (1997, March). Key words for use in rfcs to indicate requirement levels. Technical Report RFC 2119, Network Working Group.

[Cattel and Barry 2000] Cattel, R. and D. K. Barry (2000). *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers.

[Clark and Warmer 2002] Clark, T. and J. Warmer (2002). *Object Modeling with the OCL : The Rationale behind the Object Constraint Language*. Springer.

[Codd 1970] Codd, E. F. (1970). A relational model of data for large shared data banks. *Commun. ACM 13*(6), 377–387.

[Collet and Vignola 2000] Collet, P. and G. Vignola (2000). Towards a consistent viewpoint on consistency for persistent applications. In *Objects and Databases*, pp. 47–60.

[Cook and Rosenberger 2005] Cook, W. R. and C. Rosenberger (2005). Native queries for persistent objects.

[Dmitriev 2001] Dmitriev, M. (2001, May). *Safe Class and Data Evolution in Large and Long-Lived Java Applications*. Ph. D. thesis, University of Glasgow.

[Elmasri and Navathe 2003] Elmasri, R. and S. B. Navathe (2003). *Fundamentals of Database Systems, 4th Edition*. Addison Wesley.

[Evans 1999] Evans, H. (1999). Why object serialization is inappropriate for providing persistence in java. Technical report.

[Franconi, Grandi, and Mandreoli 2000] Franconi, E., F. Grandi, and F. Mandreoli (2000). A semantic approach for schema evolution and versioning in object-oriented databases. *Lecture Notes in Computer Science 1861*, 1048–??

[GigaSpaces 2005] GigaSpaces (2005, November). http://www.gigaspaces.com/. website.

[Grehan 2005] Grehan, R. (2005, November). Open a s.o.d.a. web site, www.fawcette.com/javapro/2003_04/magazine/columns/javatogo/.

[Group 2005] Group, J. D. O. E. (2005, August). Java data objects 2.0: Proposed final draft. Technical Report JSR 243, Sun Microsystems Inc.

[Group 2004] Group, O. M. (2004, October). Unified modeling language specification. Version 2.0.

[Harrington 2000] Harrington, J. L. (2000). *Object-Oriented Database Design*. Morgan Kaufmann.

[Hibernate 2005] Hibernate (2005, November). Tools for hibernate 2.x, http://www.hibernate.org/256.html. website.

[IBM Research Centre San Jose 1978] IBM Research Centre San Jose, C. (1978). Ibm report on the contents of a sample of programs surveyed. Technical report, IBM.

[James Gosling 1996] James Gosling, Bill Joy, G. S. (1996). *The Java Language Specification*. Number ISBN 0-201-63451-1. 5F, No.7, Lane 50, Sec.3 Nan Kang Road Taipei, Taiwan: Addison-Wesley.

[Jordan 2004] Jordan, M. (2004, September). A comparative study of persistence mechanisms for the java platform. Technical Report SMLI TR-2004-136, Sun Microsystems Laboratories.

[Jordan and Atkinson 2000] Jordan, M. and M. Atkinson (2000, December). Orthogonal persistence for the java platform: Specification and rationale. Technical Report SMLI TR-2000-94, Sun Microsystems Laboratories.

[Lewis, Mathiske, and Gafter 2000] Lewis, B., B. Mathiske, and N. Gafter (2000, October). Architecture of the pevm: A high-performance orthogonally persistent java virtual machine. Technical Report SMLI TR-2000-93, Sun Microsystems Laboratories.

[Meyer 1992] Meyer, B. (1992). *Eiffel: the language*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

[Meyer 1997] Meyer, B. (1997). *Object-oriented Software Construction* (2nd ed.). Prentice Hall PTR.

[Microsoft 2002] Microsoft (2002). C sharp language specification. Technical report, Microsoft Corporation.

[Middlegen 2005] Middlegen (2005, November). http://boss.bekk.no/boss/middlegen/. website.

[Stonebraker and Moore 1996] Stonebraker, M. and D. Moore (1996). *Object-Relational DBMSs: The Next Great wave*. Morgan Kaufmann Publishers, Inc.

[Zloof 1977] Zloof, M. M. (1977). Query-by-example: A data base language. *IBM Systems Journal 16*(4), 324–343.

❖ A

Sample Application Source Code

Common POJO classes

package `ch.ethz.se.base.model`

Class `Account`

```
package ch.ethz.se.base.model;

public class Account {
    private long id;

    private Customer owner;

    private Branch branch;

    private float balance;

    private AccountType accountType;

    public float getBalance() {
        return balance;
    }

    public void setBalance(float balance) {
        this.balance = balance;
    }

    public Branch getBranch() {
        return branch;
    }
```

```
    public void setBranch(Branch branch) {
        this.branch = branch;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public Person getOwner() {
        return owner;
    }

    public void setOwner(Customer owner) {
        this.owner = owner;
    }

    public AccountType getAccountType() {
        return accountType;
    }

    public void setAccountType(AccountType type) {
        this.accountType = type;
    }
}
```

Class `AccountType`

```
package ch.ethz.se.base.model;

import java.util.Set;

public class AccountType {
    private long id;

    private String name;
```

```
    private Set<Account> accounts;

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String toString() {
        return "id: " + id + ", name: " + name;
    }

    public Set<Account> getAccounts() {
        return accounts;
    }

    public void setAccounts(Set<Account> accounts) {
        this.accounts = accounts;
    }
}
```

Class `Branch`

```
package ch.ethz.se.base.model;

import java.util.Set;

public class Branch {
    private long id;
```

```
    private String name;

    private Set<Account> accounts;

    private Set<Employee> employees;

    public Set<Employee> getEmployees() {
        return employees;
    }

    public void setEmployees(Set<Employee> employees) {
        this.employees = employees;
    }

    public Set<Account> getAccounts() {
        return accounts;
    }

    public void setAccounts(Set<Account> accounts) {
        this.accounts = accounts;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

Class `Customer`

```
package ch.ethz.se.base.model;

import java.util.Set;

public class Customer extends Person {
    private String type;

    private Set<Account> accounts;

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public Set<Account> getAccounts() {
        return accounts;
    }

    public void setAccounts(Set<Account> accounts) {
        this.accounts = accounts;
    }

}
```

Class `Employee`

```
package ch.ethz.se.base.model;

public class Employee extends Person {
    private float salary;

    private String position;

    private String workingDepartment;

    private Branch workingBranch;
```

```
    public String getPosition() {
        return position;
    }

    public void setPosition(String position) {
        this.position = position;
    }

    public float getSalary() {
        return salary;
    }

    public void setSalary(float salary) {
        this.salary = salary;
    }

    public Branch getWorkingBranch() {
        return workingBranch;
    }

    public void setWorkingBranch(Branch workingBranch) {
        this.workingBranch = workingBranch;
    }

    public String getWorkingDepartment() {
        return workingDepartment;
    }

    public void setWorkingDepartment(String workingDepartment) {
        this.workingDepartment = workingDepartment;
    }

}
```

Class `Person`

```
package ch.ethz.se.base.model;

import java.util.Date;

public class Person {
```

```
private long id;

private String name;

private Date birthdate;

private String gender;

private String nationality;

public String toString() {
    return "name: " + name + " gender: " + gender + " nationality: "
            + nationality + " nr of accounts";
}

public Date getBirthdate() {
    return birthdate;
}

public void setBirthdate(Date birthdate) {
    this.birthdate = birthdate;
}

public String getGender() {
    return gender;
}

public void setGender(String gender) {
    this.gender = gender;
}

public long getId() {
    return id;
}

public void setId(long id) {
    this.id = id;
}

public String getName() {
```

```
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getNationality() {
        return nationality;
    }

    public void setNationality(String nationality) {
        this.nationality = nationality;
    }

}
```

Test class implemented for Hibernate

```
import java.util.Calendar;
import java.util.Date;
import java.util.Iterator;
import java.util.List;

import org.hibernate.Session;
import org.hibernate.Transaction;

import ch.ethz.se.base.model.Account;
import ch.ethz.se.base.model.AccountType;
import ch.ethz.se.base.model.Branch;
import ch.ethz.se.base.model.Customer;
import ch.ethz.se.base.model.Employee;
import ch.ethz.se.base.model.Person;
import ch.ethz.se.hibernate.HibernateUtil;

public class EventManager {

    private void createAccountType(String type) {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();
```

```
    AccountType accType = new AccountType();

    accType.setName(type);

    session.save(accType);

    tx.commit();
    HibernateUtil.closeSession();
}

private void createPerson(String name, Date birthdate, String gender,
        String nationality) {

    Session session = HibernateUtil.currentSession();
    Transaction tx = session.beginTransaction();

    Person person = new Person();
    person.setGender(gender);
    person.setName(name);
    person.setNationality(nationality);
    person.setBirthdate(birthdate);

    session.save(person);

    tx.commit();
    HibernateUtil.closeSession();
}

private void createCustomer(String name, Date birthdate, String gender,
        String nationality, String type) {

    Session session = HibernateUtil.currentSession();
    Transaction tx = session.beginTransaction();

    Customer cust = new Customer();
    cust.setBirthdate(birthdate);
    cust.setGender(gender);
    cust.setName(name);
    cust.setNationality(nationality);
    cust.setType(type);
```

```
        session.save(cust);

        tx.commit();
        HibernateUtil.closeSession();

    }

    private void createBranch(String name) {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();

        Branch branch = new Branch();

        branch.setName(name);

        session.save(branch);

        tx.commit();
        HibernateUtil.closeSession();
    }

    private void createEmployee(String name, Date birthdate, String gender,
            String nationality, float salary, String position,
            String workingDepartment, Branch workingBranch) {
        Employee employee = new Employee();
        employee.setName(name);
        employee.setBirthdate(birthdate);
        employee.setGender(gender);
        employee.setNationality(nationality);
        employee.setSalary(salary);
        employee.setPosition(position);
        employee.setWorkingDepartment(workingDepartment);
        employee.setWorkingBranch(workingBranch);

        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();

        session.save(employee);
```

```
    tx.commit();
    HibernateUtil.closeSession();
}

private void createAccount(Customer owner, Branch branch, AccountType type) {

    Session session = HibernateUtil.currentSession();
    Transaction tx = session.beginTransaction();

    Account acc = new Account();
    acc.setBalance(0);
    acc.setBranch(branch);
    acc.setOwner(owner);
    acc.setAccountType(type);

    session.save(acc);

    tx.commit();
    HibernateUtil.closeSession();
}

private Branch getBranchByName(String name) {

    Session session = HibernateUtil.currentSession();
    Transaction tx = session.beginTransaction();

    Branch b = (Branch) session.createQuery(
            "FROM ch.ethz.se.Branch as branch " + "WHERE branch.name = ?")
            .setString(0, name).uniqueResult();

    tx.commit();
    HibernateUtil.closeSession();

    return b;
}

private Customer getCustomerByName(String name) {
    Session session = HibernateUtil.currentSession();
    Transaction tx = session.beginTransaction();
    Customer c = (Customer) session.createQuery(
```

```
                "FROM ch.ethz.se.Customer as customer "
                        + "WHERE customer.name = ?").setString(0, name)
                .uniqueResult();
        tx.commit();
        HibernateUtil.closeSession();

        return c;
    }

    private AccountType getAccountTypeByName(String name) {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();
        AccountType at = (AccountType) session.createQuery(
                "FROM ch.ethz.se.AccountType as at " + "WHERE at.name = ?")
                .setString(0, name).uniqueResult();
        tx.commit();
        HibernateUtil.closeSession();

        return at;
    }

    private AccountType getAccountType(String id) {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();

        AccountType b = (AccountType) session.load(AccountType.class, id);

        tx.commit();
        HibernateUtil.closeSession();

        return b;
    }

    private List<AccountType> listAccountTypes() {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();

        List<AccountType> result = session.createQuery("from AccountType")
                .list();
```

```
        tx.commit();
        HibernateUtil.closeSession();

        return result;
    }

    private List<Person> listPersons() {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();

        List result = session.createQuery("from Person").list();

        tx.commit();
        HibernateUtil.closeSession();

        return result;
    }

    private List<Customer> listCustomers() {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();

        List result = session.createQuery("from Customer").list();

        tx.commit();
        HibernateUtil.closeSession();

        return result;
    }

    private void addAccountTypeToAccount(AccountType accType, Account acc) {
        Session session = HibernateUtil.currentSession();
        Transaction tx = session.beginTransaction();

        accType.getAccounts().add(acc);

        tx.commit();
        HibernateUtil.closeSession();
    }
```

```
public static void main(String[] args) {
    EventManager main = new EventManager();

    main.createBranch("Zurich");
    main.createBranch("Bern");
    main.createBranch("Geneve");
    main.createBranch("Luzern");

    main.createAccountType("Savings Account");
    main.createAccountType("Checking Account");
    main.createAccountType("Loan Account");

    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.DAY_OF_MONTH, 26);
    cal.set(Calendar.MONTH, Calendar.FEBRUARY);
    cal.set(Calendar.YEAR, 1979);
    main
            .createPerson("Joao Tobias", cal.getTime(), "male",
                    "Brazilian");

    cal = Calendar.getInstance();
    cal.set(Calendar.DAY_OF_MONTH, 1);
    cal.set(Calendar.MONTH, Calendar.MARCH);
    cal.set(Calendar.YEAR, 1981);
    main.createCustomer("Ana Hickmann", cal.getTime(), "female",
            "Brazilian", "Exclusiv");

    cal = Calendar.getInstance();
    cal.set(Calendar.DAY_OF_MONTH, 4);
    cal.set(Calendar.MONTH, Calendar.JUNE);
    cal.set(Calendar.YEAR, 1975);
    main.createEmployee("Angelina Jolie", cal.getTime(), "female",
            "American", 100000, "Manager", "Remittance", main
                    .getBranchByName("Zurich"));

    cal = Calendar.getInstance();
    cal.set(Calendar.DAY_OF_MONTH, 16);
    cal.set(Calendar.MONTH, Calendar.FEBRUARY);
    cal.set(Calendar.YEAR, 1972);
    main.createEmployee("Sarah Clarke", cal.getTime(), "female",
```

```
        "American", 200000, "Director", "Remittance", main
                .getBranchByName("Zurich"));

cal = Calendar.getInstance();
cal.set(Calendar.DAY_OF_MONTH, 9);
cal.set(Calendar.MONTH, Calendar.JULY);
cal.set(Calendar.YEAR, 1956);
main.createEmployee("Tom Hanks", cal.getTime(), "male", "American",
        300000, "Director", "Marketing", main.getBranchByName("Bern"));

cal = Calendar.getInstance();
cal.set(Calendar.DAY_OF_MONTH, 21);
cal.set(Calendar.MONTH, Calendar.FEBRUARY);
cal.set(Calendar.YEAR, 1979);
main.createEmployee("Jennifer Love Hewitt", cal.getTime(), "female",
        "American", 200000, "Cashier", "Front-office", main
                .getBranchByName("Luzern"));

System.out.println("Employees working in Zurich");
for (Iterator itr = main.getBranchByName("Zurich").getEmployees()
        .iterator(); itr.hasNext();) {
    Employee emp = (Employee) itr.next();
    System.out.println("- " + emp.getName());
}
Customer ana = main.getCustomerByName("Ana Hickmann");

System.out.println(ana.getName() + " has " + ana.getAccounts().size()
        + " accounts");
for (Iterator itr = ana.getAccounts().iterator(); itr.hasNext();) {
    Account acc = (Account) itr.next();
    System.out.println("- " + acc.getAccountType().getName());
}

Branch zh = main.getBranchByName("Zurich");
System.out.println(zh.getName());

AccountType savings = main.getAccountTypeByName("Savings Account");
System.out.println(savings.getName());

main.createAccount(main.getCustomerByName("Ana Hickmann"), main
```

```
                .getBranchByName("Zurich"), main
                .getAccountTypeByName("Savings Account"));
        main.createAccount(main.getCustomerByName("Ana Hickmann"), main
                .getBranchByName("Zurich"), main
                .getAccountTypeByName("Checking Account"));
        main.createAccount(main.getCustomerByName("Ana Hickmann"), main
                .getBranchByName("Zurich"), main
                .getAccountTypeByName("Loan Account"));

        ana = main.getCustomerByName("Ana Hickmann");
        System.out.println(ana.getName() + " has " + ana.getAccounts().size()
                + " accounts");
        for (Iterator itr = ana.getAccounts().iterator(); itr.hasNext();) {
            Account acc = (Account) itr.next();
            System.out.println("- " + acc.getAccountType().getName());
        }

        HibernateUtil.sessionFactory.close();
    }

}
```

The test class implemented for JDO

```
package ch.ethz.se.jpox;

import java.io.IOException;
import java.io.InputStream;
import java.util.Calendar;
import java.util.Date;
import java.util.Iterator;
import java.util.Properties;

import javax.jdo.JDOHelper;
import javax.jdo.PersistenceManager;
import javax.jdo.PersistenceManagerFactory;
import javax.jdo.Query;
import javax.jdo.Transaction;

import ch.ethz.se.base.model.Account;
import ch.ethz.se.base.model.AccountType;
```

```
import ch.ethz.se.base.model.Branch;
import ch.ethz.se.base.model.Customer;
import ch.ethz.se.base.model.Employee;

public class JPOXMain {
    PersistenceManager pm;

    public JPOXMain(String pathToConfigurationFile) {
        InputStream in = JPOXMain.class.getClass().getResourceAsStream(
                pathToConfigurationFile);
        Properties props = new Properties();

        try {
            props.load(in);
            PersistenceManagerFactory pmf = JDOHelper
                    .getPersistenceManagerFactory(props);
            pm = pmf.getPersistenceManager();
            pm.getFetchPlan().addGroup("accounts_field");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    private void createCustomer(String name, Date birthdate, String gender,
            String nationality, String type) {
        Customer customer = new Customer();
        customer.setName(name);
        customer.setBirthdate(birthdate);
        customer.setGender(gender);
        customer.setNationality(nationality);
        customer.setType(type);

        Transaction tx = pm.currentTransaction();
        try {
            tx.begin();

            pm.makePersistent(customer);

            tx.commit();
        } finally {
```

```
            if (tx.isActive()) {
                tx.rollback();
            }
        }
    }

    private void createEmployee(String name, Date birthdate, String gender,
            String nationality, float salary, String position,
            String workingDepartment, Branch workingBranch) {
        Employee employee = new Employee();
        employee.setName(name);
        employee.setBirthdate(birthdate);
        employee.setGender(gender);
        employee.setNationality(nationality);
        employee.setSalary(salary);
        employee.setPosition(position);
        employee.setWorkingDepartment(workingDepartment);
        employee.setWorkingBranch(workingBranch);

        Transaction tx = pm.currentTransaction();
        try {
            tx.begin();

            pm.makePersistent(employee);
            tx.commit();
        } finally {
            if (tx.isActive()) {
                tx.rollback();
            }
        }
    }

    public void createBranch(String name) {
        Branch branch = new Branch();
        branch.setName(name);

        Transaction tx = pm.currentTransaction();
        try {
            tx.begin();
```

```
        pm.makePersistent(branch);
        tx.commit();
    } finally {
        if (tx.isActive()) {
            tx.rollback();
        }
    }
}

public void createAccountType(String name) {
    AccountType accountType = new AccountType();
    accountType.setName(name);

    Transaction tx = pm.currentTransaction();
    try {
        tx.begin();

        pm.makePersistent(accountType);
        tx.commit();
    } finally {
        if (tx.isActive()) {
            tx.rollback();
        }
    }
}

public void createAccount(Customer owner, Branch branch, AccountType type) {
    Account account = new Account();
    account.setOwner(owner);
    account.setBranch(branch);
    account.setBalance(0);
    account.setAccountType(type);

    Transaction tx = pm.currentTransaction();
    try {
        tx.begin();

        pm.makePersistent(account);

        tx.commit();
```

```
        } finally {
            if (tx.isActive()) {
                tx.rollback();
            }
        }
    }

    public Branch getBranchByName(String name) {
        Transaction tx = pm.currentTransaction();
        tx.begin();
        Query query = pm
                .newQuery("SELECT FROM ch.ethz.se.jpox.persistent.Branch "
                        + "WHERE name == \"" + name + "\"");
        query.setUnique(true);

        Branch branch = (Branch) query.execute();
        tx.commit();

        return branch;
    }

    public Customer getCustomerByName(String name) {

        Transaction tx = pm.currentTransaction();
        tx.begin();
        Query query = pm
                .newQuery("SELECT FROM ch.ethz.se.jpox.persistent.Customer "
                        + "WHERE name == \"" + name + "\"");
        query.setUnique(true);

        Customer customer = (Customer) query.execute();

        tx.commit();

        return customer;
    }

    public AccountType getAccountTypeByName(String name) {
        Transaction tx = pm.currentTransaction();
        tx.begin();
```

```
    Query query = pm
            .newQuery("SELECT FROM ch.ethz.se.jpox.persistent.AccountType "
                    + "WHERE name == \"" + name + "\"");
    query.setUnique(true);

    AccountType accountType = (AccountType) query.execute();
    tx.commit();

    return accountType;
}

private void close() {
    if (!pm.isClosed()) {
        pm.close();
    }
}

public static void main(String[] args) {
    JPOXMain main = new JPOXMain("/jdo.properties");

    main.createBranch("Zurich");
    main.createBranch("Bern");
    main.createBranch("Geneve");
    main.createBranch("Luzern");

    main.createAccountType("Savings Account");
    main.createAccountType("Checking Account");
    main.createAccountType("Loan Account");

    Calendar cal = Calendar.getInstance();

    cal = Calendar.getInstance();
    cal.set(Calendar.DAY_OF_MONTH, 1);
    cal.set(Calendar.MONTH, Calendar.MARCH);
    cal.set(Calendar.YEAR, 1981);
    main.createCustomer("Ana Hickmann", cal.getTime(), "female",
            "Brazilian", "Exclusiv");

    cal = Calendar.getInstance();
    cal.set(Calendar.DAY_OF_MONTH, 4);
```

```
cal.set(Calendar.MONTH, Calendar.JUNE);
cal.set(Calendar.YEAR, 1975);
main.createEmployee("Angelina Jolie", cal.getTime(), "female",
        "American", 100000, "Manager", "Remittance", main
                .getBranchByName("Zurich"));

cal = Calendar.getInstance();
cal.set(Calendar.DAY_OF_MONTH, 16);
cal.set(Calendar.MONTH, Calendar.FEBRUARY);
cal.set(Calendar.YEAR, 1972);
main.createEmployee("Sarah Clarke", cal.getTime(), "female",
        "American", 200000, "Director", "Remittance", main
                .getBranchByName("Zurich"));

cal = Calendar.getInstance();
cal.set(Calendar.DAY_OF_MONTH, 21);
cal.set(Calendar.MONTH, Calendar.FEBRUARY);
cal.set(Calendar.YEAR, 1979);
main.createEmployee("Jennifer Love Hewitt", cal.getTime(), "female",
        "American", 200000, "Cashier", "Front-office", main
                .getBranchByName("Zurich"));

cal = Calendar.getInstance();
cal.set(Calendar.DAY_OF_MONTH, 9);
cal.set(Calendar.MONTH, Calendar.JULY);
cal.set(Calendar.YEAR, 1956);
main.createEmployee("Tom Hanks", cal.getTime(), "male", "American",
        200000, "Director", "Marketing", main.getBranchByName("Bern"));

System.out.println("Employees working in Zurich");
for (Iterator itr = main.getBranchByName("Zurich").getEmployees()
        .iterator(); itr.hasNext();) {
    Employee emp = (Employee) itr.next();
    System.out.println("- " + emp.getName());
}

main.createAccount(main.getCustomerByName("Ana Hickmann"), main
        .getBranchByName("Zurich"), main
        .getAccountTypeByName("Savings Account"));
main.createAccount(main.getCustomerByName("Ana Hickmann"), main
```

```
                .getBranchByName("Zurich"), main
                .getAccountTypeByName("Checking Account"));
        main.createAccount(main.getCustomerByName("Ana Hickmann"), main
                .getBranchByName("Zurich"), main
                .getAccountTypeByName("Loan Account"));

        Customer ana = main.getCustomerByName("Ana Hickmann");

        System.out.println(ana.getName() + " has " + ana.getAccounts().size()
                + " accounts");
        for (Iterator itr = ana.getAccounts().iterator(); itr.hasNext();) {
            Account acc = (Account) itr.next();
            System.out.println("- " + acc.getAccountType().getName());
        }
        main.close();
    }
}
```

The test class for TopLink

```
package ch.ethz.se.toplink;

import java.util.Calendar;
import java.util.Date;
import java.util.Iterator;
import java.util.List;
import java.util.Vector;

import ch.ethz.se.model.Account;
import ch.ethz.se.model.AccountType;
import ch.ethz.se.model.Branch;
import ch.ethz.se.model.Customer;
import ch.ethz.se.model.Employee;
import ch.ethz.se.model.Person;
import oracle.toplink.expressions.Expression;
import oracle.toplink.expressions.ExpressionBuilder;
import oracle.toplink.sessions.UnitOfWork;
import oracle.toplink.threetier.ClientSession;
import oracle.toplink.threetier.Server;
import oracle.toplink.tools.schemaframework.SchemaManager;
import oracle.toplink.tools.sessionmanagement.SessionManager;
```

```
public class Main {
    ClientSession session;

    SessionManager sm;

    public Main() {
        sm = SessionManager.getManager();
        Server serverSession = (Server) sm.getSession("BankSession");
        session = serverSession.acquireClientSession();
        SchemaManager schemaManager = new SchemaManager(serverSession);
        schemaManager.createSequences();
        session.logMessages();
    }

    private void createAccountType(String type) {
        UnitOfWork uow = session.acquireUnitOfWork();

        AccountType accType = new AccountType();
        AccountType accTypeClone = (AccountType) uow.registerObject(accType);
        accTypeClone.setName(type);

        uow.commitAndResume();

    }

    private void createPerson(String name, Date birthdate, String gender,
            String nationality) {

        UnitOfWork uow = session.acquireUnitOfWork();

        Person person = new Person();
        Person personClone = (Person) uow.registerObject(person);

        personClone.setGender(gender);
        personClone.setName(name);
        personClone.setNationality(nationality);
        personClone.setBirthdate(birthdate);

        uow.commitAndResume();
```

```
}

private void createCustomer(String name, Date birthdate, String gender,
        String nationality, String type) {

    UnitOfWork uow = session.acquireUnitOfWork();

    Customer cust = new Customer();
    Customer custClone = (Customer) uow.registerObject(cust);
    custClone.setBirthdate(birthdate);
    custClone.setGender(gender);
    custClone.setName(name);
    custClone.setNationality(nationality);
    custClone.setType(type);

    uow.commitAndResume();

}

private void createBranch(String name) {
    UnitOfWork uow = session.acquireUnitOfWork();

    Branch branch = new Branch();
    Branch branchClone = (Branch) uow.registerObject(branch);
    branchClone.setName(name);

    uow.commitAndResume();
}

private void createEmployee(String name, Date birthdate, String gender,
        String nationality, float salary, String position,
        String workingDepartment, Branch workingBranch) {
    UnitOfWork uow = session.acquireUnitOfWork();

    Employee employee = new Employee();
    Employee employeeClone = (Employee) uow.registerObject(employee);
    Branch branchClone = (Branch) uow.registerObject(workingBranch);
    uow.validateObjectSpace();

    employeeClone.setName(name);
```

```
        employeeClone.setBirthdate(birthdate);
        employeeClone.setGender(gender);
        employeeClone.setNationality(nationality);
        employeeClone.setSalary(salary);
        employeeClone.setPosition(position);
        employeeClone.setWorkingDepartment(workingDepartment);
        employeeClone.setWorkingBranch(branchClone);

        uow.commitAndResume();
    }

    private void createAccount(Customer owner, Branch branch, AccountType type) {

        UnitOfWork uow = session.acquireUnitOfWork();

        Account acc = new Account();
        Account accClone = (Account) uow.registerObject(acc);
        Customer customerClone = (Customer) uow.registerObject(owner);
        Branch branchClone = (Branch) uow.registerObject(branch);
        AccountType accTypeClone = (AccountType) uow.registerObject(type);

        accClone.setBalance(0);
        accClone.setBranch(branchClone);
        accClone.setOwner(customerClone);
        accClone.setAccountType(accTypeClone);

        uow.commitAndResume();

    }

    private Branch getBranchByName(String name) {
        UnitOfWork uow = session.acquireUnitOfWork();

        ExpressionBuilder eb = new ExpressionBuilder();
        Expression exp = eb.get("name").equal(name);
        Vector branches = session.readAllObjects(Branch.class, exp);

        Branch b = null;
        for (Iterator itr = branches.iterator(); itr.hasNext();) {
            b = (Branch) itr.next();
```

```
    }

    uow.commitAndResume();

    return b;
}

private Customer getCustomerByName(String name) {
    UnitOfWork uow = session.acquireUnitOfWork();

    ExpressionBuilder eb = new ExpressionBuilder();
    Expression exp = eb.get("name").equal(name);
    Vector customers = session.readAllObjects(Customer.class, exp);

    Customer c = null;
    for (Iterator itr = customers.iterator(); itr.hasNext();) {
        c = (Customer) itr.next();
    }

    uow.commitAndResume();
    return c;
}

private AccountType getAccountTypeByName(String name) {

    UnitOfWork uow = session.acquireUnitOfWork();
    ExpressionBuilder eb = new ExpressionBuilder();
    Expression exp = eb.get("name").equal(name);
    Vector accTypes = session.readAllObjects(AccountType.class, exp);

    AccountType at = null;
    for (Iterator itr = accTypes.iterator(); itr.hasNext();) {
        at = (AccountType) itr.next();
    }

    uow.commitAndResume();

    return at;
}
```

```
private AccountType getAccountType(long id) {
    UnitOfWork uow = session.acquireUnitOfWork();

    ExpressionBuilder eb = new ExpressionBuilder();
    Expression exp = eb.get("id").equal(id);
    Vector accTypes = session.readAllObjects(AccountType.class, exp);

    AccountType at = null;
    for (Iterator itr = accTypes.iterator(); itr.hasNext();) {
        at = (AccountType) itr.next();
    }

    uow.commitAndResume();
    return at;
}

private List<AccountType> listAccountTypes() {
    UnitOfWork uow = session.acquireUnitOfWork();

    Vector accTypes = session.readAllObjects(AccountType.class);

    uow.commitAndResume();
    return accTypes;
}

private List<Person> listPersons() {
    UnitOfWork uow = session.acquireUnitOfWork();

    Vector persons = session.readAllObjects(Person.class);

    uow.commitAndResume();
    return persons;
}

private List<Customer> listCustomers() {
    UnitOfWork uow = session.acquireUnitOfWork();

    Vector customers = session.readAllObjects(Customer.class);

    uow.commitAndResume();
```

```
    return customers;
}

private void addAccountTypeToAccount(AccountType accType, Account acc) {
    UnitOfWork uow = session.acquireUnitOfWork();

}

private void close() {
    sm.destroyAllSessions();
}

public static void main(String[] args) {
    Main main = new Main();
    System.out.println("Creating branches");
    main.createBranch("Zurich");
    main.createBranch("Bern");
    main.createBranch("Geneve");
    main.createBranch("Luzern");

    System.out.println("Creating Account types");
    main.createAccountType("Savings Account");
    main.createAccountType("Checking Account");
    main.createAccountType("Loan Account");

    System.out.println("Creating Person - Joao");
    Calendar cal = Calendar.getInstance();
    cal.set(Calendar.DAY_OF_MONTH, 26);
    cal.set(Calendar.MONTH, Calendar.FEBRUARY);
    cal.set(Calendar.YEAR, 1979);
    main
            .createPerson("Joao Tobias", cal.getTime(), "male",
                    "Brazilian");

    System.out.println("Creating Customer Ana Hickmann");
    cal = Calendar.getInstance();
    cal.set(Calendar.DAY_OF_MONTH, 1);
    cal.set(Calendar.MONTH, Calendar.MARCH);
    cal.set(Calendar.YEAR, 1981);
    main.createCustomer("Ana Hickmann", cal.getTime(), "female",
```

```
                "Brazilian", "Exclusiv");

        System.out.println("Creating Employee - Angelina Jolie");
        cal = Calendar.getInstance();
        cal.set(Calendar.DAY_OF_MONTH, 4);
        cal.set(Calendar.MONTH, Calendar.JUNE);
        cal.set(Calendar.YEAR, 1975);
        main.createEmployee("Angelina Jolie", cal.getTime(), "female",
                "American", 100000, "Manager", "Remittance", main
                        .getBranchByName("Zurich"));

        System.out.println("Creating Employee - Sarah Clarke");
        cal = Calendar.getInstance();
        cal.set(Calendar.DAY_OF_MONTH, 16);
        cal.set(Calendar.MONTH, Calendar.FEBRUARY);
        cal.set(Calendar.YEAR, 1972);
        main.createEmployee("Sarah Clarke", cal.getTime(), "female",
                "American", 200000, "Director", "Remittance", main
                        .getBranchByName("Zurich"));

        System.out.println("Creating Employee - Tom Hanks");
        cal = Calendar.getInstance();
        cal.set(Calendar.DAY_OF_MONTH, 9);
        cal.set(Calendar.MONTH, Calendar.JULY);
        cal.set(Calendar.YEAR, 1956);
        main.createEmployee("Tom Hanks", cal.getTime(), "male", "American",
                300000, "Director", "Marketing", main.getBranchByName("Bern"));

        System.out.println("Creating Employee - Jennifer Love Hewitt");
        cal = Calendar.getInstance();
        cal.set(Calendar.DAY_OF_MONTH, 21);
        cal.set(Calendar.MONTH, Calendar.FEBRUARY);
        cal.set(Calendar.YEAR, 1979);
        main.createEmployee("Jennifer Love Hewitt", cal.getTime(), "female",
                "American", 200000, "Cashier", "Front-office", main
                        .getBranchByName("Luzern"));

        System.out.println("Employees working in Zurich");
        for (Iterator itr = main.getBranchByName("Zurich").getEmployees()
                .iterator(); itr.hasNext();) {
```

```
    Employee emp = (Employee) itr.next();
    System.out.println("- " + emp.getName());
}
Customer ana = main.getCustomerByName("Ana Hickmann");

System.out.println(ana.getName() + " has " + ana.getAccounts().size()
        + " accounts");
for (Iterator itr = ana.getAccounts().iterator(); itr.hasNext();) {
    Account acc = (Account) itr.next();
    System.out.println("- " + acc.getAccountType().getName());
}

Branch zh = main.getBranchByName("Zurich");
System.out.println(zh.getName());

AccountType savings = main.getAccountTypeByName("Savings Account");
System.out.println(savings.getName());

System.out.println("Creating Savings Account for Ana");
main.createAccount(main.getCustomerByName("Ana Hickmann"), main
        .getBranchByName("Zurich"), main
        .getAccountTypeByName("Savings Account"));

System.out.println("Creating Checking Account for Ana");
main.createAccount(main.getCustomerByName("Ana Hickmann"), main
        .getBranchByName("Zurich"), main
        .getAccountTypeByName("Checking Account"));

System.out.println("Creating Loan Account for Ana");
main.createAccount(main.getCustomerByName("Ana Hickmann"), main
        .getBranchByName("Zurich"), main
        .getAccountTypeByName("Loan Account"));

ana = main.getCustomerByName("Ana Hickmann");
System.out.println(ana.getName() + " has " + ana.getAccounts().size()
        + " accounts");
for (Iterator itr = ana.getAccounts().iterator(); itr.hasNext();) {
    Account acc = (Account) itr.next();
    System.out.println("- " + acc.getAccountType().getName());
}
```

```
            main.close();
        }
    }
```

Classes for the Prevayler implementation of the Sample example.

package `ch.ethz.prevayler.persistent`

Class `Account`

```
package ch.ethz.prevayler.persistent;

import java.io.Serializable;

public class Account implements Serializable {
    private long id;

    private Person owner;

    private Branch branch;

    private float balance;

    private AccountType type;

    public float getBalance() {
        return balance;
    }

    public void setBalance(float balance) {
        this.balance = balance;
    }

    public Branch getBranch() {
        return branch;
    }

    public void setBranch(Branch branch) {
        this.branch = branch;
    }
```

```
    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public Person getOwner() {
        return owner;
    }

    public void setOwner(Person owner) {
        this.owner = owner;
    }

    public AccountType getType() {
        return type;
    }

    public void setType(AccountType type) {
        this.type = type;
    }

}
```

Class `AccountType`

```
package ch.ethz.prevayler.persistent;

import java.io.Serializable;

public class AccountType implements Serializable {
    private String id;

    private String name;

    public String getId() {
        return id;
    }
```

```
    public void setId(String id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

}
```

Class Branch

```
package ch.ethz.prevayler.persistent;

import java.io.Serializable;

public class Branch implements Serializable {
private String id;
private String name;

public String getId() {
return id;
}
public void setId(String id) {
this.id = id;
}
public String getName() {
return name;
}
public void setName(String name) {
this.name = name;
}
}
```

Class Customer

```
package ch.ethz.prevayler.persistent;

public class Customer extends Person {
```

```
    private static final long serialVersionUID = -3503432363029608373L;

    private String type;

    public Customer() {
        super();
    }

    @Override
    public String toString() {
        return super.toString();
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }
}
```

Class `Employee`

```
package ch.ethz.prevayler.persistent;

public class Employee extends Person {

    private static final long serialVersionUID = -4173430077944133643L;

    private float salary;

    private String position;

    private String workingDepartment;

    private Branch workingBranch;

    public String getPosition() {
        return position;
```

```
    }

    public void setPosition(String position) {
        this.position = position;
    }

    public float getSalary() {
        return salary;
    }

    public void setSalary(float salary) {
        this.salary = salary;
    }

    public Branch getWorkingBranch() {
        return workingBranch;
    }

    public void setWorkingBranch(Branch workingBranch) {
        this.workingBranch = workingBranch;
    }

    public String getWorkingDepartment() {
        return workingDepartment;
    }

    public void setWorkingDepartment(String workingDepartment) {
        this.workingDepartment = workingDepartment;
    }

}
```

Class `Person`

```
package ch.ethz.prevayler.persistent;

import java.io.Serializable;
import java.util.Date;

public class Person implements Serializable {
```

```
private static final long serialVersionUID = -3423659509898047345L;

private long id;

private String name;

private Date birthdate;

private String gender;

private String nationality;

public String toString() {
    return "name: " + name + "\nid: " + id + "\ngender: " + gender
            + "\nnationality: " + nationality + "\n";
}

public Date getBirthdate() {
    return birthdate;
}

public void setBirthdate(Date birthdate) {
    this.birthdate = birthdate;
}

public String getGender() {
    return gender;
}

public void setGender(String gender) {
    this.gender = gender;
}

public long getId() {
    return id;
}

public void setId(long id) {
    this.id = id;
}
```

```
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getNationality() {
        return nationality;
    }

    public void setNationality(String nationality) {
        this.nationality = nationality;
    }

}
```

Command classes for Prevayler:

```
package ch.ethz.prevayler.bo;

import java.util.Date;

import org.prevayler.TransactionWithQuery;

import ch.ethz.prevayler.BankingSystem;
import ch.ethz.prevayler.persistent.AccountType;

public class CreateAccountType implements TransactionWithQuery {
    AccountType accType;

    public CreateAccountType(String id, String name) {
        accType = new AccountType();
        accType.setId(id);
        accType.setName(name);
    }

    public Object executeAndQuery(Object prevalentSystem, Date executionTime)
            throws Exception {
```

```
        return ((BankingSystem) prevalentSystem).addAccountType(accType);
    }

}

-

package ch.ethz.prevayler.bo;

import java.util.Date;

import org.prevayler.TransactionWithQuery;

import ch.ethz.prevayler.BankingSystem;
import ch.ethz.prevayler.persistent.Branch;

public class CreateBranch implements TransactionWithQuery {
    Branch branch;

    public CreateBranch(String id, String name) {
        branch = new Branch();
        branch.setId(id);
    }

    public Object executeAndQuery(Object prevalentSystem, Date executionTime)
            throws Exception {
        Branch b = ((BankingSystem) prevalentSystem).addBranch(branch);
        return b;
    }

}

-

package ch.ethz.prevayler.bo;

import java.util.Date;

import org.prevayler.TransactionWithQuery;
```

```
import ch.ethz.prevayler.BankingSystem;
import ch.ethz.prevayler.persistent.Customer;

public class CreateCustomer implements TransactionWithQuery {
    Customer cust;

    public CreateCustomer(String name, Date birthdate, String gender,
            String nationality, String type) {
        cust = new Customer();
        cust.setName(name);
        cust.setBirthdate(birthdate);
        cust.setGender(gender);
        cust.setNationality(nationality);
        cust.setType(type);
    }

    public Object executeAndQuery(Object prevalentSystem, Date executionTime)
            throws Exception {
        return ((BankingSystem) prevalentSystem).addCustomer(cust);
    }

}

-

package ch.ethz.prevayler.bo;

import java.util.Date;

import org.prevayler.TransactionWithQuery;

import ch.ethz.prevayler.BankingSystem;
import ch.ethz.prevayler.persistent.Branch;
import ch.ethz.prevayler.persistent.Employee;

public class CreateEmployee implements TransactionWithQuery {
    Employee emp;

    public CreateEmployee(String name, Date birthdate, String gender,
            String nationality, float salary, String position,
```

```
            String workingDepartment, Branch workingBranch) {
        emp = new Employee();
        emp.setName(name);
        emp.setBirthdate(birthdate);
        emp.setGender(gender);
        emp.setNationality(nationality);
        emp.setSalary(salary);
        emp.setPosition(position);
        emp.setWorkingDepartment(workingDepartment);
        emp.setWorkingBranch(workingBranch);
    }

    public Object executeAndQuery(Object prevalentSystem, Date executionTime)
            throws Exception {
        return ((BankingSystem) prevalentSystem).addEmployee(emp);
    }

}
```

-

```
package ch.ethz.prevayler.bo;

import java.util.Date;

import org.prevayler.TransactionWithQuery;

import ch.ethz.prevayler.BankingSystem;
import ch.ethz.prevayler.persistent.Person;

public class CreatePerson implements TransactionWithQuery {
    private Person person;

    public CreatePerson(String name, Date birthdate, String gender,
            String nationality) {
        person = new Person();
        person.setName(name);
        person.setBirthdate(birthdate);
        person.setGender(gender);
        person.setNationality(nationality);
```

```
        }

        public Object executeAndQuery(Object prevalentSystem, Date executionTime)
                throws Exception {
            Person p = ((BankingSystem) prevalentSystem).addPerson(person);
            return p;
        }

    }
-
```

The Main class which controls the snapshot.

```
    package ch.ethz.prevayler;

    import java.io.IOException;

    import org.prevayler.Prevayler;
    import org.prevayler.PrevaylerFactory;

    public class Main {
        public final static String DATA_FOLDER = "data";

        public final static long SNAPSHOT_INTERVAL = 500;

        public static void main(String[] args) {
            try {
                final Prevayler prevayler = PrevaylerFactory.createPrevayler(
                        new BankingSystem(), DATA_FOLDER);
                Thread snapShotThread = new Thread() {
                    public void run() {
                        System.out.println("Starting snapshot system...");
                        while (true) {
                            try {
                                Thread.sleep(SNAPSHOT_INTERVAL);
                                prevayler.takeSnapshot();
                            } catch (InterruptedException e) {

                                e.printStackTrace();
                            } catch (IOException e) {
```

```
                            e.printStackTrace();
                        }

                    }
                }
            };

            snapShotThread.start();
        } catch (IOException e) {

            e.printStackTrace();
        } catch (ClassNotFoundException e) {

            e.printStackTrace();
        }
    }
}

-
```

The prevalent system

```
package ch.ethz.prevayler;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

import ch.ethz.prevayler.persistent.Account;
import ch.ethz.prevayler.persistent.AccountType;
import ch.ethz.prevayler.persistent.Branch;
import ch.ethz.prevayler.persistent.Customer;
import ch.ethz.prevayler.persistent.Employee;
import ch.ethz.prevayler.persistent.Person;
import ch.ethz.prevayler.persistent.Transaction;
import ch.ethz.prevayler.persistent.TransactionType;

public class BankingSystem implements Serializable {
    private List<Person> persons = new ArrayList<Person>();
```

```
    private long nextPersonId = 1;

    private List<Branch> branches = new ArrayList<Branch>();

    private List<AccountType> accountTypes = new ArrayList<AccountType>();

    private List<Customer> customers = new ArrayList<Customer>();

    private long nextCustomerId = 1;

    private List<Employee> employees = new ArrayList<Employee>();

    private long nextEmployeeId = 1;

    private List<TransactionType> transactionTypes = new ArrayList<TransactionType>()

    private List<Transaction> transactions = new ArrayList<Transaction>();

    private long nextTransactionId = 1;

    private List<Account> accounts = new ArrayList<Account>();

    private long nextAccountId = 1;

    public Person addPerson(Person person) {
        person.setId(nextPersonId++);
        persons.add(person);
        return person;
    }

    public Branch addBranch(Branch branch) {
        branches.add(branch);
        return branch;
    }

    public AccountType addAccountType(AccountType accType) {
        accountTypes.add(accType);
        return accType;
    }
```

```
public Customer addCustomer(Customer cust) {
    cust.setId(nextCustomerId);
    customers.add(cust);
    return cust;
}

public Employee addEmployee(Employee emp) {
    emp.setId(nextEmployeeId++);
    employees.add(emp);
    return emp;
}

public TransactionType addTransactionType(TransactionType transType) {
    transactionTypes.add(transType);
    return transType;
}

public Transaction addTransaction(Transaction trans) {
    trans.setId(nextTransactionId++);
    transactions.add(trans);
    return trans;
}

public Account addAccount(Account account) {
    account.setId(nextAccountId++);
    accounts.add(account);
    return account;
}

/*****************************************************************************
 * ********************Getters and Setters**********************
 *****************************************************************************/

public List<Person> getPersons() {
    return persons;
}

public long getNextPersonId() {
    return nextPersonId;
}
```

```
    public List<Branch> getBranches() {
        return branches;
    }

    public List<AccountType> getAccountTypes() {
        return accountTypes;
    }

    public List<Customer> getCustomers() {
        return customers;
    }

    public long getNextCustomerId() {
        return nextCustomerId;
    }

    public List<Employee> getEmployees() {
        return employees;
    }

    public long getNextEmployeeId() {
        return nextEmployeeId;
    }

    public List<TransactionType> getTransactionTypes() {
        return transactionTypes;
    }

    public List<Account> getAccounts() {
        return accounts;
    }
}

-
```

Storing sample data.

```
package ch.ethz.prevayler;

import java.io.IOException;
```

```
import java.util.Calendar;
import java.util.GregorianCalendar;

import org.prevayler.Prevayler;
import org.prevayler.PrevaylerFactory;

import sun.util.calendar.CalendarDate;
import sun.util.calendar.CalendarUtils;

import ch.ethz.prevayler.bo.CreateAccountType;
import ch.ethz.prevayler.bo.CreateBranch;
import ch.ethz.prevayler.bo.CreateCustomer;
import ch.ethz.prevayler.bo.CreateEmployee;
import ch.ethz.prevayler.bo.CreatePerson;
import ch.ethz.prevayler.bo.CreateTransactionType;
import ch.ethz.prevayler.persistent.Branch;

public class Store {
    public static final String DATA_FOLDER = "data";

    public static void main(String[] args) {
        try {
            final Prevayler prevayler = PrevaylerFactory.createPrevayler(
                    new BankingSystem(), DATA_FOLDER);
            Calendar cal = GregorianCalendar.getInstance();
            cal.set(Calendar.DATE, 26);
            cal.set(Calendar.MONTH, Calendar.FEBRUARY);
            cal.set(Calendar.YEAR, 1979);
            prevayler.execute(new CreatePerson("Joao Tobias",
                    cal.getTime(), "male", "Brazilian"));

            prevayler.execute(new CreatePerson("Ana Hickmann", cal.getTime(),
                    "female", "Brazilian"));

            prevayler.execute(new CreateAccountType("PLA", "Platina"));
            prevayler.execute(new CreateBranch("ZH", "Zurich"));

            prevayler.execute(new CreateTransactionType("WI", "Withdraw"));
            prevayler.execute(new CreateTransactionType("TR", "Transfer"));
```

```
            prevayler.execute(new CreateCustomer("Angelina Jolie", cal
                    .getTime(), "female", "American", "Exclusiv"));

            Branch b1 = (Branch) prevayler.execute(new CreateBranch("LA",
                    "Los Angeles"));
            prevayler.execute(new CreateEmployee("Jack Bauer", cal.getTime(),
                    "male", "American", 2000000, "Federal Agent", "CTU", b1));

        } catch (ClassNotFoundException cnfe) {
            cnfe.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

–

Retrieving data

```
package ch.ethz.prevayler;

import java.io.IOException;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Iterator;
import java.util.List;

import org.prevayler.Prevayler;
import org.prevayler.PrevaylerFactory;

import ch.ethz.prevayler.bo.CreatePerson;
import ch.ethz.prevayler.persistent.Customer;
import ch.ethz.prevayler.persistent.Person;

public class Retrieve {
    public static final String DATA_FOLDER = "data";

    public static void main(String[] args) {
```

```
        try {
            final Prevayler prevayler = PrevaylerFactory.createPrevayler(
                    new BankingSystem(), DATA_FOLDER);
            List<Person> list = ((BankingSystem) prevayler.prevalentSystem())
                    .getPersons();

            for (Iterator itr = list.iterator(); itr.hasNext();) {
                Person person = (Person) itr.next();
                System.out.println(person.toString());
            }

            List<Customer> cList = ((BankingSystem) prevayler.prevalentSystem())
                    .getCustomers();

            for (Iterator itr = cList.iterator(); itr.hasNext();) {
                Customer customer = (Customer) itr.next();
                System.out.println(customer.toString());
            }
        } catch (ClassNotFoundException cnfe) {
            cnfe.printStackTrace();
        } catch (IOException ioe) {
            ioe.printStackTrace();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

The source code of the sample application implemented using **Db4o**

```
using System;

namespace ch.ethz.se
{
  public class Account
  {
    private long id;
    private Customer owner;
    private Branch branch;
    private float balance;
    private AccountType type;
```

```
public Account()
{
}

public long Id
{
  get
  {
    return id;
  }
  set
  {
    id = value;
  }
}

public Customer Owner
{
  get
  {
    return owner;
  }
  set
  {
    owner = value;
  }
}

public Branch Branch
{
  get
  {
    return branch;
  }
  set
  {
    branch = value;
  }
}
```

```
    public float Balance
    {
      get
      {
        return balance;
      }
      set
      {
        balance = value;
      }
    }

    public AccountType Type
    {
      get
      {
        return type;
      }
      set
      {
        type = value;
      }
    }
  }
}

using System;

namespace ch.ethz.se
{
  public class AccountType
  {
    private string id;
    private string name;

    public AccountType()
    {
    }

    public string Name
```

```
    {
      get
      {
        return name;
      }
      set
      {
        name = value;
      }
    }

    public string Id
    {
      get
      {
        return id;
      }
      set
      {
        id = value;
      }
    }
  }
}

using System;
using System.Collections;

namespace ch.ethz.se
{
  public class Branch
  {
    private String id;
    private String name;
    private ArrayList employees = new ArrayList();

    public Branch()
    {
    }
```

```
    public string Name
    {
      get
      {
        return name;
      }
      set
      {
        name = value;
      }
    }

    public string Id
    {
      get
      {
        return id;
      }
      set
      {
        id = value;
      }
    }

    public ArrayList Employees
    {
      get
      {
        return employees;
      }
    }
  }
}


using System;

namespace ch.ethz.se
{
  public class Customer : Person
  {
```

```
    string type;

    public Customer()
    {
    }

    public string Type
    {
      get
      {
        return type;
      }
      set
      {
        type = value;
      }
    }
  }
}

using System;

namespace ch.ethz.se
{
  public class Employee : Person
  {
    private float salary;
    private String position;
    private String workingDepartment;
    private Branch workingBranch;

    public Employee()
    {
    }

    public float Salary
    {
      get
      {
        return salary;
```

```
  }
  set
  {
    salary = value;
  }
}

public string Position
{
  get
  {
    return position;
  }
  set
  {
    position = value;
  }
}

public string WorkingDepartment
{
  get
  {
    return workingDepartment;
  }
  set
  {
    workingDepartment = value;
  }
}

public Branch WorkingBranch
{
  get
  {
    return workingBranch;
  }
  set
  {
    workingBranch = value;
```

```
        }
      }
    }
  }

  using System;

  namespace ch.ethz.se
  {
    public class Person
    {
      long id;
      string name;
      DateTime birthdate;
      string gender;
      string nationality;


      public Person()
      {
      }

      public long Id
      {
        get
        {
          return id;
        }
        set
        {
          id = value;
        }
      }

      public string Name
      {
        get
        {
          return name;
        }
```

```
  set
  {
    name = value;
  }
}

public DateTime Birthdate
{
  get
  {
    return birthdate;
  }
  set
  {
    birthdate = value;
  }
}

public string Gender
{
  set
  {
    gender = value;
  }
  get
  {
    return gender;
  }
}

public string Nationality
{
  set
  {
    nationality = value;
  }
  get
  {
    return nationality;
  }
```

```
    }

    override public string ToString()
    {
      return name + "/" + gender;
    }
  }
}
```

The test class implemented for db4o.

```
using System;
using System.IO;
using com.db4o;
using com.db4o.f1;
using com.db4o.query;
using System.Collections;

namespace ch.ethz.se.main
{
  public class MainController
  {
    private ObjectContainer db;

    public MainController(ObjectContainer db)
    {
      this.db = db;
    }

    private void createPerson(
      String name,
      DateTime birthdate,
      String gender,
      String nationality)
    {
      Person person = new Person();
      person.Name = name;
      person.Birthdate = birthdate;
      person.Gender = gender;
      person.Nationality = nationality;
```

```
  if(db.get(person).next() == null)
  {
    db.set(person);
    db.commit();
  }
  else
  {
    throw new AlreadyExistException("This Person already exists!");
  }
}

private void createCustomer(
  String name,
  DateTime birthdate,
  String gender,
  String nationality,
  String type)
{
  Customer customer = new Customer();
  customer.Name = name;
  customer.Birthdate = birthdate;
  customer.Gender = gender;
  customer.Nationality = nationality;
  customer.Type = type;

  db.set(customer);
  db.commit();
}

private void createEmployee(
  String name,
  DateTime birthdate,
  String gender,
  String nationality,
  float salary,
  String position,
  String workingDepartment,
  Branch workingBranch)
{
```

```
  Employee employee = new Employee();
  employee.Name = name;
  employee.Birthdate = birthdate;
  employee.Gender = gender;
  employee.Nationality = nationality;
  employee.Salary = salary;
  employee.Position = position;
  employee.WorkingDepartment = workingDepartment;
  employee.WorkingBranch = workingBranch;

  db.set(employee);
  db.commit();

}

public void createBranch(String name)
{
  Branch branch = new Branch();
  branch.Name = name;


  db.set(branch);
  db.commit();

}

public void createAccountType(String name)
{
  AccountType accountType = new AccountType();
  accountType.Name = name;

  db.set(accountType);
  db.commit();
}

public void createAccount(
  Customer owner,
  Branch branch,
  AccountType type)
{
```

```
    Account account = new Account();
    account.Owner = owner;
    account.Branch = branch;
    account.Balance = 0;
    account.Type = type;

    db.set(account);
    db.commit();
  }

  public Branch getBranchByName(String name)
  {
    Branch proto = new Branch();
    proto.Name = name;
    Branch branch = (Branch) db.get(proto).next();
    db.commit();

    return branch;
  }

  public Customer getCustomerByName(String name)
  {
    Customer proto = new Customer();
    proto.Name = name;
    Customer customer = (Customer) db.get(proto).next();
    db.commit();

    return customer;
  }

  public AccountType getAccountTypeByName(String name)
{
    AccountType proto = new AccountType();
    proto.Name = name;
    AccountType accountType = (AccountType) db.get(proto).next();

    db.commit();
    return accountType;
  }
```

```
public ArrayList getEmployeesByBranch(Branch branch)
{
  Query query = db.query();
  query.constrain(typeof(Employee));
  query.descend("workingBranch").constrain(branch.Name);
  ObjectSet os = query.execute();
  ArrayList list = new ArrayList();
  while (os.hasNext())
  {
    list.Add((Employee) os.next());
  }
  db.commit();
  return list;
}

public ArrayList getAccountsByCustomer(Customer customer)
{
  Query query = db.query();
  query.constrain(typeof(Account));
  query.descend("owner").descend("name").constrain(customer.Name);
  ObjectSet os = query.execute();
  ArrayList list = new ArrayList();
  while (os.hasNext())
  {
    list.Add((Account) os.next());
  }
  db.commit();
  return list;
}

public void close()
{
  db.close();
}

ublic static void Main()
{
  ObjectContainer db = Db4o.openFile(Util.YapFileName);
  MainController main = new MainController(db);
```

```
main.createBranch("Zurich");
main.createBranch("Bern");
main.createBranch("Geneve");
main.createBranch("Luzern");

main.createAccountType("Savings Account");
main.createAccountType("Checking Account");
main.createAccountType("Loan Account");

DateTime date = new DateTime(1979, 2, 26);
try
{
  main.createPerson("Joao Tobias", date, "male", "Brazilian");
}
catch(AlreadyExistException aee)
{
  Console.WriteLine(aee.Message);
}

date = new DateTime(1981,3,1);
main.createCustomer("Ana Hickmann", date, "female", "Brazilian", "Exclusiv");

date = new DateTime(1975, 6, 4);

main.createEmployee("Angelina Jolie",
  date,
  "female",
  "American",
  100000,
  "Manager",
  "Remittance",
  main.getBranchByName("Zurich"));

date = new DateTime(1972, 2, 16);

main.createEmployee("Sarah Clarke",
  date,
  "female",
  "American",
```

```
    200000,
    "Director",
    "Remittance",
    main.getBranchByName("Zurich"));

  date = new DateTime(1979, 2, 21);

  main.createEmployee("Jennifer Love Hewitt",
    date,
    "female",
    "American",
    200000,
    "Cashier",
    "Front-office",
    main.getBranchByName("Zurich"));

  date = new DateTime(1956, 7, 9);

  main.createEmployee("Tom Hanks",
    date,
    "male",
    "American",
    200000,
    "Director",
    "Marketing",
    main.getBranchByName("Bern"));

  Console.WriteLine("Employees working in Zurich");

  Branch bp = new Branch();
  bp.Name = "Zurich";
  ArrayList list = main.getEmployeesByBranch(bp);

  foreach(Employee emp in list)
  {
    Console.WriteLine("- " + emp.Name);
  }

  main.createAccount(
    main.getCustomerByName("Ana Hickmann"),
```

```
        main.getBranchByName("Zurich"),
        main.getAccountTypeByName("Savings Account"));
      main.createAccount(
        main.getCustomerByName("Ana Hickmann"),
        main.getBranchByName("Zurich"),
        main.getAccountTypeByName("Checking Account"));
      main.createAccount(
        main.getCustomerByName("Ana Hickmann"),
        main.getBranchByName("Zurich"),
        main.getAccountTypeByName("Loan Account"));

      Customer ana = main.getCustomerByName("Ana Hickmann");

      list = main.getAccountsByCustomer(ana);

      Console.WriteLine(ana.Name + " has " + list.Count + " accounts");
      foreach(Account acc in list)
      {
        Console.WriteLine("- " + acc.Type.Name);
      }

      db.delete(typeof(Person));
      db.commit();
      main.close();
    }
  }
}
```